Minimal Cake Pattern in Swift

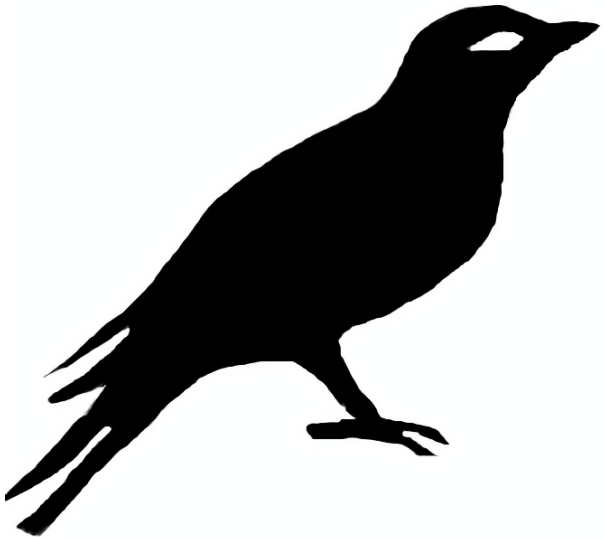
kbkz.tech #11

吉村 優

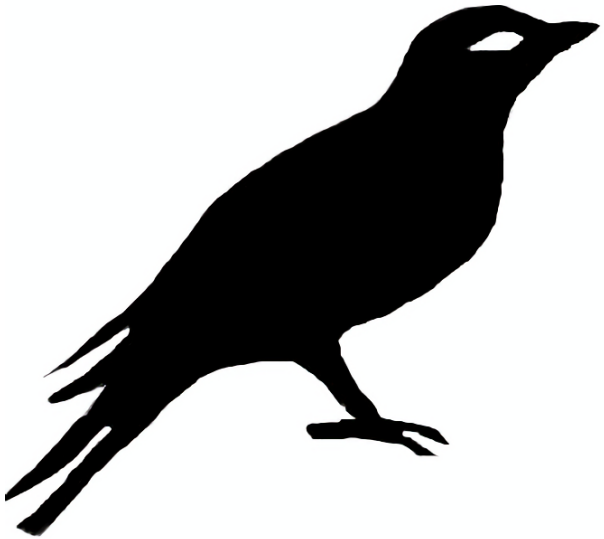
https://twitter.com/_yyu_
http://qiita.com/yyu
https://github.com/y-yu

September 16, 2016
(Commit ID: 5acb721)

自己紹介

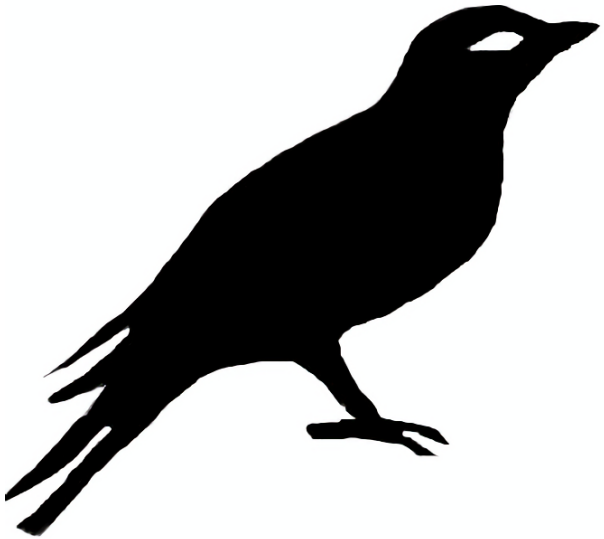


自己紹介



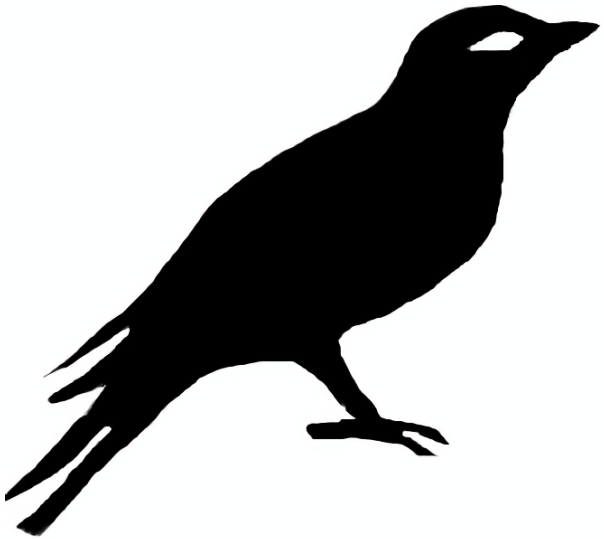
- Scala を書く仕事に従事

自己紹介



- Scala を書く仕事に従事
- 趣味は LaTeX と暗号技術

自己紹介



- Scala を書く仕事に従事
- 趣味は LaTeX と暗号技術
- Swift は最近はじめた初心者

DI（Dependency Injection）とは？

DI（Dependency Injection）とは？

Dependency Injection とは？

“

- *Dependency* とは実際にサービスなどで使われるオブジェクトである
- *Injection* とは *Dependency* オブジェクトを使うオブジェクトに渡すことである

”

Scala における最適な *Dependency Injection* の方法を考察する *[1]*

DIのメリット

DIのメリット

メリット

“

”

Scala における最適な *Dependency Injection* の方法を考察する *[1]*

DIのメリット

メリット

“

- コンポーネント同士が疎結合になる

”

Scala における最適な *Dependency Injection* の方法を考察する *[1]*

DIのメリット

メリット

“

- コンポーネント同士が疎結合になる
- クライアントの動作がカスタマイズ可能になる

”

Scala における最適な *Dependency Injection* の方法を考察する *[1]*

DIのメリット

メリット

“

- コンポーネント同士が疎結合になる
- クライアントの動作がカスタマイズ可能になる
- 依存オブジェクトのモック化によるユニットテストが可能になる

”

Scala における最適な *Dependency Injection* の方法を考察する *[1]*

DIのメリット

メリット

“

- コンポーネント同士が疎結合になる
- クライアントの動作がカスタマイズ可能になる
- 依存オブジェクトのモック化によるユニットテストが可能になる

”

Scala における最適な *Dependency Injection* の方法を考察する *[1]*

どうやってやるの？

Swift における代表的な DI 手法

Swift における代表的な DI 手法

Swift における代表的な DI 手法

“

”

Dependency Injection in Swift 2.x[2]

Swift における代表的な DI 手法

Swift における代表的な DI 手法

“

- *Swinject* を用いた動的な *DI*

”

Dependency Injection in Swift 2.x[2]

Swift における代表的な DI 手法

Swift における代表的な DI 手法

“

- *Swinject* を用いた動的な *DI*
- *Cake Pattern* を用いた静的な *DI*

”

Dependency Injection in Swift 2.x[2]

Swift における代表的な DI 手法

Swift における代表的な DI 手法

“

- *Swinject* を用いた動的な *DI*
- *Cake Pattern* を用いた静的な *DI*

”

Dependency Injection in Swift 2.x[2]

これら以外にはないの？

Swift における代表的な DI 手法

Swift における代表的な DI 手法

“

- *Swinject* を用いた動的な *DI*
- *Cake Pattern* を用いた静的な *DI*

”

Dependency Injection in Swift 2.x[2]

これら以外にはないの？

Cake Pattern の仲間 “Minimal Cake Pattern” を紹介！

Example

こんな機能を作りたい

*ソルトを使ったハッシュ関数のこと

Example

こんな機能を作りたい

`HashPasswordService`

パスワードを鍵付きハッシュ関数*でハッシュ化する機能

*ソルトを使ったハッシュ関数のこと

Example

こんな機能を作りたい

`HashPasswordService`

パスワードを鍵付きハッシュ関数*でハッシュ化する機能

必要な機能

*ソルトを使ったハッシュ関数のこと

Example

こんな機能を作りたい

HashPasswordService

パスワードを鍵付きハッシュ関数*でハッシュ化する機能

必要な機能

- 設定ファイルを読み込む機能

*ソルトを使ったハッシュ関数のこと

Example

こんな機能を作りたい

`HashPasswordService`

パスワードを鍵付きハッシュ関数*でハッシュ化する機能

必要な機能

- 設定ファイルを読み込む機能
- パスワードをハッシュ化する機能

*ソルトを使ったハッシュ関数のこと

Example

こんな機能を作りたい

HashPasswordService

パスワードを鍵付きハッシュ関数*でハッシュ化する機能

必要な機能

- 設定ファイルを読み込む機能
- パスワードをハッシュ化する機能

設定ファイルを読み込むのはなぜ？

*ソルトを使ったハッシュ関数のこと

Example

考えること

Example

考えること

- ソルトをハードコードするのは微妙

Example

考えること

- ソルトをハードコードするのは微妙
- ソルトは設定ファイルに保存する

Example

考えること

- ソルトをハードコードするのは微妙
- ソルトは設定ファイルに保存する
- 一方、テストの時はファイルから読み込みたくない

Example

考えること

- ソルトをハードコードするのは微妙
- ソルトは設定ファイルに保存する
- 一方、テストの時はファイルから読み込みたくない

ファイル IO に失敗したらテストが失敗する！

Example

考えること

- ソルトをハードコードするのは微妙
- ソルトは設定ファイルに保存する
- 一方、テストの時はファイルから読み込みたくない

ファイル IO に失敗したらテストが失敗する！

- 本実装はソルトを設定ファイルから読み込んで、テストの時はハードコードしたソルトを使う

Example

考えること

- ソルトをハードコードするのは微妙
- ソルトは設定ファイルに保存する
- 一方、テストの時はファイルから読み込みたくない

ファイル IO に失敗したらテストが失敗する！

- 本実装はソルトを設定ファイルから読み込んで、テストの時はハードコードしたソルトを使う

二つの実装が必要

設定ファイルを読み込む機能

まずは設定ファイルを読み込む部分をつくる

設定ファイルを読み込む機能

まずは設定ファイルを読み込む部分をつくる

1. インターフェース `ReadConfigService` を作成

```
protocol ReadConfigService {
  var configName: String { get }

  func readSalt() -> String
}
```

設定ファイルを読み込む機能

まずは設定ファイルを読み込む部分をつくる

1. インターフェース ReadConfigServiceを作成

```
protocol ReadConfigService {
  var configName: String { get }

  func readSalt() -> String
}
```

2. 実装を投入

```
extension ReadConfigService {
  private func readFile() -> Optional<String> {
    if ファイルをオープンする {
      return Optional.Some(ファイルの中身)
    } else {
      return Optional.None
    }
  }
}
```

設定ファイルを読み込む機能

4. メイン実装を ReadConfigServiceImplを作成

```
class ReadConfigServiceImpl: ReadConfigService {
  let configName: String

  init(_ str: String) {
    configName = str
  }

  func readSalt() -> String {
    // 本当はもっとちゃんとやる……
    return readFile()!
  }
}
```

設定ファイルを読み込む機能

4. メイン実装を ReadConfigServiceImplを作成

```
class ReadConfigServiceImpl: ReadConfigService {
  let configName: String

  init(_ str: String) {
    configName = str
  }

  func readSalt() -> String {
    // 本当はもっとちゃんとやる……
    return readFile()!
  }
}
```

設定ファイルのパスを引数で受け取って、ファイルをオープンしてソルトを読み込む

設定ファイルを読み込む機能

5. モック実装 ReadConfigServiceMockImplを作成

```
class ReadConfigServiceMockImpl: ReadConfigService {
  var configName: String = "dummy"
  let dummySalt: String

  init(_ salt: String) {
    dummySalt = salt
  }

  func readSalt() -> String {
    return dummySalt
  }
}
```

設定ファイルを読み込む機能

5. モック実装 ReadConfigServiceMockImplを作成

```
class ReadConfigServiceMockImpl: ReadConfigService {
  var configName: String = "dummy"
  let dummySalt: String

  init(_ salt: String) {
    dummySalt = salt
  }

  func readSalt() -> String {
    return dummySalt
  }
}
```

設定ファイルのパスはダミー、ソルトもコンストラクタの引数で与えられた値を必ず返す

設定ファイルを読み込む機能

5. モック実装 ReadConfigServiceMockImplを作成

```
class ReadConfigServiceMockImpl: ReadConfigService {
  var configName: String = "dummy"
  let dummySalt: String

  init(_ salt: String) {
    dummySalt = salt
  }

  func readSalt() -> String {
    return dummySalt
  }
}
```

設定ファイルのパスはダミー、ソルトもコンストラクタの引数で与えられた値を必ず返す

つまり、設定ファイルにはアクセスしない！

設定ファイルを読み込む機能

6. 依存を示すインターフェース UsesReadConfigServiceを作成

```
protocol UsesReadConfigService {
  var readConfigService: ReadConfigService { get }
}
```

設定ファイルを読み込む機能

6. 依存を示すインターフェース `UsesReadConfigService` を作成

```
protocol UsesReadConfigService {
  var readConfigService: ReadConfigService { get }
}
```

あとで使います

パスワードをハッシュ化する機能

次に、パスワードをハッシュ化する部分を作成

パスワードをハッシュ化する機能

次に、パスワードをハッシュ化する部分を作成

1. インターフェースを作成

```
protocol HashPasswordService: UsesReadConfigService {
  func hashBySha1(password: String) -> String
}
```

パスワードをハッシュ化する機能

次に、パスワードをハッシュ化する部分を作成

1. インターフェースを作成

```
protocol HashPasswordService: UsesReadConfigService {
  func hashBySha1(password: String) -> String
}
```

ReadConfigServiceに依存することを示す

パスワードをハッシュ化する機能

次に、パスワードをハッシュ化する部分を作成

1. インターフェースを作成

```
protocol HashPasswordService: UsesReadConfigService {
  func hashBySha1(password: String) -> String
}
```

ReadConfigServiceに依存することを示す

2. 実装を投入

```
extension HashPasswordService {
  func hashBySha1(password: String) -> String {
    let data = password + readConfigService.readSalt()
    // このコードはイメージです
    return SHA1(data).toString()
  }
}
```

パスワードをハッシュ化する機能

1. メイン実装を作成

```
class HashPasswordSerivceImpl: HashPasswordService {
  let readConfigService: ReadConfigService

  init(_ configFile: String) {
    readConfigService = ReadConfigServiceImpl(configFile)
  }
}
```

パスワードをハッシュ化する機能

1. メイン実装を作成

```
class HashPasswordSerivceImpl: HashPasswordService {
  let readConfigService: ReadConfigService

  init(_ configFile: String) {
    readConfigService = ReadConfigServiceImpl(configFile)
  }
}
```

ReadConfigServiceのメイン実装を DI

パスワードをハッシュ化する機能

1. メイン実装を作成

```
class HashPasswordSerivceImpl: HashPasswordService {
  let readConfigService: ReadConfigService

  init(_ configFile: String) {
    readConfigService = ReadConfigServiceImpl(configFile)
  }
}
```

ReadConfigServiceのメイン実装を DI

2. テスト実装を作成

```
class HashPasswordServiceTestImpl: HashPasswordService {
  var readConfigService: ReadConfigService =
    ReadConfigServiceMockImpl("dummySalt")
}
```

パスワードをハッシュ化する機能

1. メイン実装を作成

```
class HashPasswordSerivceImpl: HashPasswordService {
  let readConfigService: ReadConfigService

  init(_ configFile: String) {
    readConfigService = ReadConfigServiceImpl(configFile)
  }
}
```

ReadConfigServiceのメイン実装を DI

2. テスト実装を作成

```
class HashPasswordServiceTestImpl: HashPasswordService {
  var readConfigService: ReadConfigService =
    ReadConfigServiceMockImpl("dummySalt")
}
```

ReadConfigServiceのモック実装を DI

パスワードをハッシュ化する機能

11. モックも作成

```
class HashPasswordServiceMockImpl: HashPasswordService {
  var readConfigService: ReadConfigService =
    ReadConfigServiceMockImpl("dummySalt")

  func hashBySha1(password: String) -> String {
    return password + "_dummySalt"
  }
}
```

パスワードをハッシュ化する機能

11. モックも作成

```
class HashPasswordServiceMockImpl: HashPasswordService {
  var readConfigService: ReadConfigService =
    ReadConfigServiceMockImpl("dummySalt")

  func hashBySha1(password: String) -> String {
    return password + "_dummySalt"
  }
}
```

モックはハッシュ値を計算しない

パスワードをハッシュ化する機能

11. モックも作成

```
class HashPasswordServiceMockImpl: HashPasswordService {
  var readConfigService: ReadConfigService =
    ReadConfigServiceMockImpl("dummySalt")

  func hashBySha1(password: String) -> String {
    return password + "_dummySalt"
  }
}
```

モックはハッシュ値を計算しない

たとえこのサービスの実装が変更されても、依存するサービスのテストは落ちない

テスト

作成した `HashPasswordService` のテストを作る

テスト

作成した HashPasswordService のテストを作る

```
import XCTest
@testable import MCPExample

class HashPasswordServiceTest: XCTestCase {
  // テスト用の実装を使う
  var sut: HashPasswordService = HashPasswordServiceTestImpl()

  func testHashBySha1() {
    XCTAssert(sut.hashBySha1("hoge") == "
    ba5b31f489676c1545a9f175867274c8c21b8f8b")
  }
}
```

テスト

作成した HashPasswordService のテストを作る

```
import XCTest
@testable import MCPExample

class HashPasswordServiceTest: XCTestCase {
  // テスト用の実装を使う
  var sut: HashPasswordService = HashPasswordServiceTestImpl()

  func testHashBySha1() {
    XCTAssert(sut.hashBySha1("hoge") == "
    ba5b31f489676c1545a9f175867274c8c21b8f8b")
  }
}
```

設定ファイルを読まない！

DI の漏れ

DIの漏れ

プログラマのミスで次のようにDIを忘れることがある

```
class HashPasswordServiceNgImpl: HashPasswordService {
    func hashBySha1(password: String) -> String {
        return "ng"
    }
}
```

DIの漏れ

プログラマのミスで次のようにDIを忘れることがある

```
class HashPasswordServiceNgImpl: HashPasswordService {
    func hashBySha1(password: String) -> String {
        return "ng"
    }
}
```

コンパイルエラー

```
HashPasswordService.swift:41:7: Type 'HashPasswordServiceNgImpl'
does not conform to protocol 'UsesReadConfigService'
```

DI の漏れ

プログラマのミスで次のように DI を忘れることがある

```
class HashPasswordServiceNgImpl: HashPasswordService {
    func hashBySha1(password: String) -> String {
        return "ng"
    }
}
```

コンパイルエラー

```
HashPasswordService.swift:41:7: Type 'HashPasswordServiceNgImpl'
does not conform to protocol 'UsesReadConfigService'
```

実行前に DI 漏れを検出！

DI の漏れ

プログラマのミスで次のように DI を忘れることがある

```
class HashPasswordServiceNgImpl: HashPasswordService {
    func hashBySha1(password: String) -> String {
        return "ng"
    }
}
```

コンパイルエラー

```
HashPasswordService.swift:41:7: Type 'HashPasswordServiceNgImpl'
does not conform to protocol 'UsesReadConfigService'
```

実行前に DI 漏れを検出！

依存が増えると効果的！

DIの漏れ

次のように依存が増えていくと、DI 漏れがありえてくる

```
protocol CreateUserService:
  UsesSessionService,
  UsesUserRepository,
  UsesClock,
  UsesApplicationLogger,
  UsesUserConfig,
  UsesRandomGenerator {

  func create(user: User) -> Future<Session>
}
```

DIの漏れ

次のように依存が増えていくと、DI 漏れがありえてくる

```
protocol CreateUserService:
  UsesSessionService,
  UsesUserRepository,
  UsesClock,
  UsesApplicationLogger,
  UsesUserConfig,
  UsesRandomGenerator {

  func create(user: User) -> Future<Session>
}
```

大きなアプリケーションを起動させるのは大変！

DI の漏れ

次のように依存が増えていくと、DI 漏れがありえてくる

```
protocol CreateUserService:
  UsesSessionService,
  UsesUserRepository,
  UsesClock,
  UsesApplicationLogger,
  UsesUserConfig,
  UsesRandomGenerator {

  func create(user: User) -> Future<Session>
}
```

大きなアプリケーションを起動させるのは大変！

起動させなくても DI 漏れが検出できるのは便利！

まとめ

Minimal Cake Pattern のメリット

まとめ

Minimal Cake Pattern のメリット

- コンパイラによる静的なチェックにより、DI 漏れを検出できる

まとめ

Minimal Cake Pattern のメリット

- コンパイラによる静的なチェックにより、DI 漏れを検出できる
- DI のための特別なライブラリが必要ない

まとめ

Minimal Cake Pattern のメリット

- コンパイラによる静的なチェックにより、DI 漏れを検出できる
- DI のための特別なライブラリが必要ない
- 普通の Cake Pattern に比べてシンプルである

目次

参考文献

[1] 結城清太郎.
Scala における最適な dependency injection の方法を考察する ～なぜドワンゴアカウントシステムの生産性は高いのか～, 2015.

[2] ゴミ人間.
Dependency injection in swift 2.x, 2016.

[3] 吉村優.
Swift でも minimal cake pattern, 2016.

[4] Takashi Tayama.
Minimal cake pattern のお作法, 2015.

Thank you for listening!
Any question?