The Two Sides of Testing

Why test?

Our tests

Our tests

→ 111 test files

Our tests

- → 111 test files
- → 1,221 test cases

Our tests

- → 111 test files
- → 1,221 test cases
- → 999 unit tests

Our tests

- → 111 test files
- → 1,221 test cases
- → 999 unit tests
- → 222 screenshot tests

Our tests

- → 111 test files
- → 1,221 test cases
- → 999 unit tests
- → 222 screenshot tests
- → 655 screenshots

Our tests

- → 111 test files
- → 1,221 test cases
- → 999 unit tests
- → 222 screenshot tests
- → 655 screenshots
- → 5,382 assertions

Our tests

- → 111 test files
- → 1,221 test cases
- → 999 unit tests
- → 222 screenshot tests
- → 655 screenshots
- → 5,382 assertions
- → 50% of code is tests

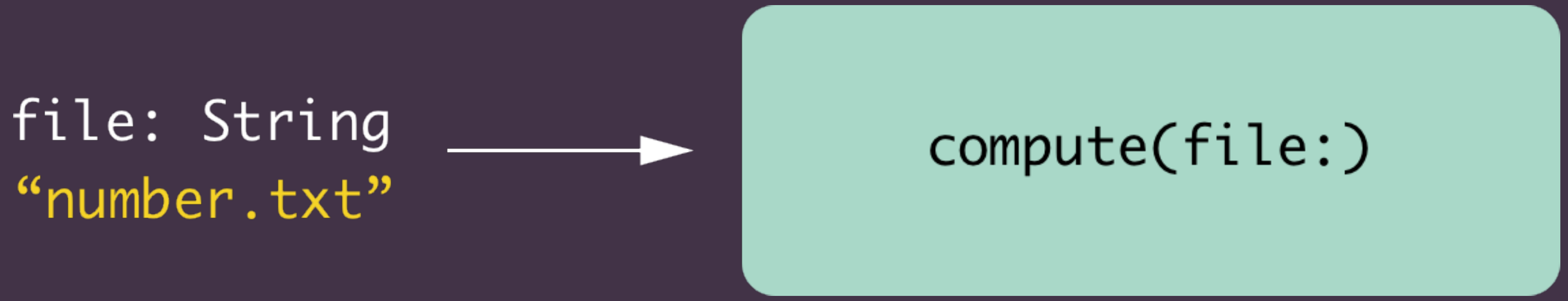
A case study

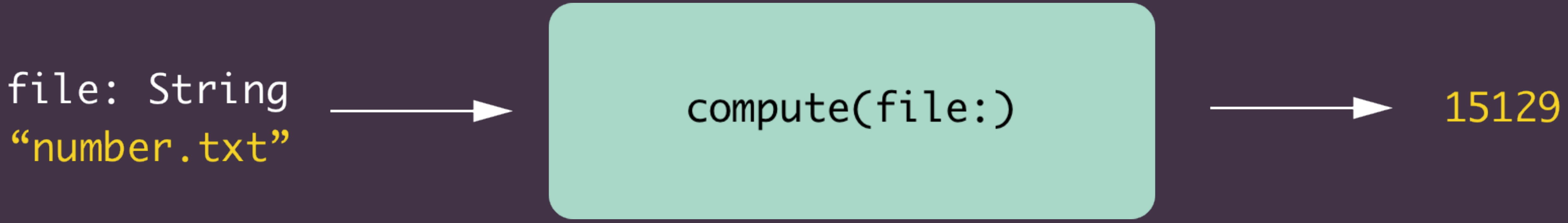
```swift
/**
 * Reads a number from a file on disk, performs a computation, prints
 * the result to the console, and returns the result.
 */
func compute(file: String) -> Int {
```

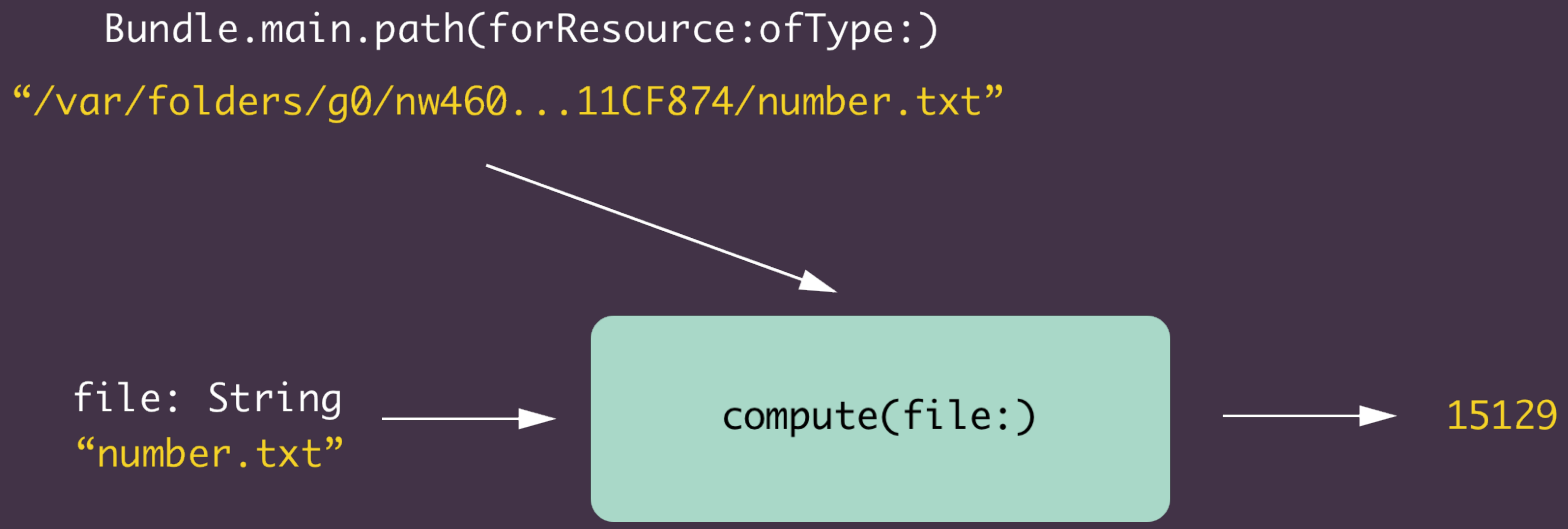
`

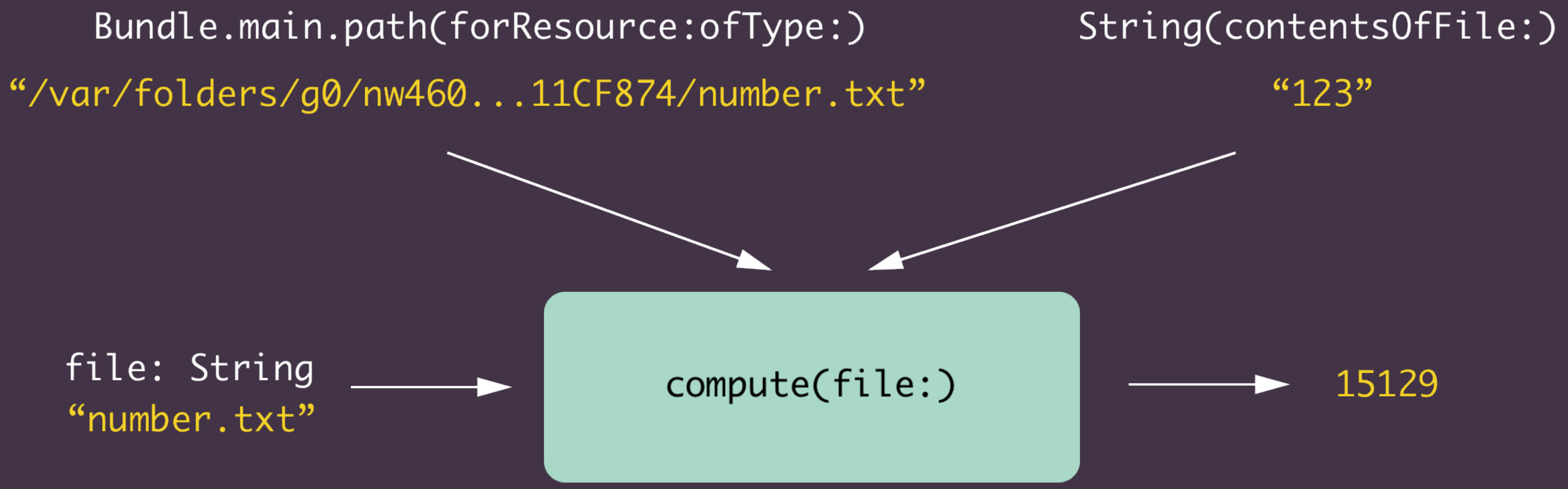
```swift
/**
 * Reads a number from a file on disk, performs a computation, prints
 * the result to the console, and returns the result.
 */
func compute(file: String) -> Int {

  let value = Bundle.main.path(forResource: file, ofType: nil)
    .flatMap { try? String(contentsOfFile: $0) }
    .flatMap { Int($0) }
    ?? 0

  let result = value * value

  print("Computed: \(result)")

  return result
}
```

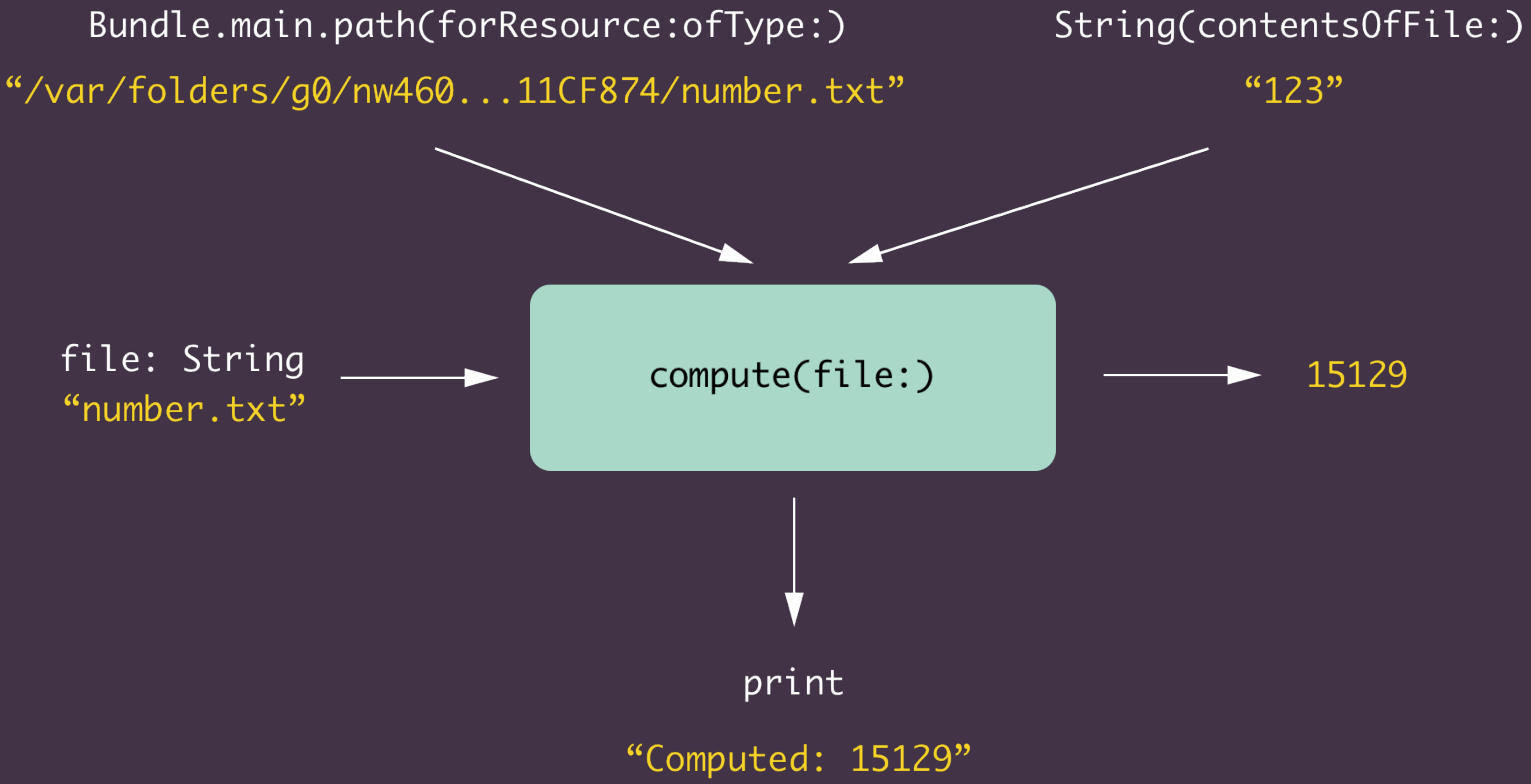
```swift
/**
 * Reads a number from a file on disk, performs a computation, prints
 * the result to the console, and returns the result.
 */
func compute(file: String) -> Int {

  let value = Bundle.main.path(forResource: file, ofType: nil)   // "/var/folders/.../number.txt"
    .flatMap { try? String(contentsOfFile: $0) }                 // "123"
    .flatMap { Int($0) }                                         // 123
    ?? 0                                                         // 123

  let result = value * value                                     // 15129

  print("Computed: \(result)")                                   // "Computed: 15129\n"

  return result                                                  // 15129
}

compute(file: "number.txt")                                      // 15129
compute(file:)
```

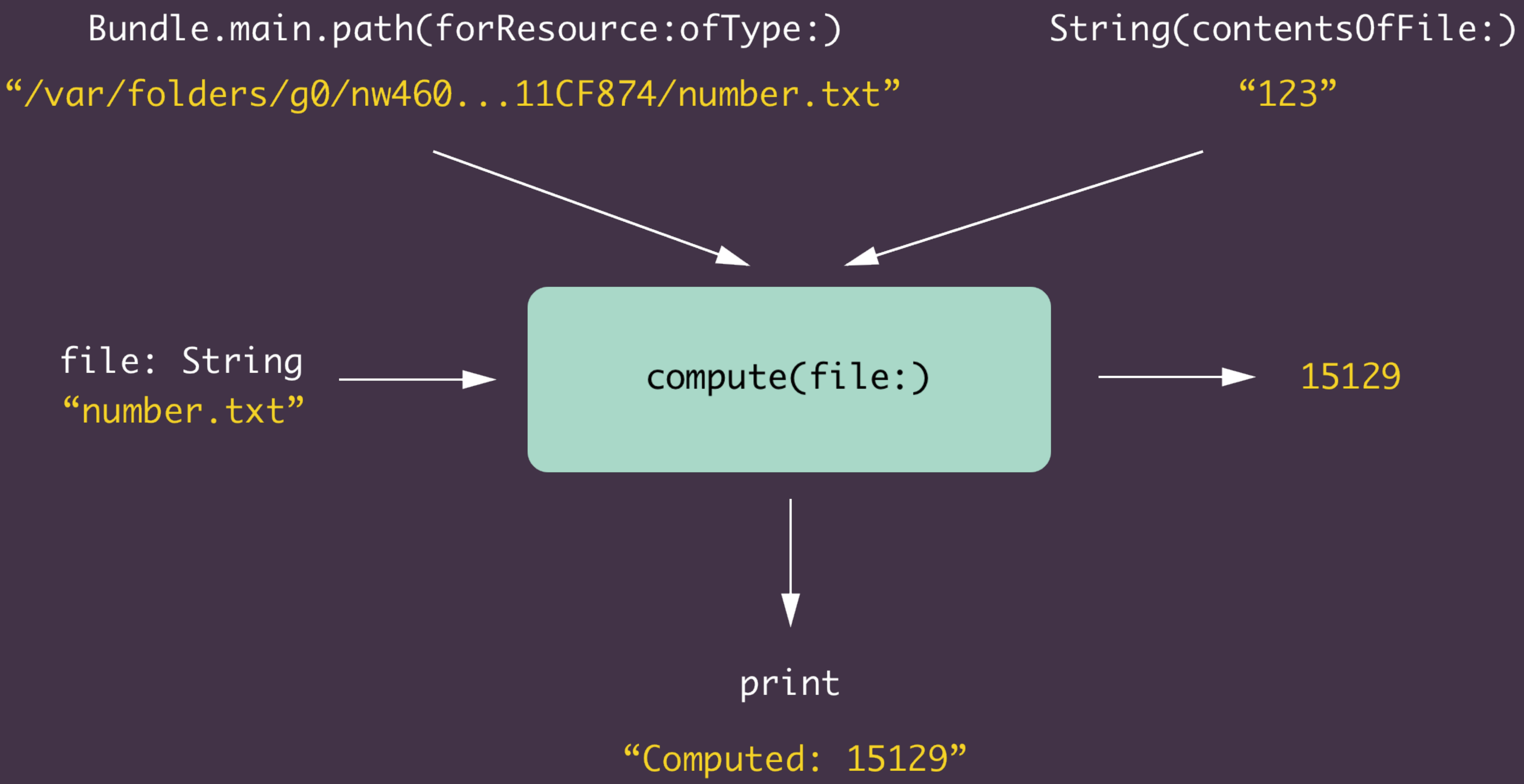

file: String
“number.txt”
compute(file:)


file: String
“number.txt”
compute(file:)
15129


Bundle.main.path(forResource:ofType:)
“/var/folders/g0/nw460...11CF874/number.txt”
file: String
“number.txt”
compute(file:)
15129


Bundle.main.path(forResource:ofType:)
“/var/folders/g0/nw460...11CF874/number.txt”
String(contentsOfFile:)
“123”
file: String
“number.txt”
compute(file:)
15129


Bundle.main.path(forResource:ofType:)
“/var/folders/g0/nw460...11CF874/number.txt”
String(contentsOfFile:)
“123”
file: String
“number.txt”
compute(file:)
15129
print
“Computed: 15129”


Bundle.main.path(forResource:ofType:)
"/var/folders/g0/nw460...11CF874/number.txt"
String(contentsOfFile:)
"123"
file: String
"number.txt"
compute(file:)
15129
print
"Computed: 15129"

Testing output

Testing output

Side effects

Side effects

An expression is said to have a “side effect” if its execution makes an observable change to the outside world.

How do you test code with side effects?

A better way to handle side effects

A better way to handle side effects

Try to *describe* effects as much as possible without actually performing the effects.

A better way to handle side effects

```
func compute(file: String) -> (Int, String) {

  let value = Bundle.main.path(forResource: file, ofType: nil)
    .flatMap { try? String(contentsOfFile: $0) }
    .flatMap { Int($0) }
    ?? 0

  let result = value * value

  return (result, "Computed: \(result)")
}
```

Testing Input

Testing Input

Co-effects

Testing Input

Co-effects

i.e. the “dual” of side effects

Co-effects

If an effect is a change to the outside world after executing an expression…

…then…

…a co-effect is the state of the world that the expression needs in order to execute.

Co-effects

An expression is said to have a “co-effect” if it requires a particular state of the world in order to execute.

How do you test code with co-effects?

A better way to handle co-effects

```
struct Environment {

}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
}
```

```
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
  let dateProtocol: DateProtocol.Type
}
```

```
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
  let dateProtocol: DateProtocol.Type
  let language: Language
}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
  let dateProtocol: DateProtocol.Type
  let language: Language
  let mainBundle: BundleProtocol
}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
  let dateProtocol: DateProtocol.Type
  let language: Language
  let mainBundle: BundleProtocol
  let reachability: SignalProducer<Reachability, NoError>
}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
  let dateProtocol: DateProtocol.Type
  let language: Language
  let mainBundle: BundleProtocol
  let reachability: SignalProducer<Reachability, NoError>
  let scheduler: DateSchedulerProtocol
}
```

```swift
struct Environment {
  let apiService: ServiceProtocol
  let cookieStorage: HTTPCookieStorageProtocol
  let currentUser: User?
  let dateProtocol: DateProtocol.Type
  let language: Language
  let mainBundle: BundleProtocol
  let reachability: SignalProducer<Reachability, NoError>
  let scheduler: DateSchedulerProtocol
  let userDefaults: UserDefaultsProtocol
}
```

A better way to handle co-effects

Refactor

Refactor

```
Bundle.main.path(forResource: file, ofType: nil)
```

Refactor

```
protocol BundleProtocol {
  func path(forResource name: String?, ofType ext: String?) -> String?
}

extension Bundle: BundleProtocol {}
```

Refactor

```
struct SuccessfulPathForResourceBundle: BundleProtocol {
  func path(forResource name: String?, ofType ext: String?) -> String? {
    return "a/path/to/a/file.txt"
  }
}

struct FailedPathForResourceBundle: BundleProtocol {
  func path(forResource name: String?, ofType ext: String?) -> String? {
    return nil
  }
}
```

Refactor

```
String(contentsOfFile: file)
```

Refactor

```swift
protocol ContentsOfFileProtocol {
  static func from(contentsOfFile file: String) throws -> String
}

extension String: ContentsOfFileProtocol {
  static func from(contentsOfFile file: String) throws -> String {
    return try String(contentsOfFile: file)
  }
}
```

Refactor

```
struct IntContentsOfFile: ContentsOfFileProtocol {
  static func from(contentsOfFile file: String) throws -> String {
    return "123"
  }
}

struct NonIntContentsOfFile: ContentsOfFileProtocol {
  static func from(contentsOfFile file: String) throws -> String {
    return "asdf"
  }
}
```

Refactor

```
func compute(file: String,
             bundle: BundleProtocol = Bundle.main,
             contentsOfFileProtocol: ContentsOfFileProtocol.Type = String.self) -> (Int, String) {

  let value = bundle.path(forResource: file, ofType: nil)
    .flatMap { try? contentsOfFileProtocol.from(contentsOfFile: $0) }
    .flatMap { Int($0) }
    ?? 0

  let result = value * value

  return (result, "Computed: \(result)")
}
```

Conclusion

Conclusion

The two things that make testing difficult are **effects** and **co-effects**.

Conclusion

To tame **effects**, think of them as data in their own right, and you simply describe the effect rather than actually perform it.

A naive interpreter can perform the effects somewhere else.

Conclusion

To tame **co-effects**, put them all in one big ole global struct, and don't ever access a global unless it is through that struct.

Thanks