13. User interface and business logic are developed separately

You have two separate teams working on the application. One team consists of front-end specialists, who are capable of making a really beautiful user interface. The other team isn't as good at building user interfaces, but it's really good at writing business logic.

Also, what you intend to do is make the user interface compatible with several different types of back-ends. Perhaps, the user interface is built by using a technology that can run on any operating system, such as Electron, while there are different versions of back-end components available for different operating systems.

Suitable design patterns

Bridge

Bridge is the design pattern that was developed specifically to solve this problem. It is a way of developing two parts of an application independently of each other.

When Bridge is used, the UI part of the application is known as the interface, while the back-end business logic part of the application is known as the implementation. This is not to be confused with interface and implementation as object-oriented programming concepts. In this case, both the user interface and the back end would have various interfaces and concrete classes that

implement them. So, what is known as the interface doesn't only consist of interfaces, and what is known as implementation doesn't exclusively consist of concrete classes.

Usually, Bridge is designed up-front and developers agree how the interface and implementation are to communicate with one another. After that, both of these components can be developed independently. One team will focus on business logic, while another team will focus on usability.

With this design, implementations can be swapped. So, as long as the access points of the implementation are what the interface expects them to be, any implementation can be used.

Using different implementations for different operating systems or data storage technologies is one of the examples. However, you can also develop a very simple implementation with faked data for the sole purpose of testing the user interface.

Why would you use Bridge

- You can develop the user interface and business logic independently.
- You can easily plug the user interface into a different back-end with the same access point signatures.
- Those who are working on the user interface don't have to know the implementation details of the business logic.
- Open-closed principle is enforced.
- The single responsibility principle is well implemented.

Facade

Facade, which we had a look at in **chapter 12**, can be used in certain circumstances, although it's often less suitable than Bridge.

For example, imagine that you have to access the back end of the app via WSDL, which would have some auto-generated code

associated with it. This is where a Facade class would be helpful, as it will abstract away all complex implementation details of this communication mechanism.

This is applicable to scenarios where the business logic layer is hosted by a third party. Likewise, if, for whatever reason, the business logic and the UI applications cannot be designed together up-front, it also might be a suitable scenario to use Facade. This is especially true when the back-end business logic application has a complex access API.

But if you don't have to deal with the complex contracts between front-end and back-end components, then facade is not the most useful design pattern to solve this specific problem.

Why would you use Facade instead of Bridge

- Easier to implement when the service with the business logic is hosted by a third party.
- Easier to implement when the interface and the implementation cannot be designed upfront.

Proxy

Proxy design pattern that we had a look at in **chapter 12** is useful if your application is relatively simple.

Essentially, you may have some back-end classes and their simplified representations that the UI components will interact with. In this case, you may have interchangeable implementations of back-end classes that the same proxies can deal with.

As Proxy delivers results without necessarily relying on the object it is providing an abstraction for, it's especially useful in situations where the back-end service with the business logic is expected to be modified and redeployed frequently. This way, the UI would still be fully operational even while the back-end is being redeployed.

Likewise, as the main purpose of Proxy is to deliver results to the client without having to trigger the actual business logic each time, it's a very useful design pattern to implement in a situation where triggering the actual business logic is computationally expensive.

Why would you use Proxy instead of Bridge

- No outage of user-accessible functionality during back-end re-deployment.
- Much better performance when the business logic is computationally expensive.