Библиотека KDDD

Содержание

KDDD (Kotlin DDD) — библиотека базовых типов, предназначенная для построения моделей предметных областей на языке Котлин согласно с тактическими *DDD*-патернами — *Entity* и *Value Object*. Состоит из закрытой **ФСТ**, аннотаций и регламента построения **CDT**.

Мотивация

> В сложной иерархически организованной системе рост разнообразия на верхнем уровне обеспечивается ограничением разнообразия на предыдущих уровнях, и наоборот, рост разнообразия на нижнем уровне разрушает верхний уровень организации (то есть, система как таковая гибнет).
>
> — Закон Е. А. Седова

Котлин является языком программирования общего назначения, предоставляющий очень широкие возможности для разработчика. В частности, — для проектирования собственных структур данных (типов). К сожалению, такая широта зачастую оборачивается появлением порочных практик, таких как мутабельность публичных свойств, одержимость примитивами (*primitive obsession*), нарушениями принципов *SOLID* и пр.

При проектировании **МПО** необходимо создать собственную **ФСТ** с обусловленными бизнес-требованиями правилами и ограничениями, возможностью валидации объектов при их создании, покрытию юнит-тестами логики **МПО**. В свою очередь такая **ФСТ** должна создаваться в рамках аксиоматики базовой **ФСТ** высшего уровня, как основы, и которая бы воплощала концепцию тактических *DDD*-патернов — *Entity* и *Value Object*.

Ограничения должны применяться на самых ранних этапах. Отчасти это возможно решить на уровне компиляции путем создания контрактов (абстрактых методов) в базовых интерфейсах и проверкой их логики юнит-тестами. Отчасти — наличием регламента, задающим эти ограничения. Проверка соблюдения регламента может осуществляться за счет дополнительных инструментов автоматизации: валидаторы кода, линтеры, кодогенераторы.

Цель

- Создание закрытой базовой **ФСТ**, воплощающую концепцию тактических *DDD*-патернов.
- Обеспечение механизма валидации **CDT** при создании объектов.
- Создание регламента проектирования **CDT**.

Базовая ФСТ

Интерфейсы **Kddd** наследуются от корневого sealed-интерфейса `Kddd` который имеет метод `validate()` предназначенный для валидации объектов при их создании. В переопределенных методах `validate()` должна реализовываться логика валидации объекта и выкидываться исключение `IllegalStateException` в рамках этой логики.

`IEntity : Kddd`

Определяет тактический *DDD*-патерн *Entity*. Содержит поля:

- `id` — идентичность сущности.
- `content` — *Value Object* содержимого сущности.

Регламент требует, что реализация этого интерфейса должна переопределять контракт `hashCode()/equals()`, завязав их на поле `id`. Отдельная группировка свойств сущности в свойство `content` обусловлена подходом, обоснованным у Вдадимира Хорикова.

`ValueObject : Kddd`

Определяет тактический *DDD*-патерн *Value Object*. Sealed-интерфейс со следующими наследниками:

`ValueObject.Data`

Предназначен для проектирования **CDT** с несколькими свойствами и последующей имплементации **CDT** в виде `data class`. Декларирует метод `<T: Kddd, A: Kddd> fork(vararg args: A): T` для возможности копирования объекта этого типа и аналогичен методу `copy()` у `data class`. Реализация этого метода описана в **Регламенте**.

`ValueObject.Value<BOXED: Any>`

Предназначен для проектирования **CDT** с одним свойством и последующей имплементации **CDT** в виде `value class`. Тип свойства определяется параметризованным типом `BOXED` и

может быть либо **ПТ**, либо **ТОН**. Обусловлено борьбой с *одержимостью примитивами*. Декларирует метод `<T : Value<BOXED>> fork(boxed: BOXED): T` для возможности копирования объекта этого типа и аналогичен методу `copy()` у `data class`. Реализация этого метода описана в **Регламенте**.

Метод `fork()` в контрактах интерфейсов `ValueObject` призван обеспечить возможность создания функционала/операций над этими типами на этапе декларации интерфейсов и, соответственно, компиляции, что обеспечивает возможность сразу создавать и запускать юнит-тесты с соответств прежде, чем будут .

Аннотации

public annotation class KDGeneratable(val implementationName: String = "", val dsl: Boolean = true, val json: Boolean = false,)

Регламент

IMPORTANT	Слова «**MUST**», «**MUST NOT**», «**REQUIRED**», «**SHALL/SHOULD**», «**MAY**» и «**OPTIONAL**» в нижеследующих списках понимаются как определено в RFC 2119.

CDT может быть реализован двумя способами:

1. Опосредовано через интерфейс-наследник **Kddd** и последующей имплементацией в классе.
2. Непосредственно как класс-наследник **Kddd**.

Первый вариант крайне рекомендуется, т.к. разделение абстракции и имплементации удобно своей гибкостью, использованием *Dependency Injection*, юнит-тестированием и, самое главное, — возможностью кодогенерации имплементации.

Интерфейс CDT

Тип **CDT** (`Cdt`):

1. Должен быть (**MUST**) `interface`.
2. Должен (**MUST**) наследоваться от соответствующего подтипа **Kddd**.
3. Свойства не должны быть (**MUST NOT**) мутабельными, быть: `val`. Мутабельность осуществляется через метод `fork()`.
4. Свойства могут быть (**MAY**) нуллабельными.
5. Если родительский тип `ValueObject.Data`:
    1. Свойства должны быть (**MUST**) типом **CDT**, либо коллекциями (`Set`, `List`, `Map`) [TODO: еще и `enum`].
    2. Параметризованные типы коллекций должны быть (**MUST**) типом **CDT**, либо

коллекциями с параметризованными типами. Таким образом возможна вложенность коллекций, например: `List<Map<Cdt, Set<Cdt>>` и т.д.

3. Типы свойств могут быть (**MAY**) определены в отдельном **CDT**, либо внутри данного типа (*nested*).

6. Если родительский тип `ValueObject.Value<BOXED : Any>`:
    1. Параметризованный тип должен быть (**MUST**) либо **ПТ**, либо **ТОН**.
7. Должен (**SHOULD**) переопределять метод `validate()`, который будет вызываться перед созданием объекта. В нем пишется логика проверки валидности свойств и параметров и выкидывается `IllegalStateException` в случае непрохождения проверки. Может быть (**MAY**) пустым, если логика валидации не задана.
8. Может (**MAY**) содержать методы, декларирующие/реализующие функционал модели.
9. Может (**OPTIONAL**) предваряться `KDDD`-аннотациями.

Имплементация CDT

Тип **CDT** (`CdtImpl`):

1. Должен быть (**MUST**) классом-наследником типов **Kddd** прямо или опосредовано через интерфейс **CDT**.
2. Должен иметь (**MUST**) приватный конструктор. Объект класса создается через билдер.
3. Должен (**MUST**) вызывать метод `validate()` внутри конструктора `init`.
4. Если родительский тип `ValueObject.Data`:
    1. Должен быть (**MUST**) `data class`.
    2. Должен иметь (**MUST**) сопутствующий класс `Builder`, реализующий паттерн *Строитель* (С.м. пример ниже).
    3. Должен иметь (**MUST**) метод `toBuilder()`, создающий и возвращающий объект `Builder`.
    4. Должен (**MUST**) переопределять метод `fork()` (С.м. пример ниже).

    ```
    @Suppress("UNCHECKED_CAST")
    override fun <T : Kddd, A : Kddd> fork(vararg args: A): T =
        Builder().apply {
            // инициализация свойств билдера из текущего объекта
        }.build() as T
    ```

5. Если родительский тип `ValueObject.Value`:
    1. Должен быть (**MUST**) `value class`
    2. Свойство `boxed` должно быть (**MUST**) либо **ПТ**, либо **ТОН**.
        1. Если свойство `boxed` является **ТОН**, то должен иметь (**MUST**) метод `parse()` в `companion object`, который десериализует объект **ТОН**:

```
public companion object {
    public fun parse(src: String): CdtImpl =
        CdtImpl(/* Создать объект 'TOH' из строки `src` */)
```

3. Должен иметь (**MUST**) реализацию билдера в виде оператора `invoke()` в `companion object`:

```
public companion object {
    public operator fun invoke(boxed: BoxedType): Cdt = CdtImpl(boxed)
}
```

4. Должен (**MUST**) переопределять метод `fork()`:

```
@Suppress("UNCHECKED_CAST")
override fun <T : ValueObject.Value<BoxedType>> fork(boxed: BoxedType): T =
CdtImpl(boxed) as T
```

6. Должен (**SHOULD**) переопределять метод `validate()`, который будет вызываться перед созданием объекта. В нем пишется логика проверки валидности свойств и параметров и выкидывается IllegalStateException в случае непрохождения проверки. Может быть (**MAY**) пустым, если логика валидации не задана.

Примеры

Пример спроектированного **CDT** *для моделирования точки с двумя координатами.*

```
public interface Point : ValueObject.Data {
    public val x: Coordinate
    public val y: Coordinate

    override fun validate() {
        // Здесь можно задать границы модели и провалидировать консистентность
свойств.
        check(x.boxed in 0..1000)
        check(y.boxed in 0..1000)
    }

    public operator fun plus(other: Point): Point =
        fork(x + other.x, y + other.y)

    public operator fun minus(other: Point): Point =
        fork(x - other.x, y - other.y)

    public operator fun times(other: Point): Point =
        fork(x * other.x, y * other.y)
```

```
    public interface Coordinate : ValueObject.Value<Int> {
        override fun validate() {}

        public operator fun plus(other: Coordinate): Coordinate =
            fork(boxed + other.boxed)

        public operator fun minus(other: Coordinate): Coordinate =
            fork(boxed - other.boxed)

        public operator fun times(other: Coordinate): Coordinate =
            fork(boxed * other.boxed)
    }
}
```

Пример имплементации **CDT**.

```
@ConsistentCopyVisibility
public data class PointImpl private constructor(
    override val x: Point.Coordinate,
    override val y: Point.Coordinate
) : Point {
    init {
        validate()
    }

    @Suppress("UNCHECKED_CAST")
    override fun <T : Kddd, A : Kddd> fork(vararg args: A): T {
        val ret = Builder().apply {
            x = args[0] as Point.Coordinate
            y = args[1] as Point.Coordinate
        }.build() as T
        return ret
    }

  public fun Point.toBuilder(): PointImpl.Builder {
    val ret = PointImpl.Builder()
    ret.x = x
    ret.y = y
    return ret
  }

  @JvmInline
  public value class CoordinateImpl private constructor(
    override val boxed: Int,
  ) : Point.Coordinate {
    init {
      validate()
    }
    override fun toString(): String = boxed.toString()
```

```
    @Suppress("UNCHECKED_CAST")
    override fun <T : ValueObject.Boxed<Int>> fork(boxed: Int): T =
CoordinateImpl(boxed) as T

    public companion object {
      public operator fun invoke(boxed: Int): Point.Coordinate = CoordinateImpl(boxed)
    }
  }

  public class Builder {
    public lateinit var x: Point.Coordinate

    public lateinit var y: Point.Coordinate

    public fun build(): PointImpl {
      require(::x.isInitialized) { "Property 'PointImpl.x' is not set!" }
      require(::y.isInitialized) { "Property 'PointImpl.y' is not set!" }
      return PointImpl(x = x,y = y,)
    }
  }
}
```

Также в этом интерфейсе можно определить, например, метод расчета расстояния до другой точки или вынести такой функционал этой модели в функцию-расширение как *use case* и покрыть юнит-тестом.

Термины и определения

МПО

Модель Предметной Области (Domain Model) — совокупность типов данных и их функционала. Определение по М. Фаулеру.

Пользователь

Разработчик, использующий данную библиотеку для проектирования собственных типов (**CDT**) для некоторого своего домена.

ФСТ

Формальная система типов, построенная на заданной аксиоматике — постулатах, определяющих допустимые границы значений и операции над типами.

ПТ

Примитивный тип Котлин: `String`, `Int`, `Boolean`, и т.д.

ТОН

Тип общего назначения из стандартных пакетов Java и Котлин, не требующих подключения специальных зависимостей: `File`, `UUID`, `URI`, и т.д.

Kddd

Корневой тип библиотеки `KDDD`.

CDT

Customer Domain Type — проектируемый **Пользователем** собственный тип структуры данных.