Developer Toolkit

Thank you for downloading my Developer Toolkit. This is just the starting point of your training journey. At the end of it, you will become a top software developer. I promise you.

The Toolkit is your warm up to get in touch with some of the core practices that I use every day in my professional and personal software development projects.

This is part of the knowledge that allowed me to talk at international conferences and become a professional Software Developer first, and a Director of Engineering later in the greatest European startup hub, namely the exciting city of Berlin.

I'm happy to share all my knowledge with you. Feel free to get in touch through my Twitter or LinkedIn profile if you have any doubt or something is not clear enough. If you prefer, drop me an email at ask@nicolopignatelli.me.

Don't forget to follow me on Medium for my latest posts on Software Development. You can also subscribe to my newsletter to be one of the first readers when I publish something and never miss an article.

Please feel free to share this toolkit, if you like. Sharing is caring :-)

How to use this toolkit

The toolkit is a list of definitions optionally followed by code samples.
Study them. Internalize them in your development style and apply them constantly. You will see a dramatic increase in the quality of your code. I promise.

Go through the definitions multiple times, take time to understand them and make sure to try them out by writing some code by yourself.

Consult the toolkit frequently, if you need, whenever you are coding. It will help you internalize the concepts faster and let you focus on your actual coding tasks.

Allow yourself some days of practice, improvements will be visible soon.

Please also remind that this toolkit is a quick reference for complex concepts. As such, it doesn't aim to exhaustively cover every bit of knowledge around the presented contents.

Enough with explanations, now. Let's get started :-)

Toolkit

Value Object

Consider a VO as an atomic unit of your code. Avoid playing with primitives and wrap them in a VO before any manipulation.
A VO has three fundamental characteristics:

- Immutability
- By value equality

- Self-consistency

Don't add getters to your VOs by default. There are good chances you are not going to need them.

A VO is immutable. It means that you cannot change its internal value(s) after creating it. No setters allowed.
This brings two advantages:

- you can freely share any VO by reference without any worry that it will be modified in another part of the code;
- it improves the semantics of your methods when you want to manipulate the VO, create a new instance or extract data out of it.

Two VOs with the same values are indistinguishable one from the other. This means you can test for equality by checking their internal values.
Hint: in most OOP languages, you can access the private properties of another object of the same class.

A VO must check for the consistency of its values in the constructor. If a value is invalid in the VO context, throw a meaningful exception.

@Java example

```
final class Locale {
   private String language;
   private String country;

   public static Locale fromString(String localeString) {
      Pattern p = Pattern.compile("^([a-z]{2})_([A-Z]{2})$");
      Matcher m = p.matcher(localeString);

      if (m.find()) {
         String language = m.group(1);
         String country = m.group(2);

         return new Locale(language, country);
      }

      throw new InvalidLocaleString(localeString);
   }

   public Locale(String language, String country) {
      if (!language.matches("^[a-z]{2}$")) {
         throw new InvalidLanguage(language);
      }

      if (!country.matches("^[A-Z]{2}$")) {
         throw new InvalidCountry(country);
      }

      this.language = language;
      this.country = country;
   }
```

```
    public boolean sameAs(Locale anotherLocale) {
        return language == anotherLocale.language &&
                country == anotherLocale.country;
    }

    public String toString() {
        return String.format("%s_%s", language, country);
    }
}
```

Entity and Aggregate

Unlike Value Objects…

- Entities are objects you can **univocally identify** by an identity property. Typically, this property is a Universal Unique Identifier (UUID) injected in the Entity constructor as first parameter. Wrapping the UUID in a Value Object is a good practice for comparison and explicit typing purposes.
- Their state optionally mutates according to their internal rules when you call a public method on them. Defining plain property setters on an Entity class is a bad practice.
- They have a lifecycle that represent their status in the reference domain.

An Aggregate is the encapsulation and manipulation unit that is managed by a Service. Typically, an Aggregate is an Entity with Value Objects as its properties and available domain operations as public methods.

@Java example

```
import java.util.UUID;

final class Customer {
    private CustomerId customerId;
    private Name name;
    private AccountStatus accountStatus;

    public static Customer createAnonymous() {
        return new Customer(
                new CustomerId(UUID.generateRandom()),
                new Name("Anonymous"),
                new AccountStatus("Active")
        )
    }

    public Customer(CustomerId customerId, Name name, AccountStatus accountStatus) {
        this.customerId = customerId;
        this.name = name;
        this.accountStatus = accountStatus;
    }

    public void deanonimize(Name name) {
        if (!isAnonymous()) {
            throw new CustomerAlreadyDeanonymized();
        }
```

```
        this.name = name;
    }

    public void deactivate() {
        if (!isActive()) {
            throw new AccountAlreadyDeactivated();
        }

        this.accountStatus = new Status("Deactivated");
    }

    private boolean isAnonymous() {
        return name.sameAs(new Name("Anonymous"));
    }

    private boolean isActive() {
        return accountStatus.sameAs(new Status("Active"));
    }
}
```

Service (aka Aggregate Root)

A Service is the point of contact for clients making requests to your Aggregates.
Its first, most important responsibility is to protect invariants around your Aggregates.
When a request comes in, a Service does the following:

1. it translates request parameters to objects belonging to the Aggregate domain;
2. it loads one or more Aggregates from their Repositories, or create a new one;
3. it checks whether the request is valid in regard to the business invariants;
4. it calls the corresponding Aggregate method to execute the domain logic requested by the client;
5. in case the business logic is successfully executed, it saves the Aggregate back to the Repository;
6. it returns to the client a Success or a Failure response that optionally wraps domain values or any occurred errors.

```
@Java example
final class Service {
    private Articles articles;
    private Publisher publisher;

    public Service(Articles articles, Publisher publisher) {
        this.articles = articles;
        this.publisher = publisher;
    }

    public Result createArticle(UUID articleUuid, String title) {
        try {
            articleId = new ArticleId(articleUuid);
            articleTitle = new Title(title);

            if (this.articles.existsOneWithTitle(articleTitle)) {
```

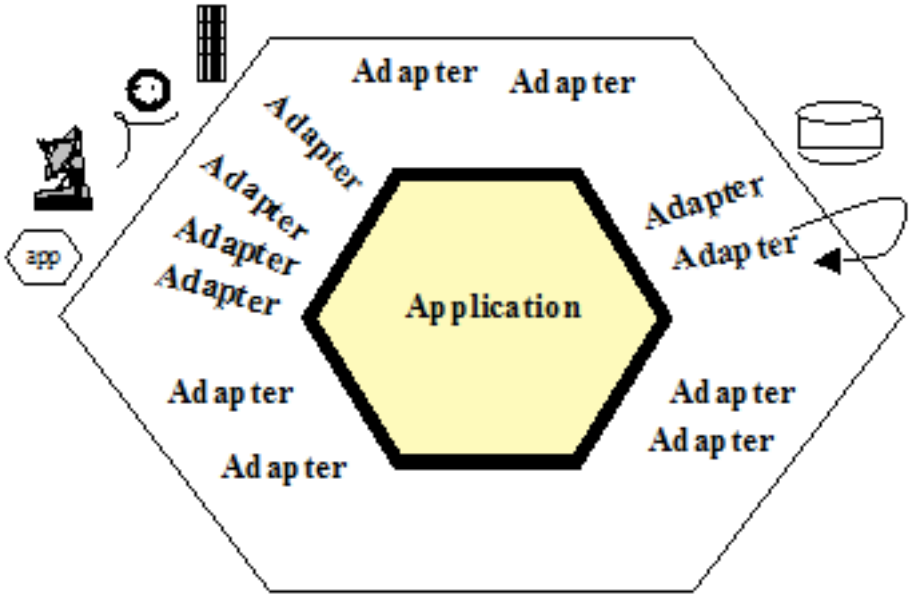
```
            throw new ArticleWithSameTitleAlreadyExists(articleTitle);
        }

        article = new Article(articleUuid, title);
        this.articles.save(article);

        this.publisher.publish(
            new ArticleWasCreated(article)
        );

        return new Success(articleId);
      } catch (Exception ex) {
        return new Failure(ex);
      }
    }
}
```

Hexagonal architecture (aka Ports and Adapters)



It's an application design pattern. It moves the focus from a classic layered architecture to an inside / outside one.
The inside part is your business logic, while the outside one includes any client that accesses the business logic and any storage or messaging system.

When applied, your business logic:

- doesn't know anything about any outside part;
- doesn't depend on the specific storage implementations of your Aggregates;
- can be tested in isolation;
- is all contained in the inside part; no business logic goes outside of this core;

Publish/Subscribe

It's a software messaging pattern where the Publisher of a message doesn't intend the message to be received by a specific recipient (aka Subscriber). The only responsibility of a Publisher is to categorize the message by one or more specific classes the Subscribers are aware of.

In turn, the Subscribers express interest in one or more classes of messages. When a message of a specific class is published, all the interested Subscribers will receive the message.

Typically, there is a middleware that takes care of routing a message from the Publisher to the Subscribers, so that they don't have to directly know and talk to each other.

This pattern is useful when integrating different Services. Services are the encapsulation point for your Aggregates, so they offer a natural boundary for defining an isolated unit of software that has no external dependencies.

For two isolated Services to react to each others' changes, the only way is through an agnostic Publish/Subscribe pattern. The only point of contact is the class and data structure of the exchanged messages.

Composition, interfaces and final implementations

Do not use inheritance to extend the behavior of your classes. Use composition instead.
Only inject class dependencies through the constructor. No setters allowed.
Only inject interfaces as class dependencies. No concrete implementations allowed.
If you feel you're injecting too many dependencies in a class, rethink your design in terms of class responsibilities and the interface segregation principle.
Use a Dependency Injection Container (or an equivalent object building mechanism) to manage the construction of your objects.
Using these rules will dramatically improve the quality of your code by making it decoupled, testable and extendable without any unwanted side-effect.

Please refer to this post for an in-deep explanation.

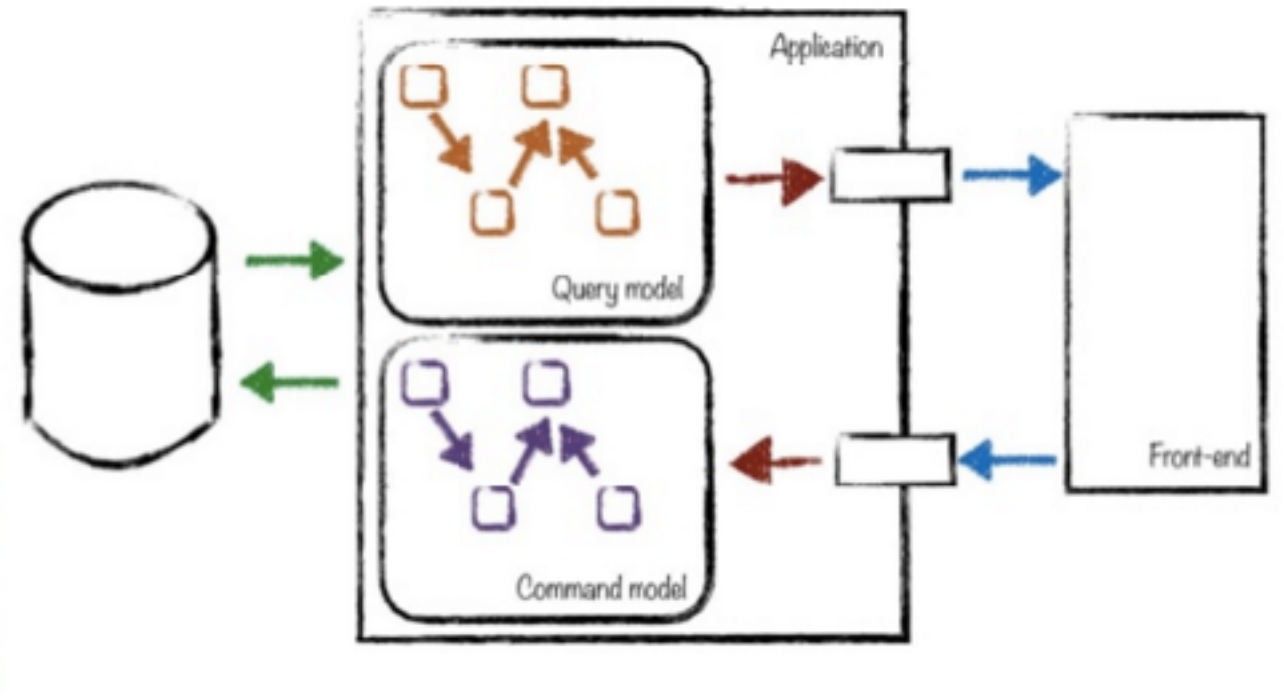
Testing

There are many testing strategies in software development, but I find the following more effective for a balance between robust and lean development:

- test your Services first, without mocking anything but their point of communication with the external world (e.g. the Publisher); this will give enough guarantees about the Service functionality without sacrificing future refactoring speed;
- test your outside clients in isolation, without reaching out to the inner Services;
- test your whole system from the UI down with an automated test suite; avoid manual clicking around, which makes very difficult to catch regressions.
- unit test a method if you are not sure enough about its behavior;
- I also suggest to add unit tests after each bug found, to prevent future regressions

There's a lot of talk around full unit code coverage and test driven development. I don't think they are beneficial in every situation. Your code is unique and you have to find the right balance between test coverage and code fluidity. Anyway, be aware this is **not** an excuse to leave your code untested.

CQRS



It's an acronym for *Command Query Responsibility Segregation*.
In simple words, it means you have at least two representations of your domain objects, with different purposes and interfaces:

- one for changing the state of your system; it's where your business rules and invariants are, it's the natural home to Services and Aggregates;
- one or more for getting information out of your system, without requesting any change.

When you need to display information to your users, do not add any getter to your Aggregates and expose them. Instead, for each of your display use cases, load the information from the storage into ad-hoc classes with getters only.

Applying CQRS dramatically simplifies the design and implementation of your domain classes. It decouples two very different responsibilities (reading and writing) in two or more models.

What's next?

The patterns and techniques I presented in this Developer Toolkit are really only the tip of the iceberg, but if you start applying them constantly you will see the benefit very soon.

I left hereunder a reference for you to read further about some of the concepts I showed you in this Toolkit. There's a lot of information to go through, but if you read the contents of the Toolkit things will be easier to categorize and understand.

Once again, feel free to throw some questions at me at ask@nicolopignatelli.me. Let's also connect through my Twitter or LinkedIn profile.

Talk to you soon!

Nicolò Pignatelli

References

- Value Object
- What is the difference between Entities and Value Object?
- Aggregates, Entities and Value Object in Domain Driven Design
- Hexagonal architecture
- Publish/Subscribe pattern
- CQRS