Design Document for Pocket Chef

Group 10

Nate Howe

Tyler Cook

Scott Hood

Bryant Baltes

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Change** |
| 0.1 | 04/13/14 | NH | Initial Document |
| 0.2 | 04/13/14 | NH | Filled out basic information and introduction. |
| 0.3 | 04/13/14 | TC | Added in block diagram |
| 0.4 | 04/13/14 | NH | 5.1.1 through 5.1.3 |
| 0.5 | 04/13/14 | NH | First design issue |
| 0.6 | 4/13/14 | TC | Second design issue, User Interfaces and Data Access Interfaces |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 3](#_Toc84977455)

[1.1 Purpose 3](#_Toc84977456)

[1.2 Scope 3](#_Toc84977457)

[1.3 Definitions, Acronyms, Abbreviations 3](#_Toc84977458)

[1.4 Design Goals 3](#_Toc84977459)

[2 References 4](#_Toc84977460)

[3 Decomposition Description 4](#_Toc84977461)

[3.1 Module Decomposition 4](#_Toc84977462)

[3.2 Concurrent Process 4](#_Toc84977463)

[3.3 Data Decomposition 4](#_Toc84977464)

[3.4 STATES 4](#_Toc84977465)

[4 Dependency Description 4](#_Toc84977466)

[4.1 Intermodule Dependencies 4](#_Toc84977467)

[4.2 InterProcess Dependencies 4](#_Toc84977468)

[4.3 Data Dependencies 4](#_Toc84977469)

[5 Interface Description 4](#_Toc84977470)

[5.1 Module Interface **Error! Bookmark not defined.**](#_Toc84977471)

[5.2 Process Interface 4](#_Toc84977472)

[6 Detailed Design 4](#_Toc84977473)

[7 Design Rationale 4](#_Toc84977474)

[7.1 Design Issues 4](#_Toc84977475)

[7.2 <Issue 1> **Error! Bookmark not defined.**](#_Toc84977476)

[7.3 <Issue 1> 4](#_Toc84977477)

[8 Traceability 4](#_Toc84977478)

# Introduction

## Purpose

To describe interfaces between subsystems and design decisions made for Pocket Chef.

## Scope

The connections and interfaces between subsystems and components.

## Definitions, Acronyms, Abbreviations

|  |  |
| --- | --- |
| Term | Description |
|  |  |
|  |  |

## Design Goals

<Put in order of your team’s priorities. Also, explain what each item means and be very specific about each item you put in here.>

1. Response Time:

* Our application should be very responsive and quick to use, since it is supposed to save time for users.

1. Reliability

* Users should be able to rely on our application to provide meaningful data and results (scanned items should appear correctly, suggested recipes should be specific to each user).

1. Scalability

* With more recipes added to the application, and more users using it concurrently, our application should improve and scale well.

1. …
2. …..
3. ….
4. …..

# References

NA

# Decomposition Description

## Module Decomposition

### 

## Concurrent Process

<Give an overall description of the different processes, threads, instantiations>

### <Process 1 > Description

<Use collaboration diagram or sequence diagram to depict an use-case that uses this process(es)/thread(s). Also describe overall operation of the process. Also – describe what the threads do.>

### <Process 2> Description

<Repeat for as many subsystems that you have>

## Data Decomposition

### <Class 1> Description

### <Class 2> Description

## STATES

### <State/System 1 > Description

### <State/System 2> Description

# Dependency Description

## Intermodule Dependencies

## InterProcess Dependencies

## Data Dependencies

# Interface Description

## Module Interface

### Pocket Chef Application Interface.

### ExpirationManager getExpirationManager() -- Gets the ExpirationManager object, which is used to manage expiration notifications.

UserManager getUserManager() -- Gets the UserManager object, which is used to get and set the User object.

### Expiration Manager

### void start(ArrayList<Food> foods, Context c, NotificationManager nm, int userID) --Starts a multi-threaded expiration service execution.

### void finish() -- Cancels the asynchronous operation.

### boolean isDone() -- Checks if the asynchronous operation has finished

### User Manager

### User getUser() – Gets the current active user.

### void setUser() – Sets the current user.

### Food Inventory

###### addItem(Food item) – Adds an item to the inventory

toString() – Returns a string detailing information about each item in the inventory.

ArrayList<Food> getItems() – Returns the full ArrayList of Foods

### User Interface

### Int getuserID() – Returns the user ID of the object.

### String getUsername() – Returns the username of the object.

### String getPassword() – Returns the hashed password of the object.

### ArrayList<Recipe> getRecipes() – Returns the list of user’s recipes.

### Recipe getSelected() – Returns the previously selected recipe by the user.

### Boolean addRecipe(Recipe rec) – Takes in the recipe object to add to the user recipe list, and returns true if successful.

### Boolean addInventoryItem(Food item) – Takes in the food object to add to the inventory, returns true if successful.

### Data Access Interface

#### DataTable ConvertResultSetToDataTable(ResultSet rs) – Takes in an SQL result set data type and converts it to a Google API DataTable. Returns the new data table.

### Connection OpenConnection() – Opens a new connection to the database, returns the new connection.

### Void CloseConnection(Connection con, ResultSet rs) – Closes the connection and result set parameters (if applicable), returns nothing.

### Food

### getName() – returns the name of the food item.

### setName(String givenName) – sets the name of the food item to the given name

### sendFoodToDB(int userID) – sends the food item to the database based on the user id

### removeFoodFromDB(int foodID) – removes this food item from the database based on its foodID

### 

### List of External Interfaces:

### Google/Guava DataTable APIs

### ZBarScanner APIs

### JDBC APIs

## Process Interface

### <Process 1> Interface

### <Process 2> Interface

# Detailed Design

NOT REQUIRED <Java Docs to be used instead>

# Design Rationale

## Design Issues

### Data Flow

#### Description

### In Android, each “screen” comprises of an activity, which has its own encapsulated data and views. Data cannot explicitly be passed from one activity to another like we would in typical object-oriented applications. To pass data from one activity to another, we had to determine the best technique for our application. One object, the User, needed to be accessed and/or modified from each activity, entailing we would have to pass around the user from one activity from another, like handing off a baton.

#### Factors affecting Issue

### Android is the main factor causing the issue. In a typical object-oriented application, we could easily pass around objects without a problem. But in Android, each activity uses its own piece of memory, and objects created in one activity cannot be accessed by another unless we apply a technique to make the object visible or persist.

#### Alternatives and their pros and cons

a.) Java’s serialization API to pass an object from one activity from another by serializing the data.

Pros:

1. Simple and easiest solution to implement.
2. Rigid design.

Cons:

1. Slowest solution. The application would take a huge performance hit if we used this solution.

1. Handing off data from one activity from another. The code quickly becomes convoluted if every activity has to pass information to the next activity and back to the previous activity.

b.) Android’s parcelable API to also serialize the data and pass from one activity to another.

Pros:

1. Fast. Android optimized this form of serialization for devices using the OS.

2.) Robust design.

Cons:

1. Once again, it’s clunky to hand off one activity from another, passing information to the next activity and back to the previous one.

c.) Extending the application class with a singleton pattern, allowing us to access the object we needed at any time.

Pros:

1.) Fast

2.) Feels object-oriented

3.) If data is accessed by every activity, this approach is useful because it makes the      object accessible globally.

Cons:

1. If the object held by the singleton is changed by multiple threads or activities, thread-safety must be considered.
2. Full visibility. If the object is used by only a few activities, we might not want a global object.

3.) Clunky get and set methods.

#### Resolution of Issue

### Extending application class with singleton pattern. Since the object we need to access is the user and their information, it makes the most sense to make that object accessible on a global scale, as long as the application is alive. It cut down on repeated code in every activity, because every activity needed to modify the user’s information one way or another (their inventory, or their recipes, etc). The singleton is implemented in its own clean class, and the activities can simply get and set the information when needed. This ended up being one line of code versus twenty or more. Also, our code is easier to follow and much more maintainable using this solutions. And because responsiveness is a top goal for our project, serialization was not an optimal choice.

## <Issue 1>

### Description

### Factors affecting Issue

### Alternatives and their pros and cons

### Resolution of Issue

# Traceability

|  |  |  |
| --- | --- | --- |
| **No** | **Use Case/ Non-functional Description** | **Subsystem/Module/classes that handles it** |
| 1 |  |  |
| 2 |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

FEEL FREE TO ADD APPENDICES AS NEEDED. UPDATE TOC BEFORE SUBMITTING