Store App: Pattern Documentation

Ricardo Antonio Gutiérrez Esparza

To develop this application, a total of four design patterns were implemented. A brief description of them is given in this document, which serves as documentation.

Creational Patterns

Singleton

We need to keep a log of some of the events that happen while someone is operating the application. These events happen in different parts of the codebase; however, we need to make sure that the settings chosen by the user are kept during runtime. We use the singleton pattern to make sure that the instantiation happens only once and that it can be accessed from every class, with the possibility of modifying its structure with the help of another pattern.

For this specific use case, singleton also helps us by lazy initializing the logger: it won't start registering events until the user sets it up, which makes for a faster start up, especially because it does not have to open any text files until it is required to.

The UML diagram can be seen with the next pattern because, as I mentioned before, the logger uses a structural pattern for a different purpose.

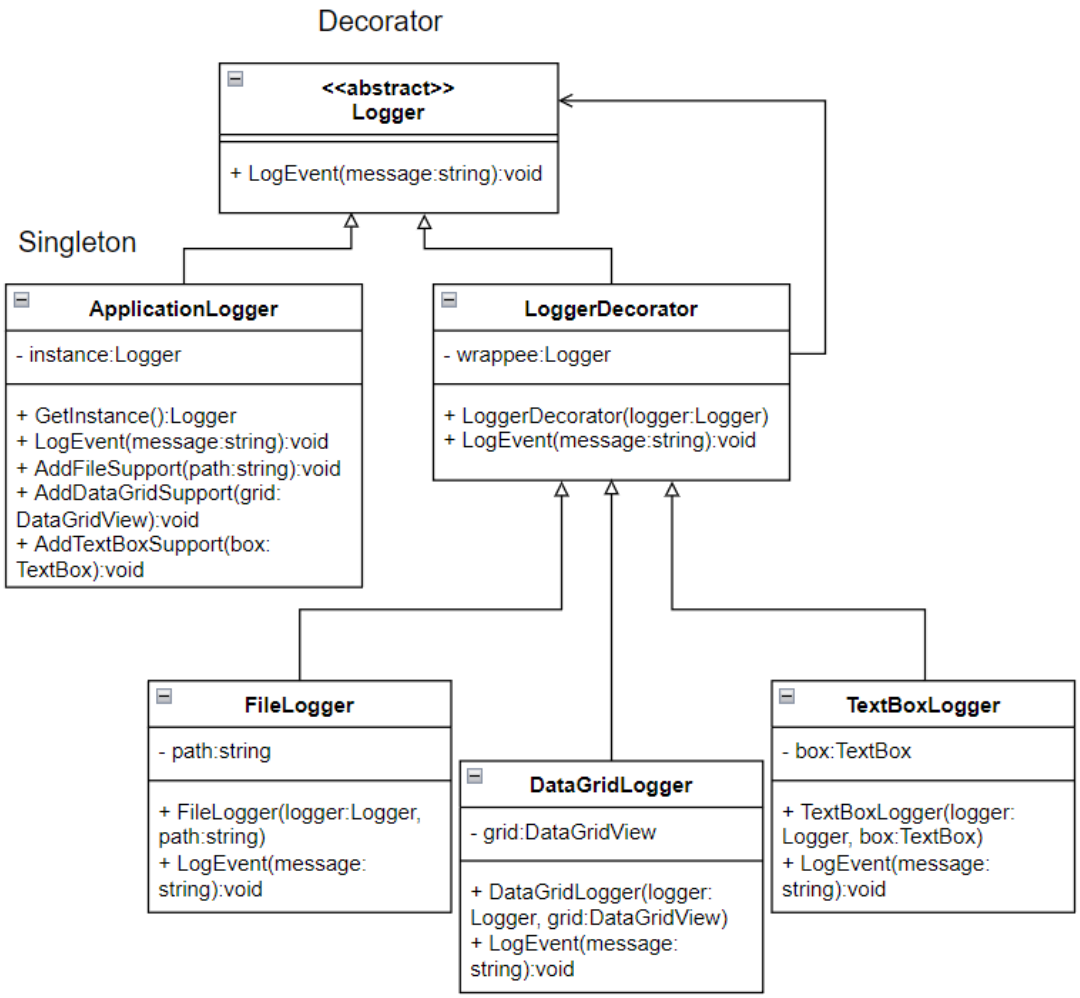
Structural Patterns

Decorator

A requirement for the logger is that it may have different outputs, some of them selected more than once. To handle this without having to keep track of multiple objects (one for each output), the decorator pattern was implemented. Every time that the user specifies a new type of output, we decorate our logger with a class that handles the logs for that specific type of output. Remember that our logger is a singleton, so we are decorating the same instance all throughout the runtime.

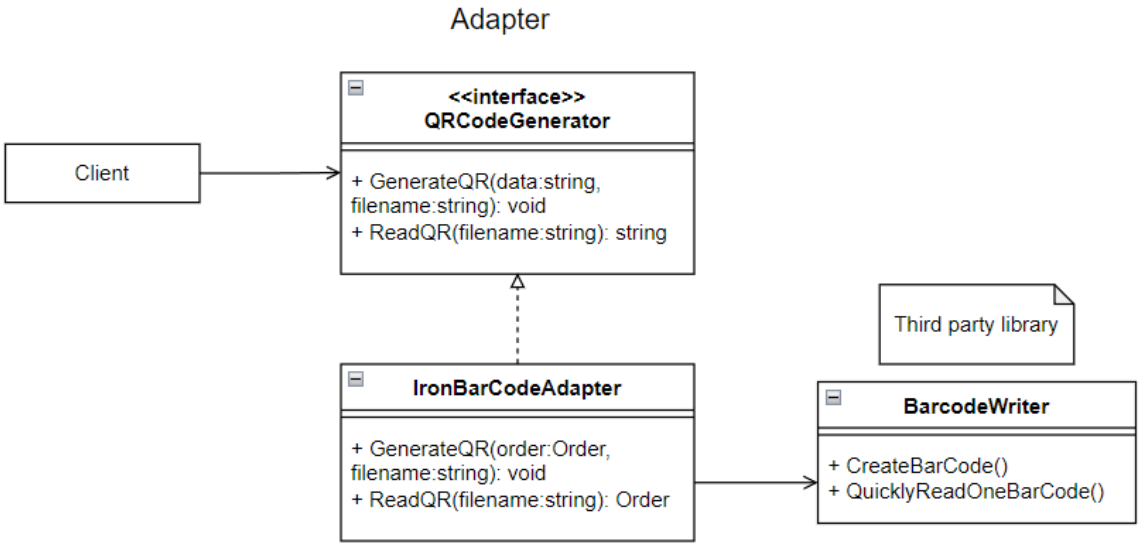
While it is simple and convenient, the downside of this approach is that we cannot remove an output during runtime (or is not that simple). This is not a big deal since it is not a requirement but is something to consider if we want to implement this feature in the future.

Here is the UML diagram of all the classes involved (including the singleton mentioned before).



Adapter

We are required to use a third-party API for QR code scanning and encoding. Some of these libraries provide a lot of functionalities, but we only need the mentioned use cases. We do not want to tie our code to a specific library, because we may want to change it at any time in the future to a better or faster one. For this reason, an adapter was implemented. We specify the two procedures that we need, and we also leave an interface for future API changes that will not require us to update the whole codebase.



Behavioral

Command

Although it was not required, I thought it would be nice to have an undo option when taking orders. As we know, the command pattern is excellent for this use case, so one was implemented for the whole process of creating an order (which includes adding products, creating the new image, and the undo option).

Implementing this pattern also allows us to schedule the execution of some commands. We could, for example, create a command to notify admins about new orders, and schedule them to be sent at an appropriate time.