-->

Milestone 2 (M2) - Domain Primitives, and Testing

The main objective of this milestone is to replace the standard base types (`String` , `Float` , ...) for concepts like "money" and "email address" with proper **domain primitives**. This is a very common pattern in Domain-Driven Design (DDD). Please watch the Domain Primitive video for the properties of domain primitives. Those domain primitives will provide:

- instantiation via a factory method, with built-in parameter validation, so that only *valid* objects will be instantiated
- basic operations (e.g. `addTo` , `subtractFrom` , and `multiplyBy` for money)
- some complex, domain-specific operations

E1: Implement Domain Primitives Basic Methods and Factories, According to Given Interfaces

In package `thkoeln.archilab.ecommerce.usecases.domainprimitivetypes` you find several interfaces that describe domain primitives (marked in yellow in the diagram below). Your task is to create an implementing class (marked in green) for each interface.

These classes will encapsulate simple, single-value types such as a postal code or email address, but also multi-property types like physical address or money. The more functionality you can move down to this "base layer", the more compact and "uncluttered" your code will be. Implement all given domain primitive interfaces. Your implementations must be located in package `thkoeln.archilab.ecommerce.domainprimitives` .

In addition to the methods required by the interface, each of your domain primitive implementations **must** have a static factory method named `of(...)` for creating instances of the primitive. Unfortunately, Java interfaces cannot contain static methods. Therefore, the parameters and validation rules for the factory method are specified **as a Javadoc comment at the end of each interface**.

There are a number of general do's and don'ts that you should follow when implementing value objects (domain primitives are always value objects). More details can be found in the ArchiLab Anti-Patterns bei "DDD mit Spring JPA" infopage. Amongst the most relevant ones (which we will check) are:

- Value Objects are immutable, i.e. never changed
- They are considered equal if all their attributes are equal
- Validation is checked at instantiation time, so there will never be an invalid instance

- Domain primitives will NEVER reference entities, because they are intended to be "base types" that you can use everywhere in your code. With entity references, you would introduce a cyclic dependency.

E2: Write Unit Tests for Selected Domain Primitive Methods

You need to write own unit tests for the following methods in your domain primitives, altogether at least 6 tests:

- `EmailAddressType.sameIdentifyerDifferentDomain(...)`
- `EmailAddressType.sameDomainDifferentIdentifyer(...)`
- `PostalCodeType.distance(...)`

Place your test classes in package `thkoeln.archilab.ecommerce.test.domainprimitives.owntests` (this is already there, as an empty package). Please make sure that you also cover edge and out-of-bounds cases. See the ArchiLab video on Unit Testing 101 for more information on how to write suitable tests.

There will be additional acceptance tests in place, which will fail in case the functionalities are improperly implemented. HThere are also tests that check if your own tests are there, and test the right methods.

NOTE: You need to use JUnit5 (imports must be `org.junit.jupiter.api.Test`).

E3: Update Your Code to Use Domain Primitives

You will notice that the interfaces in `thkoeln.archilab.ecommerce.usecases` have changed, and now use the domain primitives (as descriptive interfaces). You need to adapt your code to use these domain primitives. Please use them in your entity and service classes, where they fit. It is advisable to use **adapter services** as an anti-corruption layer, so that you can use *your own* domain primitive classes in your code, instead of the domain primitive interfaces (where you have no control over, and which won't work as `@Embedded` references in JPA). This is described in the ArchiLab infopage "Adapter-Pattern (a.k.a. Anti-Corruption-Layer)".

When you update your code, the task description and the rules (your own packages and classes located below `thkoeln.archilab.ecommerce.solution`) remain the same as in M1.