Software Maintence and Evolution Project
Test-Describer in CI/CD

Daniel Demeter
Massimo Bertocchi

1 Introduction

For our project, Massimo and I worked on extending the tool called Test-Describer. Its purpose is to annotate automatically generated tests in Java projects, so that developers can better understand what the tests are actually testing. Better understanding results in a higher rate of bug prediction, which results in faster development, and developers' time being saved.

2 Objectives

Our goals for this project were to incorporate Test-Describer into automation pipelines to better streamline a development workflow. We chose two of the most popular tools: Jenkins and Concourse. With both, we were able to achieve an automated workflow.

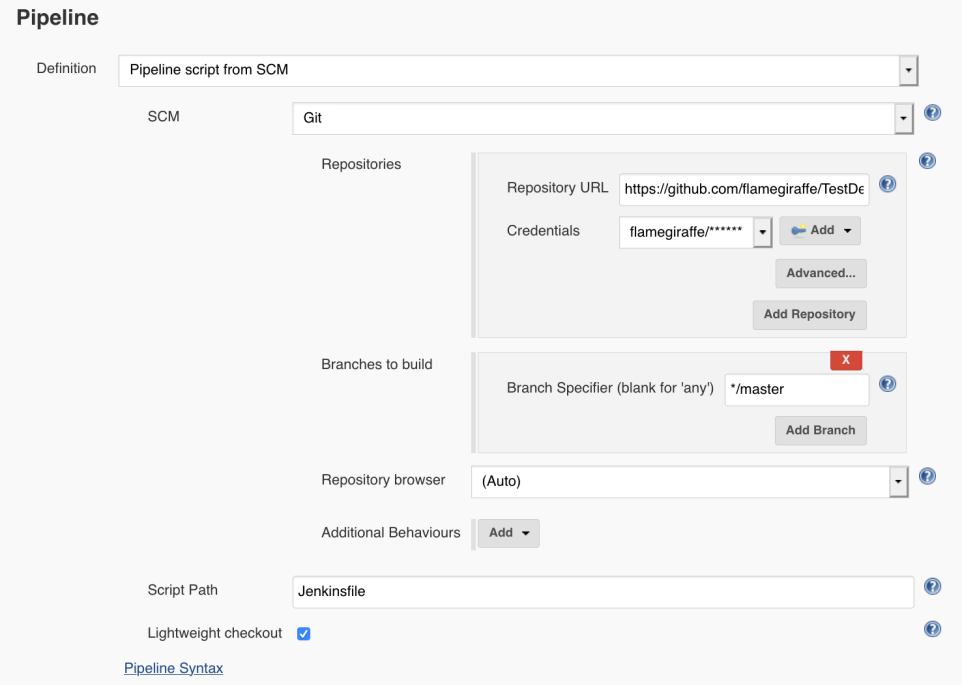
3 Process

Below, we will explain all the steps necessary to replicate our process, as well as the pitfalls we should avoid. First of all, we need to have a Java project with uncommented unit tests inside. This project needs to have its own public GitHub repository.

3.1 Jenkins

Once Jenkins is set up and running on a local server, we need to create a *Pipeline*. In its setup, we need to enable the checkbox for *GitHub project*, and supply its URL to Jenkins.

Then, under the Pipeline settings, we need to check *Pipeline script from SCM* from the dropdown menu for *Definition*. As our SCM (Source Control Management), we need to choose Git. As our *Repository URL*, we once again need to supply our GitHub project's URL.



With this setup, the pipeline will do the following: whenever we build our pipeline, it will fetch our project from the GitHub URL we provided, try and find a Jenkinsfile in the directory, and execute the steps outlined in that Jenkinsfile. As such, the next step we need to take is to include a Jenkinsfile in our Java project's root directory. This is not a text file, it does not have an extension. On OSX machines, such a file can be created easily from the Terminal with the command *"touch Jenkinsfile"*, and on Windows machines, in the Command Prompt with the command *"type nul > Jenkinsfile"*.

For our pipeline in Jenkins to work, we first need to install Docker as well. Docker needs to be running the entire time Jenkins is running as well, for everything to go smoothly.

Furthermore, we need access to a runnable jar of the Test-Describer project. Currently, after building the jar myself, it lives in my GitHub account. For future work, it would make sense to have a dedicated, unified repository for Test-Describer.

Now that we have everything set up, we need to create the steps for our pipeline inside our Jenkinsfile. I decided to have two *stages*, called 'fork' and 'test'. In the fork stage, we simply clone the repository of Test-Describer GitHub project if we don't already have it, or update it to its most current version if we do. In the test stage, we run the jar from the project we just cloned. However,

this needs further steps.

Because a jar is a java executable file, it needs java to work. As such, we need to include a *JDK* in the tools section of our Jenkinsfile. We also need to go to *Manage Jenkins* > *Global Tool Configuration* > *JDK*. Here, we need to tell Jenkins where to find the latest Java development kit. The following setup worked for me, but bear in mind that as Jenkins is interacting with a third-party system (Oracle), this can break at any time.

Now that we have our java environment set up as well, we need one last things before our Jenkinsfile should be ready to go. In our Java project, we need to have a text file that contains all of the paths to the files and folders Test-Describer needs. They need to be in order, such that the first line is the path to the source folder, the second is to the bin folder, and every pair of lines thereafter are the paths to corresponding class files and test files. The Test-Describer implementation reads these in this exact order, so any deviation from this order will not work.

Finally, our Jenkinsfile should be ready. I decided to separate the scripts for each stage into individual bash files. Below are my scripts for the Jenkinsfile, the fork stage script, and the test stage script, respectively.

```
pipeline {
    agent any
    tools {
        jdk 'JDK11'
    }
    stages {
        stage('fork') {
                steps {
                        sh 'echo "this is from the jenkinsfile"'
                        sh 'bash buildbash.sh'
                }
        }
        stage('test') {
                steps{
                        sh 'bash testbash.sh'
                }
        }
    }
}
```

```
#!/bin/bash

if [ ! -d "SME19_fork" ]
then
        git clone https://github.com/flamegiraffe/SME19_fork.git
        echo 'getting a fork!'
else
        git -C SME19_fork fetch --all
        git -C SME19_fork reset --hard origin/master
        echo 'already got a fork'
fi
```

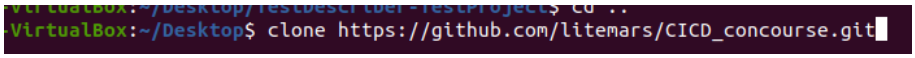
```
#!/bin/bash

echo "starting test from bash script"
echo $JAVA_HOME
java -jar SME19_fork/workspace_implementation/TD-test.jar paths.txt
```

When we build our Jenkins pipeline, what we should see is that for every test file we supplied Test-Describer, another is created with *withDescription* tagged on to its name's end.

To further automate the process, we can create git hooks to automatically build our Jenkins pipeline whenever we commit or push. Such processes are well-documented online and are easily usable with this pipeline, but out of the scope of this project.

3.2 Concourse

As for Jenkins, also Concourse runs on our local server, otherwise we could run on a private server if we would like to do. First of all we have to clone a Github repository in your computer.



Here you can find out step by step how to set up your concourse project with the Test-Describer.

There are some important files to run concourse. The first file that we're going to face on is the a docker-compose.yml. As we know Concourse runs as a Server and you have to deploy concourse using Docker container. In this file there are all setting in order to deploy your container and you have the credential, that you can modify, to access into the server afterwards.

Now there is a particular CLI that concourse use to interact with the system, it's called fly. You can download it from the main page of concourse or directly from your localhost after your docker is up. With this CLI you can configure the pipeline and see the status of the system.

The concourse core is the pipeline, so you need to configure it to write an yml file with all the necessary configurations, as it follows.

```yaml
resources:
- name: repo
  type: git
  source:
    uri: https://github.com/litemars/CICD_concourse.git

jobs:
- name: java
  plan:
  - get: repo
    trigger: true
  - task: java
    file: repo/concourse/task/task.yml
```

In the first part, you have to specify the Git repository which you want to work on and assign an Alias in order to use afterwards. Thus, you should change the path to your Github repository.

In the second part there are the jobs, it's the name for the actual steps of the pipeline, you can define how many jobs as you want, in my example I just set one job and it's defined in another yml called task.yml. Therefore, if you want to create more process, you have to copy/paste the last part, give it a different name and change the yml path. There is a line called "trigger"

```
platform: linux

image_resource:
  type: docker-image
  source:
    repository: openjdk

#What resources from the pipeline we want as input
inputs:
- name: repo
#Run the script that exists in the resource we requested
run:
  path: repo/concourse/script/start.sh
```

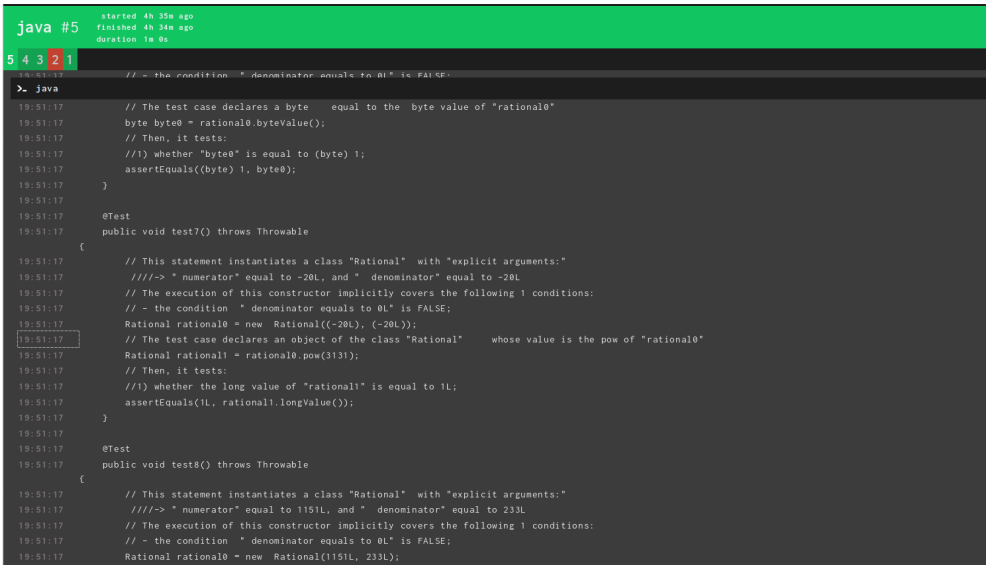
In this file there a few setups, as I said before Concourse runs in a docker container and here we have to define which type of docker is and which instance of docker is. Lighter the instance is faster the process runs, for instance I define the openjdk such as image. There is a catalogue in docker website where you can find the image more suitable for you. Then, you define the real process as a bash file, this file will run in your docker container that you decided before, and it will run like a normal bash file. Therefore, we can write your own bash and run the command that you need. Finally, we have configured our concourse

```
#!/bin/sh

echo "Hello-TD!"

pwd
cd repo/concourse/script

java -jar TD-test.jar paths.txt
```

environment, in the main Folder there are 2 directory "concourse" where there are the setting and test. In this last directory there is the project the we want

to test and what you have to do is swap this folder with the your own project. Now that we have set up our project, we need to configure the path file as in Jenkins. In the script folder, we need to have a text file that contains all of the paths to the files and folders Test-Describer needs. They need to be in order, such that the first line is the path to the source folder, the second is to the bin folder, and every pair of lines thereafter are the paths to corresponding class files and test files. The TestDescriber implementation reads these in this exact order, so any deviation from this order will not work.

In the end, we have create our instance of concourse and now to see how our pipeline goes on, we go to localhost where we can find the pipeline, click on that and you can see the Test-Describer works on your own project. All the output will display on our interface on concourse.



4 Post-Presentation Update

The jar now takes as an input an xml file instead of a text file. It should contain the paths, as well as a verbose-flag to control the level of logs from TestDescriber. An example of such an xml is below.

paths.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
        <path>
                <src>/Users/Dani/Downloads/workspace2-modified-implementation/Task1/src/</src>
                <bin>/Users/Dani/Downloads/workspace2-modified-implementation/Task1/bin/</bin>
                <class>org/magee/math/Rational.java</class>
                <testClass>org/magee/math/TestRational.java</testClass>
        </path>
        <verbose>false</verbose>
</input>
```