Glide

Existing Test Cases

Introduction

Glide has a relatively exhaustive test suite with a total of 1457 test cases. All the system's tests are located in the library-test folder as shown in Figure 1. Its test library contains subpackages that mirror those of the main library. For example, for the ThumbFetcher class in the mediastore file path, there is a JUnit ThumbFetcherTest class in the same file path convention, making it easy and intuitive to find the java class files for their corresponding JUnit test cases, and vice versa.
Additionally, we found another set of test classes under annotations in the root file that contains additional JUnit assertion testing for the AppGlideModule class as shown in Figure 2.

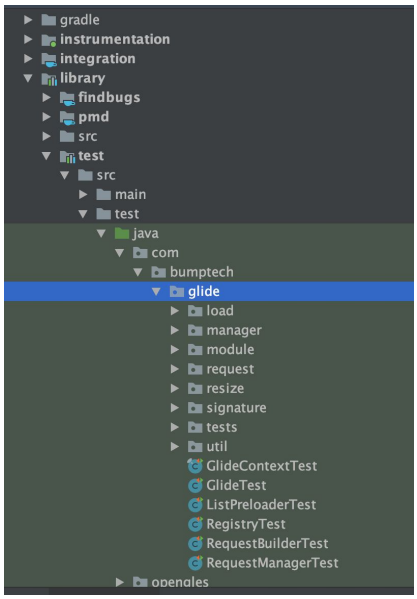

Figure 1

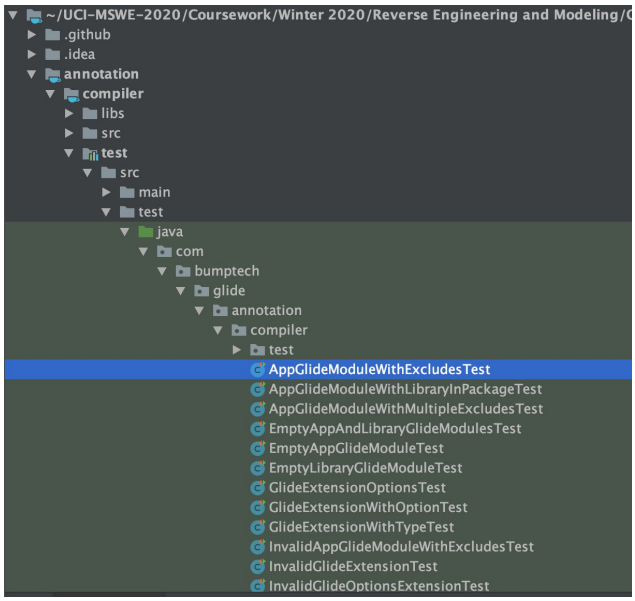

Figure 2

Existing Test Cases

Analysis

Types of Tests

We took a look at each of the test classes and determined that they were a combination of JUnit tests and Mockito mock tests. JUnit tests are typically utilized for functional testing. Mock tests are valuable in determining interactions between classes and

methods. Each test case is also run with the Robolectric framework, which enables writing and running tests on a continuous integration environment without additional set up.

Test Implementations

In functional testing, we want to observe expected outputs from methods in our system given specific inputs. JUnit allows us to test in this style, by asserting an expected value given certain method calls with particular input. A principle within functional testing is to use partitioning to select important input variables such as boundary values. The developers who wrote the tests exhibit this principle. For example in GlideUrlTest, the test covers various instances of http URL inputs to fetch image resources, such as an empty URL, null URL, URL created with different types, and so on. Each test covers a case of input value and asserts expected output values accordingly.

Test Cases

1. **StreamEncoderTest**

StreamEncoderTest tests the functionality of encoding a string data into a byte array input stream in the system's resource processing feature. It first sets up a stream encoder that uses LRU strategy to keep the pool under the maximum byte size. It also creates a new file that is based on the absolute path to the application specific cache directory on the file system that will help manage the cache memory in the disk space during run time. The test method then uses a fake string data, converts it into a byte array input stream, encodes it into the file mentioned earlier, the tests through an assert case to verify that the conversion of the byte array data input stream extracted from the file to a string value is equal to the initial fake string data. The file is then released at tear down.

This interesting test case substantiated our initial understanding of resource encoders and how they work. In addition, we learned that the getCacheDir() method is what manages the performance of local cache memory when the system gets low on memory. Tracing the method's usage in other files revealed that a similar method was being implemented for other file types such as cached photos (i.e. using getPhotoCacheDir()).

2. **StreamGifDecoderTest**

StreamGifDecoderTest tests the handling of situations in the gif resource decoding process under various situations. It tests when the resource input stream is enabled but

the resource is not recognized as a gif, when the gif header is not disabled but and the animation is not set, when gif animation is enabled, and when it is disabled. The set-up includes parsing the resource header into an array list, creating a StreamGifDecoder to convert the input stream to a byte buffer and pass the buffer to a wrapped decoder, and lastly creating a default option that sets the options for headers and animations for gif.

Throughout the test case, the decoder handling different situations based on the manipulation of options is tested to assert the expected outcomes. Since we previously did not look at how options can be manipulated through options.set() and overlooked the possibility that combinations of options, such as gif headers and gif animations can result in false outputs. The test case exemplified unique situations in which the combination of options set to a given gif decoder results in false handling.

3. **ListPreLoaderTest**

ListPreLoaderTest tests that the models needed to be loaded for the list of resources to display in the list adapter are loaded properly and have correct start and end positions. The method to pre load such items is set to users scrolling on the app interface. The tests cover various instances such as when the resource items are called in the order of decreasing and increasing item index, when the item index is less than zero, when the count of items has surpassed the total count of existing items, etc.

The test reveals the model provider and adapter design behind how the resource items are displayed on a scrollable interface. We learned through the test cases and the related classes that the model was designed to save on memory cache while giving the appearance of an infinitely large memory cache.