Class: 1819\_GRLA08002\_01

**Assignment**: Software Engineering Assignment 2 - A context-aware quick lunch recommendation system

**Student Name**: Grant Martin

**Student Number**: B00340010

**Institution**: University of the West of Scotland

**Campus**: Lanarkshire

HastEaters – Software Implementation

A context-aware quick lunch recommendation system

Purpose of document: This document details how the software developers implemented the software to generate lunch time restaurant recommendations based on the users inputted preferences and the design patterns used to construct this programme and how the code was tested.

Link GitHub Repository: <https://github.com/grantmartin96/RecommendationComponent>

Contents

[Introduction 3](#_Toc4708695)

[Development Request 3](#_Toc4708696)

[Main User Cases for the Recommendation Component of the HastEaters Mobile Application 4](#_Toc4708697)

[Inputting Eating Preferences 4](#_Toc4708698)

[Recommend A Restaurant 5](#_Toc4708699)

[Original Software Architecture Design for HastEaters Recommendation Component 6](#_Toc4708700)

[Class Diagram of Original Design 7](#_Toc4708701)

[New Software Architecture Design for Implementation 8](#_Toc4708702)

[Preference Class 8](#_Toc4708703)

[Restaurant Class 9](#_Toc4708704)

[Recommendation Class 9](#_Toc4708705)

[User Class 10](#_Toc4708706)

[Design Method 11](#_Toc4708707)

[What are Design Patterns? 11](#_Toc4708708)

[What is the Factory Method Design Method? 11](#_Toc4708709)

[What is the Decorator Design Method? 11](#_Toc4708710)

[Implementation of Design Patterns: Factory Method 12](#_Toc4708711)

[Creator Classes Explanation 12](#_Toc4708712)

[Product Classes Explanation 12](#_Toc4708713)

[Implementation of Design Patterns: Decorator Method 13](#_Toc4708714)

[Preference Class Diagram Explained 13](#_Toc4708715)

[Why I decided to use the Decorator Pattern 13](#_Toc4708716)

[Package Diagram of Recommendation Component 14](#_Toc4708717)

[Business Package 14](#_Toc4708718)

[UnitTesting Package 15](#_Toc4708719)

[MockTesting Package 15](#_Toc4708720)

[Unit Testing 16](#_Toc4708721)

[Stubbing 16](#_Toc4708722)

[Conclusion 16](#_Toc4708723)

[References 18](#_Toc4708724)

# Introduction

Based on the HastEaters SAD (Software Architecture Document) produced for assignment 1 in trimester 1, I was requested to build and implement the functional code for the ‘Recommendation’ component of the lunch time restaurant recommendation HastEaters app designed. To successfully implement the code of the ‘Recommendation’ component I will need to analyse my original design and the assignment requirements to update and modify my original design to generate a new implemented solution for the ‘Recommendation’ component of the HastEaters app.

# Development Request

The implementation of the ‘Recommendation’ component has the following restrictions that must be followed during implementation, these restrictions were included as only the ‘Recommendation’ component of the app was being built at this moment so the app would not be fully functional to get data from API’s and other components of the HastEaters application.

* No user interface should be provided.
* You are expected to create an API that will provide services to the User Interface.
* A comprehensive Test harness must be provided with the solution.
* Recommendation is handled by a third party who will provide the following procedures:
  + A GetRecommendation services that accepts location and distance and returns the name and location of a recommended restaurant and the expected wait time.
  + This system is not operational yet, you must design and build your unit and integration test around this limitation.

# Main User Cases for the Recommendation Component of the HastEaters Mobile Application

## Inputting Eating Preferences

**Description**: The user can complete their eating preferences in the app by completing a form and the forms will be stored within the app, as the user will be able to save their preferences. This form will consist of fields labelled: Dietary restrictions (text field), Allergy information (text field), type of restaurant (sit in or take away), Cuisine (dropdown list with options: Italian, Asian, Chinese, American, etc), type of foods (dropdown list with options: burgers, pizza, sandwiches, etc). Once the preferences have been submitted the app will display the user top preferences for lunch.

**Pre-requisites**:

* No user interface is required at this stage of implementation

**Conversational use case:**

1. The HastEaters app presents the user with the current preferences of the user (if the user has already submitted preferences in the past, the maximum number of preferences a customer can have is 3 at any one time).
2. The user selects (turn on or turn off) preferences
3. Periodically the mobile app requests the server to serve the current state of the preferences (will need to determine how often the app asks the server if the user has updated their preferences)

**Related use cases:**

* List the best matching restaurants (The app will return the top 3 restaurants for each user preferences)
* Recognize user preferences

## 

## Recommend A Restaurant

**Description**: The mobile app will recommend restaurants based on the user’s preferences, the users available time for lunch according to the user’s calendar and the total time for the round trip (including sit in or take away restaurants). User will be able to select 1 or many preferences and the app will return the top 3 restaurants for each selected preference.

**Pre-requisites:**

* No user interface is required at this stage of implementation
* The working code will not be connected to the user’s calendar at this stage in the implementation
* The app will not be able to communicate with Google Maps API and Google Awareness API at this stage in the implementation
* At this point in the implementation the algorithm for generating restaurant recommendations will not be functional

**Conversational use case**

1. The mobile app will present recommended restaurants sorted by lowest round-trip time.
2. Each recommendation must include the restaurant name, distance, round trip time and the current number of people at the restaurant.
3. If the user chooses a recommended restaurant, the app must provide the option to integrate the destination to the navigational application on the user’s mobile phone (this will be done using Google Maps)

**Related use cases**

* Light notification restaurant recommendation.

# Original Software Architecture Design for HastEaters Recommendation Component

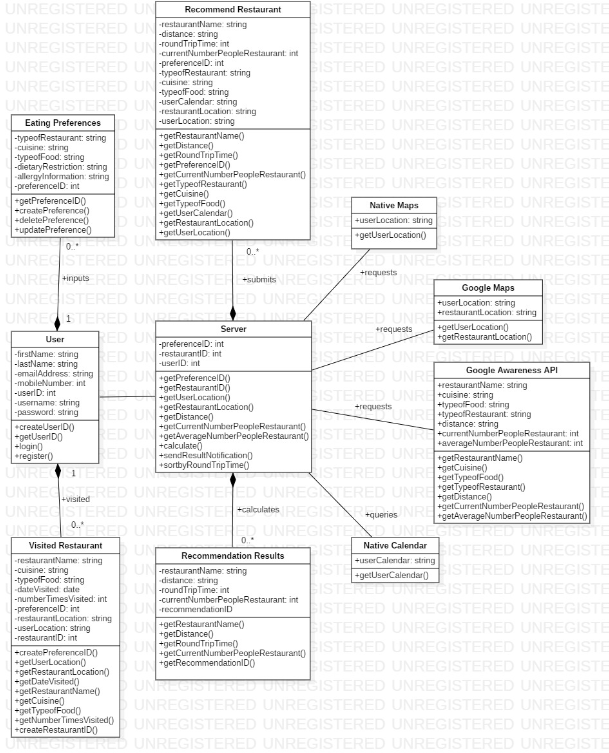
This section of the documentation will focus on detailing the ‘Recommendation’ component of the HastEaters application originally designed by myself, to see the full Software Architecture Document (SAD) please view the appendix document “SAD HastEaters Grant Martin Submission from trimester 1” submitted as part of this assignment submission.

Please note the original design proposal was based on implementing third party API’s and designing the implementation of the full app and not just the ‘Recommendation’ component.

Below I have detailed how my original design has changed and been modified to create a new design proposal to implement the Recommendation Component of the HastEaters application. Please see original class diagram on the following page.

# Class Diagram of Original Design

* The Classes highlighted with a green border, are classes which will be required to build the Recommendation Component
* The Classes highlighted with a red border, are classes which are not required as part of the Recommendation Component.
* The Classes highlighted with a blue border, are classes which will be mocked to provide mock data to allow for testing and proving the Recommendation Component solution is functional.



* The reason why we are not using the ‘Visited Restaurant’ class is because the user currently has no visited restaurants, also the users visited restaurants would need to be stored in a database and then retrieved when called and this database has not yet been designed or implemented.
* The Reason why the following Classes: Native Maps/ Google Maps/ Native Calendar/ Google Awareness API are not needed to implement the Recommendation Component is as part of this assignment the Recommendation Component is not required to interact with external third party API’s, instead we will mock this data during implementation.
* The Reason why we don’t need the ‘User’ class is that it is only contains information about the user and is not used as part of the Recommendation Component.

# New Software Architecture Design for Implementation

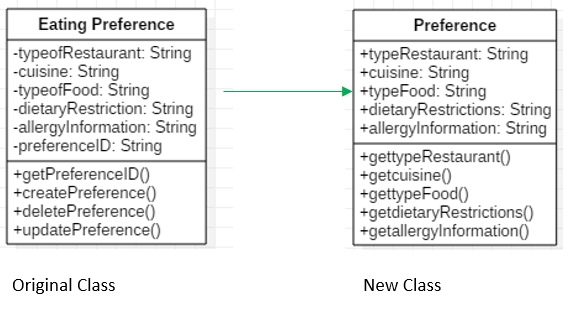
This section of the document will compare the original design and will show what changes were made to the original design to allow me to implement the working code for the Recommendation Component of the HastEaters application.

The design delivered in this section of the documentation is the design used to build and implement the Recommendation Component. This new and improved design also considers Design Patterns; however, the original Software Architecture Document did not consider any Design Patterns or restrictions in using third party API’s during implementation and the original design attempted to cover the full implementation of the HastEaters application.

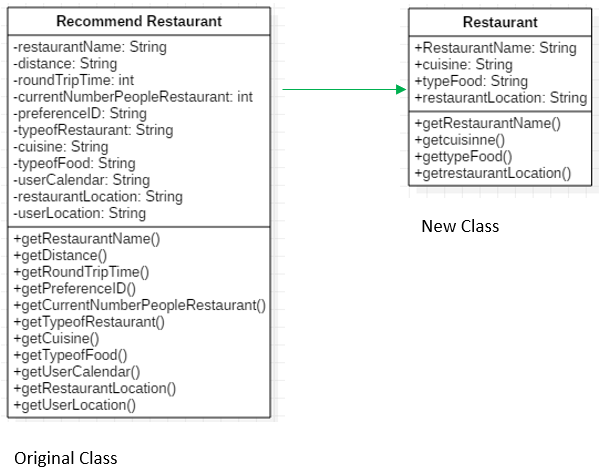
The next stage in the Software Architecture Design of the Recommendation Component was deciding which classes to use based on the original design for the implementation of the Recommendation component. The classes below show my original classes design and the new updated classes for implementation of the Recommendation component.

Firstly, I decided I needed the following main classes: Preferences, Restaurant, and Recommendation. Next, I analysed my original classes and the Recommendation component requirements to generate updated classes which would allow me to build and implement the Recommendation component of the HastEaters application.

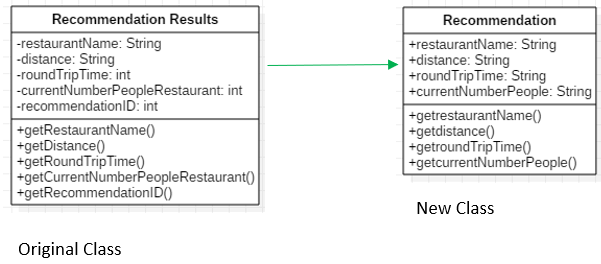
## Preference Class



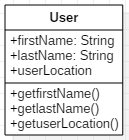
## Restaurant Class



## Recommendation Class



## User Class



A newly designed “User” class was implemented to identify the user of the application, but most importantly to retrieve the user’s location as the users live location is needed to perform the calculation to return restaurant recommendations.

# Design Method

For the implementation of the ‘Recommendation’ component of the HastEaters application, the functional code for the ‘Recommendation’ package is written to follow the Creational Design Method: The Factory Method. The functional code for the ‘Preference’ package is written to follow the Structural Design Method: Decorator.

## What are Design Patterns?

[1] Design patterns represent coding best practices which are used and understood by experienced developers. They also represent solutions to common code software problems.

Design Patterns in Software Development were first introduced by the ‘Gang of Four’ in the book entitled “Design Patterns - Elements of Reusable Object-Oriented Software” this book established common design patterns which are commonly used today. The design patterns were divided into three categories: Creational, Behavioural and Structural. [1]

## What is the Factory Method Design Method?

**[2] The Factory Method Pattern defines an interface for creating an object, but let’s subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses. [2]**

Definition of Factory Pattern defined by the Gang of Four in Design Patterns - Elements of Reusable Object-Oriented Software:

[3] “Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses” [3]

## What is the Decorator Design Method?

[4] The Decorator pattern is a Structural Design Method, which allows developers to add new functionality to existing objects without changing the existing structure. This pattern also acts as a wrapper to the existing class, adding extra functionality. [4]

Definition of Decorator Pattern defined by the Gang of Four in Design Patterns - Elements of Reusable Object-Oriented Software:

[3] “Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviours” [3]

# Implementation of Design Patterns: Factory Method

### Creator Classes Explanation

[2] RecommendationStore: The ‘RecommendationStore’ is an abstract creator class, which defines the abstract method for its subclasses to implement, to generate restaurant recommendations.

RecommendationStoreBuilder: This class is a ‘Concrete Class’. Concrete Classes produce products and in this case they product restaurant recommendations.

RecommendationStoreBuilder: The ‘createRecommendation()’ method is the factory method used to produce restaurant recommendation.

### Product Classes Explanation

Recommendation: The factory (RecommendationStoreBuilder) produces the restaurant recommendations and in the ‘RecommendationStore’ the product is ‘Recommendation’.

Recommendation1, Recommendation2, Recommendation3: These are the Concrete Products which are produced by the factory. [2]

# Implementation of Design Patterns: Decorator Method

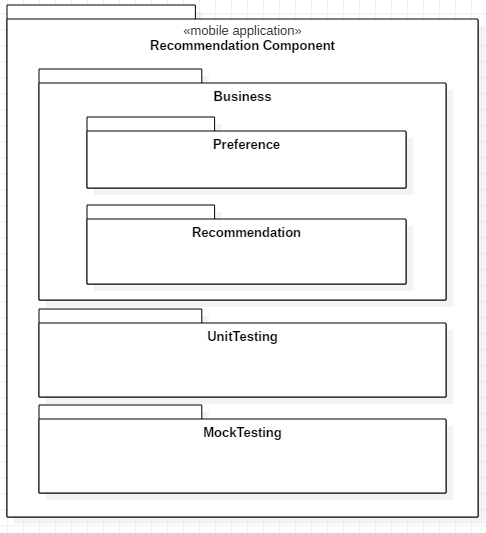
## Preference Class Diagram Explained

* [5] The Component “Preference” defines the “Preference” interface for the objects that can have responsibilities added to them.
* The Concrete Component classes highlighted in red defines the objects where additional responsibilities can be attached.
* The Decorator “PreferenceDecorator” maintains a reference to the component “Preference” object and defines an interface that conforms to the Component’s interface.
* The Concrete Decorator classes highlighted in green adds responsibilities to the component.

## Why I decided to use the Decorator Pattern

I decided to use the Decorator pattern because in the Preference package I needed a design pattern which could be easily maintained and could handle the creation of new functionality or in this case new user preferences being added/removed and edited without having to update all the classes contained within the Preference package and with the Decorator class (PreferenceDecorator) wrapping the Component class (Preference), the ConcreteDecorators classes (American and Indian) add new preferences to the original Component. [5]

# Package Diagram of Recommendation Component



The Recommendation Component implementation was divided into 3 main packages: Business, UnitTesting, MockTesting. With the Business package containing a further 2 sub-packages, so in total I have implemented 5 packages.

## Business Package

The Business package contains the business working code, which contains classes for creating user preferences, creating restaurant recommendation, getting restaurant information and the users live location, all of these classes and packages make up the main body of the mobile application and are responsible for generating the user’s restaurant recommendations. The sub-package “Preference” was implemented to contain the code for storing the user’s preferences which are then used to generate the restaurant recommendations. The “Preference” package uses the Structural design pattern the Decorator, this allows the user to easily add, remove and edit preferences which can be used to generate restaurant recommendations.

The sub-package “Recommendation” was implemented to contain the code for the Factory Design Method, which was used to generate restaurant recommendations based on the user’s preferences. The other two classes within the Business Package are: User and Restaurant. The User class gets the users location and the Restaurant class gets the information about the restaurant being recommended.

## UnitTesting Package

The UnitTesting contains all the unit tests of the Business code using JUnit 5 I was able to test each of the methods contained within the Business code classes.

## MockTesting Package

This package contains classes where I used test stubs to mock data results, as in the implementation of the Recommendation component I was not implementing third party API’s to provide information to my Business code. To test the Business code, I provide a mock response in a separate class and then called this mock response to verify that my expected results matched my actual result manually generated in my test stubs.

# Unit Testing

In the implementation of the Recommendation Component, I created a package called “UnitTesting” which contains unit tests for each of the classes contained within the Business package. To perform the unit tests I used JUnit 5 to run my tests. The unit tests test each individual method of an individual class in isolation to confirm that each individual method is working as expected and the ‘expected’ and ‘actual’ values match. This allowed me to analyse and identify bugs in my code early in the implementation process; as if the test failed I could compare the ‘expected’ and ‘actual’ result and easily identify and fix bugs until the test passed.

# Stubbing

In the implementation of the Recommendation Component I have created a package called “MockTesting” which contains the classes which mock responses from the third party API’s I would need to use in the actual application implementation. In these classes I use test stubs to provide mock data, which I have manually inputted into a class and verify that the test stubs are then retrieved when called in the test class. In the test classes for the mock API’s I used JUnit 5 to test the methods being tested retrieve the correct test stub, with the correct values by using “assertEquals” in JUnit 5 to verify that the ‘expected’ and ‘actual’ values for the methods match and successfully pass the test.

In the “MockTesting” package I mocked a response from Google Awareness API which is a third party library which gathers information about places and locations provided by Google. In the real implementation of the application Google Awareness API will be used to provide the information about the recommended restaurants. In this package I created a ‘GoogleAwarenessMockInterface’ and ‘GoogleAwarenessMock’ class where I created a test stub which provided a mock: “restaurantName”, “cuisine”, “typeFood” and “restaurantLocation” which I called in the class “GoogleAwarenessMockTest” to test my stub response against ‘expected’ results which I inputted within the test class.

In the “MockTesting” package I mocked a response from the Native Maps provider of the user’s smartphone device. In the real implementation of the application the Native Maps provider API will be used to provide the co-ordinates of the user’s live location. In this package I created a ‘NativeMapInterface’ and ‘NativeMapMock’ class where I created a test stub which provided a mock “userLocation” which I called in the class “NativeMapInterfaceTest” to test my stub response against ‘expected’ results which I inputted within the test class.

# Conclusion

In conclusion I believe I have delivered and implemented the Recommendation Component of the HastEaters application to a level which delivers proof of concept and the next steps to build the full implementation of the HastEaters app would be to start integrating third party API libraries instead of using test stubs for mocking data, start to implement UI (User Interface) so users can interact with the application and integrating all the independent business code, so that they communicate with each other to generate restaurant recommendations based on the users preferences inputted into the application.

To further improve the implementation of the Recommendation Component I would have liked to have been able to carry out test stubs using the Recommendation classes contained within the Business package and the Recommendation sub-package, however, I was unable to get test stubs to work as the RecommendationStore builder was an abstract class and needed to be public, but this change broke my Recommendation factory. If I had more time to implement the Recommendation component, I would have liked to have investigated a solution to this problem.

However, I do believe I carried out effective unit testing of the classes contained within the business package. I was also able to implement two design methods into the Business code, I was able to implement the creational design method, the factory method for producing restaurant recommendations and I was able to implement the structural design method, the decorator method for adding, removing and updating user preferences without impacting the structure of the ‘Preference’ package.

# References

[1] www.tutorialspoint.com. (2019). *Design Pattern Overview*. [online] Available at: https://www.tutorialspoint.com/design\_pattern/design\_pattern\_overview.htm [Accessed 27 Mar. 2019].

[2] O’Reilly | Safari. (2019). *Head First Design Patterns*. [online] Available at: https://www.oreilly.com/library/view/head-first-design/0596007124/ch04.html [Accessed 27 Mar. 2019].

[3] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (n.d.). *Design Patterns - Elements of Reusable Object-Oriented Software*.

[4] www.tutorialspoint.com. (2019). *Design Patterns Decorator Pattern*. [online] Available at: https://www.tutorialspoint.com/design\_pattern/decorator\_pattern.htm [Accessed 27 Mar. 2019].

[5] dzone.com. (2019). *Decorator Pattern Tutorial with Java Examples - DZone Java*. [online] Available at: https://dzone.com/articles/design-patterns-decorator [Accessed 27 Mar. 2019].