Aidan Duffy

MET CS 665

Boston University

Assignment 5, Task 1: Description of Patterns of Twitter4J

Design Pattern #1: Singleton

In the TwitterAPIMonitor.java file, this utilizes a singleton design pattern where there can only be one API Monitor created in the program. The creator notes in their comments that they opted for a singleton design, but they could have used a factory design to avoid this. The function of this is simply to handle the parsing of URLs and what the developer calls "wire off logic". Excluding the constructor and get methods, the only method in this file is "methodCalled" which uses a pattern matcher to parse URL information. As this is a singleton with a fairly singular purpose, with the exclusion of creating an APIMonitorFactory, I did not see much room for adding additional files.

Design Pattern #2: Factory

Twitter4J utilizes a factory design for its implementation of Twitter's REST API with four files specifically. The two interfaces can be found in Twitter.java and TwitterBase.java. The actual factory is in the aptly named TwitterFactory.java, and the actual implementations can be found in the TwitterImpl.java and TwitterBaseImpl.java. The first interface, Twitter, lays the groundwork for the fundamental operations of the API for Twitter, such as pulling up a user's timeline or DMs. The TwitterBase interface sets the stage for background authorization and backend operations for users through getting user's unique IDs & screen names as well as the particular instance's authorization scheme and configuration information. The TwitterBaseImpl file implements the TwitterBase interface, among others, and is the base class for Twitter instances. These objects are tied to a specific instance of an associated user's login, storing screen name, id, URL information, the http client, as well as the authorization and configuration information. The methods implemented in this file initialize a Twitter instance based on a relationship to a factory object (init function), process http requests, return authorization tokens for outside methods to confirm what users have access to. Finally, TwitterImpl completes the vast majority of operations (see the sheer size of it on UML diagram…). An instance of this object – which is created from the TwitterFactory object as a singleton based on access tokens or authorization data – implements the Twitter interface and extends the TwitterBaseImpl class and handles all the following resources for the API: timelines, tweets, searches, DMs, followers, users, favorites, suggested users, user lists (subscribers, members, etc.), saved searches, geo/places, trends, spam reporting, and help.

The most obvious way to introduce new files would be to modularize this code as it is flexible and simple, but it is difficult to understand at times and all these operations (hundreds of methods) are all completed by this one object. I think that this code could be *significantly* more readable and easily understood if the developer distributed the operations among other objects that are dedicated to each of these tasks – or at least group them. For instance, search and saved search resources could be pooled to a secondary object, or timelines, tweets, and favorites could be pooled together or even places and trends as many of their methods rely on one another. As I mentioned, the fact that all these operations are centralized here makes the file too large and difficult to understand at times.

Design Pattern #3: Observer

The TwitterBaseImpl.java, as I mentioned above, implements the HttpResponseListener interface, specifically its httpResponseReceived method, which alters a number of objects based off of an HttpResponseEvent which is based off of the HttpResponse object. This listener, based off the HTTP event response, updates all of the existing rateLimitStatus objects as well as the status code object. This method in TwitterBaseImpl easily centralizes everything within this observer design pattern.

I did not see much room for adding additional files.