Healthy Eating App  
Detailed Design Description

**1 Introduction**

The purpose of this report is to provide detailed design models of the Healthy Eating application software. In particular, the work done in Part 1 of the project was reflected and expanded on. The scope of this report is limited to the software architecture summary, identification of modules and control flow, structural design specification and behavioral design specification. It should be noted that names of only the core classes are included. The key data structures and algorithms are mentioned for only the notable classes. Furthermore the report focuses on the implementation and reasoning behind the usage of only the most prominent design patterns and identification of the essential modules.

**2 Software Architecture Summary**

The software architectural style was changed from the repository style (as identified in Part 1 of the Project) to Model-View-Presenter. The reason behind this change was due to the fact that the team decided to use the Google Web Toolkit (GWT), a Java software development framework. Using this framework made it easier to focus on implementing design patterns and optimizing the Healthy Eating application. Furthermore, the framework ensured the presence of cross browser compatibility and other non-functional ‘nice to have’ features. The following text will provide brief overview of the newly adapted software architecture. The diagram below visualizes this new architectural style:

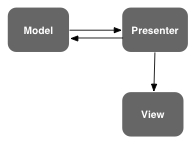


Figure: Diagram showing the MVP architectural style

The Model, View and Present can each be thought of as different subsystems that can be represented as separate packages. In the case of the Healthy Eating application the Model would consist of the Database Access Object where all queries to the database would be made. The View would consist of the different pages that comprise the application. The Presenter would house all the application’s logic including handling when to perform data synchronization and controlling view transitions. The main connectors include remote procedure calls to the server (between Presenter and Model), SQL queries to the database, call back routines and method calls between all three subsystems.

**3 Identifying Modules and Control Flow**

**3.1. Explanation of the Rationale Used in Module Identification and Encapsulation, and Selection of Applicable Design Patterns**

Modules were created in adherence to the Model-View-Presenter architectural style. Consequently, notable functionalities within the application had a View and Presenter class. For instance, Login had a LoginView class and a LoginPresenter class. The View class’ identification was made possible by the fact that GWT uses widgets in order to develop the application. Each widget is an individual class with certain attributes. In order to maintain cohesion and decrease coupling, widgets that served the same purpose were made part of one custom class. This custom class represented a widget composed of other widgets. It should be noted that this rationale also ensured that encapsulation was achieved seeing as each module contains the necessary resources to perform its main function (i.e. LoginView’s main function is to display Login information; this class has all the resources it needs to do so). Furthermore, the Presenter class simply has the logic associated with the corresponding View class. For instance, the LoginPresenter has the logic for user authentication.

Moreover, two structural design patterns, Composite and Proxy, were applied to refine the design of the software. The Composite design pattern was chosen because the application had widgets made up of other widgets. The following UML diagram further highlights the way the design pattern was used in the design:

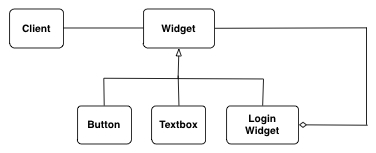


Figure: Example of use of the Composite design pattern in the application

In addition, the Proxy design pattern was used. In particular a protection proxy was utilized for authentication. Via the proxy design pattern, administrators were allowed unrestricted access whereas normal users were allowed to see only a subset of the Access object. The rationale behind the use of this pattern was to provide different actors with different accesses to the same object. The UML diagram below highlights how the design pattern was used in the application:

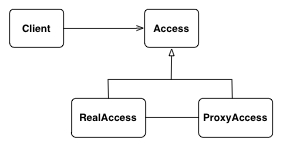


Figure: Example of use of the Proxy design pattern in application

**3.2 Overall Control Flow**

The overall control flow is centralized. The reason behind this is that the view transition logic is controlled by an onValueChange() method, this provides a centralized way of navigating within the app. For instance the onValueChange() method receives an event of type ValueChangeEvent. The value of this event is then stored inside a string token. Subsequently a series of if-else statements determine what the new view should be, based on what the token value is. In summary, seeing as navigating the whole application is controlled by one method, the control flow design is centralized.