PACEMAKER ANDROID APP – PATTERN SOLUTION

Design Patterns

Abstract

The purpose of this document is to outline the design patterns used in the creation of the Pacemaker Android Application

Colm Carew

Email: colmcarew2@gmail.com

Student ID: 20053766

Module: Design Patterns

Course: MSc. Communications Software

Date: 17/04/2016

Patterns Used

- **Creational Patterns**
  - Builder
  - Singleton
- **Structural Patterns**
  - Adapter
  - Façade
- **Behavioural Patterns**
  - Memento
  - Model View Controller
  - Strategy
- **Architectural Pattern**
  - Half Sync / Half Async

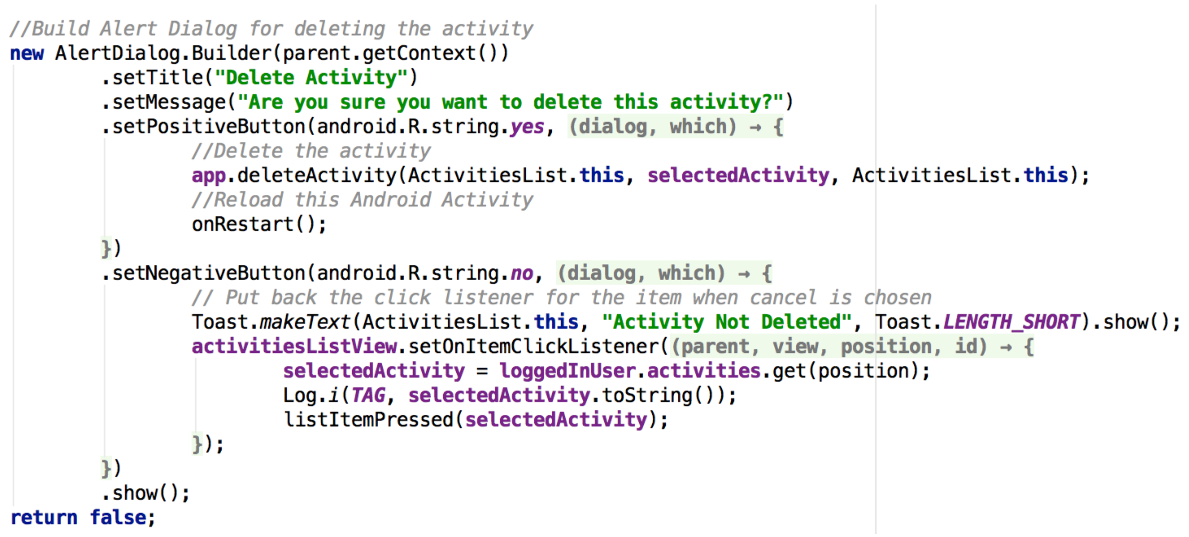
Builder

The builder pattern is a creational pattern used for creating complex objects.

The builder pattern solves the problem of creating an object with many parameters. Some of these parameters may be mandatory and others will be optional. This eliminates the need for multiple overridden constructors.

The builder pattern is robust and more maintainable however it is quite verbose and requires code duplication.

The builder pattern can be seen implemented in the ActivitiesList activity in the Android project.



```java
//Build Alert Dialog for deleting the activity
new AlertDialog.Builder(parent.getContext())
        .setTitle("Delete Activity")
        .setMessage("Are you sure you want to delete this activity?")
        .setPositiveButton(android.R.string.yes, (dialog, which) → {
                //Delete the activity
                app.deleteActivity(ActivitiesList.this, selectedActivity, ActivitiesList.this);
                //Reload this Android Activity
                onRestart();
        })
        .setNegativeButton(android.R.string.no, (dialog, which) → {
                // Put back the click listener for the item when cancel is chosen
                Toast.makeText(ActivitiesList.this, "Activity Not Deleted", Toast.LENGTH_SHORT).show();
                activitiesListView.setOnItemClickListener((parent, view, position, id) → {
                        selectedActivity = loggedInUser.activities.get(position);
                        Log.i(TAG, selectedActivity.toString());
                        listItemPressed(selectedActivity);
                });
        })
        .show();
return false;
```

Figure 1.0 : Builder Example

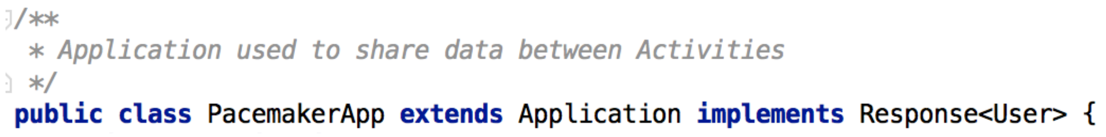
Singleton

The singleton patterns is a creational pattern and restricts the instantiation of a class to one object.

This is useful when one object is needed throughout the application.

In Android the Singleton can be accomplished via having a custom class extending the Application class. When the object of the extended application class is required in an activity they can be obtained by using the getApplication() method.

In the case of this Android App it is used for sharing the logged in user, their details and associated methods amongst the necessary Android Activities.



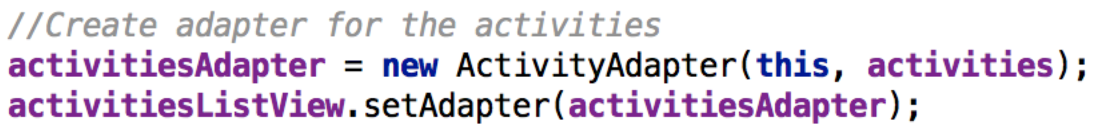
```
/**
 * Application used to share data between Activities
 */
public class PacemakerApp extends Application implements Response<User> {
```

Figure 2.0 Singleton Used in the Android App

Adapter

The adapter pattern is a structural pattern. It's purpose is to convert the interface of a class into another interface that the client expects.

In the case of this assessment the adapter pattern is used to display custom objects in a list view.



```
//Create adapter for the activities
activitiesAdapter = new ActivityAdapter(this, activities);
activitiesListView.setAdapter(activitiesAdapter);
```

Figure 3.0 : Activity Adapter being called

Façade

The face pattern is a structural pattern. The purpose of Façade is to shield the user from the complex details of the system and provide them with a simplified view for ease of use.

The best example of this from the assessment is the use of the Ion package. In order to set an ImageView to display an image obtained via GET the user would need to connect to the appropriate url, get the image + process it to become the appropriate object and then set the image view.

With Ion the user is shielded from this and all of the steps mentioned can be done in one simple line of code which can be seen below.

```
public static void setUserImage(ImageView imageView, String profilePhoto) {
    Ion.with(imageView).load(Rest.URL + "/getPicture/" + profilePhoto);
}
```

Figure 4.0 : Ion to set an ImageView from a photo obtain via GET

Memento

Memento is a behavioural pattern. The memento pattern is used to keep previous states of an object in memory.

In the case of this assessment it is used to pass objects from one Activity to another.
In one activity the object is converted to a JSON string and put into the next activity.

```
Gson gS = new Gson();
String target = gS.toJson(selectedActivity);
Intent showActivity = new Intent(this, ShowMyActivity.class);
//Put the JSON String into the next android activity
showActivity.putExtra(PacemakerENUMs.SELECTEDACTIVITY.toString(), target);
startActivity(showActivity);
```

Figure 5.0 : Putting JSON String to next activity

In the next activity the JSON string is pulled from the intent and converted back to the object it was in the last activity.

```
Gson gS = new Gson();
String target = getIntent().getStringExtra(PacemakerENUMs.SELECTEDACTIVITY.toString());
selectedActivity = gS.fromJson(target, MyActivity.class);
```

Figure 6.0 : Converting JSON string back to object

Mode View Controller

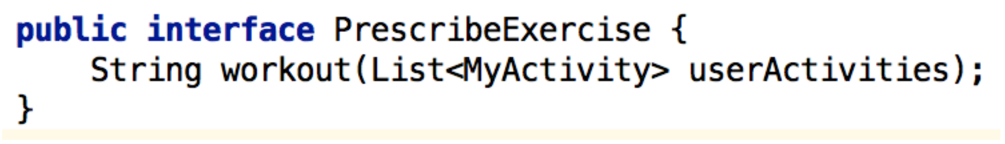
MVC is a behavioural pattern. The purpose of the MVC pattern is to split the software application into three interconnected parts the model, the view and the controller.
The model manages the data, logic and rules of the application (User, MyActivity, etc.). The view is the output representation (xml layout file). The controller (Android Activity) takes in input and converts it to commands for the model or view.

Strategy

The strategy pattern is a behavioural pattern. It allows for algorithms to be selected at runtime.

It works by having a class that uses an algorithm to contain a reference to an interface that has a method specifying a specific algorithm. Each implementation of the interface will implement a different algorithm.

In the assessment it is used for prescribing user workouts based on their desired fitness result.

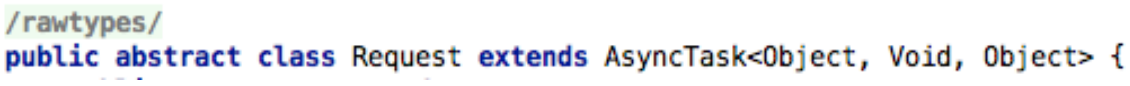


```
public interface PrescribeExercise {
    String workout(List<MyActivity> userActivities);
}
```

Figure 7.0 : Strategy Interface

Half Sync / Half Async

The purpose of this pattern is to process low-level system services asynchronously and simple application service processing synchronously. For the purpose of this assignment it allows REST requests to be performed in the background and not in the same thread as the current Android Activity. This means if a REST request fails, then the main activity does not crash.



```
/rawtypes/
public abstract class Request extends AsyncTask<Object, Void, Object> {
```

Figure 8.0 : Asynchronous Task