Integration Testing Framework Concept Guide

Table of Contents

1. What is this?

This is a concept guide for me as a developer to write down ideas and conceptional things about the Maven Integration Testing framework.

- How I think things could be done from a user perspective
- How I might implement things
- What kind of limitations I think exist or not
- I taken a deeper look into existing integration tests and check how I could handle that with current development or what's needed to target the issues etc.

WARNING	This is neither the status of the development nor something which is implemented. There are things which already implemented from this guide but they must not.

2. Overview

The expressiveness of tests is a very important part of writing integration tests or test in general. If a test is not easy to understand it is very likely not being written.

Lets take a look into the following code which gives you an impression how an integration test for a Maven Plugins/Maven Extensions/Maven-Core should look like:

```
package org.it;

import static org.assertj.core.api.Assertions.assertThat;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void the_first_test_case(MavenExecutionResult result) {
    assertThat(result)
      .build()
        .isSuccessful()
      .and()
      .project()
        .hasTarget()
          .withEarFile()
            .containsOnlyOnce("META-INF/MANIFEST.MF")
        .log()
          .info().contains("Writing data to file")
      .cache()
          .withEarFile("G:A:V")
          .withPomFile("G:A:V")
          .withMetadata().contains("xxx");
  }
}
```

3. Structuring Integration Tests

3.1. The Test Class(es)

The location of the above integration test defaults to `src/test/java/<package>/FirstMavenIT.java`. The selected name like `<any>IT.java` implies that it will be executed by Maven Failsafe Plugin by convention. This will result in a directory structure as follows:

```
.
└── src/
    └── test/
        └── java/
            └── org/
                └── it/
                    └── FirstIT.java
```

For the defined integration tests we need also projects which are the **real test cases** (Maven projects). This needs to be put somewhere in the directory tree to be easily associated with the test `FirstMavenIT`.

The project to be used as an test case is implied to be located into `src/test/resources-its/<package>/FirstMavenIT` this looks like this:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstIT/
```

But now where to put the separated **test cases**? This can easily achieved by using the **method name** within the test class `FirstIT` which is `the_first_test_case` in our example. This results in the following directory layout:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstIT/
                        └── the_first_test_case/
                            ├── src/
                            └── pom.xml
```

This approach gives us the opportunity to write several integration test cases within a single test class `FirstIT` and also separates them easily.

3.2. Test Case Execution

During the execution of the integration tests the following directory structure will be created within the `target` directory:

```
.
└─────target/
    └──────── maven-its/
        └──────── org/
            └──────── it/
                └──────── FirstIT/
                    └──────── the_first_test_case/
                        ├──────── .m2/
                        ├──────── project/
                        │     ├──────── src/
                        │     ├──────── target/
                        │     └──────── pom.xml
                        ├──────── mvn-stdout.log
                        ├──────── mvn-stderr.log
                        └──────── other logs
```

Based on the above you can see that each test case (method within the test class) has it's own local cache (`.m2/repository`). Furthermore you see that the project is built within the `project` folder. This gives you a view of the built project as you did on plain command line and take a look into it. The output of the built is written into `mvn-stdout.log` (stdout) and the output to stderr is written to `mvn-stderr.log`.

3.3. Parallelization

Based on the previous definitions and structure you can now derive the structure of the test cases as well as the resulting output in `target` directory if you take a look into the following example:

```
package org.it;

import static org.assertj.core.api.Assertions.assertThat;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void the_first_test_case(MavenExecutionResult result) {
     ...
  }
  @MavenTest
  void the_second_test_case(MavenExecutionResult result) {
     ...
  }
  @MavenTest
  void the_third_test_case(MavenExecutionResult result) {
     ...
  }
}
```

The structure of the Maven projects in `resources-its` directory:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstMavenIT/
                        ├── the_first_test_case/
                        │   ├── src/
                        │   └── pom.xml
                        ├── the_second_test_case/
                        │   ├── src/
                        │   └── pom.xml
                        └── the_this_test_case/
                            ├── src/
                            └── pom.xml
```

The resulting structure after run will look like this:

```
.
└─────target/
    └──────── maven-its/
        └──────── org/
            └──────── it/
                └──────── FirstMavenIT/
                    └──────── the_first_test_case/
                    |     ├──────── .m2/
                    |     ├──────── project/
                    |     |     ├──────── src/
                    |     |     ├──────── target/
                    |     |     └──────── pom.xml
                    |     ├──────── mvn-stdout.log
                    |     ├──────── mvn-stderr.log
                    |     └──────── other logs
                    ├──────── the_second_test_case/
                    |     ├──────── .m2/
                    |     ├──────── project/
                    |     |     ├──────── src/
                    |     |     ├──────── target/
                    |     |     └──────── pom.xml
                    |     ├──────── mvn-stdout.log
                    |     ├──────── mvn-stderr.log
                    |     └──────── other logs
                    └──────── the_third_test_case/
                        ├──────── .m2/
                        ├──────── project/
                        |     ├──────── src/
                        |     ├──────── target/
                        |     └──────── pom.xml
                        ├──────── mvn-stdout.log
                        ├──────── mvn-stderr.log
                        └──────── other logs
```

So this means we can easily parallelize the execution of each test case `the_first_test_case`, `the_second_test_case` and `the_third_test_case` cause each test case is decoupled from each other.

to make separated from log files and local cache. The result of this setup is that each test case is completely separated from each other test case and gives us an easy way to parallelize the integration test cases in a simple way.

4. Ideas

4.1. Separation of the cache (aka Local Maven Repository)

`@MavenRepository` should be implemented as separate Extension or separate annotation?

Currently the definition for the cache would be defined in one go with the `MavenJupiterExtension` annotations which implies the following test cases would assume that the cache is defined for all tests which means globally to the given class which in the following is not correct as it is newly defined for the `NestedExample` class. If I redefined the `@MavenJupiterExtension(mavenCache=MavenCache.Global)` on the nested class `NestedExample` it would result into having an other cache for the nested class but not what I wanted to have.

So the cache definition should **not** being made in relationship with the `MavenJupiterExtension` annotation.

```
@MavenJupiterExtension(mavenCache = MavenCache.Global)
class MavenIntegrationExampleNestedGlobalRepoIT {

  @MavenTest
  void packaging_includes(MavenExecutionResult result) {
  }

  @MavenJupiterExtension
  class NestedExample {

    @MavenTest
    void basic(MavenExecutionResult result) {
    }

    @MavenTest
    void packaging_includes(MavenExecutionResult result) {
    }

  }

}
```

The solution would be to have a separate annotation for the `@MavenRepository` to define the cache. So the following code shows directly that the repository is defined on the highest class level which can be inherited automatically. The annotation in its default form defines the repository to be defined in `.m2/repository`. It might be a good idea to make it configurable(?) If we like to change the behaviour in derived class the annotation can be added on the derived classes as well.

```
@MavenJupiterExtension
@MavenRepository
class MavenIntegrationExampleNestedGlobalRepoIT {

  @MavenTest
  void packaging_includes(MavenExecutionResult result) {
  }

  @MavenJupiterExtension
  class NestedExample {

    @MavenTest
    void basic(MavenExecutionResult result) {
    }

    @MavenTest
    void packaging_includes(MavenExecutionResult result) {
    }

  }

}
```

The following gives you an impression of making the repository defined in another directory. (This would overwrite the default.)

```
@MavenJupiterExtension
@MavenRepository(".anton")
class MavenIntegrationExampleNestedGlobalRepoIT {

  @MavenTest
  void packaging_includes(MavenExecutionResult result) {
  }

  @MavenJupiterExtension
  class NestedExample {

    @MavenTest
    void basic(MavenExecutionResult result) {
    }

    @MavenTest
    void packaging_includes(MavenExecutionResult result) {
    }
  }
}
```

The annotation is better decision to be open for later enhancements if we think about separating

repositories for releases, snapshots etc. So this annotation could easily enhanced with parameters like the following:

```
import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
@MavenJupiterExtension
@MavenRepository(releases=".releases", snapshots=".snapshots")
class IntegrationIT {

}
```

4.2. Mock Repository Manager

The Mock Repository Manager is as the name implies a mock for a repository. This is sometimes useful to test things like creating releases Maven Release Plugin or define particular content for remote repositories within integration tests for the Versions Maven Plugin.

In general there are coming up the following questions:

- Based on the parallel nature of those integration tests we need to prevent using the same port for each execution. This needs to be injected into the appropriate test run. Usually we would use `localhost:Port` (Is `localhost` sufficient?).
- A repository manager can be used to deploy artifacts (during a test) into it and afterwards check the content somehow. (For example if checksum have been correctly created and deployed).
- A repository manager could be used to download artifacts from it. ? Test Case? (Reconsider?)
- Reuse of existing repos (filled up with special dependencies) in several tests cases to prevent copying of all artifacts?

```
@MavenJupiterExtension
@MavenMockRepositoryManager
class FirstMavenIT {

  @MavenTest
  void the_first_test_case(MavenExecutionResult result) {
    //
  }

}
```

We need to assume that for the execution of Mock Repository Manager we need to have a `settings.xml` template available which can be filled with the current values and being placed into the resulting test case directory.

After running an integration test with support of the Mock Repository Manager the directory structure looks like the following:

```
.
└─────target/
    └──────── maven-its/
        └──────── org/
            └──────── it/
                │    settings.xml (Template)
                └──────── FirstMavenIT/
                    └──────── the_first_test_case/
                        ├──────── .m2/
                        ├──────── project/
                        │     ├──────── src/
                        │     ├──────── target/
                        │     └──────── pom.xml
                        ├──────── mvn-stdout.log
                        ├──────── mvn-stderr.log
                        ├──────── settings.xml
                        └──────── other logs
```

There are several things to be defined like the source repository which contains artifacts already installed an repository

The default directory where to find artifacts which are already within the repository can be found in a directory called `.mrm` at the same level as the `@MavenMockRepositoryManager` annotation.

The position where we defined the `@MavenMockRepositoryManager` annotation shows us on which level we would like to support the usage of it. The above example defines it on integration test class level which means all methods/nested classes will inherit it by default if not overwritten.

The following examples shows that the mock repository manager will only be used for the single test case `the_second_test_case`.

```
@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void the_first_test_case(MavenExecutionResult result) {
    //
  }

  @MavenTest
  @MavenMockRepositoryManager
  void the_second_test_case(MavenExecutionResult result) {
    //
  }

}
```

If we would like to have a mock repository manager should be used for a larger number of tests we

could define the annotation `@MavenMockRepositoryManager` on a separate class/interface which is implemented/extends from for the classes which should be used.

4.2.1. Implementation Hints

- Maybe we can simply use the mrm modules like `mrm-api`, `mrm-servlet` and `mrm-webapp`.

4.3. Setup Projects

We have in general three different scenarios.

Scenarios

- Project setup for a single test case
- Project setup for a number of test cases.
- Global setup projects which should be executed only once.

4.3.1. Setup Project for single test case

Based on the nested class option in JUnit jupiter it would be the best approach to express that via nested class with only a single test case and an appropriate `@BeforeEach` method which describes the pre defined setup.

```
package org.it;

import static org.assertj.core.api.Assertions.assertThat;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import org.junit.jupiter.api.Nested;

@MavenJupiterExtension
class FirstMavenIT {
  @Nested
  class TestCaseWithSetup {
      @BeforeEach
      void beforeEach(MavenExecutionResult result) {
        //..
      }

      @MavenTest
      void the_first_test_case(MavenExecutionResult result) {
         ...
      }
  }

  @MavenTest
  void the_first_test_case(MavenExecutionResult result) {
     ...
  }

  @MavenTest
  void the_second_test_case(MavenExecutionResult result) {
     ...
  }

}
```

4.3.2. Setup Project for a number of test cases

The best and simplest solution would be to use the `@BeforeEach` annotation. That would make the intention of the author easy to understand and simply being expressed.

The disadvantage of this setup would be to execute a full maven build for the setup project within the beforeEach method for each test case method.

One issue is the question where to put the cache for all those test cases?

One requirement based on the above idea is to use the same cache for the beforeEach and the appropriate test case. What about parallelization? The beforeEach and the particular test case must be using the same cache otherwise we have no relationship between the beforeEach method and

the particular test cases? Is this a good idea? (We have made the assumption if not defined different that each test case is using a separate cache) It could assumed having a global cache for test cases which are within the nested class?

```
package org.it;

import static org.assertj.core.api.Assertions.assertThat;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import org.junit.jupiter.api.BeforeEach;

@MavenJupiterExtension
class FirstMavenIT {

  @BeforeEach
  void beforeEach(MavenExecutionResult result) {
    //..
  }

  @MavenTest
  void the_first_test_case(MavenExecutionResult result) {
     //...
  }

  @MavenTest
  void the_second_test_case(MavenExecutionResult result) {
     //...
  }
  @MavenTest
  void the_third_test_case(MavenExecutionResult result) {
     //...
  }
}
```

Baseds on the previously written the conclusion would be to make it possible to use inheritance between the test classes to express a setup/beforeach for a hierarchie of integration test cases which from my point of view sounds like a bad idea? Need to reconsider?

4.4. General Setup Repositories

General Setup repositories which already contains particular dependencies which are needed for test cases. Here we need to make it possible having a local repository to be pre defined on a test case base or on test class or even on several classes or all tests.

The simplest solution would be to create a directory called something like `.predefined-repo` in a particular directory level which implies that this directory will be used as a repository. This can be taken as a pre installed local cache with particular dependencies etc.

Let us take a look at the example:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstIT/
                        └── the_first_test_case/
                            ├── .predefined-repo
                            ├── src/
                            └── pom.xml
```

This would mean that the `.predefined-repo` contains already installed artifacts etc. which can be used to run a test against this based on the method name `the_first_test_case` this is limited to a single test method.

This can be made a more general thing to define it on a class level like the following:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstIT/
                        ├── .predefined-repo
                        └── the_first_test_case/
                        │   ├── src/
                        │   └── pom.xml
                        └── the_second_test_case/
                            ├── src/
                            └── pom.xml
```

This would mean having a predefined repository defined for all test cases within the whole test class (`the_first_test_case` and `the_second_test_case`).

If we move that directory level up like the following:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    ├── .predefined-repo
                    └── FirstIT/
                        └── the_first_test_case/
                            ├── src/
                            └── pom.xml
```

This would mean that the predefined repository is available for all integration test classes within the whole package inclusive all sub packages.

4.5. General Setup Repositories incl. Snapshots

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstIT/
                        └── the_first_test_case/
                            ├── .pre-release-repo
                            ├── .pre-snapshot-repo
                            ├── src/
                            └── pom.xml
```

This would mean that the `.pre-release-repo` contains already installed artifacts etc. The `.pre-snapshot-repo` contains snapshots of particular aritifacts.

To get above usable in Maven you have to have a `settings.xml` which contains the appropriate configuration which looks like this:

We have to define the `central` repo and the snapshot repo. This will limit the access of this build to outside repositories.

settings.xml

```
<settings>
    <profiles>
        <profile>
            <id>it-repo</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
                <repository>
                    <id>local.central</id>
                    <url>file:///Users/xxx/.m2/repository</url>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </repository>
                <repository>
                    <id>local.snapshot</id>
                    <url>file:///Users/xxxx/project/m2snapshots</url>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </repository>
            </repositories>
            <pluginRepositories>
                <pluginRepository>
                    <id>local.central</id>
                    <url>file:///Users/khmarbaise/.m2/repository</url>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </pluginRepository>
            </pluginRepositories>
        </profile>
    </profiles>
</settings>
```

5. Real Life Examples

Within this chapter we describe different integration test cases which are done in integration tests with maven-invoker or with other tests for different maven plugins etc. to see if we missed something which is needed to get that framework forward.

5.1. Maven Assembly plugin

5.1.1. Custom-ContainerDescriptorHandler Test Case

https://github.com/apache/maven-assembly-plugin/blob/master/src/it/projects/container-descriptors/custom-containerDescriptorHandler

Example Test case `custom-containerDescriptorHandler` from Maven Assembly Plugin:

```
custom-containerDescriptorHandler (master)$ tree
.
├── assembly
│   ├── a.properties
│   ├── pom.xml
│   └── src
│       ├── assemble
│       │   └── bin.xml
│       └── config
│           ├── a
│           │   └── file.txt
│           └── b
│               └── file.txt
├── handler-def
│   ├── pom.xml
│   └── src
│       └── main
│           └── resources
│               └── META-INF
│                   └── plexus
│                       └── components.xml
├── invoker.properties
├── pom.xml ①
└── verify.bsh
```

① What is the purpose of this pom file?

Based on the `invoker.properties` file this test case is divided into two steps: The first step is to `install` the `handler-def` project into local cache and second run `package` phase on the project `assembly`.

invoker.properties

```
invoker.project.1=handler-def
invoker.goals.1=install

invoker.project.2=assembly
invoker.goals.2=package
```

The question is coming up how can we translate that to the new integration test framework. The simple answer is like this:

CustomContainerDescriptorHandlerIT.java

```
package org.it;

import static com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenRepository;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.TestMethodOrder;

@MavenJupiterExtension
@MavenRepository
@TestMethodOrder(OrderAnnotation.class)
class CustomContainerDescriptorHandlerIT {

  @MavenTest(goals = {"install"})
  @Order(10)
  void handler_ref(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }

  @MavenTest
  void assembly(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
    // check content of the `assembly/target/ directory
    // Details see https://github.com/apache/maven-assembly-
plugin/blob/master/src/it/projects/container-descriptors/custom-
containerDescriptorHandler/verify.bsh
  }

}
```

Currently this test case contains a single issue which means it uses an project which is run as a general setup project from Maven Invoker Plugin. https://github.com/apache/maven-assembly-plugin/tree/master/src/it/it-project-parent

Based on this setup you will get separated log files for each run in it's own directory not concatenated into a single file.

5.1.2. Grouping Test Cases

This will result in grouping tests within the single class.

Thinking into another level a test could look like this:

ContainerDescriptorHandlerIT.java

```
package org.it;

import static com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenRepository;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(OrderAnnotation.class)
@MavenJupiterExtension
class ContainerDescriptorsIT {

  @Nested
  @MavenRepository
  class CustomContainerDescriptorHandler {

    @MavenTest(goals = {"install"})
    @Order(10)
    void handler_ref(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
    }

    @MavenTest
    void assembly(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
      // check content of the `assembly/target/ directory
      // Details see https://github.com/apache/maven-assembly-
plugin/blob/master/src/it/projects/container-descriptors/custom-
containerDescriptorHandler/verify.bsh
    }
  }

  @Nested
  @MavenRepository
  class ConfiguredHandler {
```

```
    @MavenTest(goals = {"install"})
    @Order(10)
    void handler_ref(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
    }

    @MavenTest
    void assembly(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
      // check content of the `assembly/target/ directory
      // Details see https://github.com/apache/maven-assembly-
plugin/blob/master/src/it/projects/container-descriptors/custom-
containerDescriptorHandler/verify.bsh
    }
  }

}
```

5.2. Versions Maven Plugin

5.2.1. The Test case Example 1

Several of the integration test cases for the Versions Maven Plugins are using the following content for the `invoker.properties` (or very similar)

invoker.properites

```
invoker.goals=${project.groupId}:${project.artifactId}:${project.version}:compare-
dependencies
invoker.systemPropertiesFile = test.properties
```

and the `test.properties` files looks like this:

test.properties

```
remotePom=localhost:dummy-bom-pom:1.0
reportOutputFile=target/depDiffs.txt
```

so the first part in `invoker.properties` which contains `invoker.goals` means to call Maven like this:

```
mvn ${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies
```

where a placeholder `${project.groupId}` is being replaced with the `groupId` of the project (plugin) which the tests should run on. `${project.artifactId}` will be replaced with the `artifactId` and `${project.version}` with the version of the project. In the end a call will look like this:

```
mvn org.codehaus.mojo:versions-maven-plugin:2.7.0-SNAPSHOT:compare-dependencies
```

Now let us come to the `test.properties` which is simply being translated to the following: (backslashes are only added to make it more readable)

```
mvn org.codehaus.mojo:versions-maven-plugin:2.7.0-SNAPSHOT:compare-dependencies \
    -DremotePom="localhost:dummy-bom-pom:1.0" \
    -DreportOutputFile="target/depDiffs.txt"
```

Now let us assume we could translate that very easy:

FirstIT.java

```
@MavenJupiterExtension
class CustomContainerDescriptorHandlerIT {

  @MavenTest(goals = {
"${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies"})
  void calling_a_goal(...) {
    ...
  }

  @MavenTest(goals = {
"${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies"},
    systemProperties = {
      "remotePom=localhost:dummy-bom-pom:1.0",
      "reportOutputFile=target/depDiffs.txt"
  })
  void calling_a_goal_with_sytem_properties(...) {
    ...
  }
}
```

Now I'm asking why do we use this bunch of placeholders `${project.groupId}:${project.artifactId}:${project.version}`. Only based on the fear that the groupId or artifactId or version could change. A change in groupId or artifactId is very rare. I've never seen a change in groupId nor artifactId in plugin projects. What changes more often is the version of the artifact which means with each release. So it would make sense to define for the version a placeholder like `${project.version}`.

NOTE	Based on the approach to simply read the `pom.xml` file of the project under test this can be solved easily. This makes it also possible to run the IT within the IDE.

5.2.2. Testcase

5.2.3. Test Case IT-SET-001

The following `invoker.properties` describes a test case which comprises of two consecutive calls of Maven on the same directory (project):

it-set-001

```
invoker.goals.1=${project.groupId}:${project.artifactId}:${project.version}:set
-DnewVersion=2.0
invoker.nonRecursive.1=true
invoker.buildResult.1=success

invoker.goals.2=${project.groupId}:${project.artifactId}:${project.version}:set
-DnewVersion=2.0 -DgroupId=* -DartifactId=* -DoldVersion=*
invoker.nonRecursive.2=true
invoker.buildResult.2=success
invoker.description.2=Test the set mojo when the new version is the same as the old
version, using wildcards. This kind of build used to fail accourding the issue 83 from
github.
```

The above means to execute on the same project several executions of maven calls. This breaks at the moment the idea of separation of the builds by method.

This might be expressed by using `@MavenProject` annotation which defines such thing. The name of the method can be a sub directory which contains `mvn-stdout.log` etc.

NOTE	We should make the `@MavenRepository` part of `@MavenProject`.

ITSETIT.java

```
@TestMethodOrder(OrderAnnotation.class)
@MavenJupiterExtension
class setVersionIT {

  @Nested
  @MavenRepository
  @MavenProject("set_001") //Define the project to be used. Only valid on Nested class
or root class.
  @DisplayName("Test the set mojo when the new version is the same as the old version,
using wildcards. This kind of build used to fail accourding the issue 83 from github.
")
  class Set001 {
    @MavenTest(options = {"-N"}, goals = {
"${project.groupId}:${project.artifactId}:${project.version}:set"}
      systemProperties = {"newVersion=2.0"})
    @Order(10)
    void first_test(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
    }

    @MavenTest(options = {"-N"}, goals = {
"${project.groupId}:${project.artifactId}:${project.version}:set"}
      systemProperties = {"newVersion=2.0", "groupId=*", "artifactId=*",
"DoldVersion=*"})
    @Order(20)
    @DisplayName("where setup two is needed.")
    void second_test(MavenExecutionResult result) {
      assertThat(result).isFailure();
    }

}
```

5.2.4. Test Case UPDATE-CHILD-MODULES-001

Think about the following:

invoker.properties

```
# first check that the root project builds ok
invoker.goals.1=-o validate
invoker.nonRecursive.1=true
invoker.buildResult.1=success

# second check that adding the child project into the mix breaks things
invoker.goals.2=-o validate
invoker.nonRecursive.2=false
invoker.buildResult.2=failure

# third fix the build with our plugin
invoker.goals.3=${project.groupId}:${project.artifactId}:${project.version}:update-
child-modules
invoker.nonRecursive.3=true
invoker.buildResult.3=success

# forth, confirm that the build is fixed
invoker.goals.4=validate
invoker.nonRecursive.4=false
invoker.buildResult.4=success
```

This could be translated into the following:

UpdateChildModuleIT.java

```
@TestMethodOrder(OrderAnnotation.class)
@MavenJupiterExtension
class UpdateChildModuleIT {

  @Nested
  @MavenRepository
  @MavenProject("name-x") //Define the project to be used.
  class One {
    @MavenTest(options = {"-o"}, goals = { "validate" })
    @Order(10)
    void first_test(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
    }

    @MavenTest(options = {"-o"}, goals = { "validate" })
    @Order(20)
    @DisplayName("where setup two is needed.")
    void second_test(MavenExecutionResult result) {
      assertThat(result).isFailure();
    }

    @MavenTest(options = {"-N"}, goals = {
"${project.groupId}:${project.artifactId}:${project.version}:update-child-modules" })
    @Order(30)
    @DisplayName("where setup two is needed.")
    void third_test(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
    }

    @MavenTest(goals = { "validate" })
    @Order(10)
    void forth_test(MavenExecutionResult result) {
      assertThat(result).isSuccessful();
    }
  }

}
```

6. Open Things

Things which currently not working or net yet tested/thought about

- ☐ A build/tool(s) running without relation to Maven? This means we only need to define what we start simply a different thing than Maven. Would we like to support this?
- ☐ POM Less builds currently not tried. Calling only a goal like `site:stage` ?
- ☐ Setup projects which should be run

- ☐ General Setup repositories which already contain particular dependencies which are needed for test cases. Here we need to make it possible having a local repository to be pre defined on a test case or on a more general way.
- ☐ Support for a mock repository manager (mrm) to make tests cases with deploy/releases etc. possible. A thought might be to integrate the functionality of mrm into this extension and somehow configure that for the test cases?
- ☐ Support for Mock Repository Manager

7. Ideas

7.1. IDE Integration

- If we change the code of a plugin within the IDE the Integration test will not test against the changed code only against the latest built jar files. The IDE compiles the changes code into `target/classes`... something about the classpath?
- Tricky idea: If we start an integration test we could check if the class files are newer than the created jar file and build via `mvn package` the project under test and copy them into the appropriate directories and then run the test as usual.
- Assertion Idea

```
assertThat(result)
  .project()
    .hasTarget()
      .withEarFile()
        .containsOnlyOnce("META-INF/MANIFEST.MF");

assertThat(result)
  .project()
    .log()
      .info().contains("Writing data to file");

assertThat(result)
  .cache()
      .hasEarFile("G:A:V")
      .hasPomFile("G:A:V")
      .hasMetadata("G:A")
        .contains("xxx");
```

7.2. Log Assertion

We have at the moment at least three different outputs:

1. The stdout as `mvn-stdout.log`
2. The stderr as `mvn-stderr.log`

3. The list of used command line parameters `mvn-arguments.log`

filename.java

```
assertThat(result).isSuccessful().out()...
assertThat(result).out().warn()
```