Chapter 53: Design Patterns - Structural

Design patterns are general solutions to problems that frequently occur in software development. The following are templates of standardized best practices in structuring and designing code, as well as examples of common contexts in which these design patterns would be appropriate.

Structural design patterns focus on the composition of classes and objects to create interfaces and achieve greater functionality.

Section 53.1: Adapter

Adapters are used to convert the interface of a given class, known as an **Adaptee**, into another interface, called the **Target**. Operations on the Target are called by a **Client**, and those operations are *adapted* by the Adapter and passed on to the Adaptee.

In Swift, Adapters can often be formed through the use of protocols. In the following example, a Client able to communicate with the Target is provided with the ability to perform functions of the Adaptee class through the use of an adapter.

```
// The functionality to which a Client has no direct access
class Adaptee {
    func foo() {
        // ...
    }
}

// Target's functionality, to which a Client does have direct access
protocol TargetFunctionality {
    func fooBar() {}
}

// Adapter used to pass the request on the Target to a request on the Adaptee
extension Adaptee: TargetFunctionality {
    func fooBar() {
        foo()
    }
}
```

Example flow of a one-way adapter: `Client -> Target -> Adapter -> Adaptee`

Adapters can also be bi-directional, and these are known as **two-way adapters**. A two-way adapter may be useful when two different clients need to view an object differently.

Section 53.2: Facade

A **Facade** provides a unified, high-level interface to subsystem interfaces. This allows for simpler, safer access to the more general facilities of a subsystem.

The following is an example of a Facade used to set and retrieve objects in UserDefaults.

```
enum Defaults {

    static func set(_ object: Any, forKey defaultName: String) {
        let defaults: UserDefaults = UserDefaults.standard
        defaults.set(object, forKey:defaultName)
        defaults.synchronize()
```

```
    }

    static func object(forKey key: String) -> AnyObject! {
        let defaults: UserDefaults = UserDefaults.standard
        return defaults.object(forKey: key) as AnyObject!
    }

}
```

Usage might look like the following.

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

The complexities of accessing the shared instance and synchronizing UserDefaults are hidden from the client, and this interface can be accessed from anywhere in the program.