

Taking the boilerplate out of your tests with Sourcery



XCTest





XCTest



```
import XCTest

class MyTests: XCTestCase {

    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the correct results.
    }

}
```



XCTest





Sourcey 101

What is Sourcery?

What is Sourcery?

"Sourcery is a **code generator** for the Swift language, built on top of Apple's own SourceKit. It extends the language abstractions to allow you to **generate boilerplate code** automatically."¹

¹ <https://github.com/krzysztofzablocki/Sourcery/blob/master/README.md>

What can we do with Sourcery?

What can we do with Sourcery?

```
enum Direction {  
    case up  
    case right  
    case down  
    case left  
}
```

```
Direction.allCases // [.up, .right, .down, .left]
```

What can we do with Sourcery?

Since Swift 4.2, this can be achieved through the `CaseIterable` protocol.

But before, generating `.allCases` was a great use case to illustrate how Sourcery works.

Let's implement it 💪

1st - Setting Up Sourcery

Setting Up Sourcery

```
$ brew install sourcery
```

Setting Up Sourcery

Xcode Project > Build Phases > New Run Script Phase

```
sourcery \
--sources <sources path> \
--templates <templates path> \
--output <output path>
```

2nd - Phantom Protocol

Phantom Protocol

```
protocol EnumIterable { }
```

```
extension Direction: EnumIterable { }
```

3rd - Sourcery Template

Sourcery Template

```
{% for enum in types.implementing.EnumIterable|enum %}

{% if not enum.hasAssociatedValues %}

{{ enum.accessLevel }} extension {{ enum.name }} {
    static let allCases: [{{ enum.name }}] = [
{% for case in enum.cases %}
    .{{ case.name }} {% if not forloop.last %} , {% endif %}
{% endfor %}
    ]
}

{% endif %}

{% endfor %}
```

4th - Generated Code

Generated Code

Build your target

Generated Code

```
internal extension Direction {  
    static let allCases: [Direction] = [  
        .up ,  
        .right ,  
        .down ,  
        .left  
    ]  
}
```

Generated Code

Add the generated file to your project

That's it 

To Sum Up

To Sum Up

- Sourcery parses your source code
- It then uses it to execute templates
- Those templates generate new source code
- Your project can use this generated code

**End of
Sourcery 101**

**Back to
writing tests**

```
func testEquality() {
    let personA = Person(firstName: "Charlie", lastName: "Webb", age: 10, hasDriverLicense: false, isAmerican: true)
    let personB = Person(firstName: "Charlie", lastName: "Webb", age: 11, hasDriverLicense: false, isAmerican: true)

    XCTAssertEqual(personA, personB)
}
```

✖ XCTAssertEqual failed: ("Person(firstName: "Charlie", lastName: "Webb", age: 10, hasDriverLicense: false, isAmerican: true)") is not equal to ("Person(firstName: "Charlie", lastName: "Webb", age: 11, hasDriverLicense: false, isAmerican: true)") - ✖



A human-readable diff would be nice

Incorrect age: expected 10, received 11

Diffing methods are a perfect example of boilerplate

```
internal extension Person {

    func diff(against other: Person) -> String {
        var result = [String]()

        if self.firstName != other.firstName {
            var diff = "Incorrect firstName: "
            diff += "expected \(self.firstName), "
            diff += "received \(other.firstName)"

            result.append(diff)
        }
        if self.lastName != other.lastName {
            var diff = "Incorrect lastName: "
            diff += "expected \(self.lastName), "
            diff += "received \(other.lastName)"

            result.append(diff)
        }
        if self.age != other.age {
            var diff = "Incorrect age: "
            diff += "expected \(self.age), "
            diff += "received \(other.age)"

            result.append(diff)
        }
        if self.hasDriverLicense != other.hasDriverLicense {
            var diff = "Incorrect hasDriverLicense: "
            diff += "expected \(self.hasDriverLicense), "
            diff += "received \(other.hasDriverLicense)"

            result.append(diff)
        }
        if self.isAmerican != other.isAmerican {
            var diff = "Incorrect isAmerican: "
            diff += "expected \(self.isAmerican), "
            diff += "received \(other.isAmerican)"

            result.append(diff)
        }
        return result.joined(separator: ". ")
    }
}
```

How about we use Sourcery to generate it?

1st - Phantom Protocol

```
protocol Diffable { }
```

```
extension Person: Diffable { }
```

2nd - Sourcery Template

```
{% for type in types.implementing.Diffable %}

{{ type.accessLevel }} extension {{ type.name }} {

    func diff(against other: {{ type.name }}) -> String {
        var result = [String]()

        {% for variable in type.variables %}
            if self.{{ variable.name }} != other.{{ variable.name }} {
                var diff = "Incorrect {{ variable.name }}: "
                diff += "expected \\\(self.{{ variable.name }}), "
                diff += "received \\\(other.{{ variable.name }})"

                result.append(diff)
            }
        {% endfor %}

        return result.joined(separator: ". ")
    }
}

{% endfor %}
```

3rd - Generated Code

```
internal extension Person {

    func diff(against other: Person) -> String {
        var result = [String]()

        if self.firstName != other.firstName {
            var diff = "Incorrect firstName: "
            diff += "expected \(self.firstName), "
            diff += "received \(other.firstName)"

            result.append(diff)
        }
        if self.lastName != other.lastName {
            var diff = "Incorrect lastName: "
            diff += "expected \(self.lastName), "
            diff += "received \(other.lastName)"

            result.append(diff)
        }
        if self.age != other.age {
            var diff = "Incorrect age: "
            diff += "expected \(self.age), "
            diff += "received \(other.age)"

            result.append(diff)
        }
        if self.hasDriverLicense != other.hasDriverLicense {
            var diff = "Incorrect hasDriverLicense: "
            diff += "expected \(self.hasDriverLicense), "
            diff += "received \(other.hasDriverLicense)"

            result.append(diff)
        }
        if self.isAmerican != other.isAmerican {
            var diff = "Incorrect isAmerican: "
            diff += "expected \(self.isAmerican), "
            diff += "received \(other.isAmerican)"

            result.append(diff)
        }
        return result.joined(separator: ". ")
    }
}
```

4th - Updated Tests

```
let personA = Person(firstName: "Charlie", lastName: "Webb", age: 10, hasDriverLicense: false, isAmerican: true)
let personB = Person(firstName: "Charlie", lastName: "Webb", age: 11, hasDriverLicense: false, isAmerican: true)

XCTAssertEqual(personA, personB, personA.diff(against: personB))
```

✖ XCTAssertEqual failed: ("Person(firstName: "Charlie", lastName: "Webb", age: 10, hasDriverLicense: false, isAmerican: true)") is not equal to ("Person(firstName: "Charlie", lastName: "Webb", age: 11, hasDriverLicense: false, isAmerican: true)") - Incorrect age: expected 10, received 11 ✖



How about we craft more complex tools?

Classic MVVM Architecture

```
protocol UserService {
    func fetchUserName() -> String
}

class ViewModel {

    var userNameUpdated: ((String) -> Void)?

    private let service: UserService

    init(service: UserService) {
        self.service = service
    }

    func fetchData() {
        let userName = self.service.fetchUserName()

        self.userNameUpdated?(userName)
    }
}
```

How do we test a component with dependencies?

Obvious, just inject mocked dependencies!

How do we write mocked dependencies?

How about we just don't? (And let Sourcery do it)

Generating Mocked Implementations

1st - Phantom Protocol

```
protocol MockedImplementation { }

protocol UserService: MockedImplementation {
    func fetchUserName() -> String
}
```

2nd - Sourcery Template

```
{% for protocol in types.implementing.MockedImplementation|protocol %}

{{ protocol.accessLevel }} class Mocked{{ protocol.name }}: {{ protocol.name }} {

{% for method in protocol.methods %}

var {{ method.callName }}CallCounter: Int = 0
var {{ method.callName }}ReturnValue: {{ method.returnTypeName }}?

func {{ method.name }} -> {{ method.returnTypeName }} {
    {{ method.callName }}CallCounter += 1
    return {{ method.callName }}ReturnValue!
}

{% endfor %}
}

{% endfor %}
```

3rd - Generated Code

```
internal class MockedUserService: UserService {  
  
    var fetchUserNameCallCounter: Int = 0  
    var fetchUserNameReturnValue: String?  
  
    func fetchUserName() -> String {  
        fetchUserNameCallCounter += 1  
        return fetchUserNameReturnValue!  
    }  
}
```

4th - Writing Tests

```
class ViewModelTests: XCTestCase {

    func testUserServiceCalls() {
        let mockedService = MockedUserService()
        let viewModel = ViewModel(service: mockedService)

        mockedService.fetchUserNameReturnValue = "John Appleseed"

        viewModel.fetchData()

        XCTAssertEqual(mockedService.fetchUserNameCallCounter, 1)
    }
}
```

No more boilerplate

No more boilerplate

Now we only focus on writing tests for the business logic 

Of course, there's a lot more features we could add:

- variables to store arguments
- calling completion handlers
- dealing with throwing functions
- etc.

No more boilerplate

Sourcery actually ships with a template that takes care of all those needs: [AutoMockable](#)

(But beware, it is MUCH harder to understand 🙄)

We are now able to generate tools for testing...

...but there's still room to go even further!

Testing Dependency Injection

Dependency Injection

Many apps rely on **Swinject** to provide the architectural basis of dependency injection.

Dependency Injection

```
import Swinject

class ViewModelAssembly: Assembly {

    func assemble(container: Container) {
        container.register(UserService.self) { _ in return ImplUserService() }
        container.register(ViewModel.self) { resolver in
            let service = resolver.resolve(UserService.self)!

            return ViewModel(service: service)
        }
    }
}
```

Dependency Injection

```
class GreetingsViewControllerAssembly: Assembly {

    func assemble(container: Container) {
        container.register(GreetingsViewController.self) { resolver in
            let viewModel = resolver.resolve(ViewModel.self)!
            let viewController = UIStoryboard(name: "Main", bundle: nil)
                .instantiateViewController(withIdentifier: "GreetingsViewController")
                as! GreetingsViewController

            viewController.viewModel = viewModel

            return viewController
        }
    }
}
```

Dependency Injection

```
struct ViewControllerFactory {  
  
    static var greetingsVC: GreetingsViewController {  
        let assembler = Assembler([ViewModelAssembly(), GreetingsViewControllerAssembly()])  
  
        return assembler.resolver.resolve(GreetingsViewController.self)!  
    }  
}
```

Swinject **relies** exclusively on **runtime** checks

So we need **tests**, to avoid production **crashes**

How do we test those injections?

How do we test those injections?

Let's reason about the situation:

- The dependencies follow a tree structure
- The view controllers are the roots of those trees
- By instantiating them, we trigger the whole injection process

How do we test those injections?

Conclusion: we need to write tests that attempt to instantiate all the controllers.



WRITE TESTS



**HAVE SOURCERY
WRITE TESTS**

Sourcery Template

```
import XCTest
@testable import YourApp

class DependencyInjectionTests: XCTestCase {

    func testDependencyInjection() {
        {% for variable in type["ViewControllerFactory"].staticVariables %}
            _ = ViewControllerFactory.{% variable.name %}_
        {% endfor %}
    }
}
```

Generated Code

```
import XCTest
@testable import TestSourcery

class DependencyInjectionTests: XCTestCase {

    func testDependencyInjection() {
        _ = ViewControllerFactory.greetingsVC
    }
}
```

That's it!

As **new controllers** are added to the factory, the corresponding tests will be **written automatically** 

No more room for mistakes, pretty cool!

Recap

Recap

- Sourcery is really easy to set up, don't feel scared to try it!
- (It's also easy to take it out of a project, should you need)
- It's a great tool to avoid writing boilerplate code by hand
- Tests are a perfect place to use Sourcery, because they involve lots of boilerplate
- Every time a "one-size-fits-all" approach makes sense, there's a good chance Sourcery can help

Don't rely on Humans  **to do the job of a Robot** 

Pro Tips

Be careful!

Be careful!

Sourcery lets us manipulate **familiar concepts** (types, protocols, etc.) in an **unfamiliar manner**.

We get to look at our code the same way that Xcode does.

This is not an approach we are used to, and it is very easy to fail to consider some edge cases



First, ask Google

First, ask Google

Templates tend to be much more complicated than initially thought (remember AutoMockable).

A little Google search might just save you a lot of time by pointing you in the right direction.

Should I commit generated files?

Should I commit generated files?

We might initially think that generated code should not be versionned.

However, you should seriously consider versionning it.

This way, **changes** to generated files **will appear during code review**, providing the opportunity to check that everything still works fine.

If you don't version generated files, it becomes really easy to forget that they even exist...



Taking the boilerplate out of your tests with Sourcery

Vincent Pradeilles [@v_pradeilles](https://twitter.com/v_pradeilles) – Worldline 

