

The Simple Command Framework . . . . .	2
Documentation version 2.0.2 . . . . .	4
Github Repository . . . . .	5
SCF in 5 minutes . . . . .	6
Learning by example . . . . .	10
Getting started . . . . .	11
Building a chain of responsibility . . . . .	13
Configure chains with xml . . . . .	17
Configure chains with the Spring framework . . . . .	19
Building a process . . . . .	22
Further functionalities . . . . .	29
The behavior of commands in the SCF . . . . .	31
About the architecture . . . . .	33
Best practices . . . . .	35
Realeasenotes . . . . .	36
Simple Execution Example . . . . .	39

# The Simple Command Framework



Verweise

STATUS

## SIMPLE COMMAND FRAMEWORK

### Inhalt

- [1 Simple Command Framework Version 2.0.2](#)
- [2 SCF in 5 minutes](#)
- [3 SCF by example](#)
- [4 Resources](#)
  - [4.1 Documentation](#)
  - [4.2 License](#)
  - [4.3 Download](#)
  - [4.4 Maven](#)

### Simple Command Framework Version 2.0.2

⚠ This site is in progress. Some links maybe are broken.

This is the homepage of the Simple Command Framework (SCF). The aim of this framework is to offer mechanism to write software which fits the SOLID principles. SOLID means:

- Single Responsibility Principle
- Open / Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

This mnemonic acronym was introduced by Michael Feathers for the "first five principles" which later identified by Robert C. Martin in the early 2000s that stands for five basic principles of object oriented programming and design (<http://social.technet.microsoft.com/wiki/contents/articles/18033.software-design-principles.aspx>).

### SCF in 5 minutes

### SCF by example

### Resources

The framework is 100% test driven development so you've 100% testing code coverage. The code is also tested with the PI Test-Framework (<http://pitest.org/>). Also it is evaluated with SonarQube and has no technical debt (<http://www.sonarqube.org/>).

### Documentation

You may browse the java doc of the latest version here: <https://mwolff.info/apidocs/simple-command/>

### License

/\*\*

Simple Command Framework.

Framework for easy building software that fits the SOLID principles.

Download: <https://github.com/simplecommand/command>

Copyright (C) 2015 - 2021 Manfred Wolff

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

\*/

## Download

You can download the framework as sources or clone the repository: <https://github.com/simplecommand/command>

You can also download the jar from the central maven repository:  
<https://repo1.maven.org/maven2/org/mwolff/command/2.0.0/> (last released version)

## Maven

Since version 1.3.2 the framework is part of the central maven repository.  
To use it you need a dependency in your pom.xml

### pom.xml

```
<!-- command framework -->
<dependency>
  <groupId>org.mwolff</groupId>
  <artifactId>command</artifactId>
  <version>2.0.2</version>
</dependency>
```

# Documentation version 2.0.2

Inhalt	Verweise
<ul style="list-style-type: none"><li>• <a href="#">Github Repository</a></li><li>• <a href="#">SCF in 5 minutes</a></li><li>• <a href="#">Learning by example</a><ul style="list-style-type: none"><li>• <a href="#">Getting started</a></li><li>• <a href="#">Building a chain of responsibility</a></li><li>• <a href="#">Configure chains with xml</a></li><li>• <a href="#">Configure chains with the Spring framework</a></li><li>• <a href="#">Building a process</a></li></ul></li><li>• <a href="#">Further functionalities</a></li></ul>	<div>STATUS</div>

The Simple Command Framework (SCF) is open source and licensed under the GNU Lesser General Public Licence

```
/**
 * Simple Command Framework.
 *
 * Framework for easy building software that fits the SOLID principles.
 * @author Manfred Wolff <m.wolff@neusta.de>
 *
 * Download: https://github.com/simplecommand/command
 *
 * Copyright (C) 2012-2021 Manfred Wolff and the simple command community
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301
 * USA
 */
```

# Github Repository

## Inhalt

- [Clone the repository](#)
- [Resources](#)

## Verweise

STATUS

⚠ This is the documentation of the current version 2.0.0

## Clone the repository

Browse on any directory of your filesystem.

```
git clone https://github.com/simplecommand/command.git
cd command
git checkout Release/2.0.0
```

Then you've the 2.0.0 version sources which is covered in this documentation.

If you want to participate to the project please mail [command@manfred-wolff.de](mailto:command@manfred-wolff.de).

## Resources

You can download the jar file of the last version here:

Version	Downloadlink
Major 1.4.0	<a href="#">command-1.4.0.jar</a>
Major 1.5.0	<a href="#">command-1.5.0.jar</a>
Major 1.5.1	<a href="#">command-1.5.1.jar</a>
2.0.0-RC1	<a href="#">command-2.0.0-RC1.jar</a>
Major 2.0.0	<a href="#">command-2.0.0.jar</a>

You can browse here the latest java doc: <http://mannewolff.info/apidocs/simple-command/>

To use this branch in your java application just add a dependency to your POM:

### Maven dependency

```
<dependency>
  <groupId>org.mwolff</groupId>
  <artifactId>command</artifactId>
  <version>2.0.0</version>
</dependency>
```

If you want to know which versions are in the central maven repository you can browse <http://search.maven.org/#search%7Cga%7C1%7Corg.mwolff>

Non major versions you've to install manually:

```
mvn install:install-file -Dfile=<path-to-file> -DgroupId=org.mwolff \
  -DartifactId=command -Dversion=<version> -Dpackaging=jar
```

Next --> [SCF in 5 minutes](#)

# SCF in 5 minutes

## Inhalt

- The aim of the framework
- Browse the JavaDoc for more information
- Start implementing your first test
- Start implementing your first command
- Create a second command
- Sticking both commands together
- Conclusion

## Verweise

[API Doc](#)  
[Bitbucket Repository](#)

STATUS

⚠ This documentation covers the version 2.0.2 of the framework.

## The aim of the framework

Splitting complex requirements into little pieces of software is a good approach in software development. Think of micro services: Small pieces of software fulfill a part of functionality. If you put them together you've a powerful application. The smallest unit in the SCF is the Command object. A Command is a java class which provides a little piece of functionality in its `executeCommand()` - method. The glue between several Command objects is the `ParameterObject`. This can be anything derived from `java.lang.Object` which is not final (of course you want to change values).

Add the framework into your maven project

```
<dependency>
  <groupId>org.mwolff</groupId>
  <artifactId>command</artifactId>
  <version>2.0.2</version>
</dependency>
```

To run unit tests add the following dependency in maven for supporting JUnit 5:

maven junit dependency

› [Quelle erweitern](#)

## Browse the JavaDoc for more information

<https://mwolff.info/apidocs/simple-command/>

## Start implementing your first test

```
1 package org.mwolff.helloworld;
2
3
4 import static org.hamcrest.CoreMatchers.*;
5 import static org.hamcrest.MatcherAssert.assertThat;
6 import static org.junit.Assert.*;
7
8 import org.junit.Test;
9 import org.mwolff.command.interfaces.CommandTransition;
10 import org.mwolff.command.parameterobject.DefaultParameterObject;
11 import org.mwolff.command.parameterobject.GenericParameterObject;
12
13 public class HelloWorldCommandTest {
14
15     /**
16      * Tests if getInstance() returns unique instances and the right inte
17      */
18     @Test
19     public void getInstanceTest() throws Exception {
20         HelloWorldCommand hello = HelloWorldCommand.getInstance();
21         HelloWorldCommand world = HelloWorldCommand.getInstance();
22         assertThat(HelloWorldCommand.getInstance(), instanceOf(HelloWorldCommand.class));
```

```

23         assertThat(hello, is(not(world)));
24     }
25
26     /**
27      * Tests if the command sets a token named helloworld.message with the
28      * value "Hello World".
29      */
30     @Test
31     public void testExecuteHelloWorld() throws Exception {
32         GenericParameterObject context = DefaultParameterObject.getInstance();
33         CommandTransition result = HelloWorldCommand.getInstance().execute(context);
34         assertThat(result, is(CommandTransition.SUCCESS));
35         assertThat(context.getString("helloworld.message"), is("Hello World"));
36     }
37 }

```

- **Line 18:** The first test is a test of the factory method. It is best practice to provide a factory method for each command.
- **Line 29:** Here is the actual implementation of the executeCommand() method. For a simple use of commands this is the entry point of your business logic.
- **Line 30:** The framework works with **parameter objects**. SCF offers a generic implementation of such an object. It is best practice to create your own object but if you want to reuse the command in several environments it make sense just work with the generic one.

We want to assert that a key is generated in the command with value "Hello World" and the command executes with result SUCCESS.

## Start implementing your first command

```

package org.mwolff.helloworld;

import org.mwolff.command.interfaces.Command;
import org.mwolff.command.interfaces.CommandTransition;
import org.mwolff.command.parameterobject.GenericParameterObject;

public class HelloWorldCommand implements Command<GenericParameterObject> {

    public static final HelloWorldCommand getInstance() {
        return new HelloWorldCommand();
    }

    @Override
    public CommandTransition executeCommand(GenericParameterObject parameterObject) {
        parameterObject.put("helloworld.message", "Hello World");
        return CommandTransition.SUCCESS;
    }
}

```

To prevent to implement the legacy methods all the time, you can implement an AbstractDefaultCommand.

## Create a second command

```

package org.mwolff.helloworld;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.junit.Assert.*;

import org.junit.Test;
import org.mwolff.command.DefaultCommandContainer;
import org.mwolff.command.interfaces.CommandContainer;
import org.mwolff.command.interfaces.CommandTransition;
import org.mwolff.command.parameterobject.DefaultParameterObject;
import org.mwolff.command.parameterobject.GenericParameterObject;

public class SecondCommandTest {

    @Test
    public void getInstanceTest() throws Exception {
        SecondCommand instance = SecondCommand.getInstance();
    }
}

```

```

        assertThat(instance, instanceOf(SecondCommand.class));
    }

    @Test
    public void testHowAreYou() throws Exception {
        GenericParameterObject parameterObject = DefaultParameterObject.getInstance();
        parameterObject.put(SecondCommand.helloworld_message, "");
        CommandTransition result = SecondCommand.getInstance().executeCommand(parameterObject);
        assertThat(parameterObject.getString(SecondCommand.helloworld_message), is(""));
        assertThat(result, is(CommandTransition.SUCCESS));
    }
}

```

```

package org.mwolff.helloworld;

import org.mwolff.command.AbstractDefaultCommand;
import org.mwolff.command.interfaces.CommandTransition;
import org.mwolff.command.parameterobject.GenericParameterObject;

public class SecondCommand extends AbstractDefaultCommand<GenericParameterObject> {

    public static final String helloworld_message = "helloworld.message";

    public static final SecondCommand getInstance() {
        return new SecondCommand();
    }

    @Override
    public CommandTransition executeCommand(GenericParameterObject parameterObject) {
        String message = parameterObject.getString(helloworld_message);
        message += " How are you?";
        parameterObject.put(helloworld_message, message);
        return CommandTransition.SUCCESS;
    }
}

```

## Sticking both commands together

```

@Test
public void testBothTogether() throws Exception {

    GenericParameterObject parameterObject = DefaultParameterObject.getInstance();
    CommandContainer<GenericParameterObject> container = new DefaultCommandContainer();

    CommandTransition result = container
        .addCommand(HelloWorldCommand.getInstance())
        .addCommand(SecondCommand.getInstance())
        .executeCommand(parameterObject);

    assertThat(parameterObject.getString(SecondCommand.helloworld_message), is("How are you?"));
    assertThat(result, is(CommandTransition.SUCCESS));
}

```

## Conclusion

SCF is a framework is need for all situations where you

- want to write code fragments, which are loose coupled.
- want to write command chains which you will execute.
- want to write strategies.

SCF comes with interfaces and default implementation.

This example is part of simple-command-tools. [You can clone it or download it directly here.](#)

[Learn more about the Simple Command Framework.](#)





## Learning by example

### Inhalt

- [Getting started](#)
- [Building a chain of responsibility](#)
- [Configure chains with xml](#)
- [Configure chains with the Spring framework](#)
- [Building a process](#)

### Verweise

STATUS



This is the documentation of the current version 2.0.0

# Getting started

## Inhalt

- [Maven archetype](#)
- [Starting the project](#)
- [Importing to the IDE](#)
- [What is out of the box](#)

## Verweise

STATUS

## Maven archetype

A good approach to start a new project is working with maven archetypes. SCF offers an archetype to start with. Maven downloads the archetype automatically. Current version of the archetype is 2.0.2


## Starting the project

Create an directory, browse to the directory and type in a shell:

```
mvn archetype:generate -DarchetypeGroupId=org.mwolff -DarchetypeArtifactId=comm
```

Then you have to answer some questions regarding groupId, artefactId, version etc.

```
mwolff@laptopc3100:~/tmp$ mvn archetype:generate -DarchetypeGroupId=org.mwolff
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] Generating project in Interactive mode
[WARNING] Archetype not found in any catalog. Falling back to central repository
[WARNING] Use -DarchetypeRepository=<your repository> if archetype's repository
Define value for property 'groupId': : de.mycompany
Define value for property 'artifactId': : myfirstproject
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': de.mycompany: :
Confirm properties configuration:
groupId: de.mycompany
artifactId: myfirstproject
version: 1.0-SNAPSHOT
package: de.mycompany
Y: : Y
[INFO] -----
[INFO] Using following parameters for creating project from Old (1.x) Archetype
[INFO] -----
[INFO] Parameter: basedir, Value: /home/mwolff/tmp
[INFO] Parameter: package, Value: de.mycompany
[INFO] Parameter: groupId, Value: de.mycompany
[INFO] Parameter: artifactId, Value: myfirstproject
[INFO] Parameter: packageName, Value: de.mycompany
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /home/mwolff/tmp/myfirs
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 25.620 s
[INFO] Finished at: 2017-08-31T13:13:26+02:00
[INFO] Final Memory: 15M/221M
[INFO] -----
```

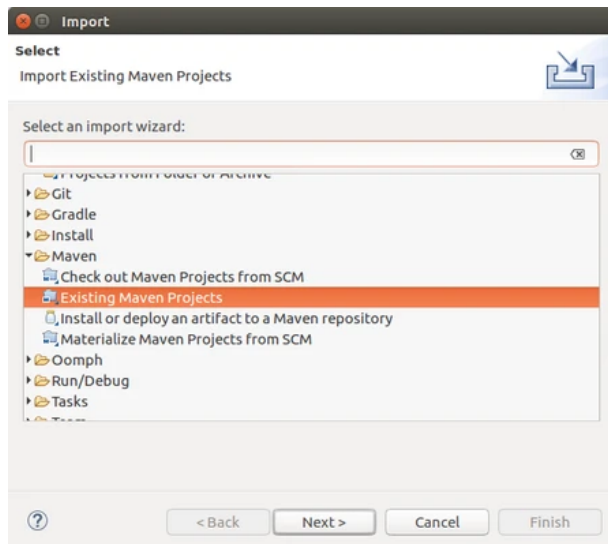
 If you want to work with an other version of the framework you've to change it in your pom.xml. Current release version is 2.0.2.

After the process is finished you have a ready-to-go project.

```
mwolff@laptopc3100:~/tmp/example$ ll
insgesamt 12
drwxr-xr-x 3 mwolff users 4096 Aug 31 12:53 ./
drwxr-xr-x 5 mwolff users 4096 Aug 31 12:52 ../
drwxr-xr-x 3 mwolff users 4096 Aug 31 12:53 myfirstproject/
```

## Importing to the IDE

In eclipse or IntelliJ you may now import your project as a maven project.



## What is out of the box

You first have a pom with many dependencies:

**pom.xml with ll necessary artifacts**

[Quelle erweitern](#)

Of course the dependency to the current SCF is configured as well. Secondly the simplecommandframework.xsd for validating XML driven configuration for commands and processes are inserted in the resources folder. You can just start write your first test.

Next -> [Building a chain of responsibility](#)

# Building a chain of responsibility

## Content

- [User Story](#)
- [Architecture](#)
- [Test driven approach for the first strategy, the first chain](#)
- [Test driven approach for the second chain command](#)
- [Stick them both together](#)

## Links to other resources

[The behavior of commands in the SCF](#)

STATUS

## User Story

**As a Users I want to load transparent files for various sources so that nobody cares where the file is located.**

Acceptance criteria:

- The module just gets a name of a file e.g. `demofile.xml`.
- To start the module just write an integration test for it.
- First the file is interpreted as a file laying in the classpath e.g. `/demofile.xml`
- Secondly the file is interpreted as a file laying in the filesystem e.g. `/demofile.xml` as it may lays in root directory.
- The chain feedbacks "DONE" if the work is done, true otherwise (this is just the behavior of the chain functionality).

## Architecture

First some theory: The chain of responsibility design pattern is described as followed:

"In **object-oriented design**, the **chain-of-responsibility pattern** is a **design pattern** consisting of a source of **command objects** and a series of **processing objects**.<sup>[1]</sup> Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain."  
([https://en.wikipedia.org/wiki/Chain-of-responsibility\\_pattern](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern))

This is actually an good example for a **chain of responsibility design pattern**. Every chain is a strategy to load a file from a dedicated source. If it was successful the chain can be aborted, otherwise the next strategy has to overtake. Another good example is a filter mechanism where each command does a little logic to handle part of the "problem".

Actually you have to possibilities to implement the stuff:

1. Providing an `executeCommand()` method which just returns *DONE* or *NEXT*, where *NEXT* means the next command should overtake.
2. Providing an `executeCommand()` method that returns *SUCCESS* if the operation fails (because *SUCCESS* is mapped to *NEXT* in the `AbstractDefaultChainCommand` implementation) and returns *DONE* if the operation succeed because *DONE* means the work of the whole chain is competed. I don't recommend this way, because it is just a little bit confusing.

We take the first approach for the first strategy and the second for the other implementation.

See more in [The behavior of commands in the SCF](#).

## Test driven approach for the first strategy, the first chain

Assuming we [build up a project with the maven archetype for the SCF](#), we can start immediately.

### FileLoaderCommandTest.java

```
package org.mwolff.resourceloader.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.*;

import org.junit.Test;
import org.mwolff.command.interfaces.CommandTransition;

public class FileLoaderCommandTest {

    private final String filename = "src/test/resources/test.txt";

    @Test
    public void testReadFileFails() throws Exception {
        final CommandTransition result = new FileLoaderCommand().executeCommand(
            filename);
        assertThat(result, is(CommandTransition.NEXT));
    }
}
```

```

    }

    @Test
    public void testReadFile() throws Exception {
        final CommandTransition result = new FileLoaderCommand().executeCommand
        assertThat(result, is(CommandTransition.DONE));
    }
}

```

Here one solution which fulfills the test (you may have a better one) 😊

**Hint**  
To get the test green you need an empty file named "test.txt" placed in the src/text/resources folder.

#### FileLoaderCommand.java

```

package org.mwolff.resourceloader.commands;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

import org.mwolff.command.chain.AbstractDefaultChainCommand;
import org.mwolff.command.interfaces.CommandTransition;

public class FileLoaderCommand extends AbstractDefaultChainCommand<String> {

    @Override
    public CommandTransition executeCommandAsChain(String parameterObject) {

        try {
            new FileInputStream(new File(parameterObject));
        } catch (FileNotFoundException e) {
            return CommandTransition.NEXT;
        }
        return CommandTransition.DONE;
    }
}

```

As you see in this example the ParameterObject can be a final Object as well if you don't want to change values of the object.

## Test driven approach for the second chain command

#### ClassLoaderCommandTest.java

```

package org.mwolff.resourceloader.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

import org.junit.Test;
import org.mwolff.command.interfaces.CommandTransition;

public class ClassLoaderCommandTest {

    private final String classPathResource = "test.txt";

    @Test
    public void testReadBadFile() throws Exception {
        final CommandTransition result = new ClassLoaderCommand().executeComman
        assertThat(result, is(CommandTransition.NEXT));
    }

    @Test
    public void testReadGoodFile() throws Exception {
        final CommandTransition result = new ClassLoaderCommand().executeComman
        assertThat(result, is(CommandTransition.FAILURE));
    }
}

```

```
}
```

#### ClassLoaderCommand

```
package org.mwolff.resourceloader.commands;

import java.io.InputStream;

import org.mwolff.command.chain.AbstractDefaultChainCommand;
import org.mwolff.command.interfaces.CommandTransition;

public class ClassLoaderCommand extends AbstractDefaultChainCommand<String>{

    @Override
    public CommandTransition executeCommandAsChain(String parameterObject) {
        final String filename = parameterObject;
        final InputStream inputStream = this.getClass().getResourceAsStream("/"
        if (inputStream == null) {
            return CommandTransition.NEXT;
        }
        return CommandTransition.FAILURE;
    }
}
```

As you see the second implementation has the disadvantage that you have to return SUCCESS even the operation fails because of the mapping.

## Stick them both together

For this use case: Last but not least we need an integration test to stick all things together. If one command succeeded the whole chain returns DONE, NEXT otherwise.

```
package org.mwolff.resourceloader.integrationtests;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.*;

import org.junit.Test;
import org.mwolff.command.DefaultCommandContainer;
import org.mwolff.command.builder.XMLChainBuilder;
import org.mwolff.command.interfaces.CommandContainer;
import org.mwolff.command.interfaces.CommandTransition;
import org.mwolff.resourceloader.commands.ClassLoaderCommand;
import org.mwolff.resourceloader.commands.FileLoaderCommand;

public class ChainIntegrationTest {

    @Test
    public void testGreenPathClassloader() throws Exception {
        CommandContainer<String> commandContainer = new DefaultCommandContainer
        CommandTransition result = commandContainer
            .addCommand(new ClassLoaderCommand())
            .addCommand(new FileLoaderCommand())
            .executeCommandAsChain("test.txt");

        assertThat(result, is(CommandTransition.FAILURE));
    }

    @Test
    public void testGreenPathXML() throws Exception {
        XMLChainBuilder<String> builder = new XMLChainBuilder<>("/chain.xml");
        CommandTransition result = builder.executeCommandAsChain("test.txt");
        assertThat(result, is(CommandTransition.DONE));
    }

    @Test
    public void testRedPathClassloader() throws Exception {

        new ClassLoaderCommand();
    }
}
```

```

        new FileLoaderCommand();

        CommandContainer<String> commandContainer = new DefaultCommandContainer();
        CommandTransition result = commandContainer
            .addCommand(new ClassLoaderCommand())
            .addCommand(new FileLoaderCommand())
            .executeCommandAsChain("nobodyKnows.txt");

        assertThat(result, is(CommandTransition.NEXT));
    }
}

```

Of course in a real world project you need in this case more tests: File exists but no classpath ... etc.

[Feel free to clone the examples repository to browse.](#)

Next -> [Configure chains with xml](#)



# Configure chains with xml

## Inhalt

- [Review of the previous lecture](#)
- [About chain builder](#)
- [Refactoring of the last example](#)
- [Conclusion](#)

## Verweise

STATUS

## Review of the previous lecture

Let's look to the example we discussed before: [Building a chain of responsibility](#). Remember we build the chain manually with java code:

```
CommandContainer<String> commandContainer = new DefaultCommandContainer<>();
boolean result = commandContainer
    .addCommand(new ClassLoaderCommand())
    .addCommand(new FileLoaderCommand())
    .executeCommandAsChain("test.txt");
```

SCF offers two more possibilities to build chains:

- Building chains with xml, we will discuss here.
- Building chains with a dependency framework like spring, [we will discuss in the next section](#).

Check out the example to follow it by code <https://github.com/simplecommand/simple-command-tools.git> mainly the resource loader project.

## About chain builder

A chain builder has the following behavior:

1. It is a command. So you actually can add it to a command container.
2. It is a command container, so you can call `executeCommand()` or `executeCommandAsChain()` on it.
3. The `ChainBuilder` interface offers a method `buildChain()` which returns a `CommandContainer`.

There are two concrete builder:

- The `XMLChainBuilder`, we want to discuss here.
- The `InjectionChainBuilder`, [we want to cover in the next section](#)

 The `ChainBuilder` interface is removed in 2.0.0 because it was only a marker interface.

## Refactoring of the last example

Refactoring is quite simple. We offer a second integration test to build the chain via XML.

```
@Test
public void testGreenPathXML() throws Exception {
    XMLChainBuilder<String> builder = new XMLChainBuilder<>("/chain.xml");
    CommandTransition result = builder.executeCommandAsChain("test.txt");
    assertThat(result, is(CommandTransition.DONE));
}
```

For the implementation of this test we just have to offer an xml file called `chain.xml`. It has to be in the class path. For a convenient building of the file SCF offers a XSD.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<process xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="simplecommandframework.xsd">
    <action class="org.mwolff.resourceloader.commands.ClassLoaderCommand" />
    <action class="org.mwolff.resourceloader.commands.FileLoaderCommand" />
</process>
```

## Conclusion

This solution fits the open close principle: You can modify the software without modifying the code. The next strategy is just an implementation of a command plus the enhancement of the xml file.

Next .> [Configure chains with the Spring framework](#)

# Configure chains with the Spring framework

## Inhalt

- [Some words about spring](#)
- [Configuration of spring](#)
- [Inject the commands](#)
- [The final test](#)

## Verweise

STATUS

## Some words about spring

Spring is a javaEE framework which offers many functionality. For our purpose we use the injection framework and the test framework. You can add them easily to your pom.xml

```
<properties>
  <springframework.version>5.3.4</springframework.version>
</properties>

<dependencies>

  .....

  <!-- Spring -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${springframework.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${springframework.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${springframework.version}</version>
  </dependency>

  .....
</dependencies>
```

If you are interested in spring please browse the original documentation: <https://spring.io/docs>

## Configuration of spring

In real world projects you need to initialize the application context of spring. Either for a web application or for a standalone application. [Browse here the spring documentation](#).

In our case we initialize the context in a test. The first test is to insure that the chain builder is injected.

```
1 package org.mwolff.resourceloader.integrationtests;
2
3
4 import static org.junit.Assert.*;
5 import javax.annotation.Resource;
6 import org.junit.Test;
7 import org.junit.runner.RunWith;
8 import org.mwolff.command.chain.InjectionChainBuilder;
9 import org.springframework.test.context.ContextConfiguration;
10 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
11
12 @RunWith(SpringJUnit4ClassRunner.class)
13 @ContextConfiguration("classpath:application.xml")
14 public class SpringChainIntegrationTest {
```

```

15
16     @Resource
17     InjectionChainBuilder<String> injectionChainBuilder;
18
19     @Test
20     public void testInjectionSuccessful() throws Exception {
21         assertThat(injectionChainBuilder, CoreMatchers.instanceOf(Chaint
22     }
    }

```

- Line 11 tells junit to take the spring unit class loader.
- Line 12 is actually the initialization of the application context.
- Line 15 is the injection point for the chainbulder.

The test asserts that the builder is successfully integrated.

To implement the test you've in Spring three possibilities:

- Configure the framework with xml.
- Configure the framework only annotation based (since Spring 3)
- Configure the framework with java configuration (since Spring 4)

For our example we use the first ability:

#### application.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee" xmlns:lang="http://ww
    xmlns:tx="http://www.springframework.org/schema/tx" xmlns:util="http://www.
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
        http://www.springframework.org/schema/aop http://www.springframework.or
        http://www.springframework.org/schema/context http://www.springframewor
        http://www.springframework.org/schema/jee http://www.springframework.or
        http://www.springframework.org/schema/lang http://www.springframework.o
        http://www.springframework.org/schema/tx http://www.springframework.org
        http://www.springframework.org/schema/util http://www.springframework.o
    default-autowire="byName">

    <context:annotation-config />

    <bean name="injectionChainBuilder" class="org.mwolff.command.chain.Injectio
</bean>

</beans>

```

Now the test runs green.

## Inject the commands

The next step is to inject the commands. The test should only test if two commands are injected in the builder.

```

@Test
public void testCommandsAreInjected() throws Exception {
    @SuppressWarnings("unchecked")
    List<Command<String>> commandList = (List<Command<String>>) ReflectionTestU
    assertThat(commandList.size(), is(2));
}

```

The implementation is done by spring itself, just inject the commands in the application.xml of spring framework.

```

<bean name="classLoaderCommand" class="org.mwolff.resourceloader.commands.Class
<bean name="fileLoaderCommand" class="org.mwolff.resourceloader.commands.FileLo

<bean name="injectionChainBuilder" class="org.mwolff.command.chain.InjectionCha
    <property name="commands">
        <list>
            <ref bean="classLoaderCommand"/>
            <ref bean="fileLoaderCommand"/>
        </list>
    </property>

```

```
</bean>
```

## The final test

Everything is done. Now we need the final integration test, which should be green a well.

```
package org.mwolff.resourceloader.integrationtests;

import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;
import java.util.List;
import javax.annotation.Resource;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mwolff.command.Command;
import org.mwolff.command.CommandTransitionEnum.CommandTransition;
import org.mwolff.command.chain.InjectionChainBuilder;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.util.ReflectionTestUtils;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:application.xml")
public class SpringChainIntegrationTest {

    @Resource
    InjectionChainBuilder<String> injectionChainBuilder;

    @Test
    public void testInjectionSuccessful() throws Exception {
        assertThat(injectionChainBuilder, notNullValue());
    }

    @Test
    public void testCommandsAreInjected() throws Exception {
        @SuppressWarnings("unchecked")
        List<Command<String>> commandList = (List<Command<String>>) ReflectionTestUtils
        .getFieldValue(this, "injectionChainBuilder", "commandList");
        assertThat(commandList.size(), is(2));
    }

    @Test
    public void testGreenPathSpring() throws Exception {
        CommandTransition result = injectionChainBuilder.executeCommandAsChain(
            "greenPathSpring");
        assertThat(result, is(CommandTransition.DONE));
    }
}
```

Next -> [Building a process](#)

## Building a process

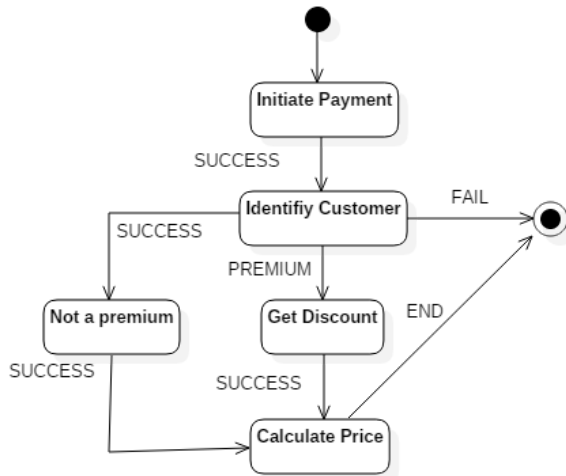
### Inhalt

- Designing a process
- Starting with the InitiatePaymentCommand
- Creating the XML
- Implementing the IdentifyCustomerProcess
- Implementing the GetDiscountProcess and SuccessProcessCommand
- Implementing the whole process in XML

### Verweise

STATUS

## Designing a process



Just look to this little process. Some nodes just do some business logic and succeeded. Other nodes has different - up to three - transitions. The only ability to stick processes together is an xml format. We want to implement everything simple and with TDD approach.

Here some requirements for the show case implementation.

- All process nodes which returns only *SUCCESS* will be represented through an *SuccessProcessCommand*.
- Initiate payment calculates a random number.
- If the random number is odd it is a premium customer, other wise not a premium.
- If the random number is mod 3 than the process should fail.
- The basket has an amount of \$100.
- The Discount is 10%

**⚠** The process mode is only applicable with the *DefaultCommandContainer*. It makes no sense just to execute a single node, which might make sense in the other modes (command, chain of responsibility). Actually you should use the XML notation and the *XMLChainBuilder* (which returns a *CommandContainer*) to build up and start a process.

## Starting with the InitiatePaymentCommand

We will implement the *ParameterObject* in an TDD approach as well so it will be grown during implementation. We start with the first process:

### Parameterobject

```
package org.mwolff.processdemo.commands;

public class PaymentParameterObject {

    private String customerAccountNumber;

    public String getCustomerAccountNumber() {
        return customerAccountNumber;
    }
}
```

```

    public void setCustomerAccountNumber(String customerAccountNumber) {
        this.customerAccountNumber = customerAccountNumber;
    }
}

```

I actually write no tests for POJOs. The setter and getter will be tested transitive. If the coverage detect an method which is not tested I delete it, because officially it is not used.

#### Test

```

package org.mwolff.processdemo.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.*;

import org.junit.Test;

import junit.framework.TestCase;

public class InitiatePaymentCommandTest extends TestCase {

    @Test
    public void testExecuteMethod() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        InitiatePaymentCommand initiatePaymentCommand = new InitiatePaymentComm
        String next = initiatePaymentCommand.executeAsProcess(paymentParameterO
        assertThat(next, is("SUCCESS"));
        String number = paymentParameterObject.getCustomerAccountNumber();
        assertThat(number, notNullValue());
    }

    @Test
    public void testTestMode() throws Exception {
        InitiatePaymentCommand initiatePaymentCommand = new InitiatePaymentComm
        initiatePaymentCommand.setProcessID("INITIATE");
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setTestmode(true);
        paymentParameterObject.setCustomerAccountNumber("100");
        String next = initiatePaymentCommand.executeAsProcess(paymentParameterO
        assertThat(next, is("SUCCESS"));
        String number = paymentParameterObject.getCustomerAccountNumber();
        assertThat(number, is("100"));
        String breadCrumb = paymentParameterObject.getBreadCrumb();
        assertThat(breadCrumb, is(" => INITIATE"));
    }
}

```

The test asserts only the result of the command (should be SUCCESS) and that a number is generated.

#### ① Hint

Beside that I generally use bread crumbs when I work with the process mode. The bread crumb indicates the steps, a process walk through. That is only necessary for integration tests. For the "bread crumb tests" I set a test mode to true in the parameter object.

#### Implementation

```

package org.mwolff.processdemo.commands;

import java.util.Random;

import org.mwolff.command.interfaces.CommandTransition;
import org.mwolff.command.process.AbstractDefaultProcessCommand;
import org.springframework.util.StringUtils;

public class InitiatePaymentCommand extends AbstractDefaultProcessCommand<Payme

    @Override
    public String executeAsProcess(PaymentParameterObject context) {

```

```

        executeCommand(context);
        return "SUCCESS";
    }

    @Override
    public CommandTransition executeCommand(PaymentParameterObject parameterObj) {
        if (!parameterObject.isTestmode()) {
            Random rn = new Random();
            int n = Integer.MAX_VALUE;
            int i = rn.nextInt() % n;
            parameterObject.setCustomerAccountNumber(Integer.valueOf(i).toString());
        } else {
            String breadCrumb = parameterObject.getBreadCrumb();
            if (StringUtils.isEmpty(breadCrumb)) breadCrumb = "";
            breadCrumb = breadCrumb + " => " + getProcessID();
            parameterObject.setBreadCrumb(breadCrumb);
        }
        return CommandTransition.SUCCESS;
    }
}

```

#### Important

It is very important that you inherit from `AbstractDefaultProcessCommand` because here is the logic implemented for the process handling.

## Creating the XML

### Integrationtest

```

package org.mwolff.processdemo.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.*;
import org.junit.Test;
import org.mwolff.command.builder.XMLChainBuilder;

public class IntegrationTest {
    @Test
    public void testAll() throws Exception {
        ChainBuilder<PaymentParameterObject> xmlChainBuilder = new XMLChainBuilder();
        PaymentParameterObject paymentParameterObject = new PaymentParameterObject();
        String result = xmlChainBuilder.executeAsProcess("START", paymentParameterObject);
        assertThat(result, is("END"));
        String number = paymentParameterObject.getCustomerAccountNumber();
        assertThat(number, notNullValue());
    }
}

```

For the XML we provide an end note, which is part of the framework.

### process.xml

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<process xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="simplecommandframework.xsd">
    <action class="org.mwolff.processdemo.commands.InitiatePaymentCommand" id="START">
        <transition name="SUCCESS" to="END"></transition>
    </action>
    <action class="org.mwolff.command.process.DefaultEndCommand" id="END"/>
</process>

```

## Implementing the IdentifyCustomerProcess

This is the key of the process because we've three different transitions.

- The transition is FAIL, if the created number is MOD 3.
- The transition is PREMIUM, if the created number is odd.
- The transition is SUCCESS, otherwise.



## Test

```
package org.mwolff.processdemo.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

import org.hamcrest.CoreMatchers;
import org.junit.Test;

public class IdentifyCustomerProcessTest {

    @Test
    public void testFailure() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setCustomerAccountNumber("42");
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("FAIL"));
    }

    @Test
    public void testPremiumCustomer() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setCustomerAccountNumber("41");
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("PREMIUM"));
    }

    @Test
    public void testNoPremiumCustomer() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setCustomerAccountNumber("40");
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("SUCCESS"));
    }

    @Test
    public void testTestMode() throws Exception {
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        identifyCustomerProcess.setProcessID("IDENTIFY");
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setTestmode(true);
        paymentParameterObject.setCustomerAccountNumber("40");
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("SUCCESS"));
        String breadCrumb = paymentParameterObject.getBreadCrumb();
        assertThat(breadCrumb, is(" => IDENTIFY"));
    }
}
```

## Implementation

```
package org.mwolff.processdemo.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.*;

import org.hamcrest.CoreMatchers;
import org.junit.Test;

public class IdentifyCustomerProcessTest {

    @Test
    public void testFailure() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setCustomerAccountNumber("42");
```

```

        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("FAIL"));
    }

    @Test
    public void testPremiumCustomer() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setCustomerAccountNumber("41");
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("PREMIUM"));
    }

    @Test
    public void testNoPremiumCustomer() throws Exception {
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setCustomerAccountNumber("40");
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("SUCCESS"));
    }

    @Test
    public void testTestMode() throws Exception {
        IdentifyCustomerProcess identifyCustomerProcess = new IdentifyCustomerP
        identifyCustomerProcess.setProcessID("IDENTIFY");
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setTestmode(true);
        paymentParameterObject.setCustomerAccountNumber("40");
        String next = identifyCustomerProcess.executeAsProcess(paymentParameter
        assertThat(next, CoreMatchers.is("SUCCESS"));
        String breadcrumb = paymentParameterObject.getBreadCrumb();
        assertThat(breadcrumb, is(" => IDENTIFY"));
    }
}

```

## Implementing the GetDiscountProcess and SuccessProcessCommand

We don't provide the implementation in detail, because it is almost the same as we saw in `InitiatePaymentCommand`. Feel free:

- To browse the solution in our github
- Clone the Simple Command Tools: <https://github.com/simplecommand/simple-command-tools.git>

## Implementing the whole process in XML

For the final integration test **we need an ability to test the different process flows**. For that we introduce a test mode. If test equals true, the account will not created but injected. Secondly we introduce a breadcrumb to be sure the right process steps are created. So we've to enhance all tests and all commands with test mode and breadcrumb. We skip this section here because it is quite easy and [you can still see it in the code of our repository](#). Here's the final integration test and the final xml file.

### Integration test

```

package org.mwolff.processdemo.commands;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.*;

import org.junit.Test;
import org.mwolff.command.builder.XMLChainBuilder;

public class IntegrationTest {

    @Test
    public void testPremium() throws Exception {
        XMLChainBuilder<PaymentParameterObject> xmlChainBuilder = new XMLChainB
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setTestmode(true);
        paymentParameterObject.setCustomerAccountNumber("31");
    }
}

```

```

        String result = xmlChainBuilder.executeAsProcess("START", paymentParamete
        String breadcrumb = paymentParameterObject.getBreadCrumb();
        assertThat(breadcrumb, is(" => START => IDENTIFY_CUSTOMER => GET_DISCOU
        assertThat(result, is("END"));
        String number = paymentParameterObject.getCustomerAccountNumber();
        assertThat(number, is("31"));
    }

    @Test
    public void testFail() throws Exception {
        XMLChainBuilder<PaymentParameterObject> xmlChainBuilder = new XMLChainB
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setTestmode(true);
        paymentParameterObject.setCustomerAccountNumber("30");
        String result = xmlChainBuilder.executeAsProcess("START", paymentParamete
        String breadcrumb = paymentParameterObject.getBreadCrumb();
        assertThat(breadcrumb, is(" => START => IDENTIFY_CUSTOMER"));
        assertThat(result, is("END"));
        String number = paymentParameterObject.getCustomerAccountNumber();
        assertThat(number, is("30"));
    }

    @Test
    public void testNotAPremium() throws Exception {
        XMLChainBuilder<PaymentParameterObject> xmlChainBuilder = new XMLChainB
        PaymentParameterObject paymentParameterObject = new PaymentParameterObj
        paymentParameterObject.setTestmode(true);
        paymentParameterObject.setCustomerAccountNumber("20");
        String result = xmlChainBuilder.executeAsProcess("START", paymentParamete
        String breadcrumb = paymentParameterObject.getBreadCrumb();
        assertThat(breadcrumb, is(" => START => IDENTIFY_CUSTOMER => NOT_A_PREM
        assertThat(result, is("END"));
        String number = paymentParameterObject.getCustomerAccountNumber();
        assertThat(number, is("20"));
    }
}

```

#### Process definition

```

<?xml version="1.0" encoding="ISO-8859-1" ?>

<process xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="simplecommandframework.xsd">

    <action class="org.mwolff.processdemo.commands.InitiatePaymentCommand"
        id="START">
        <transition name="SUCCESS" to="IDENTIFY_CUSTOMER"></transition>
    </action>

    <action class="org.mwolff.processdemo.commands.IdentifyCustomerProcess"
        id="IDENTIFY_CUSTOMER">
        <transition name="FAIL" to="END"></transition>
        <transition name="PREMIUM" to="GET_DISCOUNT"></transition>
        <transition name="SUCCESS" to="NOT_A_PREMIUM"></transition>
    </action>

    <action class="org.mwolff.processdemo.commands.GetDiscountProcessCommand"
        id="GET_DISCOUNT">
        <transition name="SUCCESS" to="CALCULATE_PRICE"></transition>
    </action>

    <action class="org.mwolff.processdemo.commands.SuccessCommand"
        id="NOT_A_PREMIUM">
        <transition name="SUCCESS" to="CALCULATE_PRICE"></transition>
    </action>

    <action class="org.mwolff.processdemo.commands.SuccessCommand"
        id="CALCULATE_PRICE">
        <transition name="SUCCESS" to="END"></transition>
    </action>

```

```
<action class="org.mwolff.command.process.DefaultEndCommand" id="END" />
</process>
```

Next --> [About the architecture](#)

## Further functionalities

### Inhalt

- [Creating Commands on the fly](#)
- [Insert commands in a predefined order](#)

### Verweise

STATUS

## Creating Commands on the fly

Because a Command object is a functional interface, you can create commands via lambdas as well.

```
@DisplayName("some tests on commands")
public class CommandTest {

    @Test
    @DisplayName("starting command as lambda")
    void commandLambdaTest() {

        var myContext = new DefaultParameterObject();

        var container = new DefaultCommandContainer<DefaultParameterObject>().a
            (context) -> {
                context.put("Hallo", "Welt");
                return CommandTransition.SUCCESS;
            };

        assertThat(container.executeCommand(myContext)).isEqualTo(CommandTransi
        assertThat(myContext.get("Hallo")).isEqualTo("Welt");
    }
}
```

## Insert commands in a predefined order

Usually Commands are executed in the order they are inserted in the CommandContainer. You can force execution in a concrete order by inserting them via priority.

```
@Test
@DisplayName("order test for command container")
void commandOrderTest() {

    Command<DefaultParameterObject> command1 = (context) -> {
        var order = context.getAsString("order");
        context.put("order", order + " -1");
        return SUCCESS;
    };

    Command<DefaultParameterObject> command2 = (context) -> {
        var order = context.getAsString("order");
        context.put("order", order + " -2");
        return SUCCESS;
    };

    var context = new DefaultParameterObject();
    var container = new DefaultCommandContainer<DefaultParameterObject>()
        .addCommand(2, command2)
        .addCommand(1, command1)
        .executeCommand(context);

    assertThat(context.getAsString("order")).isEqualTo(" -1 -2");
}
```

The XML configuration as well has the possibility to insert commands with a given order (priority).

# The behavior of commands in the SCF

## Inhalt

- [Command](#)
- [ChainCommand](#)
- [CommandContainer](#)
- [Composite Pattern](#)

## Verweise

STATUS

## Command

The command interface offers one method:

```
CommandTransition executeCommand(T parameterObject) (since 1.4.1)
```

The method has the following behavior:

- The business logic is implemented in this method.
- If something fails, `executeCommand()` returns `CommandTransition.FAILURE`.
- If everything is alright, `executeCommand()` returns `CommandTransition.SUCCESS`.

These methods are made for executing either standalone or in a chain, which runs until one Command fails.

## ChainCommand

ChainCommand objects have a additional method:

```
CommandTransition executeCommandAsChain(T parameterObject)
```

The behavior of `executeCommandAsChain()` is quite different to the usual `executeCommand()` method. In this case the command is aware of the chain and can decide if the whole work of the chain is done or if the next link of the chain should overtake.

For this the `CommandTransition` is:

- `NEXT` work is done, the next chain can overtake.
- `DONE` if everything is fine and the work of the whole chain is done.
- `FAILURE` if an error occurred and it makes no sense, that the next command should overtake.

The last situation often comes if the first command should load a resource and the resource is not available. In this case it makes no sense to continue.

If you look in the `AbstractDefaultChainCommand` class you see a translation between command and chain which might be confusing:

```
@Override
public CommandTransition executeCommandAsChain(T parameterObject) {
    final CommandTransition result = executeCommand(parameterObject);
    if (result == CommandTransition.SUCCESS)
        return CommandTransition.NEXT;
    return CommandTransition.DONE;
}
```

A Command result of `SUCCESS` means: Everything was OK, the next command can be triggered even a `FAILURE` means that an exception has occurred and the execution of the chain should be stopped. This means: If the whole logic is encapsulated in the `executeCommand()` method, sometimes the command has to return `FAILURE`, even everything is OK, because in terms of `FAILURE` sometimes the next command has to overtake. So it might be a good approach as well leaving the `executeCommand()` method blank and implement the logic in the `executeCommandAsChain()` method.

## CommandContainer

The `CommandContainer` object couples commands together. The execution of commands works as follow:

- `executeCommand()` has the same behavior. Instead of throwing an exception `CommandTransition.FAILURE` should returned, `CommandTransition.SUCCESS` otherwise.
- `executeCommandAsChain()` has the same behavior but the return value is either `NEXT` (next command should overtake), `DONE` (work is done) or `FAILURE`, some error occurred and stops the whole chain.

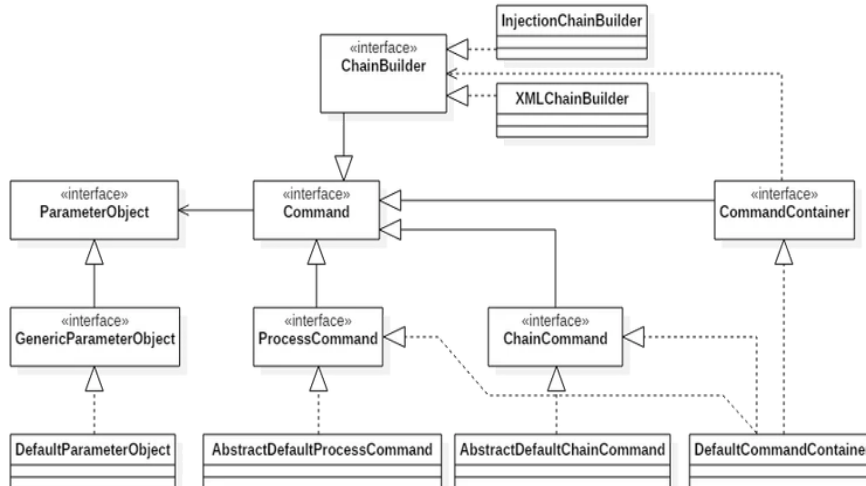
## Composite Pattern

Even a `CommandContainer` object is a `Command` as well you can add `Command` and `CommandContainer` as well in a single container. In addition you've methods to order `Command` objects in the container via priority. If two `Command` objects share the same priority the first inserted `Command` will win. [Browse the java doc to go in depth.](#)



## About the architecture

The architecture of the simple command framework is - as named - simple. It comes with a couple of interfaces and some default implementation.



For each purpose you've a dedicated interface and sometimes an abstract implementation. You can pass a parameter object among the whole chain which has to fulfill the `ParameterObject` interface. The `ParameterObject` interface is just a marker for those objects.

The framework follows the [composite pattern](#). You've a second interface which is a command as well but works as a container of commands. You can create chains with commands and command container as well.

For each interfaces you've default implementations which you can use for your own purposes.

The `AbstractDefaultChainCommand` provides an empty `execute()` method but an implementation of the chain of responsibility framework. The `executeAsChain()` method calls the `execute()` method. If there is no exception thrown it returns true. True means, the work hasn't been done and the next command should be called. If the `execute()` method throws an exception, it is an indicator that the execution of the chain should be stopped. For this the `executeAsChain()` method results false.

This is a good approach if the execution of the chain has to be stopped because an error has occurred. It may be a bad approach, if you want to stop because everything is fine. In this case you've to throw an exception: Everything is fine, the work is done. This is a little bit an anti-pattern but unfortunately there is no "positive exception" in Java.

The `DefaultCommandContainer` has convenient methods to add commands to a container. Either in natural order (just add it) or via priority. If you add two commands with the same priority the first inserted command is executed first. Even the container has the same behavior as a simple command you can execute the container in the same way as a command. You can add a container in a container as well.

For bypassing parameter objects you may want to use the `GenericParameterObject` interface, which provides methods to add key-value pairs into the object. An implementation of that is the `DefaultParameterObject`.

### A brief description of the parts of the framework

**Command:** Interface for wrapping an algorithm. Everything is a command.

**ChainCommand:** Interface for a command which works in the second mode: [Chain of Responsibility](#).

**ProcessCommand:** Interface for a command which works in the third mode: Process Mode.

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

**ParameterObject:** Marker interface for a parameter object. `ParameterObject` objects are the glue between different commands. Because of the concept of these objects the whole framework is thread-safe and supports re-entrant algorithms.

**GenericParameterObject:** Interface for a generic parameter object that holds a list of keys and values. A better approach is to create own type-safe parameter objects 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 parameter object.

**AbstractDefaultChainCommand:** 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.

**AbstractDefaultProcessCommand:** An Implementation which provides a generic method for fulfilling the process mode. Just call `executeAsProcess()`. All commands will be executed in the right way.

**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 the *composite pattern*. The **DefaultCommandContainer** actually implements all three modes.

**DefaultParameterObject:** An implementation of the **GenericParameterObject** interface to work with generic data in a generic data structure. This is not recommended because it is not type safe.

**ChainBuilder:** The **ChainBuilder** is an interface to build chains from different sources e.g. XML, java configured or any other. Concrete classes have to implement the **buildChain()** method.

**InjectionChainBuilder:** **ChainBuilder** which takes a Spring configuration or any other DI framework to build the actual chain.

**XMLChainBuilder:** **ChainBuilder** which takes an XML configuration to build the actual chain.

## Best practices

Inhalt	Verweise
<ul style="list-style-type: none"><li>• Use your own ParameterObject / Context</li><li>• Use the factory pattern to create new Command objects.</li><li>• Place the whole business logic in the executeCommand() method</li><li>• Deal with default implementations</li><li>• Configure Chains via XML / Spring</li></ul>	
	STATUS

In this section there are some best practices to work with the Simple Command Framework.

⚠ This section is still in progress

### Use your own ParameterObject / Context

It is just a question of usage:

- If you have a library of commands which you want to stick together in various project it is better to deal with generic parameter objects. In this case you can use the default implementation `DefaultParameterObject`. The disadvantage of this approach is that you've to deal with strings. Recommendation: **Use string constants to work with keys**. Every command should have constants for the possible keys inside.

```
public class SecondCommand extends AbstractDefaultCommand<GenericParameterObject> {

    public static final String helloworld_message = "helloworld.message";

    public static final SecondCommand getInstance() {
        return new SecondCommand();
    }

    @Override
    public CommandTransition executeCommand(GenericParameterObject parameterObject) {
        String message = parameterObject.getAsString(helloworld_message);
        message += " How are you?";
        parameterObject.put(helloworld_message, message);
        return CommandTransition.SUCCESS;
    }
}
```

- If you deal with parameter objects in a own chain you should provide a parameter object with get- and set-methods because it is safe. You can use any object for parameter objects still objects which are generated from other frameworks.

### Use the factory pattern to create new Command objects.

It is good practice to create domain objects via the factory pattern. You may provide an `getInstance()` method in every command or a factory object which offers methods for creation of all commands. Factory pattern gives you the opportunity to change the behavior of object creation. Because SCF works with parameter objects it is also thread safe to work with singletons across multiple threads.

### Place the whole business logic in the executeCommand() method

### Deal with default implementations

### Configure Chains via XML / Spring

# Releasenotes

## Inhalt

- Introduction
- Version 1.5.0 Releasedate: 05.09.2017
  - The following issues are resolved:
    - SCF-15: Change the execute and executeAsChain Method to a Transition Enumeration
      - Problem regarding execute():
      - New behavior:
      - Problem regarding executeAsChain()
      - New behavior:
    - SCF-16: Make buildChain() protected
  - Other solved issues:
    - SCF-2: Reorganisation of unit tests
    - SCF-5: Remove sonar issues
    - SCF-7: Remove non-javadoc issues in comments
    - SCF-8: Reorganize package structure in test (remove the commons package)
    - SCF-10/SCF-11/SCF-12/SCF-13/SCF-14: Providing a "learning by example" documentation for a better understanding
    - SCF-15: Provide a default Implementation for AbstractDefaultProcessCommand.executeAsChain()
- Version 1.5.1 Releasedate: 22.09.2017
  - SCF-22: Introduce an AbstractDefaultCommand to avoid implementing legacy methods
- Version 2.0.0 Releasedate: 15.01.2018
  - The following issues are resolved:
    - SCF-18 Remove the ChainBuilder interface in 2.0.0
    - SCF-19: Provide a factory method for a GenericParameterObject
    - SCF-23: Change CommandTransition.SUCCESS TO NEXT
    - SCF-24: Remove deprecated methods: executeCommand and executeCommandAsChain
    - SCF-31: Use standard jax parser to parse xml structure
    - SCF-34 Migrate JUnit to JUnit 5
- Version 2.0.1 Releasedate: 11. 04.2019
- Version 2.0.2 Releasedate: 28.03.2021

## Verweise

Current JavaDoc

STATUS

## Introduction

The current release (currently 2.0.1) is found in the master branch of the software.

The release under construction is found in the develop branch. Both branches have the following quality rules:

- The software compiles
- 100% code coverage
- No technical debt with sonarQube analysis

New things will be developed in branches. The software checked in in a branch has no quality issues.

## Version 1.5.0 Releasedate: 05.09.2017

### Commit Info

```
commit:
SHA-1: f36a9dd783eda5c83c4e79f58c138dd9578d8ef8
Tag Info: object f36a9dd783eda5c83c4e79f58c138dd9578d8ef8
tag 1.5.0
```

The main focus of the 1.5.0 release is the enhancing of the interfaces Command and ChainCommand. The semantics wasn't clear at all. Especially the executeAsChain() methods returns a boolean and it is hard to understand in which situations true or false will be returned.

The following issues are resolved:

### SCF-15: Change the execute and executeAsChain Method to a Transition Enumeration

Problem regarding execute():

The execute(T parameterObject) method returns void. So actually you've to throw an exception if something goes wrong. Best practice is to implement the whole business logic into this method, for other purpose (chaining,

process handling) as well. Regarding chaining you had to throw an "everything" is OK exception, which makes no sense.

#### New behavior:

There is a new method called `executeCommand(T parameterObject)`. It returns an `CommandTransition` enumeration. The execute method should return:

- *SUCCESS*, if everything was fine during execution.
- *FAILURE*, if there was an error.

The old `execute(T parameterObject)` method is still there but marked as deprecated.

#### Problem regarding `executeAsChain()`

The `executeAsChain(T parameterObject)` returns boolean. That's not a good approach. A chain should be either aborted (if the whole work is done) or succeeded which means the next command should overtake.

#### New behavior:

There is a new method called `executeCommandAsChain(T parameterObject)` which returns an `CommandTransition` enumeration as well. This method should return:

- *SUCCESS*, if the next chain should be overtake
- *DONE*, otherwise, the chain should be stopped because work is done.

The old `executeAsChain(T parameterObject)` method is still there but marked as deprecated. It will be removed in the 2.0 version of the framework.

#### SCF-16: Make `buildChain()` protected

The method is public so you can call it. But it makes no sense because the different execute-methods call `buildChain()` before the execution. So a better approach is to make it protected. For this the method was removed in the `ChainBuilder` interface. The interface is still there as a marker interface and will maybe be removed in the 1.6.0 version.

#### Other solved issues:

SCF-2: Reorganisation of unit tests

SCF-5: Remove sonar issues

SCF-7: Remove non-javadoc issues in comments

SCF-8: Reorganize package structure in test (remove the commons package)

SCF-10/SCF-11/SCF-12/SCF-13/SCF-14: Providing a "learning by example" documentation for a better understanding

SCF-15: Provide a default implementation for `AbstractDefaultProcessCommand.executeAsChain()`

## Version 1.5.1 Releasedate: 22.09.2017

commit 782e0182a0f49fcb0c2ac3f126beecce3c1c315a  
Author: Manfred Wolff <wolff@manfred-wolff.de>  
Date: Fri Sep 22 08:15:14 2017 +0200

#### SCF-22: Introduce an `AbstractDefaultCommand` to avoid implementing legacy methods

Because the old legacy methods (`execute()` and `executeAsChain()`) are part of the framework till 1.7.0 you've to implement them. The `AbstractDefaultCommand` provided a default implementation of this method. So actually you should work with the defaults named:

- `AbstractDefaultCommand` for simple execution.
- `AbstractDefaultChainCommand` for the chain of responsibility approach
- `AbstractProcessCommand` for the process behavior.

## Version 2.0.0 Releasedate: 15.01.2018

commit b7eae4c2054304e1cff5f09fe86517063627c404  
Author: Manfred Wolff <wolff@manfred-wolff.de>  
Date: Tue Jan 16 11:38:01 2018 +0100

The following issues are resolved:

#### SCF-18 Remove the `ChainBuilder` interface in 2.0.0

After `buildChain()` was marked protected (SCD-16) the `ChainBuilder` interface was a pure marker interface. So it was removed in this version.

#### SCF-19: Provide a factory method for a `GenericParameterObject`

It is best practice to work with factory methods (never create objects with the new operator).

#### SCF-23: Change `CommandTransition.SUCCESS TO NEXT`

Actually we introduced new states for the transition object:

CommandTransition value	meaning
<i>SUCCESS</i>	Return value of the executeCommand method. If everything worked fine this value should be returned.
<i>FAILURE</i>	Return value of the executeCommand method. If the command fails, this value should be returned.
<i>NEXT</i>	Return value of the executeCommandAsChain method. If the next command should overtake, this value should be returned.
<i>DONE</i>	Return value of the executeCommandAsChain method. If the work of the whole chain is done, this value should be returned

The AbstractDefaultChain command does a mapping of these commands in the default implementation.

```
@Override
public CommandTransition executeCommandAsChain(T parameterObject) {
    final CommandTransition result = executeCommand(parameterObject);
    if (result == CommandTransition.SUCCESS) {
        return CommandTransition.NEXT;
    }
    return CommandTransition.DONE;
}
```

If you use this default implementation you only have to implement the executeCommand method in chain commands as well.

All other issues of this release are documentation issues like the best practice documentation.

#### SCF-24: Remove deprecated methods: executeCommand and executeCommandAsChain

It was still confusing having two methods with almost same semantic but different behavior. For that we decided to delete all deprecations in 2.0.0.

#### SCF-31: Use standard jax parser to parse xml structure

The implementation of the XMLChainBuilder was bad. So we changed it to a standard approach using standard java sax parser. Actually we "ate our own dogfood" in this new implementation using commands to fulfill the job.

#### SCF-34 Migrate JUnit to JUnit 5

All tests are migrated to Junit5. Because of missing PIT support for Junit5 actually no PIT test runs.

### Version 2.0.1 Releasedate: 11. 04.2019

commit 7d229aa977fffcce2562f00526623353dba185d9 (HEAD -> master, origin/master, origin/develop, develop)  
 Author: mwolff <m.wolff@neusta.de>  
 Date: Thu Apr 11 08:48:13 2019 +0200

#### Releasegoal: Changing to Java 11

- ☑ SCF-3 - Change to Java 11 **FERTIG**
- ☑ SCF-4 - Change Jacoco plugin to version Release 0.8.3 **FERTIG**
- ☑ SCF-5 - Because of Mockito #1419 change to Mockito 2.23.4 **FERTIG**

### Version 2.0.2 Releasedate: 28.03.2021

Releasegoal: Refactoring for better quality. Removed some cycles showed in SonarGraph.

⚠ For this the package structure changed from 2.0.1 to 2.0.2.

# Simple Execution Example

Inhalt
<ul style="list-style-type: none"><li>• Specification<ul style="list-style-type: none"><li>• First Iteration<ul style="list-style-type: none"><li>• Epic: As a administrator of the system I need a password policy to ensure that passwords are safe.<ul style="list-style-type: none"><li>• Use Case: As a administrator I want to configure the password length.</li><li>• Use Case: A password should have at least a length of xx characters (x configurable through implementation).</li></ul></li></ul></li><li>• Implementation</li><li>• Discussion</li></ul></li></ul>

Verweise
STATUS

## Specification

### First Iteration

**Epic:** As a administrator of the system I need a password policy to ensure that passwords are safe.

**Use Case:** As a administrator I want to configure the password length.

**Use Case:** A password should have at least a length of xx characters (x configurable through implementation).

**Acceptance criteria:**

*Given:* Password length is set to 19. Password is set to "short to Validate."

*When:* Validator is executed

*Then:* Error message should be: "The Password has to be as least 19 Characters."

*Given:* Password length is set to 19. Password is set to "Long enough to Validate."

*When:* Validator is executed

*Then:* The errormessages are empty

### Implementation

The context in our case is a ParameterObject. Here is the "final" test for this (of course it was implemented in several steps)

LoginParameterTest	> Quelle erweitern
--------------------	--------------------

The Test for the validator is here.

LengthValidatorTest	> Quelle erweitern
---------------------	--------------------

The acutal Parameter Object.

LoginParameter	> Quelle erweitern
----------------	--------------------

The implementation of the Validator

LengthValidator	> Quelle erweitern
-----------------	--------------------

### Discussion

Of course in the first step it is not necessary to take the SCF. But we know (in this case we know) that there are other validators in the next iterations. For this we can start with a simple implementation and then we going to refactor to the Simple Command Framework.