Team_Costco Homework5

Team_Costco: Duo Chai, Soobin Choi, Marc Andrada

Design Patterns in Glide

Factory

Example: src/Main/request/target/ImageViewTargetFactory

- Factory design pattern is a creational pattern that provides a way to create an object based on its sub-classes[1]. In Glide, the ImageViewTargetFactory defines the Target interface class for creating a View object. Since, in Glide, a Target type can display either a Bitmap or Drawable in the ImageView, the use of factory design pattern allows what kind of object gets set on the view. As such, the factory design pattern provides flexibility to the architecture of the program as the factory allows multiple types of ImageView Targets to be created.

Builder

Example: src/Main/RequestBuilder

- Builder pattern provides an alternative and easier way to construct complex objects when coders want to build different immutable objects using the same object building process[2]. Compared to Factory Pattern, Builder pattern provides more controls during the creation process. In Glide, RequestBuilder class defines multiple ways to build a request (a request is used for loading/displaying an image in Android UI). The use of Builder Pattern allows users to build a request with different attributes by calling the methods defined in RequestBuilder such as addListener(), transition(), error(), then use load() method to load the image. Figure 1 shows the sequence diagram when the MainActivity wants to create a GlideRequest to load an image.

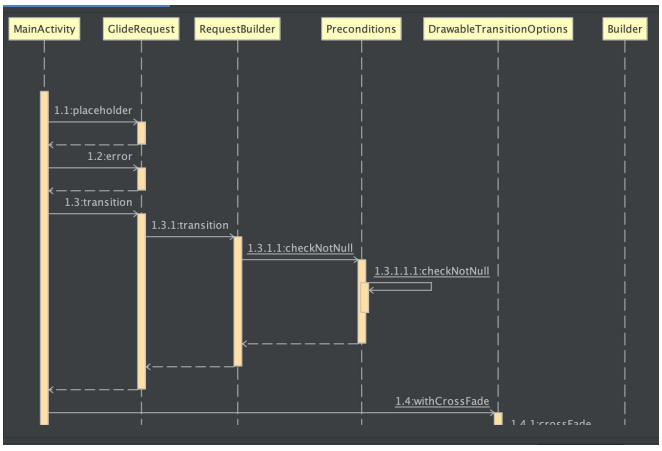

Figure 1

Module

Example: Main/load/engine/module/AppGlideModule

1 Factory Method Pattern. JavaTPoint. Accessed on March 3, 2020. https://www.javatpoint.com/factory-method-design-pattern

2 Builder Design Pattern. HowToDoInJava. Accessed on March 3, 2020. https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/

3 Module pattern. Wikipedia.org. Accessed on March 3, 2020 https://en.wikipedia.org/wiki/Module_pattern

- The module pattern is a structural pattern that groups related classes, singletons, and methods to be grouped in a single entity[3]. In Glide, AppGlideModule defines a set of dependencies and options to use when initializing Glide within an application. Additionally, there can be at most one AppGlideModule in an application, which is the essence of what the module design pattern supports. As shown in the UML diagram in Figure 2, AppGlideModule extends LibraryGlideModule and implements AppliesOptions. One or more LibraryGlideModules contained within any library or application. AppGlideModule will deal with any library conflicts and ensure all library dependencies are set for initialization of a single AppGlideModule. Overall, the module pattern provides structure to define dependencies and options for AppGlideModule.

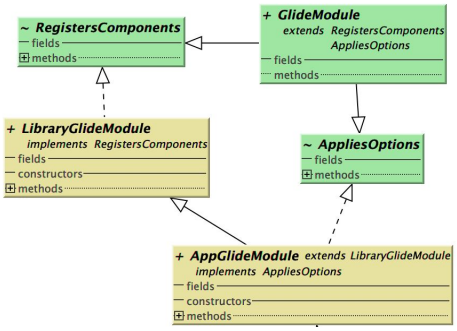

Figure 2

Listener/Observer

- src/Main/request/RequestListener
- Why is the pattern in use? Observer Pattern defines a one-to-many dependency relation so that once one subject changes its status, all the observers that are interested in this subject will be notified and changed automatically[4]. In Glide, RequestListener is responsible for src/Main/request/SingleRequest object, monitoring the status of a request during image loading. RequestListener defines a few callback methods, which will be called when image loading failed or image resource is ready. This usage of Observer Pattern provides flexible communication among request, logging system (the failure and ready information will be written in logger) and GlideException (indicating when a load fails). This communication/subscription can be added/removed easily under the benefits of Observer Pattern. Figure 3 shows the subscription relation between RequestListener and SingleRequest.

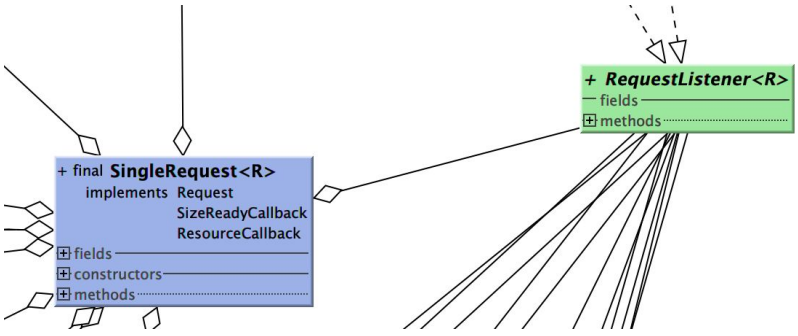

Figure 3

4 Observer Design Pattern. HowToDoInJava. Accessed on March 3, 2020. https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/

Strategy

Example: src/Main/load/resource/bitmap/DownsampleStrategy

- Strategy design pattern specifies the algorithm to implement a task in run time, which in the case of Glide would be the task to downsample images. Downsample is used in Glide during decoding and its behavior to scale images depends on the ResourceDecoder (which is a class to decode resource) and version of Android, which make up the client context. The strategy design pattern benefits the algorithm switching process when dealing with image decoding and makes the strategies clearer since strategies are all encapsulated in one class. Figure 4 shows the usages of DownsampleStrategy, showing that the DownsampleStrategy is set as one of the options for a base request and is used in the decoding process.

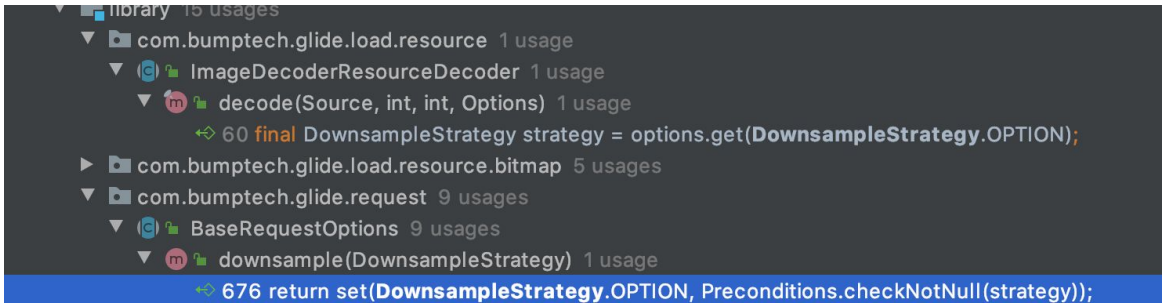

Figure 4

First Issue

The Pull Request sent to Glide can be checked here:
https://github.com/bumptech/glide/pull/4130

We have discovered a couple possible typo issues within the annotation of Glide's Encoder interface:

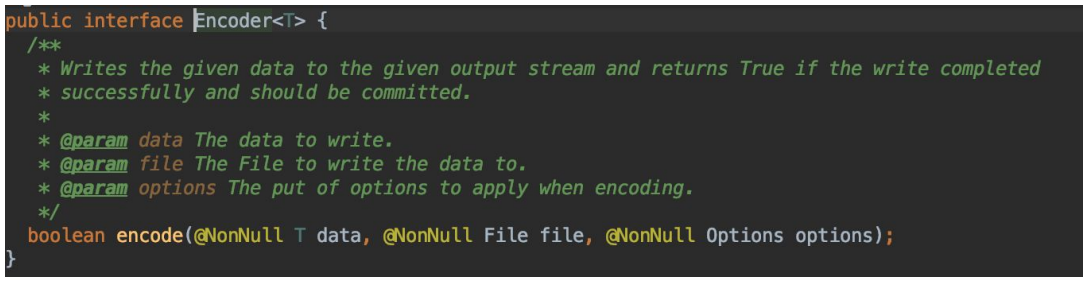

Figure 5

Issue 1: "@param file The File to write the data to." (Figure 5)

- **F** is unnecessarily capitalized

4 Observer Design Pattern. HowToDoInJava. Accessed on March 3, 2020.
https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/

- We suggest editing annotation to: “@param file The file to write the data to.”

Issue 2: “@param options The put of options to apply when encoding.” (Figure 5)

- Options.java is described as a set of Option objects placed within an ArrayMap.
- We suggest that “put” could be changed to “set” or “map” to be more clear as to what the annotation “options” represents.

4 Observer Design Pattern. HowToDoInJava. Accessed on March 3, 2020. https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/