A glimpse into Protocols with Associated Type and type Erasure

Denis Poifol

Cocoaheads Lyon, le 17 Mai 2018

// FABERNOVEL
TECHNOLOGIES

Sommaire

01 //
Protocols with Associated Type (PAT)

Declaration and implementation

```
protocol Identifiable {
    associatedtype Identifier
    var id: Identifier { get }
}

struct IntIdentifiableStruct: Identifiable {
    typealias Identifier = Int
    let id: Int
    /* Other properties */
}

struct StringIdentifiableStruct: Identifiable {

    let id: String
    /* Other properties */
}
```

Extension

```
extension Equatable where Self: Identifiable, Self.Identifier: Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool {
        return lhs.id == rhs.id
    }
}
```

Extension

```
extension Equatable where Self: Identifiable, Self.Identifier: Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool {
        return lhs.id == rhs.id
    }
}

extension Hashable where Self: Identifiable, Self.Identifier: Hashable {
    var hashValue: Int { return id.hashValue }
}
```

Extension

```
extension Equatable where Self: Identifiable, Self.Identifier: Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool {
        return lhs.id == rhs.id
    }
}

extension Hashable where Self: Identifiable, Self.Identifier: Hashable {
    var hashValue: Int { return id.hashValue }
}

extension StringIdentifiableStruct: Hashable {}
```

Existential

```
let identifiable: Identifiable        Protocol 'Identifiable' can only be used
as a generic constraint because it has Self or associated type requirements
```

Protocol with Associated Types

- Default protocol functionalities
- Placeholder for one or multiple types
- No existential type
- A given type can conform to a PAT only in one way

Note: *Protocol with associated type are not generic protocols (which do not exist in swift)*

02 //
Type Wrapping

Configurable protocol

```
protocol Configurable {
    associatedtype Model
    func configure(with model: Model)
}
```

Configurable protocol

```
protocol Configurable {
    associatedtype Model
    func configure(with model: Model)
}

extension UILabel: Configurable {
    func configure(with model: String) {
        text = model
    }
}
```

Configurable protocol

```
protocol Configurable {
    associatedtype Model
    func configure(with model: Model)
}

extension UILabel: Configurable {
    func configure(with model: String) {
        text = model
    }
}

extension UIButton: ConfigurableView {
    func configure(with model: String) {
        setTitle(model, for: .normal)
    }
}
```

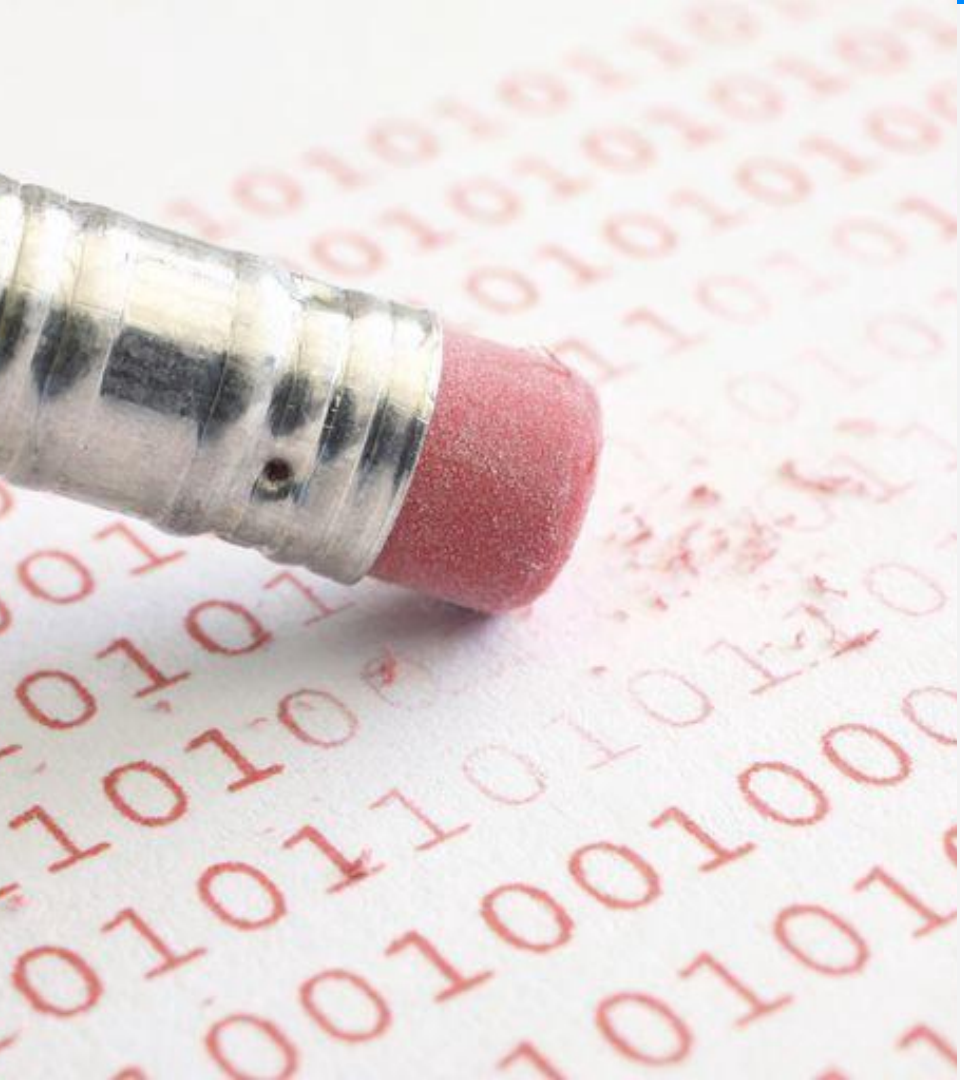
Wrap the implementing class inside of a generic type

```
struct ConfigurableWrapper<ConcreteClass: Configurable>: Configurable {
    typealias Model = ConcreteClass.Model
    private let wrappedInstance: ConcreteClass

    init(_ wrappedInstance: ConcreteClass) {
        self.wrappedInstance = wrappedInstance
    }

    func configure(with model: Model) {
        wrappedInstance.configure(with: model)
    }
}
```

Wrap the implementing class inside of a generic type

```
let label = UILabel()
let configurable = ConfigurableWrapper(label)

type(of: configurable)// ConfigurableWrapper<UILabel>
```

03 //
Type Erasure



Type Erasure

What do we want to achieve ?

Store object conforming to a PAT regardless of their actual type but defined by their associated type(s)

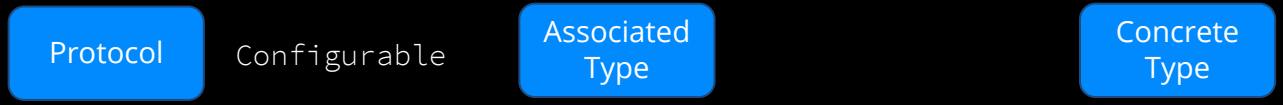
What do we want to achieve

```
protocol ViewContract {
    var successView: AnyConfigurable<String> { get }
    var goalView: AnyConfigurable<String> { get }
    var successImageView: AnyConfigurable<UIImage> { get }
}
```

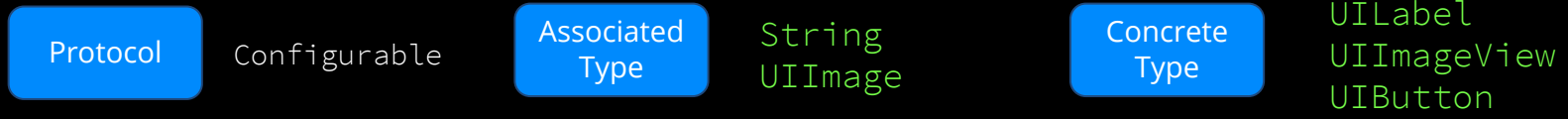
What do we want to achieve

```
protocol ViewContract {
    var successView: AnyConfigurable<String> { get }
    var goalView: AnyConfigurable<String> { get }
    var successImageView: AnyConfigurable<UIImage> { get }
}

var stringConfigurables: [AnyConfigurable<String>] = [
    successUIElement,
    goalUIElement,
]
let models = [
    "Success",
    "Present view to data layer wrapped as AnyConfigurable",
]
zip(stringConfigurable, models)
    .forEach { $0.0.configure(with: $0.1) }
```

Type Erasure Pattern in Standard Library



Type Erasure Pattern in Standard Library



Type Erasure Pattern in Standard Library

Protocol — `Configurable`

Associated Type — `String` `UIImage`

Concrete Type — `UILabel` `UIImageView` `UIButton`

Abstract Base

Private Box

Public Wrapper

Type Erasure Pattern in Standard Library

Protocol — `Configurable`

Associated Type — `String` `UIImage`

Concrete Type — `UILabel` `UIImageView` `UIButton`

Abstract Base

```
class _AnyConfigurableBase<ModelObject>: Configurable
```

Private Box

Public Wrapper

Type Erasure Pattern in Standard Library

Protocol — `Configurable`

Associated Type — `String` `UIImage`

Concrete Type — `UILabel` `UIImageView` `UIButton`

Abstract Base

```
class _AnyConfigurableBase<ModelObject>: Configurable
```

Private Box

```
class _AnyConfigurableBox<ActualType: Configurable>:_AnyConfigurableBase<ActualType.Model>
init(_ configurable: ActualType)
```

Public Wrapper

Type Erasure Pattern in Standard Library

Protocol — Configurable

Associated Type — String, UIImage

Concrete Type — UILabel, UIImageView, UIButton

Abstract Base

```
class _AnyConfigurableBase<ModelObject>: Configurable
```

Private Box

```
class _AnyConfigurableBox<ActualType: Configurable>:_AnyConfigurableBase<ActualType.Model>
init(_ configurable: ActualType)
```

Public Wrapper

```
final class AnyConfigurable<Model>: Configurable
init<Concrete: Configurable>(_ concrete: Concrete) where Concrete.Model == Model
```

Type Erasure Pattern in Standard Library

Protocol — Configurable

Associated Type — String, UIImage

Concrete Type — UILabel, UIImageView, UIButton

Abstract Base

```
class _AnyConfigurableBase<ModelObject>: Configurable
```

Private Box

```
class _AnyConfigurableBox<ActualType: Configurable>:_AnyConfigurableBase<ActualType.Model>
init(_ configurable: ActualType)
```

Public Wrapper

```
final class AnyConfigurable<Model>: Configurable
init<Concrete: Configurable>(_ concrete: Concrete) where Concrete.Model == Model
```

Type Erasure Pattern in Standard Library

Protocol — Configurable

Associated Type — String, UIImage

Concrete Type — UILabel, UIImageView, UIButton

Abstract Base

```
class _AnyConfigurableBase<ModelObject>: Configurable
```

Private Box

```
class _AnyConfigurableBox<ActualType: Configurable>:_AnyConfigurableBase<ActualType.Model>
init(_ configurable: ActualType)
```

Public Wrapper

```
final class AnyConfigurable<Model>: Configurable
init<Concrete: Configurable>(_ concrete: Concrete) where Concrete.Model == Model
```

Implementing the abstract base class

```
class _AnyConfigurableBase<ModelObject>: Configurable {

    typealias Model = ModelObject









}
```

Implementing the abstract base class

```
class _AnyConfigurableBase<ModelObject>: Configurable {

    typealias Model = ModelObject

    init() {
        guard type(of: self) != _AnyConfigurableBase.self else {
            fatalError("This class cannot be implemented")
        }
    }

    func configure(with model: ModelObject) {
        fatalError("Must be overiden")
    }
}
```

Create an amphibological box class

```
class _AnyConfigurableBox<ActualType: Configurable>:
    _AnyConfigurableBase<ActualType.Model> {

    private let configurable: ActualType

    init(_ configurable: ActualType) {
        self.configurable = configurable
    }

    override func configure(with model: Model) {
        configurable.configure(with: model)
    }
}
```

Implementing the final wrapper

```
final class AnyConfigurable<Model>: Configurable {
    private let box: _AnyConfigurableBase<Model>

    init<Concrete: Configurable>(_ concrete: Concrete)
        where Concrete.Model == Model {
        box = _AnyConfigurableBox(concrete)
    }

    func configure(with model: Model) {
        box.configure(with: model)
    }
}
```

What do we want to achieve

```
protocol ViewContract {
    var successView: AnyConfigurable<String> { get }
    var goalView: AnyConfigurable<String> { get }
    var successImageView: AnyConfigurable<UIImage> { get }
}

var stringConfigurables: [AnyConfigurable<String>] = [
    successUIElement,
    goalUIElement,
]
let models = [
    "Success",
    "Present view to data layer wrapped as AnyConfigurable",
]
zip(stringConfigurable, models)
    .forEach { $0.0.configure(with: $0.1) }
```

Implementing ViewContract

```
class ViewController: UIViewController, ViewContract {
    private(set) lazy var successView = AnyConfigurable(label)
    private(set) lazy var goalView = AnyConfigurable(button)
    private(set) lazy var successImageView = AnyConfigurable(imageView)

    private var label = UILabel()
    private var button = UIButton()
    private var imageView = UIImageView()

     // ViewController implementation

}
```

Read More

Great article from bigNerdRanch that inspired this talk
https://www.bignerdranch.com/blog/breaking-down-type-erasures-in-swift/

Swift Generic Manifesto
https://github.com/apple/swift/blob/master/docs/GenericsManifesto.md

My github with a playground related to this talk
https://github.com/denisPoifol/Talks/

Merci.

// FABERNOVEL TECHNOLOGIES

Denis Poifol

Junior Software Engineer

+33 6 58 90 85 54

denis.poifol@fabernovel.com