OCP in Practice

Dipl.-Inf. Manfred Wolff, Senior Technical Consultant at neusta software development

Abstract

Even simple implementations tends to become complex during the project life cycle. Look at this simple use case: “As a security administrator I want to have secure passwords avoiding hacking my system.” As acceptance criteria there might be several rules that password has to fulfil e.g.

- The password has to be a length of minimum 12 characters.
- There should be at least one capital letter in the password.
- There should be at least one digit in the password
- and so on

Of course the requirements grows up in a project so in the first user story there might be only two rules and during the project there are more stories that puts more rules in place. The Open-Close-Principle mentioned: *“Classes should be open for extension but closed for modification”*. How to handle? In this article I provide a simple mechanism called “simple-command-framework” where you can implement such problems in a convenient way. I’m going to provide a way (a refactoring rule) how to refactor from a common implementation to the simple-command framework.

An old idea – <devide an imperare>

Cutting software in little pieces is an old idea. There are design patterns as command-, chain of responsibility- or strategy pattern which follows the same idea. What is so important on these frameworks?

- A complex problem is split into easy or complicated problems. Each piece does a simple thing.
- The simple little peace can be easily tested.
- You can provide a context passing around all pieces to get some glue to fit them together.

The simple Idea is to split each call in a method to an object (encapsulating algorithms in objects). Look to a potential first quick implementation of our use case with the first acceptance criteria below:

```java
public class LoginValidator {

        private String passwd;
        private String errorMessage = "";

        public LoginValidator(final String passwd) {
                this.passwd = passwd;
        }

        public boolean validateLength() {

                boolean result = true;
                if (passwd.length() < 12) {
                        errorMessage =
                                "The password has to have at minimum 12 characters";
                        result = false;
                }
                return result;
        }

        public String getErrorMessage() {
                return errorMessage;
        }
}
```

Implementing the next rule has a little refactoring in place, maybe the final implementation is like that:

```
	public boolean validate() {
		boolean result = true;
		result = validateLength();
		if (result) {
			result = validateCapitalCharacter();
		}
		return result;
	}


	private boolean validateLength() {
		boolean result = true;
		if (passwd.length() < LENGTH_CONFIG) {
			errorMessage = LENGTH_VALIDATOR_ERROR;
			result = false;
		}
		return result;
	}

	private boolean validateCapitalCharacter() {
```

The problem here is the validate method. It will be grow and grow even you put more validation rules on the password. And if there are dependencies between different rules you have to handle it via control structures as well.

What about this:

```
<bean id="lengthValidator" class="de.neusta.login.validator.LengthValidator">
	<property name="length" value="14" />
</bean>

<bean id="capitalValidator" class="de.neusta.login.validator.CapitalValidator">
	<property name="countOfCapitalCharacters" value="2" />
</bean>

<bean id="chainBuilder" class="de.mwolff.command.chainbuilder.DefaultChainBuilder">
	<property name="validators">
		<list>
			<ref bean="lengthValidator"/>
			<ref bean="capitalValidator"
		</list>
	</property>
</bean>
```

The next validator need just an Implementation of the validator, wire it up in the spring.xml and then enhance the validators' properties. Perfect you think? Yes it is.

Simple implementation

The above example is a configuration file of the Spring framework. Dependency injection is the easiest way to configure algorithm to a complex structure. Of course you can create an own implementation very easy:

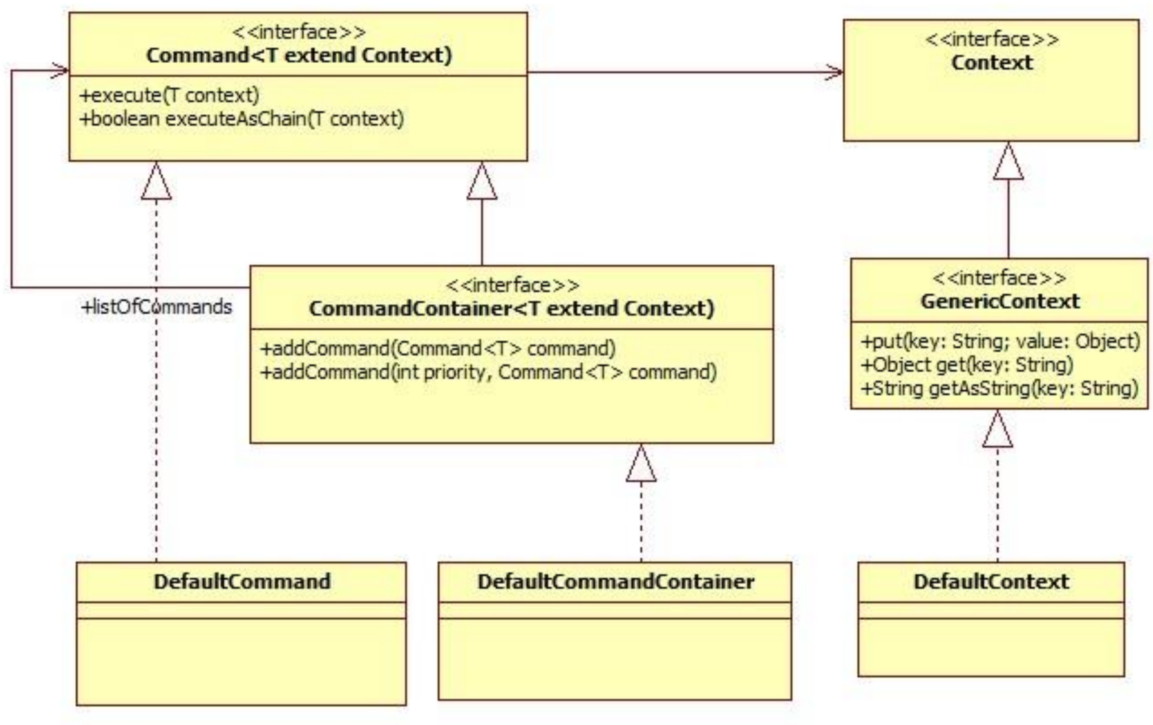
- Define an interface that provides a generic method.
- Encapsulate your algorithm in implementation classes of your interface.
- Build a list of those algorithm classes.
- Iterate through the list an execute each algorithm

Why a framework. There are some reasons:

- Benefit from best practice.

- Use something which is well tested
- Find solutions for more complex scenarios e.g. chaining, passing a context, using configuration issues.

The command framework comes with a couple of interfaces and some default implementation.


<<interface>>
Command<T extend Context)
+execute(T context)
+boolean executeAsChain(T context)
<<interface>>
Context
+listOfCommands
<<interface>>
CommandContainer<T extend Context)
+addCommand(Command<T> command)
+addCommand(int priority, Command<T> command)
<<interface>>
GenericContext
+put(key: String; value: Object)
+Object get(key: String)
+String getAsString(key: String)
DefaultCommand
DefaultCommandContainer
DefaultContext

The simple-command framework

The framework has three ideas:

1. Wrapping algorithm in classes and be able to execute the algorithm.
2. Implementing the chain-of-responsibility pattern: Just stop if one execution of an algorithm fails.
3. Ability to add glue in form of a context. The whole framework works with java generics so the context has just to implement the context interface.

The different parts of the framework:

`Command`: Interface for wrapping an algorithm.

`CommandContainer`: Interface for grouping different `command`-objects in a container. This fits the composite pattern: `CommandContainer` objects have the same behaviour as `command`-objects. For this it is possible to add `command`-objects as well as `CommandContainer` in such a container.

`Context`: Marker interface for a context. `Context`-objects are the glue between different commands. Because of the concept of contexts the whole framework is thread-safe and supports re-entrant algorithms.

`GenericContext`: Interface for a generic context that holds a list of keys and values. A better approach is to create own type-safe contents with get- and set-methods for the values (Bean convention). Because the whole framework works with generics you can instantiate each command with your own context.

`DefaultCommand`: An implementation which provides a generic method for fulfilling the chain-of-responsibility pattern. Just call `executeAsChain()`. All commands will be executes. If one command fails the execution is stopped.

`DefaultCommandContainer`: An implementation to add commands into a container. The execution of the container will execute all parts of it (either commands or command-containers). This fulfils actually the composite pattern.

`DefaultContext`: An implementation of the `GenericContext`-interface to work with generic data in a generic data structure. This is not recommended because it is not type safe.

How to refactor to this framework

There are some clean-code methodologies which has to fulfil.

SRP

The open-close principle is one part of SOLID. Another part is the single responsibility principle. *"There should only one reason to change a class (method)."* In our case the `LoginValidator` fits SRP because the validator has only one responsibility: Validating a password. So the first step is a refactoring to the SRP clean code principle.

One level of abstraction

In our case the `LoginValdator` has to levels of abstraction: Public methods has no "calculation" just calling methods. All private methods holds algorithms that are called from the public method. One level of abstraction means that each method should follow only one of this two levels I described.

Refactoring to pattern

After the software is prepared the refactoring is very easy:

1. Change the signature of all private algorithms to contexts.
2. Encapsulate each algorithm into a `command`-object.
3. Change the public method: Just add all algorithms into a `CommandContainer`.
4. Prepare the context.
5. Execute the `CommandContainer` bypassing the context either brut forced or via chaining.
6. Evaluate the context.
7. (optional) Provide a configuration of your container e.g. with the Spring framework or via the Builder-Pattern.

Now each command takes control of the whole execution. It can read and write values into the context and can <entscheiden> even the execution should going on or should stop.