Stubble: A Macro-Based Stubbing Library for Swift

Samuel Shi

Abstract

Unit testing is a crucial tool used by software developers to verify the correctness of their code. Developers often use stubs to replace the implementations of complex dependencies to make them suitable for the test environment, but the traditional approach of doing this in Swift requires significant amounts of boilerplate and source code changes. This paper presents Stubble, a macro-based stubbing library for Swift that generates stubbing infrastructure automatically, preserving the safety and flexibility of the traditional approach while avoiding its limitations.

Introduction

Unit testing is an important and widespread practice used by software developers to verify the correctness of the code they write. When testing code with dependencies that are unsuitable for the test environment, developers often replace the dependencies with stubs: altered versions of a unit's dependencies that have specialized implementations for each test case. (Kumari, 2024)

In some cases, these dependencies may be unsuitable because they cannot function properly in the test environment. For example, consider a class that manages Bluetooth communication between an app and a smart lightbulb in the end user's home. Without access to Bluetooth hardware and a physical smart bulb nearby, testing code that depends on this class becomes impossible. A developer could create a stub for this class that subverts its actual implementation and returns simulated responses from the smart bulb, allowing the depending code to be reliably tested.

Additionally, even if a dependency can properly function in the test environment, the developer may want to test specific cases in the depending code that are difficult to reliably reproduce with the dependency's real implementation. For example, imagine a method that receives an HTTP response containing the list of students currently enrolled in a course. If the developer wanted to test how this method behaves when no students are enrolled, they could stub the code performing the HTTP request to return an empty array, rather than performing the actual HTTP request.

When testing Swift code, developers typically use protocol-based polymorphism to provide different implementations of a unit's dependencies in the test and production environments. This approach is flexible and easy to understand, but it has several limitations and downsides. Namely, it requires a significant amount of boilerplate code and changes to the developer's production source code. Here, the term "production source code" refers to the original, unmodified source code that the developer maintains in their codebase and runs as a part of their production app.

This paper introduces Stubble, a stubbing library for Swift that utilizes the language's powerful macro system to reduce the boilerplate and source code changes needed to implement stubs for Swift code.

Stubbing in Swift

The typical approach used for stubbing Swift code is to implement it manually using protocols (Polaczyk, 2018). In Swift, a protocol defines a set of properties, methods, and other requirements needed to perform some task. A concrete type can then implement these requirements to *conform* to that protocol. By referring to a protocol rather than any specific concrete type, a unit of code can interact with any type that implements the desired functionality. (*Protocols*, n.d.)

When using protocols to stub an existing type, start by introducing a protocol that exactly replicates its exposed API and add a conformance to that protocol to the production implementation. Consider the following example where we'd like to stub a service class that performs an HTTP request to return the list of students currently enrolled in a class:

```
// Production code

protocol RosterServiceProtocol {
    func fetchStudents() async throws -> [Student]
}

class RosterService: RosterServiceProtocol {
    func fetchStudents() async throws -> [Student] {
        // make an HTTP request
        return response
    }
}
```

Then, in the rest of the code that depends on this type, replace all references to the original type with references to the protocol and add an initializer that accepts an instance of the dependency from the initializer's call site. In our example, we have a view model that calls the roster service's `fetchStudents()` method and groups the students by graduation year:

```
// Production code

class RosterViewModel {
    let service: RosterServiceProtocol

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

    func studentsByGraduationYear() async throws -> [Int : [Student]] {
        let students = try await service.fetchStudents()
        return Dictionary(grouping: students, by: \.graduationYear)
    }
}
```

Now, when testing any of this code, create a new implementation that conforms to the same protocol and whose behavior can be controlled. Then, create an instance of this stub type, configure it to perform the desired behavior, and pass it into the depending code's initializer. In our example, we'll create a `RosterServiceStub` class that returns the array it was initialized with, rather than performing a network request. Then, we'll create an instance that returns an empty array to test the view model's behavior when no students are currently enrolled:

```
// Test code

class RosterServiceStub: RosterServiceProtocol {
    let students: [Student]
    init(students: [Student]) { self.students = students }

    func fetchStudents() async throws -> [Student] { return students }
}

@Test
func testEmptyRoster() async throws {
    let stub = RosterServiceStub(students: [])
    let vm = RosterViewModel(service: stub)

    let result = try await vm.studentsByGraduationYear()
    #expect(result == [:])
}
```

This approach works well and has two main features:

1. **Compile-time safety:** Because of their shared protocol conformance, the Swift compiler will catch any discrepancies between the stubbed and production implementations, preventing runtime errors resulting from incorrect method signatures or property types.
2. **Flexibility:** Because developers write normal Swift code to implement their stubs, they're able to implement their stubs to perform any behavior they'd like: returning a hard-coded value, manually throwing an error, or even complex behaviors like reading sample data from a JSON file or tracking state between method calls.

While this approach is safe and flexible, it has significant limitations that make it difficult to work with:

1. **Tedious refactoring and maintenance:** Any changes made to a stubbed type's exposed API need to be mirrored in the protocol and all stub implementations, meaning that even a small change will require many source changes.
2. **Significant production code changes:** Despite the production application only using a single implementation of its dependencies, extra levels of indirection and dependency injection need to be added to effectively test its behavior.
3. **No partial stubbing:** Because a type must implement all requirements to conform to a protocol, stub implementations must implement all of a type's exposed API, even if the code under test only interacts with a small subset.

These limitations stem from one fundamental issue: developers must manually write and maintain significant amounts of boilerplate code. Swift 5.9 introduced a powerful macro system designed to "help developers reduce repetitive boilerplate" (Sandberg & Borla, 2023). So, Stubble leverages this macro system to examine the developer's source code and automatically generate the stubbing infrastructure, removing the burden and limitations caused by boilerplate code.

Swift Macros

To understand how Stubble uses macros to improve the developer experience when stubbing, let's first examine how Swift macros are expanded and what they are able to do.

How Macros are Expanded

Like macro systems in other programming languages, Swift macros expand to generate new source code that will be compiled and executed as part of the program they are used in. Different macro systems implement different mechanisms for this expansion, each having its set of own trade-offs between simplicity, safety, and power.

C's macro system uses the C preprocessor to perform simple, textual replacements of macro invocations with their bodies before compilation. Because macros only work with source code as text, there is no requirement that a macro expands to valid C code. Macro expansions resulting in poorly-formed C code can cause compilation errors that are hard to resolve, as the compiler only sees the expanded text and has no information about the original macro invocation. Additionally, because macro definitions are processed and substituted as raw text, they are unable to reason about their inputs to change the form of their expansion, which is a significant functional limitation. (Free Software Foundation, Inc, 1992)

Rust's declarative macro system takes a different, more sophisticated approach by allowing developers to define a macro as a set of pattern-matching statements. When the compiler encounters a macro invocation, it attempts to match the structure of the macro inputs' source code with the macro's pattern-matching statements and will emit the body associated with the first pattern that successfully matches. Additionally, the compiler verifies that macro expansions are valid Rust code and can give better diagnostics in the case of an invalid expansion than C. Because these macros can declaratively reason about their inputs to produce different expansions, they are much more powerful than C's text-replacement macros. But, they are still limited by the expressivity of the pattern-matching syntax. (*Macros*, n.d.)

Unlike the declarative macro systems above, Swift macros are normal Swift functions that operate on type-checked nodes of the program's abstract syntax tree (AST). This design offers three key advantages:

1. Because macros are normal Swift functions, they can use all of Swift's powerful, procedural tools to determine the form of their expansions: complex control flow, reusable helper functions, data structures to track state during expansion, etc.
2. Because macro inputs and expansions are represented as type-checked AST nodes, they are guaranteed to be valid Swift code, increasing safety and preventing the confusing compiler diagnostics seen when using C macros.
3. Having access to the macro inputs as AST nodes gives a macro an incredibly detailed view of its inputs, which it can use when generating its expansion.

Because of these choices, Swift's macro system is safer and more powerful than the two systems described previously. But, because the compiler needs to execute arbitrary Swift code to produce macro expansions, it is the most complicated of the three. Swift macros are distributed as libraries, compiled separately from the program using them, and executed as compiler plugins to perform each macro expansion. But, because of good tooling support in Xcode, the most popular IDE used to write Swift code, this complexity does not meaningfully affect the developer's experience or workflow when they begin using macros. (Gregor, 2023a)

Kinds of Swift Macros

Swift allows developers to implement two kinds of macros: freestanding and attached.

Invocations of freestanding macros appear on their own in the original source code, rather than being attached to a declaration, and generate new expressions or declarations as a function of their inputs. These are useful for utility functions that benefit from being able to reason about the source code of its inputs at compile time. For example, a `#URL(string:)` macro could parse a string literal to either emit a non-optional `URL` object or emit a compile-time error, depending on whether or not the string contains a URL:

```
let goodURL = #URL(string: "https://unc.edu") // returns a URL instance
let badURL = #URL(string: "") // Compile-Time Error: "" is not a valid URL
```

Without macros, the `URL(string:)` initializer returns an optional value–either `nil` or a `URL`–even if the validity of the string as a URL is statically checkable. (Gregor, 2023a; Lee, 2023)

Swift also allows developers to write attached macros: macros that extend or modify existing declarations. An attached macro can inspect details from a declaration's AST to determine the modifications to perform. For example, an `@MemberwiseInit` attached macro could examine the members in a struct declaration, extract the names and types of its instance properties, and generate a member-wise initializer:

```
@MemberwiseInit
struct Point {
    var x: Double
    var y: Double

    func distance(to other: Point) -> Double { ... }
}

// Expands to:

struct Point {
    var x: Double
    var y: Double

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }

    func distance(to other: Point) -> Double { ... }
}
```

The specific modifications an attached macro can make to its corresponding declaration depend on which of several *attached macro roles* it implements. Each role has a very specific purpose, such as providing a new declaration next to the one it's attached to, adding new methods and properties to a type, or rewriting the body of a method. (Gregor, 2023a)

As we discuss Stubble's implementation later in the next section, we will see many different attached macro roles and what they do.

Stubble

Stubble leverages Swift's powerful attached macros described above to examine the type the developer would like to stub and automatically generate the infrastructure needed to do so.

Here, the term infrastructure abstractly refers to the supporting code needed to provide a stub implementation for a unit of code. In the protocol-based approach, the stubbing infrastructure is the protocol definition, the conformance on the original type, and the stub implementations. The stubbing infrastructure generated by Stubble takes a different form and will be explained in detail in the following sections.

Design Goals

Specifically, Stubble is designed to preserve the benefits provided by the manual protocol-based stubbing technique and eliminate its limitations, while creating an easy-to-use API:

- **Compile-time safety:** Stub implementations should be safely interchangeable with their production counterparts, and the compiler should catch any discrepancies before runtime.
- **Flexibility:** Developers should be able to write arbitrary Swift code in their stub implementations to create any desired behavior.
- **Easy maintenance:** Changes to stubbed types should not require any manual updates to the stubbing infrastructure.
- **Minimal production code changes:** Adding stub-based tests to your project should not significantly influence how the production code is written.
- **Partial stubbing:** Developers should be able to stub a subset of a type's members, preserving the production behavior of all others.
- **Ease of use:** The syntax used to create stub implementations with Stubble should be easy to understand, and the mechanics behind how the stub implementation should be easy to understand.

What it Does

Rather than generating a protocol and separate conforming stub implementation, Stubble modifies the original type declaration to add stubbing capabilities. Developers just need to apply one of Stubble's attached macros to the declaration they would like to stub, and Stubble automatically generates all of the stubbing infrastructure inside the type itself, meaning that all references to the stubbable type in the rest of the codebase can remain unmodified.

There are two narrowly focused macros that perform the expansions needed to stub a specific kind of member: `@StubbableFunction` and `@StubbableProperty`. A third type-level macro, `@Stubbable`, inspects an entire type and applies the member-level macros to all supported members. Let's take a look at each of them more closely.

`@StubbableFunction`

Developers can add stubbing capabilities to an instance method by applying the `@StubbableFunction` attached macro to it. In the `RosterService` example seen earlier, a developer would apply `@StubbableFunction` to the `fetchStudents()` method. It would have the following expansion:

```
class RosterService {
    @StubbableFunction
    func fetchStudents() async throws -> [Student] {
        // make an HTTP request
        return response
    }
}

// Expands to:

class RosterService {
   var _fetchStudents: (() async throws -> [Student])? = nil

   func fetchStudents() async throws -> [Student] {
      if let _fetchStudents {
          return try await _fetchStudents()
      } else {
          // make an HTTP request
          return response
      }
   }
}
```

`@StubbableFunction` reads attributes from the signature of the function it is applied to in order to add an optional closure property with the same signature as the original function. Then, it modifies the body of the original function to check this property–if set, it completely delegates to the closure. Otherwise, it runs the original method implementation.

Because of the behavior of the new method body, developers can then stub the method by assigning a closure to the new closure property. In this example, a developer could stub the method to always return an empty array, rather than performing the HTTP request:

```
let service = RosterService()
service._fetchStudents = { return [] }

let students = try await service.fetchStudents() // always returns []
```

`@StubbableProperty`

Similarly, developers can add stubbing capabilities to an instance property with the `@StubbableProperty` attached macro. `@StubbableProperty` performs the expansion needed to stub a property's getter and setter by converting it from a stored property to a computed property. In Swift, computed properties don't directly store values. Instead, they provide getters and setters to indirectly retrieve and set other properties. Consider the following expansion of `@StubbableProperty`:

```
class FeatureFlag {
    @StubbableProperty
    var isEnabled: Bool

    init(isEnabled: Bool) {
        self.isEnabled = isEnabled
    }
}

// Expands to:

class FeatureFlag {
    private var _isEnabled: Bool               // (1)
    var _getIsEnabled: (() -> Bool)? = nil     // (2)
    var _setIsEnabled: ((Bool) -> Void)? = nil // (3)

    var isEnabled: Bool {
        get {                                  // (4)
            if let _getIsEnabled {
                return _getIsEnabled()
            } else {
                return _isEnabled
            }
        }
        set {                                  // (5)
            if let _setIsEnabled {
                _setIsEnabled(newValue)
            } else {
                _isEnabled = newValue
            }
        }
        @storageRestrictions(initializes: _isEnabled)
        init(initialValue) {                   // (6)
            _isEnabled = initialValue
        }
    }

    init(isEnabled: Bool) {
        self.isEnabled = isEnabled
    }
}
```

The expansion of this macro has six components, labeled as comments in the code snippet above and explained below:

(1) First, it generates the private instance variable `_isEnabled`. This becomes the backing storage for the new computed property.

(2, 3) It then generates two optional closure properties used to stub the computed property's getter and setter using the same technique as `@StubbableFunction`.

(4, 5) It then adds get and set accessors to the property, converting it to a computed property. Like the new method bodies generated by `@StubbableFunction`, these call the custom getter or setter if they've been set. Otherwise, these read from or write to the backing storage `_isEnabled` when called.

(6) Lastly, it adds an `init` accessor to the computed property. This allows the backing storage to be initialized

through the computed property wrapper, which is needed if no initial value was provided to the base property.

After applying `@StubbableProperty` to a property, a developer can then stub its getter or setter by assigning closures to the appropriate property:

```
let stub = FeatureFlag()
stub._getIsEnabled = { return true }
stub._setIsEnabled = { print("Flag is enabled: \($0)") }
```

`@Stubbable`

Lastly, Stubble offers a type-level convenience macro to add stubbing capabilities for an entire type with a single macro application rather than needing to apply macros to all of the type's members: `@Stubbable`. This macro visits each member in the type's declaration and inspects its syntax to decide if it should receive `@StubbableFunction`, `@StubbableProperty`, or no new annotations at all. In the following example, it sees an instance property and an instance method, and applies `@StubbableProperty` and `@StubbableFunction` accordingly:

```
@Stubbable
class RosterService {
    var courses: [Course] = []

    func fetchAllStudents() async throws -> [Student] {
        // fetch students in all courses from server
        return allStudents
    }
}

// Expands to:

class RosterService {
    @StubbableProperty
    var courses: [Course] = []

    @StubbableFunction
    func fetchAllStudents() async throws -> [Student] {
        // fetch students in all courses from server
        return allStudents
    }
}
```

Then, `@StubbableFunction` and `@StubbableProperty` perform their expansions as described above, and developers can stub the functions and properties as usual.

Implementation

To understand how Stubble performs these transformations, let's examine each of Stubble's macros, which attached macro roles they implement, and the procedures they execute.

`@StubbableFunction`

Swift allows a single macro definition to implement multiple roles, meaning that it will be expanded once for each of its roles. `@StubbableFunction` performs two main functions—generating the optional closure property and rewriting the function's body—and implements two attached macro roles—peer and body—accordingly.

Peer macros can inspect the declaration they're applied to and produce new declarations in its scope (Gregor et al., 2023). `@StubbableFunction`'s peer macro implementation reads the important attributes from the function signature it's applied to: the parameter list, the return type, whether or not it is asynchronous, and whether or not it can throw an error. Then, it uses these attributes to generate an optional closure property with the same function type as the one it was applied to. Consider the following example:

```
@StubbableFunction
func add(_ a: Int, to b: Int) -> Int {
    return a + b
}
// Peer macro expands to:

var _add: ((Int, Int) -> Int)? = nil

func add(_ a: Int, to b: Int) -> Int {
    return a + b
}
```

In this example, `@StubbableFunction`'s peer macro implementation saw that the function accepts two integer parameters, returns an integer, and is not asynchronous or throwing. So, it generated a closure with the same description.

`@StubbableFunction`'s peer macro implementation also includes logic for handling variadic parameters. When a Swift method has a variadic parameter, it gets packaged into an array for use inside the method. So, rather than generating a closure with a variadic parameter type, `@StubbableFunction` generates a closure with an array type for the associated parameter:

```
@StubbableFunction
func add(_ numbers: Int...) -> Int {
    var total = 0
    for number in numbers {
        total += number
    }
    return total
}

// Peer macro expands to:

var _add: (([Int]) -> Int)? = nil

func add(_ numbers: Int...) -> Int {
    var total = 0
    for number in numbers {
        total += number
    }
    return total
}
```

Body macros can inspect the function declaration they're applied to and provide a new body that will be used when the function is called (Gregor, 2023b). `@StubbableFunction`'s body macro implementation reads the original body from the function declaration and replaces it with an if statement that checks whether or not the peer closure has been set to a non-`nil` value. The if branch calls the peer closure, passing along the function's parameters. The else branch contains the original method body:

```
@StubbableFunction
func add(_ a: Int, to b: Int) -> Int {
    return a + b
}
```

```
// Body macro expands to:

func add(_ a: Int, to b: Int) -> Int {
    if let _add {
        return _add(a, b)
    } else {
        return a + b
    }
}
```

During this expansion, it is crucial that the body macro correctly builds the call to the peer closure, ensuring that it uses the correct function call semantics for the function's type. So, `@StubbableFunction` correctly adds `return`, `try`, and `await` before the closure call if needed, always using the internal parameter names, and more to ensure it produces compilable Swift code.

For example, Swift functions can omit the `return` keyword if they have a non-void return type and only contain a single expression. When `@StubbableFunction` is applied to a function with an implicit return statement, it needs to add the `return` keyword itself. So, `@StubbableFunction` will check whether the function it's applied to has a void return type, which has many spellings: no return type clause, `-> Void`, `-> Swift.Void`, `-> ()`, `-> (Void)`, etc. Then, if the function is non-void and its body only contains a single expression, it will add `return` before the original body in the else branch. Here is an example of an asynchronous, throwing method with an implicit return statement:

```
@StubbableFunction
func method() async throws -> String {
    try await anotherMethod()
}

// Body macro expands to:

func method() async throws -> String {
    if let _method {
        return try await _method()
    } else {
        return try await anotherMethod()
    }
}
```

`@StubbableProperty`

Like `@StubbableFunction`, the `@StubbableProperty` macro performs two main functions: generating the base storage, getter, and setter; and transforming the property into a computed property. So, it also implements two attached macro roles: peer and accessor.

`@StubbableProperty`'s peer macro implementation begins by reading the name, type, and optional initial value from the variable declaration it is applied to. Then, it uses these to generate the three stored properties seen earlier: a private, mutable property with the same type and initial value, an optional getter closure that accepts no arguments and returns a value of the correct type, and an optional setter closure that takes in a value of the correct type and has no return value:

```
@StubbableProperty
var x: Int = 0

// Peer macro expands to:

private var _x: Int = 0
var _getX: (() -> Int)? = nil
```

```swift
var _setX: ((Int) -> Void)? = nil

var x: Int = 0
```

Because there is much less variation in the syntax of property declarations than method declarations, this peer macro implementation is much more straightforward than `@StubbableFunction`'s. The one major constraint is that, because Swift macros only operate on syntax and have no type information, the developer must provide an explicit type for the original property and cannot use Swift's powerful type inference. When `@StubbableProperty` is applied to a property without an explicit type declaration, it emits the following error and fix-it:

```swift
@StubbableProperty
var property = 1 // Error: '@StubbableProperty' requires an explicit type. (Fix)

// Clicking 'Fix' in Xcode results in:

@StubbableProperty
var property: <#Type#> = 1
```

Accessor macros can turn stored properties into computed properties by adding custom accessors. These are often used to turn a stored property into a wrapper around some other storage with minimal boilerplate (Gregor et al., 2023). `@StubbableProperty`'s accessor macro adds `get`, `set`, and `init` accessors to the original property to turn it into a stubbable wrapper around the underscored property generated by the peer macro implementation:

```swift
@StubbableProperty
var x: Int = 0

// Accessor macro expands to:

var x: Int {
    get {
        if let _getX {
            return _getX()
        } else {
            return _x
        }
    }
    set {
        if let _setX {
            _setX(newValue)
        } else {
            _x = newValue
        }
    }
    @storageRestrictions(initializes: _x)
    init(initialValue) {
        _x = initialValue
    }
}
```

The accessor macro only has one dependency: the name of the original property. So, its implementation is very simple since it only needs to read the original name of the property and generate the accessors, using the correct derivations of the original name for the names of the base storage, getter, and setter.

`@Stubbable`

`@Stubbable` implements a single attached macro role: member attribute. Member attribute macros can modify the member declarations of the types they're applied to by adding attributes to them. In Swift, attributes are annotations prefixed with `@` that give the compiler extra information about the type they're applied to – `@objc` indicates that the declaration can be used from Objective-C, `@discardableResult` indicates that the return value of a method can be ignored without a warning, etc. But, these attributes can also be references to other attached macros. (*Attributes*, n.d.; Gregor et al., 2023)

`@Stubbable` uses this macro role to visit each member of the type to be stubbed. It first examines the member's syntax to determine if it is a property, method, or neither. If the member is a supported property or a method, `@Stubbable` will apply `@StubbableProperty` or `@StubbableFunction` accordingly. Otherwise, it will leave the member unmodified. Consider the following example:

```
@Stubbable
class RosterService {
    let baseURL: URL
    var courses: [Course] = []

    init(baseURL: URL) { self.baseURL = baseURL }

    func fetchAllStudents() async throws -> [Student] {
        // fetch students in all courses from server
        return allStudents
    }
}

// Expands to:

class RosterService {
    let baseURL: URL

    @StubbableProperty
    var courses: [Course] = []

    init(baseURL: URL) { self.baseURL = baseURL }

    @StubbableFunction
    func fetchAllStudents() async throws -> [Student] {
        // fetch students in provided courses from server
        return allStudents
    }
}
```

In this example, the `courses` property received the `@StubbableProperty` macro, which performs the expansion needed to stub properties. However, even though `baseURL` is a property, it was left unmodified because it is immutable and thus, cannot be stubbed. Similarly, the class's initializer was left unmodified because Stubble cannot stub initializers. Last, the `fetchStudents()` method received the `@StubbableFunction` macro, which performs the expansion needed to stub functions.

Additionally, `@Stubbable` will emit a descriptive compilation error if it is applied to a declaration that cannot contain stored properties, such as enums, protocols, and extensions:

```
@Stubbable
enum LoadingState {
    case idle
    case loading
    case success
    case error
}

// Results in:

@Stubbable // Error: '@Stubbable' cannot be applied to enums
enum LoadingState {
    case idle
    case loading
    case success
    case error
}
```

Writing Tests with Stubble

While Stubble's implementation is sophisticated, it is very simple to use. Using Stubble to generate stubs for your types offers a significantly improved experience compared to the manual approach described earlier. To illustrate this, let's compare the test of `RosterViewModel.studentsByGraduationYear()` using the protocol-based approach with the same test written with Stubble:

```
// Production Code

@Stubbable
class RosterService {
    func fetchStudents() async throws -> [Student] {
        // make an HTTP request
        return students
    }
}

class RosterViewModel {
    let service: RosterService

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

    func studentsByGraduationYear() async throws -> [Int : [Student]] {
        let students = try await service.fetchStudents()
        return Dictionary(grouping: students, by: \.graduationYear)
    }
}

// Test code

@Test
func testEmptyRoster() async throws {
    let service = RosterService()
    service._fetchStudents = { return [] }

    let vm = RosterViewModel(service: service)

    let result = try await vm.studentsByGraduationYear()
    #expect(result == [:])
}
```

The test written with Stubble only needs a single change to the production source code – adding that `@Stubbable` attribute to the type – meaning that there is no boilerplate code at all to write and maintain.

Achieving Design Goals

Having seen how Stubble is implemented and used, we can evaluate how effectively Stubble achieves its design goals.

Compile-Time Safety

Stubble ensures that stub implementations can be safely interchanged with their production counterparts by very carefully building the peer closure types to be exactly correct. Then, because the macro expansion is passed on to the compiler, it can validate that these closures accept the same parameters, return the same types, and are called with the correct combination of `return`, `try`, and `await`.

Additionally, because these closures are strongly typed, the compiler enforces that any stub implementations treat the parameters as their expected types and return values of the expected types.

Flexibility

Because developers provide their stub implementations as a closure, developers have access to Swift's full feature set when writing their stubs, giving them significant flexibility. A developer could write stubs that return static values, throw errors, or implement arbitrarily complex behavior:

```swift
// Return static value
let service = RosterService()
service._fetchStudents = { return [] }

// Throw errors
let service = RosterService()
service._fetchStudents = { throw URLError.notConnectedToInternet }

// Complex behavior
var students: [Student] = []

let database = DatabaseManager()
database._query = { return students }
database._insert = { students.append($0) }
database._delete = { student in
    students.removeAll(where: { $0.id == student.id })
}
```

Easy Maintenance

Because all of the stubbing infrastructure is automatically generated by Stubble during every compilation, developers do not need to make any repetitive changes when refactoring their stubbed types.

Minimal Production Code Changes

As mentioned previously, adding stubbing capabilities to a type only requires a single source change: adding the `@Stubbable` annotation. This is because Stubble adds the stubbing infrastructure to the type itself, meaning that the rest of your production code can continue referring to the stubbed type directly, and there's no need to create protocols or refactor your types to use dependency injection, as discussed earlier.

Partial Stubbing

Again, because all of Stubble's stubbing infrastructure is generated inside the type itself, developers can choose a subset of members to stub. This even allows developers greater control over how they factor and modularize their code, since you can stub and test methods of the same type simultaneously, rather than factoring out the dependencies into a separate type:

```
@Stubbable
class RosterService {
    private func fetchStudents() async throws -> [Student] {
        // make an HTTP request
        return response
    }

    func fetchStudentsByGraduationYear() async throws -> [Int : [Student]] {
        let students = try await fetchStudents()
        return Dictionary(grouping: students, by: \.graduationYear)
    }
}

@Test
func testEmptyRoster() async throws {
    let service = RosterService()
    service._fetchStudents = { return [] }

    let result = try await service.fetchStudentsByGraduationYear()
    #expect(result == [:])
}
```

Ease of Use

The syntax used to stub implementations with Stubble is incredibly easy to understand, since it simply uses variable assignments and closures, just like other Swift code that developers are used to writing. This is much easier for developers to use than the behavior-driven development-inspired domain-specific languages used in other frameworks like Mockito for Java (*Mockito Home Page*, n.d.):

```
// Stubble (Swift)
let service = RosterService()
service._fetchStudents = { return [] }

// Mockito (Java)
RosterService service = mock(RosterService.class);
when(service.fetchStudents()).thenReturn(Collections.emptyList());
```

By leveraging Swift's macro system to generate stubbing infrastructure in place, and implementing a flexible, easy-to-understand API, Stubble successfully achieves its design goals: preserving the benefits and removing the limitations of the manual, protocol-based stubbing approach and doing so with a pleasant API.

Performance

While Stubble primarily focuses on improving the developer experience when stubbing, it is important to consider its impact on the execution time of the developer's test suite as well.

To compare the runtime performance of Stubble and the protocol-based approach, let's measure the execution time of the `RosterViewModel.studentsByGraduationYear()` seen earlier when stubbing `RosterService.fetchStudents()` to return an empty array with both approaches.

Specifically, we'll run `RosterViewModel.studentsByGraduationYear()` 50,000,000 times in a tight loop, measure the total time, and divide it by 50,000,000 to find the average execution time of a single call to the method. Performing that procedure with both approaches yields the following results:

Approach	Average Method Execution Time
Stubble	570 nanoseconds
Protocol-Based	489 nanoseconds

This method took approximately 81 nanoseconds longer on my machine when using Stubble than the protocol-based approach, resulting in a roughly 16.6% increase in execution time. This increase occurs because the Stubble requires a conditional check and closure dispatch to call the stub implementation, while the protocol-based approach only requires a single dynamic dispatch. However, Stubble's developer experience and productivity improvements – reduced boilerplate, easier maintenance, etc – outweigh the negligible execution time increases of reasonably sized testing suites.

Future work

Although Stubble already provides a solid foundation for macro-based stubbing in Swift, there are several promising directions for future development to extend its customizability and functionality.

1. **Protect Release Builds:** Currently, Stubble performs the same expansions during test and release builds, meaning that there is a slight performance overhead – as a result of the conditional checks – introduced in the optimized release versions of an app distributed via the App Store, for example. If Swift macros could read compiler flags during their expansion, Stubble's macros could leave the declarations completely unmodified during release builds, eliminating any performance impacts for end users.
2. **Extend Stubbing Support**: Stubble currently supports stubbing instance methods and instance properties of a type. But, it could additionally support other kinds of members such as generic methods, static methods, static properties, subscripts, and computed properties. Adding support for these in the future would be simple due to `@Stubbable`'s composition-based approach.
3. **Advanced Configuration:** Attached macros can accept parameters, which `@Stubbable` could use to give developers granular control over its behavior: which kinds of members get stubbed, the access level of the stubbing infrastructure, etc.

Conclusion

In this paper, I have presented Stubble, a stubbing library for Swift that leverages Swift's new, powerful macro system to automate the generation of stubbing infrastructure, allowing developers to control the behavior of a unit of code's dependencies while testing. This macro-based approach provides the same type safety and flexibility as the common protocol-based stubbing technique while eliminating its boilerplate, maintenance burden, and limitations, all with an easy-to-use API.

Stubble is distributed as a Swift package and its source is available on GitHub under the MIT license (https://github.com/samrshi/Stubble).

References

Attributes. (n.d.). The Swift Programming Language. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/attributes/

Free Software Foundation, Inc. (1992). *The c preprocessor.* The University of Utah Department of Mathematics. https://www.math.utah.edu/docs/info/cpp_1.html#SEC10

Gregor, D. (2023a). *A vision for macros in swift.* Swift Evolution on GitHub. https://github.com/swiftlang/swift-evolution/blob/main/visions/macros.md

Gregor, D. (2023b). *Function body macros*. Swift Evolution on GitHub. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0415-function-body-macros.md

Gregor, D., Borla, H., & Wei, R. (2023). *Attached macros*. Swift Evolution on GitHub. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0389-attached-macros.md

Kumari, R. (2024). *What is a test double? | types & best practices*. testsigma. https://testsigma.com/blog/test-doubles/

Lee, A. van der. (2023). *Swift macros: Extend swift with new kinds of expressions*. SwiftLee. https://www.avanderlee.com/swift/macros/

Macros. (n.d.). The Rust Programming Language. https://doc.rust-lang.org/book/ch19-06-macros.html

Mockito home page. (n.d.). Mockito. https://site.mockito.org

Polaczyk, B. (2018). *Stubbing in pair with swift compiler*. Medium. https://medium.com/@londeix/stubbing-in-pair-with-swift-compiler-c951770a295b

Protocols. (n.d.). The Swift Programming Language. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/protocols/

Sandberg, A., & Borla, H. (2023). *Swift 5.9 released*. Swift. https://www.swift.org/blog/swift-5.9-released/