George Jone

8/13/2015

Final Essay - Software Design Patterns and Polymorphism Usage

I had not heard of software design patterns before taking 413. It seems like a topic that should at least have been mentioned in earlier courses so students could learn to recognize and implement them earlier on. The topic of polymorphism has been touched upon in Data Structures and Programming Methodology, but these courses never taught students how it would be applicable to real-world situations. Taking Software Development has given me the tools and knowledge required to effectively recognize these concepts and apply them to an android project. Throughout the development cycle of Food Roulette, my team and I have left numerous examples of polymorphism and design pattern usage.

A very powerful use of polymorphism in Food Roulette is the appstate, which our team referred to as the “tumor”. Named such because it is constantly active and readily accessible as long as the application is being used. The appstate extends the Application class. FoodRouletteApplication is the implementation of the tumor.

public class FoodRouletteApplication extends Application {

//This is the app state (tumor) file where all application data can be stored, until the app is terminated

// all non-primitives must be thread-safe

…

**Fig. 1.1:** FoodRouletteApplication Appstate

This class can be accessed and muted by any other class, method, or activity within Food Roulette. There are a multitude of objects which call the appstate. Our method populateUBERData is one of the many methods within our application that requires access to the global latitude and longitude variables provided by location services through the appstate. Based on the user’s location, Food Roulette will provide an estimate for the time it will take for an Uber pickup. Fig. 1.2 below is a snippet from our app that illustrates how the populateUBERData method is able to access the appstate’s global variables.

private void populateUBERData()

{

String currLat=Double.toString(\_appState.latitude);

String currLong=Double.toString(\_appState.longitude);

final String url1

="https://api.uber.com/v1/estimates/time?start\_latitude="+currLat

+"&start\_longitude="+currLong

+"&server\_token=VG3lVgGREzta1qdqqxR5dzHSRZPOFiZkYnBEpAXD";

**Fig. 1.2:** Accessing appstate’s Latitude and Longitude variables through populateUBERData

Another powerful example of Food Roulette’s usage of polymorphism is Asynctask. While we could have made the application without the use of Asynctask, we decided to incorporate it to make the application run more performantly and smoothly. Asynctasks were used throughout the application to safely thread and run heavy and time-intensive tasks concurrently. ImageLoadTask, shown below in Fig. 1.3, is one of these asynctasks. It takes a url from Yelp and displays a restaurant rating based on the url:

public class ImageLoadTask extends AsyncTask<Void, Void, Bitmap> {

private String url;

private ImageView imageView;

**Fig. 1.3:** Definition of ImageView extending AsyncTask

Shown below in Fig. 1.4, one of Asynctask’s methods doInBackGround() is overrided in ImageLoadTask, allowing for customized usage of the background tasks. In this case, we load the image in the background so it will display more immediately when the post-roulette activity is opened.

@Override

protected Bitmap doInBackground(Void... params) {

try {

URL urlConnection = new URL(url);

HttpURLConnection connection = (HttpURLConnection) urlConnection

.openConnection();

connection.setDoInput(true);

connection.connect();

InputStream input = connection.getInputStream();

Bitmap myBitmap = BitmapFactory.decodeStream(input);

return myBitmap;

} catch (Exception e) {

e.printStackTrace();

}

return null;

}

**Fig. 1.4:** doInBackground() is overrided

The Singleton is one of the design patterns we used in Foodroulette we used it in the DbAbstractionLayer class. This class was designed as a singleton because we need only one instance of a database object at once. In Fig. 2.1, the DbAbstractionLayer is shown. The class starts with a private static instance variable of type DbAbstractionLayer dbAbstractionLayer. A private constructor so that no outside class can access the singleton. Finally there is a public static method DbAbstractionLayer that returns the instance dbAbstractionLayer.

public class DbAbstractionLayer {

private static DbAbstractionLayer dbAbstractionLayer = null;

…

public static DbAbstractionLayer getDbAbstractionLayer(){

if (dbAbstractionLayer == null){

dbAbstractionLayer = new DbAbstractionLayer();

return dbAbstractionLayer;

}else{

return dbAbstractionLayer;

}

}

**Fig. 2.1:** Singleton usage in DbAbstractionLayer

We also used the builder pattern to create and modify Google map markers. The mapmakers tells the user where they are and where the business is located on the Google Map. The marker builder needs to have unique attributes for the marker’s icon, latitude, and longitude. The builder pattern makes it incredibly easy to make and modify the map markers since. Fig. 2.2 shows how we implemented the businessMarker.

businessMarker = mMap.addMarker(new MarkerOptions()

       .title(currentBusiness.name)

       .position(position)

.icon(BitmapDescriptorFactory.fromBitmap(businessIcon)));

**Fig. 2.2:** Builder pattern usage in businessMarker

This paper lists only a small sample of the polymorphism and software design patterns utilized in the creation of Food Roulette. Design patterns are prevalent throughout our application. So prevalent in fact, that I had just discovered one I had used without even knowing it! These tools allowed us to create a much more robust application in a much more refined way. Without these concepts, our code would be a complete mess, and much of it would not even work. Walking out of CSc 413, I feel much more confident in my abilities as a programmer to recognize usage of complex code concepts as well as utilize them.