OCP in Practice

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

**Abstract**

Even simple implementation tend 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.

# 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 patterns 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 easyly 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. Look to a potential first quick implementation of our use case below:

**public** **class** LoginValidator {

**private** String passwd;

**private** String errorMessage = "";

**public** LoginValidator(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.OneCapitalValidator"*>

<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 and the enhancement of 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 an execute method.
* Encapsulate your algorithm in implementation classes of your interface.
* Build a list of those algorithm classes.
* Iterate through the list an execute each algrorithm

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.