IS1220HB
SOFTWARE ENGINEERING
Final Report: Part I

An in-depth review of main characteristics,
design decisions and patterns of an
Enjoy-Your-Meal System

Paris, France
2016

Written by
LOH CHENG WAI MATHIAS
XIONG TIANKAI

In association with

CentraleSupélec

Contents

1 Foreword **3**

2 Overview **4**

2.1 Client Specifications 4

2.2 Organization of the EYMS Core 5

3 Main Characteristics **5**

3.1 `CoreSystem` 5

3.1.1 An in-depth look into `RestaurantSystem` 6

3.2 Users 7

3.2.1 `ClientUser` 7

3.2.2 `StaffUSer` 7

3.3 `Meals` 7

3.3.1 Pricing model 8

3.3.2 Price change mechanism 8

3.3.3 Ability to change ingredients 8

3.3.4 `Ingredient` class 8

3.4 `Order` 9

3.4.1 Making an order 9

3.4.2 Payment of order 9

3.5 `FidelityCard` 9

3.5.1 `BasicFidelityCard` 10
3.5.2 `PointFidelityCard` 10
3.5.3 `LotteryFidelityCard` 10
3.6 `Update` 10

4 Design patterns and model technicalities **11**
4.1 Factory Method Design Pattern 11
4.1.1 Meals analysis 11
4.1.2 Ingredients analysis 12
4.2 Strategy Design Pattern 13
4.2.1 Meals/Order/Ingredients analysis 13
4.3 Observer Design Pattern 14
4.3.1 User/RestaurantSystem/Subscriber/Publisher analysis 15

5 Moving forward **16**
5.1 Other possible implementations 16
5.1.1 Fidelity Card System 16
5.1.2 Order System 16
5.2 CLUI/GUI 16
5.3 Testing 17

6 Conclusion **17**

1 Foreword

As commissioned by the client, **Enjoy Your Meal System** (EYMS) is a project initiative aimed to develop a management software for restaurant chains according to client specifications. This project has been divided into 2 key parts, the first phase consists of the EYMS core system, which would meet all technical requirements specified by it. It would then be completed by phase 2, which serves mainly to implement a user interface to provide a platform for easy interaction between EYMS core and the client.

This report aims to guide the reader through the design process of our EYMS core. The report commences with an overview of the project, through a reconciliation of client specifications followed by how we organized the core system files. The main characteristics of our report are then demonstrated and explained. Following which, the key decision ideas and principles chosen are expounded upon and explained in details, as well as to make known the limitations of our design. Finally, we proceed to discuss about alternative and future implementations, the development status of our CLUI/GUI as well as a short note on the testing of our system before closing out with our conclusion.

It should also be noted that while phase I serves a nice midway point of the development of our system, it is still in *development phase*, and as such, our code provided may not be as well up kept as of current implementation. We seek and thank you for your understanding and patience.

Written: 20/03/2016

2 Overview

This software hosts a suite of different functions and capabilities, and in this document we aim to showcase these features and the thought processes behind our design choices. We will then conclude the paper with possible areas of improvements as well as limitations of our current design model.

We started the project by sketching a **Unified Modeling Language** (UML) document which contains the initial blueprints of all the models we intended to create, as well as the possible relationship between different objects and entities. The UML has since undergone many changes as we changed certain viewpoints of how we wanted to implement the system, and as of submission date, contains the latest model of our project.

The software was then entirely written in **Java** and along with this report, a **Javadocs** documentation has been created for the ease of use and understanding of the classes and methods of our project. Finally, in order to withstand different user-based scenarios, we have also implemented **JUnit** test cases in hopes of maximizing the minimization of errors in our methods.

2.1 Client Specifications

Before we begin analysing the in-depth details of the project, it is imperative for us to first identify the key stakeholders of the problem posed to us. By understanding the roles they play, we would then be able to draw clearer relationships between them.

Without a doubt, we would definitely need to create a `User` object that have access to the main features of our system. This `User` could either be a `Client`, the patron of the aforementioned restaurant, or a `Staff`, a person who manages the system and/or the restaurant in general.

We would also need to take into consideration the food choices that are provided by the restaurant, or otherwise, the food preferences that are sought after by `Users`. We would name this stakeholder the different `Meals` of the system. We have also taken into account the different `Ingredient` objects required by each `Meals` object.

The association between `User` and `Meals` would not be possible without a means of payment. This will be done through an `Orders` system, where transactions would

take place. It would also allow for `Clients` to observe most recent orders, as well for `Staff` to make business choices by the popularity of their `Meals`.

To increase the number of returning `Clients`, membership cards will be issued in the form of different kinds of `FidelityCards`. Different `FidelityCards` exist to provide different, non-overlapping advantages. We would also need to implement an `Update` system in to notify and send new promotions and updates to clients automatically and/or manually.

Alas, to consolidate the different aspects that we have mentioned earlier, we have a `RestaurantSystem` that settles account issues, a `LoadSystem` that stores and manages vital information (such as different `Meals`, `Orders`, `Ingredients`, among others). A `RunSystem` is also required to ensure that the system would run.

2.2 Organization of the EYMS Core

We have decided to approach this project with a consideration of the characteristics and the possible behaviours of each stakeholder `Object`. Recite as aforementioned, we have five stakeholders:

- Core system (basically, our restaurant)
- Users
- Meals
- Orders
- Fidelity cards
- Updates

Having said, each of them represents a package that we have implemented in our project. Individual designs and internal relations of them will be addressed separately.

3 Main Characteristics

3.1 `CoreSystem`

This is the core package of our programme which will initialise the system upon launch (it contains the only `main()` class of the entire project hierarchy). We start

off with a `RestaurantSystem`, which handles a "database"of users. In addition, it also controls the log-in and register features of our system. A layer above, we have a `LoadSystem Thread` object which serves as a controller and reconciliation point for all our different "databases" (for example, aforementioned `RestaurantSystem`, an `OrderSystem` to handle orders, a list of `Users`, a list of `Ingredients` and `Meals` ...). Beyond the scope of our project requirements, we have also initiated a simple (but uncompleted) **Command Line User Interface** within `LoadSystem`, which contains an algorithm and control flow of processes when our program is used by an enduser. Currently, this serves as a developmental interface for us to verify the methods and scenarios that we have created. Finally, we have `RunSystem` which starts our `LoadSystem Thread`. In particular, this is also the point in which we pre-load existing `Meals`, `Staff`, `Ingredient` objects.

3.1.1 An in-depth look into `RestaurantSystem`

Consider myself being a user. I should be either a registered user or a new user. Hence the system should start with two options: "register" and "login". The system either addresses a registered user who is ready to log in, or convert a new user to a registered user and continue the process. If the user chooses to log in, the user should have access to certain operations which depends on the identity of the user. i.e. as a `StaffUser` or a normal `ClientUser`.

The majority of functions at system level are defined in the class `RestaurantSystem`. It stores a set of users(we use a set instead of an array to avoid duplicates) and functions to manipulate such set. One important function which the class should possess is `register()` which has been implemented as a series of enquiries about identity information and registration choices. With the information recorded, the programme can continue to create a new `User` object. Another indispensable function is `login()` which checks whether the input user name and password corresponds to a registered user which is stored in the system.

After the user has logged in, he has gained access to methods available to an `Users` object. (which automatically differentiates between a `ClientUser` and a `StaffUser`)

3.2 Users

Our proposed EYMS system would be accessible to both clients of the restaurant as well as the staff members. It is thus imperative for us to distinguish between these two as both of them should have their own set of commands.

Nonetheless, they also do share similar characteristics. That is to say, a `User` is characterized by his first and last names, username, password, email address, birthday, contact details, the ability to receive updates, his fidelity card, etc. We have thus created an abstract class for `User` which contains all the specifications of what a `User` should have.

3.2.1 `ClientUser`

A client should be able to purchase meals, which in our implementation is done through an `Order` system. He also owns a `FidelityCard` which can be changed at any time. We will get to these two implementations later.

3.2.2 `StaffUSer`

We note that a `StaffUser` IS-A `ClientUser`, simply because he should also be able to purchase meals. Notably, he is also equipped with additional functions that allows him to perform other things, such as to add new orders, set pricing on meals, set special pricing on meals, etc. We have implemented this by setting `RestaurantSystem` (*Recall: RestaurantSystem is the "database" for all users registered in our system, and contains methods that* `Users` *can achieve*) methods that are accessible only to `Users` which are instances of `StaffUser`.

3.3 `Meals`

Each meal is characterised by a unique name, a set of ingredients, a description and a price.

3.3.1 Pricing model

Every `Meal` instance can be instantiated by a name, a description, price (optional) and a variable amount of `Ingredient`s. If price is not defined, it would simply be calculated by the sum of price of all ingredients. Otherwise, the price can be set by a `StaffUser` when he logs into the system.

3.3.2 Price change mechanism

The restaurant may choose to make a special offer on any meal, hence changes the price temporarily. We would then want to switch back the price to the original when special offer is no longer available. Instead of modifying the price attribute repeatedly, we added a `specialPrice` attribute to `Meal` and a switch between `price` and `specialPrice`.

3.3.3 Ability to change ingredients

Some clients may require to remove or add certain ingredients of the meal. While enabling such operation, we also need to adjust the price accordingly. We calculate the new price according the price of the changed ingredients. In the case where the modified price is lower than the original price, the price will not be reduced so that the restaurant does not lose profit. Conversely, if a `User` adds additional `Ingredient`s, he would have to subsequently pay for the extra ingredients he has ordered.

3.3.4 `Ingredient` class

`Ingredient` is a class of its own, which is contained in (as an attribute) a `Set` in each `Meals`. Each `Ingredient` object is defined by it's name, quantity, and its price per quantity. This is primarily how we calculate the default price of a `Meal` if it is not defined by a `StaffUser`. We have chosen this approach instead of defining it in the meal to simplify the operations. To assist modification of ingredients in `Meals`, methods to add or remove certain quantities of the `Ingredient` are defined. Unit price is set and total price is calculated accordingly. To be practical and to remove persistent truncation errors that comes with the primitive java `double` attribute, our prices are **always** rounded to two decimal places (to the unit of cents).

3.4 Order

3.4.1 Making an order

An Order can be made by a User, it has information about several meals and their quantities. Instead of designing it as commonly interpreted as "one order" made by one User, we think it would be better to make it more general to be a sequence of orders. As such, when accessed by a ClientUser, it can be used to confirm the Meals ordered in one instance, through the display of the Meals, the Ingredients selected as well as individual price. Alternatively, when accessed by a chef who is a StaffUser, it can display a message of all orders that have been made.

3.4.2 Payment of order

Whenever a Meal is added to an order sequence, the price of the meal is immediately calculated, based on the algorithms provided by different FidelityCards that the user owns. The fidelity cards will then update their features accordingly. (To be explained in the next section)

3.5 FidelityCard

When considering the choice of design of FidelityCard, we realized that it shares a one-to-one relationship with our Users, that is to say, every FidelityCard contains a User, while every User must own a FidelityCard. Hence, we have created an abstract Fidelity card that possesses a similar User attribute.

We have different FidelityCards avaliable, and we have created subclasses according to different specifications required (to be explained below). In our eventual CLUI, we would allow a choice for Users to change their FidelityCard, however, we have also decided that such a change would nullify any unused perks they have achieved and **Users would not be able to continue progress with perks that they have achieved should they decide to revert at a later time.**

3.5.1 BasicFidelityCard

As required, we have created a `BasicFidelityCard` that every `User` will have by default upon the creation of an account. It allows the `User` to have access to `specialOffer`s whenever a staff declares one.

During payment, the `Order` system would check if the `Meal` object is on `specialOffer`, and would calculate the `price` accordingly.

3.5.2 PointFidelityCard

When a `PointFidelityCard` is chosen, the `User` would no longer be able to access `specialOffer`s, and the system would hence not check if `specialPrices` are available. Instead, the `User` would earn a point for every 10 units of money paid. When 100 points are accumulated, the system automatically deducts 10 percent off the next meal purchased. The points would then be reset.

3.5.3 LotteryFidelityCard

When a `LotteryFidelityCard` is chosen, the `User` would again not be able to receive any `specialPrice` offer, and any points collected would be set to 0. Instead, the `User` would be given a predetermined chance (set at 5 percent by default, can be changed by a `StaffUser`) to obtain his next `Meal` for free.

3.6 Update

The system should be able to send notifications to users who have subscribed to receive updates on special offers. Hence we created an `update` package to implement such functionality. Inside the package, a `Publisher` interface and a `Subscriber` interface are written. We have chosen `RestaurantSytem` to implement `Publisher` and our `User` to implement `Subscriber`. In particular, the `notifySubcriber()` method in `Publisher` is overridden in `RestaurantSytem` to generate information of current special offers and send it to the `Subscriber`. This implementation also checks for each subscriber if it is their birthday and sends birthday wishes (and a birthday offer discount, default value = 30 percent) if it returns true

In a simpler terms, consider the following scenario. Whenever a `User` logs into the system, our system is designed such that it will automatically determine if it is his birthday (granted, he has acknowledge to share the information with us previously). If it is, we would send him a birthday message. Subsequently, if he has also opted to receive updates for special offers, he would also receive it.

From a `StaffUser` viewpoint, he would be able send notifications to `Users` which would be stored until a `User` logs in and has read the message. We will discuss the technicalities of this implementation in the next section.

4 Design patterns and model technicalities

Design patterns represent best practices to tackle commonly encountered situations in programming. For this project, we have chosen 3 key design patterns to abide by the *Open-Close principle* as closely as possible. However, such implementations also brings about limitations, and we will take this same opportunity to discuss decisions we have taken that might not comply with aforementioned principle. We have chosen to use the following design patterns:

- Factory Method Design Pattern
- Strategy Design Pattern
- Observer-Observable Design Pattern

4.1 Factory Method Design Pattern

The factory method design pattern was used to create objects in our project. This allows us to extend the range of products with minor changes to existing code, and hides implementation details from end users.

Stakeholders involved: Meals, Ingredients

4.1.1 Meals analysis

All instances of concrete `Meals` (in this case, we have predefined certain subcategories of `Meals`, such as `Appertizer`, `MainCourse` and `Dessert`) are extensions of

the abstract superclass `Meal`. This abstract superclass imposes the requirement of a constructor on them that takes in a list of essential variables for the creation of any `Meal` instance.

Each type of concrete `Meal` item would have a concrete `Meal` factory (`AppertizerFactory`, `MainCourseFactory`, `DessertFactory`) dedicated to its creation, and all concrete `Meal` factories are extensions of the abstract superclass `MealFactory`. These `MealFactory`s provides the prototype for us to create concrete `Meal` objects.

We also introduced a separate factory builder class, `MealFactoryBuilder`, which contains all existing types of factories. The details of the these factories are encapsulated in our `MealFactoryBuilder` and would be presented to the end user without exposing him to any instantiation/implementation details. Also, eventually with the implementation of our CLUI, there would be little adjustments needed for the end user to create new `Meal` items. Since the details of our factories in the `MealFactoryBuilder` are encapsulated in a `HashMap`, the client simply needs to key in the type of `Meal` as a `String` (i.e, Appertizer, MainCourse, Dessert) as one of the parameters during item creation.

To summarize, using factory method design pattern with a `MealFactoryBuilder` does improve encapsulation and remove the need to make adjustments to client code. However, it does come with its flaws. Assume we were to create a new subcategory of `Meal` item, say a `Beverage`. Creation of a `BeverageFactory` requires us to write additional code by extending our concurrent `MealFactory`. As opposed to the use of a Factory Design Pattern or an Abstract Factory Design Pattern, the creation of a new `Beverage` class would require only the creation of new `Beverage` class by extending `Meals`. In other words, creation of new factories require **more code additions to our system files**.

While this violates the open-close principle to some extent, we stand by this choice of a factory method design pattern as in reality, there are only a fixed/finite number of subcategories of meals. Hence, in practice, there is really no need for us to add additional factories to our system code on a regular basis.

4.1.2 Ingredients analysis

On the other hand of the spectrum, we have an `Ingredients` class which contains a vast number of different archetypes (spice, salty, vegetable, meat, ...). This discourages a factory method design pattern because we would require to insert a lot of code

for different archetypes.

Similarly we would not need an abstract factory design pattern because in practice, we are not expecting the client to create any instance of an ingredient.

We now have a problem of creation of *fixed, predefined* list of ingredients which *does not require any subcategories*. Hence, we have decided to utilise a simple factory design pattern for the creation of new `Ingredient` object.

4.2 Strategy Design Pattern

Strategy design pattern is commonly used to provide a solution for scenarios where one would prefer a client program that needs to behave differently, depending on the circumstance. As compared to using inheritance, a strategy design pattern provides greater flexibility, and adheres to the open-close principle. This is done defining a family of strategies and providing encapsulation for each one of them. The strategies implemented are hidden from the client.

Stakeholders involved: Meals/Order/Ingredients

4.2.1 Meals/Order/Ingredients analysis

Whenever a meal is ordered by the client, he has the choice of adding new ingredients, removing ingredients, or change the quantity of the ingredients required. Due to our current design framework of our `Ingredient` and how it interacts with our `Meal` class (A meal object has a list of ingredients object), changing the ingredient would dynamically affect other attribute in our meal object (price will be changed, ingredient quantity will be forever changed, etc). This poses a problem when repeated `Meal` object is ordered by the system.

For example, suppose `UserClient` A would like to buy a `Appertizer` salad. He decides to add an additional ingredient (cheese) to the salad. Since salad is a predefined meal, it would dynamically change the ingredients of the salad. That is to say, when `UserClient` B comes along after A, and decides to order the salad, he would be exposed to a salad object with cheese (by default, it shouldn't have cheese).

By extension, there are a couple of different approaches to solve this problem. The most obvious method is to create a **deep copy** of our salad object. Then this new

salad object would be totally independent of the one that it precedes. However, making a deep copy requires *serialization* and this could cause many problems to our implementation as we would have there are many different aspects that we have to look after (the exact format is hard to keep stable, class changes can easily make serialized data unreadable, etc). This is especially problematic for a class supposedly dynamic (we have to constantly change ingredients).

Alternatively, we could produce the same result by recreating salad as a new `Appertizer` object with different ingredients used. However, this could cause a lot of overhead in performance because in reality, especially for a mega restaurant with hundreds of customers every day, it would require a creation and storage of many salad objects with different ingredients, which would inevitably affect the performance of the system over time.

Using a strategy design pattern allows us to solve this problem by removing the need to create new objects for different purposes (that is also to say, there will only be one and only one salad object that is constantly mutated to satisfy different client demands). By creating 4 different behaviors (`AddIngredient`, `RemoveIngredient`, `ChangeQuantity` and `NormalBehavior`), it allows us to dynamically access different states of a meal object while also offering the option of returning back to it's default ingredients and ingredient quantities.

This is also a good example of abiding by the open-close principle, as the addition of new Strategy classes would not affect existing strategies. The code is also easily expandable and minimal changes to existing code is required.

4.3 Observer Design Pattern

The observer design pattern provides an efficient solution when a number of different objects (observers) needed to be notified of a change of another object (observed). It defines the way the different observers are updated when changes are made to the observed object.

Stakeholders involved: Users (observer), RestaurantSystem (observable)

4.3.1 User/RestaurantSystem/Subscriber/Publisher analysis

The need for automatic receiving of updates as a user logs into the system immediately corresponds to an **Observer Design Pattern**. However, instead of the usual implementation of an `Observable` and `Observer` class, we have created our own implementations in the form of a `Publisher` and a `Subscriber` to better suit the features that we need. We call the restaurant a `Publisher` and each `User` is regarded as a `Subscriber`.

The interface `Publisher` which is implemented by `RestaurantSystem` possesses three methods to be overridden: `subscribe(Subscriber sub)`, `unsubscribe(Subscriber unsub)`, and `notifySubscriber()`. Inside `RestaurantSystem`, they are overridden as:

- `subscribe(Subscriber sub)`: add a `Subscriber` object to the `subscriber_list` of the `RestaurantSystem`.
- `unsubscribe(Subscriber unsub)`: remove the particular `Subscriber` object from the `subscriber_list` of the `RestaurantSystem`.
- `notifySubscriber()`: send notifications to all `Subscriber` in the `subscriber_list` of the `RestaurantSystem`.

When the `RestaurantSystem` use the `notifySubscriber()` method, it compose the current special offer, then generate personalised message for each of the `Subscriber`.

In order to allow the user to be notified once logged in, we wrote a `popupMessage(Subscriber sub)` method inside `RestaurantSystem` and execute it for the user logged in.

Both `notifySubscriber()` and `popupMessage(Subscriber sub)` will check if the `Subscriber` agrees to receive birthday offer and if that day is his birthday. `birthdayOffer` is set to have higher priority than `normaldayOffer`.

This implementation ensures that each `User` and `RestaurantSystem` would have the updated restaurant special offers attributed to them. Having this implementation emulates closely systems that we see commonly in our lives everyday (for example we will always immediately receive special offers from websites whenever they publish them). It also automates the process of which a restaurant sends promotions instead of having to individually check the database of users everyday.

5 Moving forward

5.1 Other possible implementations

Due to time constraints, our phase I implementations may not be as ideal as we would imagine it to be. However, going into phase II of the project, there are certainly things that we might revisit and change in addition to the implementation of a user interface.

5.1.1 Fidelity Card System

We could implement a Strategy Design Pattern for our fidelity card system, that way we could only have one fidelity card with different behaviors (Basic, Point and Lottery) based on user's subscription. This adheres better to the open-close principle as right now if we were to add a new fidelity card type, we would have to create an entire new class to adhere to this change.

5.1.2 Order System

We could implement a Visitor Design Pattern is used when we have to perform an operation on a group of similar yet different objects. Visitors are representations of operations to be performed on the elements of an object structure. It allows us to define a new operation without changing the classes of the elements on which it operates, hence localising the change. Currently, our Order System "hard codes" the different pricing methods based on the different Fidelity Cards used. This is terrible practise as if we were to create a new Fidelity Card type, then we would have to reconsider the entire system code in order to implement it. However, with a visitor design pattern this can be prevented and it would allow an easier management of payment calculation for different card types in the future.

5.2 CLUI/GUI

Moving onto phase II, we also look to implement a command line implementation and possibly a graphical implementation of our EYMS Core. Our current (non-robust) implementation can be found in `LoadSystem` and `RunSystem`.

5.3 Testing

As of this iteration, our tests are only located in various JUnit files found in our system library, most of which aim to test the relational correspondence between targeted chosen stakeholders.

This is done as part of our Test Driven Development efforts as we are still in a developmental phase. Although incomplete, we suggest using `RunSystem` as a make-do test drive of the processes that we have implemented. It features a simple user prompt that allows us to register an account / log in to an account amongst other features. Unfortunately, it is still a **work in progress** and we will only be able to truly present a working test scenario in the next iteration (Phase II).

6 Conclusion

The EYMS Core contains the core functions that a restaurant meal system should possess. The LoadSystem/RunSystem class functions currently as an **intermediary platform** for the testing of the EYMS Core. However, one should note that it is still a work in progress (**many features are still not implemented**) and would serve as the foundation on which we build our user-interface for the next iteration.