# Factoryの自動生成によりテストを書きやすくする

# Takeshi Ihara

- AbemaTV
- Twitter: @nonchalant0303
- GitHub: Nonchalant
- Climbing 🧗
- Game 🎮

2　Factoryの自動生成によりテストを書きやすくする, iOSDC 2018 Reject Conference days1

# Test

```
struct User {
    let age: Int

    var isAdult: Bool {
        return age >= 20
    }
}

class UserTests: XCTestCase {
    func test20歳以上なら成人である() {
        let user = User(age: 20)
        XCTAssertTrue(user.isAdult)
    }

    func test20歳未満なら成人でない() {
        let user = User(age: 19)
        XCTAssertFalse(user.isAdult)
    }
}
```

# New Property

```
struct User {
    let name: String
    let age: Int

    var isAdult: Bool {
        return age >= 20
    }
}

class UserTests: XCTestCase {
    func test20歳以上なら成人である() {
        let user = User(age: 20)
        XCTAssertTrue(user.isAdult)
    }

    func test20歳未満なら成人でない() {
        let user = User(age: 19)
        XCTAssertFalse(user.isAdult)
    }
}
```

# Compile Error

```
struct User {
    let name: String
    let age: Int
    let birthday: Date

    var isAdult: Bool {
        return age >= 20
    }
}

class UserTests: XCTestCase {
    func test20歳以上なら成人である() {
        let user = User(age: 20) // Missing argument for parameter 'name' in call
        XCTAssertTrue(user.isAdult)
    }

    func test20歳未満なら成人でない() {
        let user = User(age: 19) // Missing argument for parameter 'name' in call
        XCTAssertFalse(user.isAdult)
    }
}
```

# Fragile Test

```
struct User {
    let name: String
    let age: Int

    var isAdult: Bool {
        return age >= 20
    }
}

class UserTests: XCTestCase {
    func test20歳以上なら成人である() {
        let user = User(name: "Takeshi Ihara", age: 20)
        XCTAssertTrue(user.isAdult)
    }

    func test20歳未満なら成人でない() {
        let user = User(name: "Takeshi Ihara", age: 19)
        XCTAssertFalse(user.isAdult)
    }
}
```

# Factory Pattern

## オブジェクトの生成処理を共通化する

```
struct UserFactory {
    static func provide(name: String = "", age: Int = 0) -> User {
        return User(name: name, age: age)
    }
}

class UserTests: XCTestCase {
    func test20歳以上なら成人である() {
        let user = UserFactory.provide(age: 20)
        XCTAssertTrue(user.isAdult)
    }

    func test20歳未満なら成人でない() {
        let user = UserFactory.provide(age: 19)
        XCTAssertFalse(user.isAdult)
    }
}
```

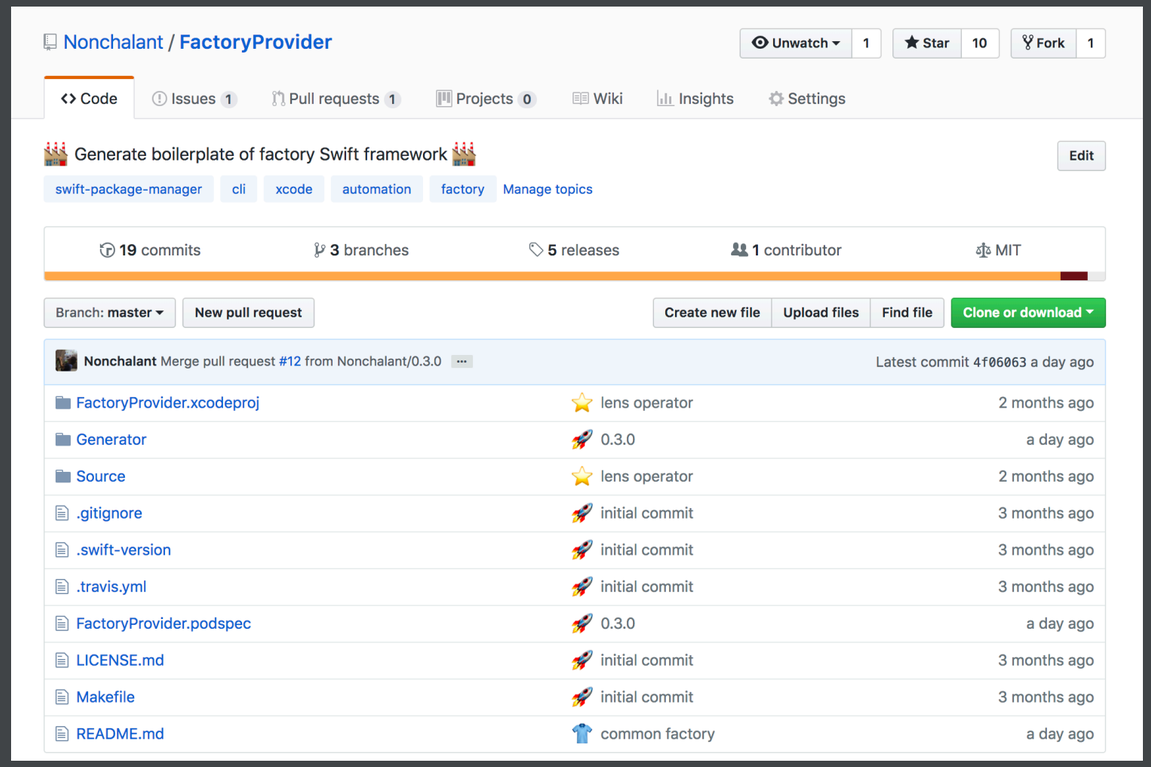

Factoryの自動生成によりテストを書きやすくする, iOSDC 2018 Reject Conference days1

# Cost of Factory

- テスト対象のオブジェクトの数だけFactoryが必要になる
  - ネストが深い型だと依存オブジェクトの数だけ必要になる
- Factoryなどを用意するコストが高いとテストを書かなくなる

# FactoryProvider

https://github.com/Nonchalant/FactoryProvider



# FactoryProvider

https://github.com/Nonchalant/FactoryProvider

- Factoryを自動生成するライブラリ
- Enum, Structが生成対象
- Lensをサポート
- ymlで設定項目を定義
- Generatorを含めるためにCocoapodsでのみインストール可能

# Factory (Struct)

型パラメータにStructを指定する

```
import FactoryProvider

let user = Factory<User>.provide()
// User(name: "", age: 0)
```

# Factory (Enum)

型パラメータにEnumを指定する

```
import FactoryProvider

let season = Factory<Season>.provide()
// Season.spring

enum Season {
    case spring
    case summer
    case automn
    case winter
}
```

# Generated Object

固定値で生成される😱

```
import FactoryProvider

var user = Factory<User>.provide()
user.name = "Takeshi Ihara" // Cannot assign to property: 'name' is a 'let' constant

struct User {
    let name: String
    let age: Int

    var isAdult: Bool {
        return age >= 20
    }
}
```

# Lens

- 不変性を保ちつつネストしたデータ構造に対するアクセスをLensの合成で表現できるようにしたもの
- 元々はHaskellの概念
- SwiftzのLens実装を独立したフレームワークとして切り出したFocusというフレームワークも存在
  - https://github.com/typelift/Focus

# Lens

```
import FactoryProvider

let user = Factory<User>.provide()
// User(name: "", age: 0)

let newUser = user |> User._age *~ 20 // User._ageをLensと呼ぶ
// User(name: "", age: 20)
```

# Lens (Nested)

```
import FactoryProvider

struct User {
    let id: UserId
    let name: String
    let age: Int
}

struct UserId {
    let value: String
}

let user = Factory<User>.provide() |> User._id * UserId._value *~ "nonchalant0303"
// User(id: UserId(value: "nonchalant0303"), name: "", age: 0)
```

# Config

## ymlファイルで設定する

```
includes: # 生成対象のStruct，Enumを含んだファイルへのパス（ファイル単位、ディレクトリ単位）
  - Input/SubInput1
  - Input/SubInput2/Source.swift
excludes: # 生成対象のStruct，Enumの例外を含んだファイルへのパス
  - Input/SubInput1/SubSubInput
  - Input/SubInput2/Source.swift
testables: # テスト対象のターゲット
  - target1
  - target2
output: output/Factories.generated.swift # 自動生成されたコードのパス
```

# Build Phases

テストの実行時にFactoryの自動生成スクリプトを呼び出す

```
"${PODS_ROOT}/FactoryProvider/generate" --config .factory.yml
# Factories.generated.swift is generated 🎉
```

# How FactoryProvider Works

2つのコードベースから成り立つ

- FactoryProviderのコード (Fixed)
- 自動生成されるコード (Generated in Build Phases)

# Providable (FactoryProvider)

Providableに準拠した型がFactoryでオブジェクトを取得できる

```
public protocol Providable {
    static func provide() -> Self
}
```

# Primitive Factory (FactoryProvider)

Primitiveな型のFactoryが定義されている (Int, Optional, String, ...)

```
extension Int: Providable {
    public static func provide() -> Int {
        return 0
    }
}

extension Optional: Providable where Wrapped: Providable {
    public static func provide() -> Optional {
        return .some(Wrapped.provide())
    }
}

...
```

# Factory (FactoryProvider)

```
struct Factory<T: Providable> {
    static func provide() -> T {
        return T.provide()
    }
}
```

# Specified Factory (Generated Code)

```
import FactoryProvider

extension User: Providable {
    static func provide() -> User {
        return User(
            id: Factory<UserId>.provide(),
            name: Factory<String>.provide(),
            age: Factory<Int>.provide()
        )
    }
}

extension UserId: Providable {
    static func provide() -> UserId {
        return UserId(
            value: Factory<String>.provide()
        )
    }
}
```

# Specified Factory (Generated Code)

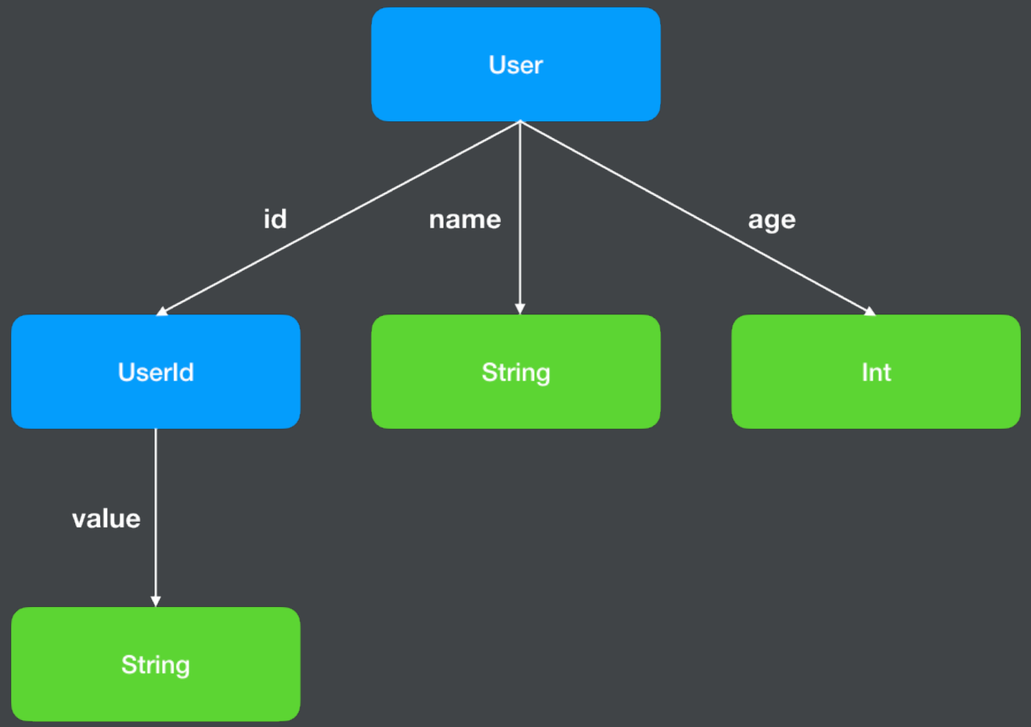
```
import FactoryProvider

extension User: Providable {
    static func provide() -> User {
        return User(
            id: Factory<UserId>.provide(),
            name: Factory<String>.provide(),
            age: Factory<Int>.provide()
        )
    }
}

extension UserId: Providable {
    static func provide() -> UserId {
        return UserId(
            value: Factory<String>.provide()
        )
    }
}
```

# Specified Factory (Generated Code)

葉がすべてPrimitiveな型になるまで木を伸ばす



# Lens (FactoryProvider)

```
public struct Lens<Whole, Part> {
    private let getter: (Whole) -> Part
    private let setter: (Part, Whole) -> Whole

    public init(getter: @escaping (Whole) -> Part, setter: @escaping (Part, Whole) -> Whole) {
        self.getter = getter
        self.setter = setter
    }

    public func get(_ from: Whole) -> Part {
        return getter(from)
    }

    public func set(_ from: Part, _ to: Whole) -> Whole {
        return setter(from, to)
    }
}
```

# Custom Operator (FactoryProvider)

```
infix operator *~: MultiplicationPrecedence
infix operator |>: AdditionPrecedence

public func * <A, B, C> (lhs: Lens<A, B>, rhs: Lens<B, C>) -> Lens<A, C> {
    return Lens<A, C>(
        getter: { a in
            rhs.get(lhs.get(a))
        },
        setter: { (c, a) in
            lhs.set(rhs.set(c, lhs.get(a)), a)
        }
    )
}

public func *~ <A, B> (lhs: Lens<A, B>, rhs: B) -> (A) -> A {
    return { a in
        lhs.set(rhs, a)
    }
}

public func |> <A, B> (x: A, f: (A) -> B) -> B {
    return f(x)
}

public func |> <A, B, C> (f: @escaping (A) -> B, g: @escaping (B) -> C) -> (A) -> C {
    return { g(f($0)) }
}
```

# Compose (FactoryProvider)

```
public func * <A, B, C> (lhs: Lens<A, B>, rhs: Lens<B, C>) -> Lens<A, C> {
    return Lens<A, C>(
        getter: { a in
            rhs.get(lhs.get(a))
        },
        setter: { (c, a) in
            lhs.set(rhs.set(c, lhs.get(a)), a)
        }
    )
}
```

# Set (FactoryProvider)

```
infix operator *~: MultiplicationPrecedence

public func *~ <A, B> (lhs: Lens<A, B>, rhs: B) -> (A) -> A {
    return { a in
        lhs.set(rhs, a)
    }
}
```

# Modify (FactoryProvider)

```
infix operator |>: AdditionPrecedence

public func |> <A, B> (x: A, f: (A) -> B) -> B {
    return f(x)
}

public func |> <A, B, C> (f: @escaping (A) -> B, g: @escaping (B) -> C) -> (A) -> C {
    return { g(f($0)) }
}
```

# Lens (Generated Code)

```
extension User {
    static var _name: Lens<User, String> {
        return Lens<User, String>(
            getter: { $0.name },
            setter: { name, base in
                User(name: name, age: base.age)
            }
        )
    }
    static var _age: Lens<User, Int> {
        return Lens<User, Int>(
            getter: { $0.age },
            setter: { age, base in
                User(name: base.name, age: age)
            }
        )
    }
}
```

# Decompose

```
let user = Factory<User>.provide() |> User._name *~ "Takeshi Ihara"
         = (User._name *~ "Takeshi Ihara")(Factory<User>.provide())
         = { user in
             Lens<User, String>(
                 getter: { $0.name },
                 setter: { name, base in
                     User(name: name, age: base.age)
                 }
             ).set("Takeshi Ihara", user)
           }(Factory<User>.provide())
         = { name, base in
              User(name: name, age: base.age)
           }("Takeshi Ihara", Factory<User>.provide())
         = User(name: "Takeshi Ihara", age: 0)
```

# Decompose

```
let user = Factory<User>.provide() |> User._name *~ "Takeshi Ihara"
         = (User._name *~ "Takeshi Ihara")(Factory<User>.provide())
         = { user in
             Lens<User, String>(
                 getter: { $0.name },
                 setter: { name, base in
                     User(name: name, age: base.age)
                 }
             ).set("Takeshi Ihara", user)
           }(Factory<User>.provide())
         = { name, base in
              User(name: name, age: base.age)
           }("Takeshi Ihara", Factory<User>.provide())
         = User(name: "Takeshi Ihara", age: 0)
```

# Decompose (|>)

```
let user = Factory<User>.provide() |> User._name *~ "Takeshi Ihara"
         = (User._name *~ "Takeshi Ihara")(Factory<User>.provide())
         = { user in
             Lens<User, String>(
                 getter: { $0.name },
                 setter: { name, base in
                     User(name: name, age: base.age)
                 }
             ).set("Takeshi Ihara", user)
           }(Factory<User>.provide())
         = { name, base in
              User(name: name, age: base.age)
           }("Takeshi Ihara", Factory<User>.provide())
         = User(name: "Takeshi Ihara", age: 0)
```

# Decompose (Lens, *~)

```
let user = Factory<User>.provide() |> User._name *~ "Takeshi Ihara"
         = (User._name *~ "Takeshi Ihara")(Factory<User>.provide())
         = { user in
             Lens<User, String>(
                 getter: { $0.name },
                 setter: { name, base in
                     User(name: name, age: base.age)
                 }
             ).set("Takeshi Ihara", user)
           }(Factory<User>.provide())
         = { name, base in
              User(name: name, age: base.age)
           }("Takeshi Ihara", Factory<User>.provide())
         = User(name: "Takeshi Ihara", age: 0)
```

# Decompose (Closure)

```
let user = Factory<User>.provide() |> User._name *~ "Takeshi Ihara"
         = (User._name *~ "Takeshi Ihara")(Factory<User>.provide())
         = { user in
             Lens<User, String>(
                 getter: { $0.name },
                 setter: { name, base in
                     User(name: name, age: base.age)
                 }
             ).set("Takeshi Ihara", user)
           }(Factory<User>.provide())
         = { name, base in
              User(name: name, age: base.age)
           }("Takeshi Ihara", Factory<User>.provide())
         = User(name: "Takeshi Ihara", age: 0)
```

# Decompose

```
let user = Factory<User>.provide() |> User._name *~ "Takeshi Ihara"
         = (User._name *~ "Takeshi Ihara")(Factory<User>.provide())
         = { user in
             Lens<User, String>(
                 getter: { $0.name },
                 setter: { name, base in
                     User(name: name, age: base.age)
                 }
             ).set("Takeshi Ihara", user)
           }(Factory<User>.provide())
         = { name, base in
              User(name: name, age: base.age)
           }("Takeshi Ihara", Factory<User>.provide())
         = User(name: "Takeshi Ihara", age: 0)
```

# Result

```
struct User {
    let name: String
    let age: Int

    var isAdult: Bool {
        return age >= 20
    }
}

import FactoryProvider

class UserTests: XCTestCase {
    func test20歳以上なら成人である() {
        let user = Factory<User>.provide() |> User._age *~ 20
        XCTAssertTrue(user.isAdult)
    }

    func test20歳未満なら成人でない() {
        let user = Factory<User>.provide() |> User._age *~ 19
        XCTAssertFalse(user.isAdult)
    }
}
```

# Demo

# Future Work (Protocol)

Not support protocol😭

```
protocol A {}


// Type 'A' does not conform to protocol 'Providable'
let a = Factory<A>.provide()
```

# Future Work (Protocol)

Extension of protocol cannot have an inheritance clause😱

```
// Extension of protocol 'A' cannot have an inheritance clause
extension A: Providable {
    public static func provide() -> Self {
        fatalError()
    }
}

let a = Factory<A>.provide()
```

# Future Work (Protocol)

Need concrete type🤔

```
protocol A: Providable {}

extension A {
    static func provide() -> Self {
        fatalError()
    }
}

// Using 'A' as a concrete type conforming to protocol 'Providable' is not supported
let a = Factory<A>.provide()
```

# Future Work (Protocol)

No problem using concrete type💚

```
protocol A: Providable {}

extension A {
    static func provide() -> Self {
        fatalError()
    }
}

struct B: A {}

let a = Factory<B>.provide()
```

# わいわいswiftc

興味を持った方はぜひ

# Future Work (interface)

User.provide()のように直接生成メソッドが呼べてしまう😱

```
extension User: Providable {
    static func provide() -> Self {
        return User(
            ...
        )
    }
}

let user = User.provide() or Factory<User>.provide()
```

# Future Work (interface)

**fileprivate**なGenericsを持つならそのStruct自身も**fileprivate**にしなくてはならない😱

```
fileprivate Protocol {
    static func provide() -> Self
}

fileprivate extension User: Providable {
    ...
}

// Generic struct must be declared private or fileprivate because its generic parameter uses a fileprivate type
struct Factory<T: Providable> {
    static func provide() -> T {
        return T.provide()
    }
}
```

# Future Work (interface)

## Factoryにすべて記述する🤔

```
struct Factory<T> {
    static func provide() -> T {
        switch T.self {
            case is User.Type:
                return User(
                    name: Factory<String>.provide(),
                    age: Factory<Int>.provide()
                )
            case is String.Type:
                return ""
            ...
            default:
                fatalError()
        }
    }
}
```

# Future Work (interface)

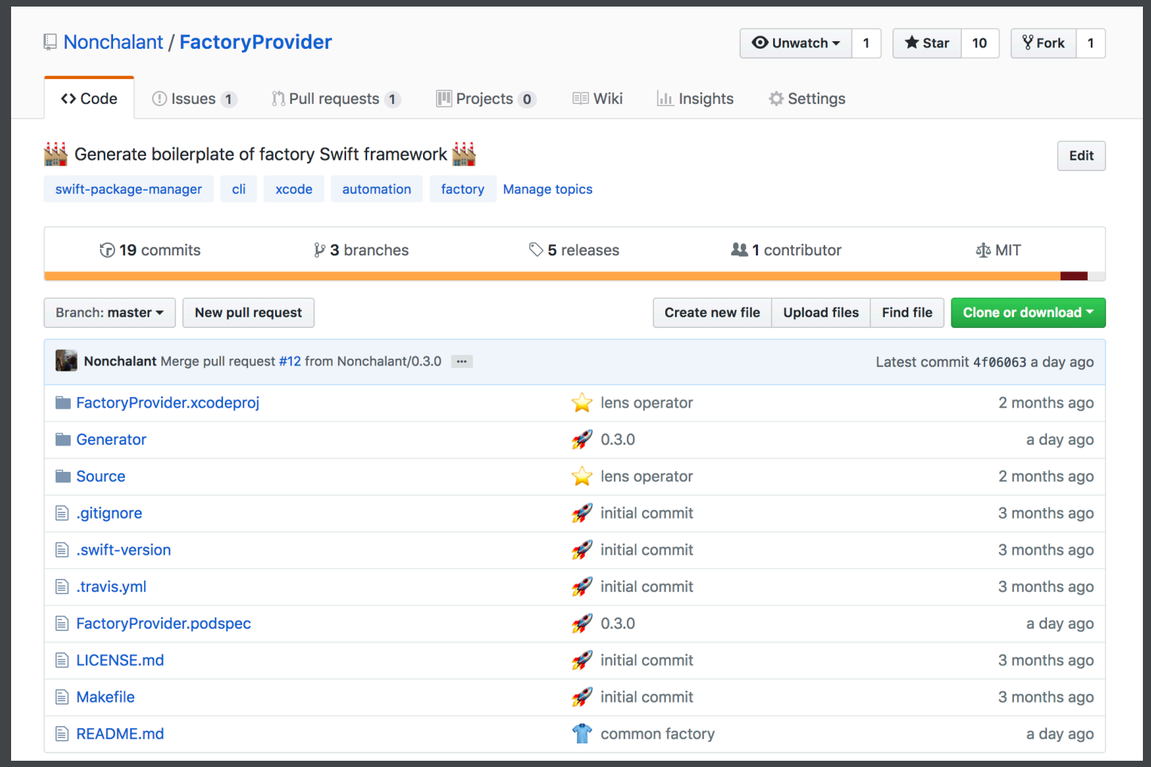
## `Inner Type`をサポートが困難😱 (パースが大変)

```
struct User {
    let id: Id
    struct Id {}
}

struct Factory<T> {
    static func provide() -> T {
        switch T.self {
            case is User.Type:
                return User(
                    id: Factory<Id>.provide() // Use of undeclared type 'Id'
                )
            ...
        }
    }
}
```

# FactoryProvider

https://github.com/Nonchalant/FactoryProvider



# Conclusion

- 「構造変化に強いテスト」が実現できた💚
- Factoryを用意する手間がなくなったのでテストに集中できる😍
- テスト対象のプロパティが分かりやすくなった✨
  - `Factory<User>.provide() |> User._age *~ 19`
- **SourceKitten + Stencil**を使ってパースして自動生成するの楽しい🚀