APP ARCHITECTURE

TEMPLATE METHODS – DIFFICULTY: HARD

INTRODUCTION

The template method pattern lets us change one or more steps in a process by encouraging subclasses and conforming types to selectively replace default implementations. Subclassing is usually designed so to add things to an existing implementation, but **the template method focuses on *replacing* implementations.**

More specifically, you replace methods that are called as part of an existing algorithm or process rather than directly by your own code, which has led to the template method sometimes being called the Hollywood principle: **"don't call us, we'll call you."**

THE PROBLEM

It's common to want to inject custom functionality into one part of a long algorithm, without wanting to replace everything. For example, you might want to subclass **UIView** then override the **drawRect()** method so you can handle custom drawing – you don't want to change anything else, but you do want the rendering system to use your custom method rather than its default.

Remember the Hollywood principle: "don't call us, we'll call you." The methods you replace are there to replace part of a large piece of work, rather than being called by you – you shouldn't ever call **drawRect()**, for example.

THE SOLUTION

Developers with a long history of object-oriented programming naturally reach for subclassing, and that's OK – a "subclass and override" solution works well enough. For example, here's a Bicycle class that is assembled by calling a number of methods:

```
class Bicycle {
    func assemble() {
        addWheels()
        addGears()
        addDrivetrain()
    }

    func addWheels() { }
    func addGears() { }
    func addDrivetrain() { }
}
```

If we wanted to make an electric bike that changed only one part of the assembly algorithm, we could subclass and replace one method like this:

```
class ElectricBicycle: Bicycle {
    override func addDrivetrain() {
        print("Adding electric drivetrain.")
    }
}
```

However, subclassing creates a problem: inheritance is the tightest form of coupling we can have in our program, directly attaching one type to another. Can you imagine how much would break if **UIResponder** were to change one day? (Spoiler: everything.)

A better solution is to use protocols with a default implementation. You'd start with this:

```
protocol Bicycle {
    func addWheels()
    func addGears()
    func addDrivetrain()
}
```

You would then provide default implementations of those methods, like this:

```
extension Bicycle {
    func assemble() {
        addWheels()
        addGears()
        addDrivetrain()
    }

    func addWheels() { }
    func addGears() { }
    func addDrivetrain() { }
}
```

Next, create protocols for each part you want to override. For example:

```
protocol HasElectricDrivetrain: Bicycle { }

extension HasElectricDrivetrain  {
    func addDrivetrain() {
        print("Adding electric drivetrain.")
    }
}
```

Now you can make new types that replace as many parts of the algorithm as you want, just by making it conform to different protocols:

```
struct ElectricBicycle: Bicycle, HasElectricDrivetrain { }
```

THE CHALLENGE

Our app has three structs responsible for rendering graphics in the detail view controller: **AppProjectRenderer**, **GameProjectRenderer**, and **TechniqueProjectRenderer**.

Right now, these structs don't share any code. **Can you use the template method pattern to make them share as much as possible?**

You can use the subclass and override approach if you want an easier challenge, otherwise go for protocol extensions.