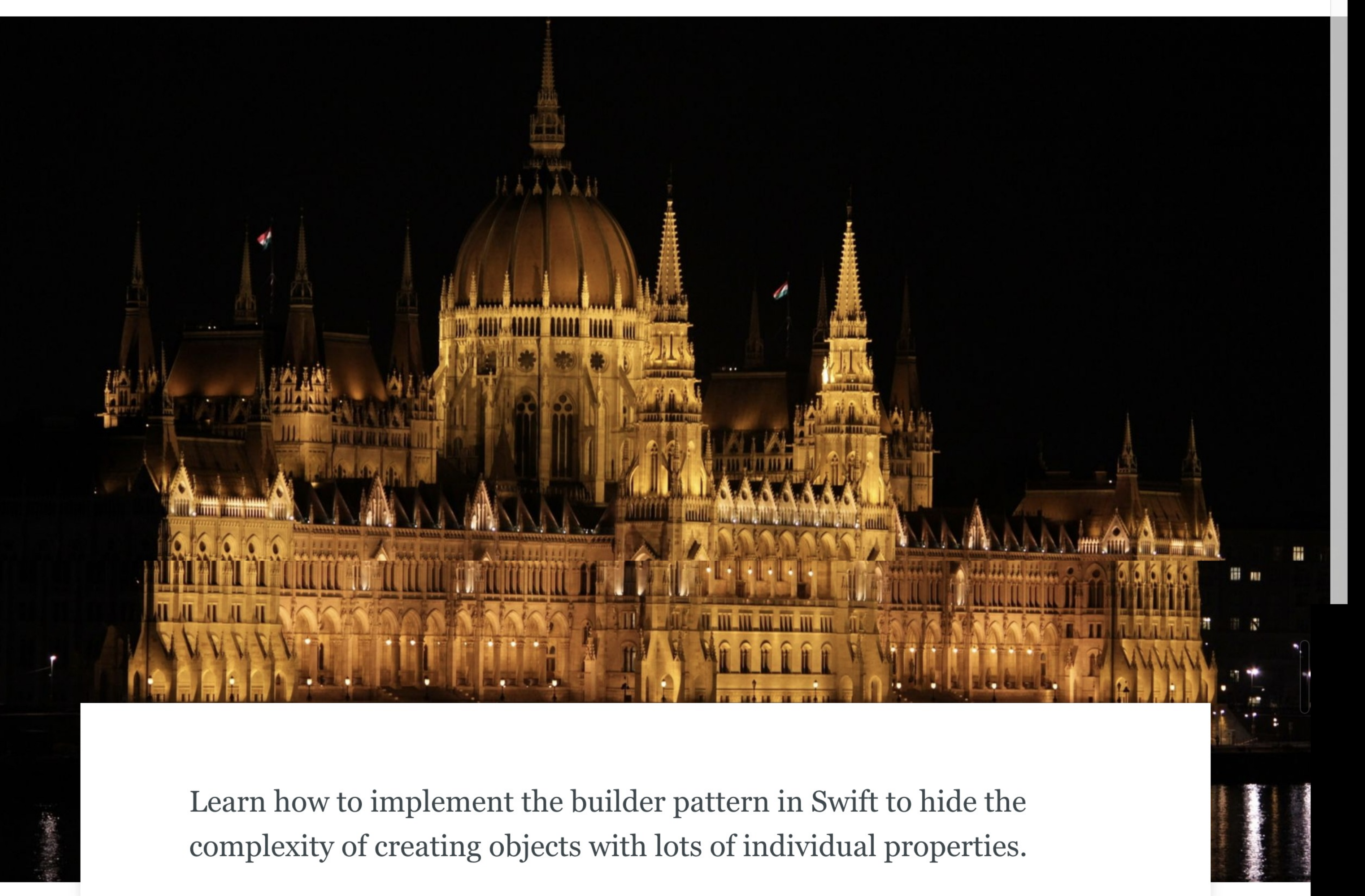
24 MAY 2018 / DESIGN PATTERNS

Swift builder design pattern



Learn how to implement the builder pattern in Swift to hide the complexity of creating objects with lots of individual properties.

How does the builder pattern work?

The builder pattern can be implemented in multiple ways, but that really doesn't matters if you understand the main goal of the pattern:

> *The intent of the Builder design pattern is to separate the construction of a complex object from its representation.*

So if you have an object with lots of properties, you want to hide the complexity of the initialization process, you could write a builder and construct the object through that. It can be as simple as a build method or an external class that controls the entire construction process. It all depends on the given environment. 🏗

That's enough theory for now, let's see the builder pattern in action using dummy, but real-world examples and the powerful Swift programming language! 💪

Simple emitter builder

I believe that SKEmitterNode is quite a nice example. If you want to create custom emitters and set properties programmatically - usually for a SpriteKit game - an emitter builder class like this could be a reasonable solution. 👾

Builder.swift 504 Bytes on GitLab

```swift
class EmitterBuilder {

    func build() -> SKEmitterNode {
        let emitter = SKEmitterNode()
        emitter.particleTexture = SKTexture(imageNamed: "MyTexture")
        emitter.particleBirthRate = 100
        emitter.particleLifetime = 60
        emitter.particlePositionRange = CGVector(dx: 100, dy: 100)
        emitter.particleSpeed = 10
        emitter.particleColor = .red
        emitter.particleColorBlendFactor = 1
        return emitter
    }
}

EmitterBuilder().build()
```

Simple theme builder

Let's move away from gaming and imagine that you are making a theme engine for your UIKit application which has many custom fonts, colors, etc. a builder could be useful to construct standalone themes. 🔨

Builder.swift 544 Bytes on GitLab

```swift
struct Theme {
    let textColor: UIColor?
    let backgroundColor: UIColor?
}

class ThemeBuilder {

    enum Style {
        case light
        case dark
    }

    func build(_ style: Style) -> Theme {
        switch style {
        case .light:
            return Theme(textColor: .black, backgroundColor: .white)
        case .dark:
            return Theme(textColor: .white, backgroundColor: .black)
        }
    }
}

let builder = ThemeBuilder()
let light = builder.build(.light)
let dark = builder.build(.dark)
```

"Chained" URL builder

With this approach you can configure your object through various methods and every single one of them will return the same builder object. This way you can chain the configuration and as a last step build the final product. ⛓

Builder.swift 1.25 KB on GitLab

```swift
class URLBuilder {

    private var components: URLComponents

    init() {
        self.components = URLComponents()
    }

    func set(scheme: String) -> URLBuilder {
        self.components.scheme = scheme
        return self
    }

    func set(host: String) -> URLBuilder {
        self.components.host = host
        return self
    }

    func set(port: Int) -> URLBuilder {
        self.components.port = port
        return self
    }

    func set(path: String) -> URLBuilder {
        var path = path
        if !path.hasPrefix("/") {
            path = "/" + path
        }
        self.components.path = path
        return self
    }

    func addQueryItem(name: String, value: String) -> URLBuilder  {
        if self.components.queryItems == nil {
            self.components.queryItems = []
        }
        self.components.queryItems?.append(URLQueryItem(name: name, value: value))
        return self
    }

    func build() -> URL? {
        return self.components.url
    }
}

let url = URLBuilder()
    .set(scheme: "https")
    .set(host: "localhost")
    .set(path: "api/v1")
    .addQueryItem(name: "sort", value: "name")
    .addQueryItem(name: "order", value: "asc")
    .build()
```

The builder pattern with a director

Let's meet the director object. As it seems like this little thing decouples the builder from the exact configuration part. So for instance you can make a game with circles, but later on if you change your mind and you'd like to use squares, that's relatively easy. You just have to create a new builder, and everything else can be the same. 🎬

Builder.swift 1.04 KB on GitLab

```swift
protocol NodeBuilder {
    var name: String { get set }
    var color: SKColor { get set }
    var size: CGFloat { get set }

    func build() -> SKShapeNode
}

protocol NodeDirector {
    var builder: NodeBuilder { get set }

    func build() -> SKShapeNode
}

class CircleNodeBuilder: NodeBuilder {
    var name: String = ""
    var color: SKColor = .clear
    var size: CGFloat = 0

    func build() -> SKShapeNode {
        let node = SKShapeNode(circleOfRadius: self.size)
        node.name = self.name
        node.fillColor = self.color
        return node
    }
}

class PlayerNodeDirector: NodeDirector {

    var builder: NodeBuilder

    init(builder: NodeBuilder) {
        self.builder = builder
    }

    func build() -> SKShapeNode {
        self.builder.name = "Hello"
        self.builder.size = 32
        self.builder.color = .red
        return self.builder.build()
    }
}

let builder = CircleNodeBuilder()
let director = PlayerNodeDirector(builder: builder)
let player = director.build()
```

Block based builders

A more swifty approach can be the use of blocks instead of builder classes to configure objects. Of course we could argue on if this is still a builder pattern or not... 😛

Builder.swift 366 Bytes on GitLab

```swift
extension UILabel {

    static func build(block: ((UILabel) -> Void)) -> UILabel {
        let label = UILabel(frame: .zero)
        block(label)
        return label
    }
}

let label = UILabel.build { label in
    label.translatesAutoresizingMaskIntoConstraints = false
    label.text = "Hello wold!"
    label.font = UIFont.systemFont(ofSize: 12)
}
```

⚠️ Please note that the builder implementation can vary on the specific use case. Sometimes a builder is combined with factories. As far as I can see almost everyone interpreted it in a different way, but I don't think that's a problem. Design patterns are well-made guidelines, but sometimes you have to cross the line.