ODSOFT Project 2024/2025 - CI/CD and Automated Testing

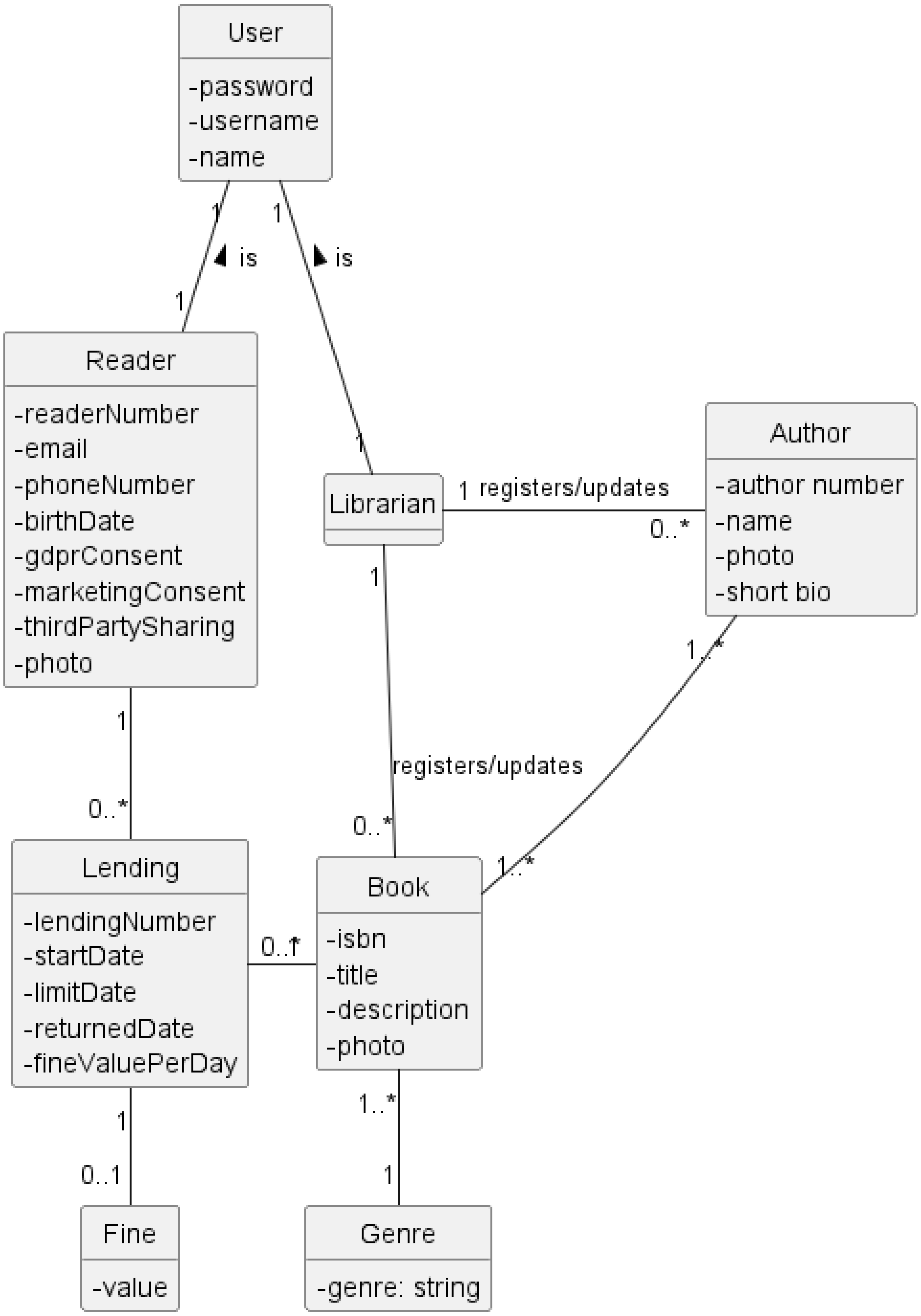
Introduction

This project focuses on automating CI/CD and testing processes for a library management system. The system offers REST endpoints to manage information on books, genres, authors, readers and loans. This project phase aims to improve the system's variability, configurability, reliability and automation.

The main goals include implementing a CI/CD pipeline in Jenkins, covering build, packaging, code analysis, test execution and deployment for local and remote environments. The project also explores advanced testing practices, including unit, integration, acceptance and mutation testing, focusing on code coverage and reporting the results.

System-as-is

Domain Model



Explanation

The domain model for this project consists of some entities and their relationships within a library management system. The User class serves as authentication to limit the access to the application , while the Librarian class represents library staff responsible for managing books and authors. The Genre class categorizes books, and the Book class contains essential details about the books themselves. The Author class has information about the authors, and the Reader class represents the persons that want to borrow some books. Lastly, the Lending class tracks the borrowing of books, managing the lending process to the readers, and the Fine class specifically handles penalties for the ones that don´t return in time the books.

The relationships between these classes illustrate the interactions within the system:

A Librarian can register and update multiple Authors and Books, establishing a one-to-many relationship. Books can be linked to multiple Authors and categorized under a single Genre, showcasing a many-to-many relationship with Authors and a one-to-many relationship with Genre. Each Lending record is associated with a single Reader and a single Book, while also allowing for the possibility of a fine, creating a comprehensive lending management system. Overall, this domain model provides a clear view and understanding of the components and their attributes in the system.

Requirements

Functional requirements describe the functionality of the system.

For this application we can find :

Book:

- I want to register a book.
- I want to update a book's data.
- I want to know the details of a book given its ISBN.
- I want to search books by genre.
- I want to return a book. If the return is overdue I'm fined by the library.

Author:

- I want to register an author.
- I want to update an author's data.
- I want to know an author's detail given its author number.
- I want to search authors by name.

Lending :

- I want to lend a book to a reader.
- I want to return a book. If the return is overdue I'm fined by the library.
- I want to know the details of a lending given its lending number

Reader:

- I want to know an author's detail given its author number.
- I want to search authors by name.
- I want to know the details of a book given its ISBN.
- I want to search books by genre.
- I want to register as a reader.
- I want to update my personal data.
- I want to know a user's detail given its reader number.
- I want to search Readers by name.
- I want to lend a book to a reader.

- I want to return a book.
- I want to know the details of a lending given its lending number.

Quality Requirements (ASR)

ASR: It is an acronym for "Architecturally Significant Requirements". A subset of the non-functional quality requirements impinging directly on the architecture and structure of the system. These sets of requirements do not concern specific functionalities but rather essential features that guarantee the system is efficient, maintainable, and scalable over time. With regards to this library management system, ASRs include the following:

Performance:

It should also be swift and efficient. For example, performance-related ASR ensures that the CI/CD pipeline finishes the whole process, which includes build and test, deployment in less than 5 minutes for quick feedback and a smooth workflow.

Reliability:

The system should be robust and reliable. One target for reliability is having over 90% code coverage from testing. This reduces the chance of any bug reaching production. Static analysis, such as provided by SonarQube, is used to find and reduce potential problems in code, such as "code smells" and unnecessary complexity.

Scalability:

The system should be in a position to support an increase in the number of users and data volumes without its current performance falling. It should be easily configurable and flexible enough to handle minor adjustments in view so that it can support additional modules and larger datasets with less extensive rewriting of code.

Maintainability:

The code structure must support easy maintenance and upgrades. Tools like CheckStyle ensure that the code follows styling standards, enhancing readability and comprehension for other developers. This requirement also covers modularity and the use of coding best practices, allowing the system to be updated with minimal impact on other parts of the codebase.

Documenting and Validating ASRs

To ensure the ASRs are met, the system includes reports and metrics in the CI/CD pipeline, with automated checks, such as:

Test Coverage: Utilizing JaCoCo to generate coverage reports and ensure critical areas of the codebase are adequately tested.

Code Analysis:

Configuring SonarQube to perform checks for complexity, code duplication, and vulnerabilities, providing ongoing reports on code quality.

Pipeline Performance Metrics:

Assessing the execution time of different pipeline stages and adjusting them as needed to meet the under-5-minutes requirement.

In conclusion of this section, we can confirm that the system is not very well prepared to be on production since it has a lot of maintainability issues, low code coverage and a lot of problems detected with jacoco and sonarqube as it will be described below.

Project Goals

1. **CI/CD Automation**
   Configure and integrate Jenkins for a complete pipeline, including:
   - Version control (SCM)
   - SonarQube and CheckStyle
   - Build and packaging (Build and Package)
   - Tests Run
   - Jacoco
   - Deploy

   In this section we will explain our thinking proccess and what we decided in order the results the team had.
2. **Automated Testing**
   - **Unit Tests**: Divided into black-box and white-box testing of domain classes.
   - **Mutation Testing**: Checking test effectiveness via mutation testing.
   - **Integration Tests**: Verifying interactions between controllers, services and repositories.
   - **Acceptance Testing**: Validating expected system behavior.
3. **Performance and Documentation**
   - Document the system's initial state (design through reverse engineering).
   - Critically analyze the pipeline's performance over time, providing evidence of improvements.

Project Structure

- **src/main/java**: Contains the system implementation.
- **src/test/java**: Contains unit, integration and mutation tests.
- **Jenkinsfile**: Configuration file for the Jenkins CI/CD pipeline.

- **Documentation**: File detailing all the team decisions during the project.

CI/CD Pipeline

A pipeline automates the software development process, ensuring code is consistently built, tested and deployed. It integrates continuous integration (CI) and continuous delivery (CD) practices, reducing manual steps, catching bugs earlier. This leads to faster, more reliable deployments and higher code quality.

1. **Source Control Management (SCM)**
   We started by implement the Source Control Management. This stage manages changes to source code over time. It keaps tracking of the code revision. It helps team collaboration by avoiding conflicts through branching and merging. It will mainly insure that the last code is built and tested automatically identifying issues earlier. For this step, we developed a stage that has the following command:

   ```
   stage('SCM') {
           steps {
               checkout scm
           }
       }
   ```

   The checkout scm command will retrieve the source code from the version control system. In this case the team used Git. It will ensure that the pipeline is dealing with the last code version and everything is up to date.
2. **Build and Packaging**
   In this tage, the team focused on compiling the source code and creating a deployable package. The build, has usual, will resolves dependencies, ensuring that the libraries and components are included in the final artifact. This step is one of the most importants because it will transform the source code into an application possible to be run. The team implement a stage dedicated to building and packaging the application:

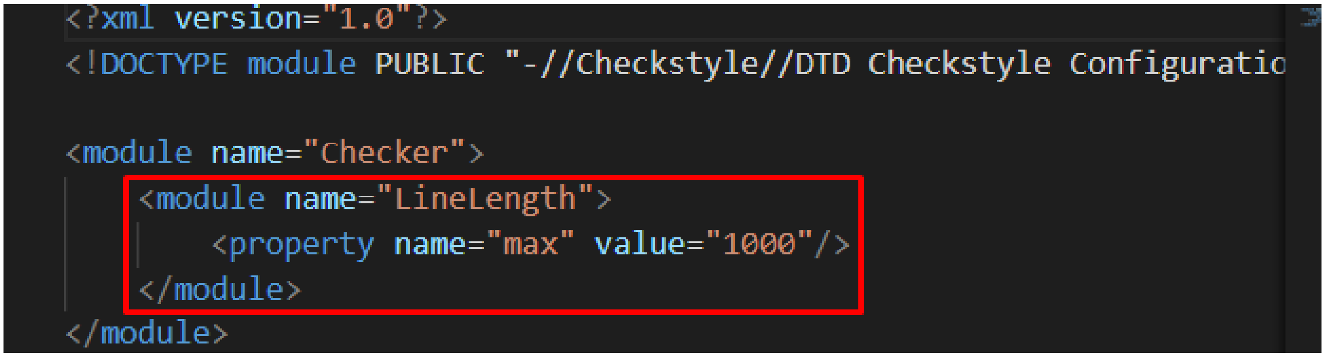
   ```
   stage('Build and Packaging') {
       steps {
           script {
               if (isUnix()) {
                   sh 'mvn clean package'
               } else {
                   bat 'mvn clean package'
               }
           }
       }
   }
   ```

   The **mvn clean package** command invokes Maven which will first clean previous builds and then compiles the source code. This ensure that a fresh build is done everytime the pipeline is run.

3. **Static Code Analysis**
   In this stage, we focused on guarantee the quality and maintainability of the code through the static analysis. This stage will check potential issues, such as code smells, bugs and vulnerabilites. With this step we can improve code quality.
   For this project, we implemented SonarQube and ScheckStyle as our static code analysis.

```
stage('Scan & Checkstyle') {
       parallel {
           stage('Scan') {
               when { expression { !isUnix() } }  // Run this stage only on Windows
               steps {
                   withSonarQubeEnv('sq-odsoft') {
                       withCredentials([string(credentialsId: 'sonar', variable: 'SONAR_TOKEN')])
                           bat "mvn org.sonarsource.scanner.maven:sonar-maven-plugin:3.9.1.2184:so
                       }
                   }
               }
           }
           stage('Checkstyle') {
               steps {
                   script {
                       if (isUnix()) {
                           sh 'mvn checkstyle:checkstyle -Dcheckstyle.failOnViolation=false'
                       } else {
                           bat 'mvn checkstyle:checkstyle -Dcheckstyle.failOnViolation=false'
                       }
                   }
               }
           }
       }
   }
```

We will discuss in another section why we decided to implement this two steps in parallel but for now we will give a brief explanation of what is happening in this stage.
Sonarqube is the first detailed stage and is used for continuous inspection of code quality. It analyzes and provides detailed reports on multiple metrics, including code coverage, complexity, duplications, etc. For this step we had to create in the jenkins a credential with the Id='sonar' and use it as a variable to the command to use. The **mvn sonar:sonar** command uses the token and triggers the Sonar analysis. By analysing the compiled classes defined with the locatin *target/classes* directory, SonarQube will initiathe the analysis and generate the reports.
With the sonarqube, we implemented CheckStyle to guarantee coding standards and best practices. This tool will check the source code with a set of defined rules. This rules are defined in the checkstyle.xml file:



```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC "-//Checkstyle//DTD Checkstyle Configuratio

<module name="Checker">
    <module name="LineLength">
        <property name="max" value="1000"/>
    </module>
</module>
```

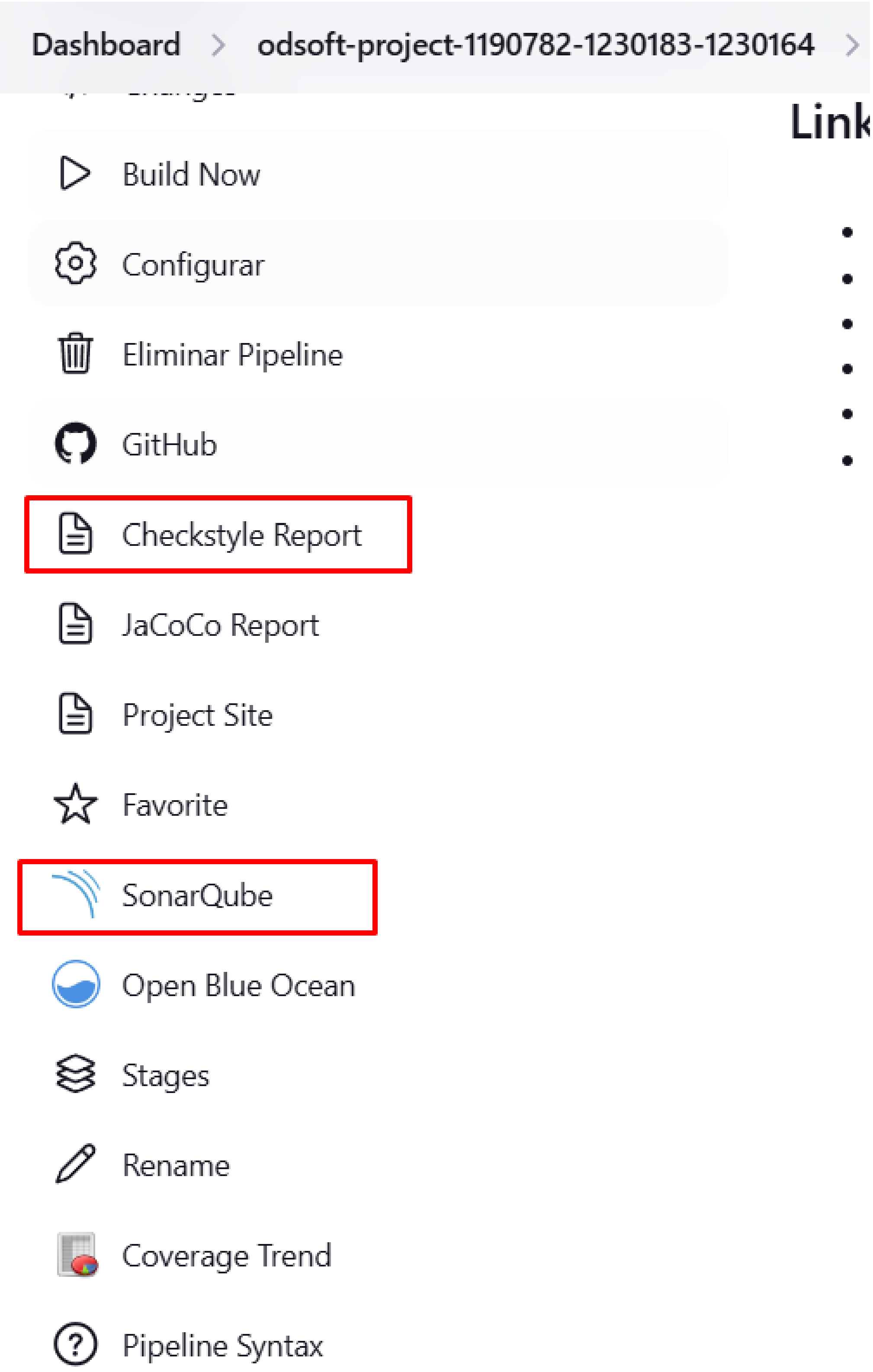
We only defined one rule: LineLength. This is the only rule defined because the source code had a lot of problems. In order for us to confirm that CheckStyle was working properly, this rule was enough. We can see in the image that the max lenght value is set to 1000. This way, it won't return any error. If we set it to 500, for example, it will return 3 errors. With the usage of the flasgs failOnViolation=false, the pipeline always runs properly even if the Checkstyle detect some needed improvements. This way the user can check the report generated and see what problems must be solved but it won't block the pipeline.

After this parallel stages we have the CheckStyle report publish:

```
stage('Publish Checkstyle Report') {
            steps {
                publishHTML([
                    reportDir: 'target/site',
                    reportFiles: 'checkstyle.html',
                    reportName: 'Checkstyle Report',
                    keepAll: true,
                    allowMissing: false,
                    alwaysLinkToLastBuild: true
                ])
            }
        }
```

It will generate a checkstyle.html file in the specified directory: target/site.

In jenkins we can see the SonarQube and CheckStyle report in the left panel of the pipeline run:



4. **Test Execution**

   Testing is one of the most important parts of the pipelines. This stage will guarantee that new introduces features won't break the previous versions and will keep the code functional.

   i. **Unit Testing**:

      Unit tests focus on validating components of the application individually, mainly domain classes. For this, in the pipeline we configured, in parallel, the run of the mutation, opaque and transparent tests. The reason why we decided to do it in parallel will be discussed in another section of this report.

Mutation tests assess the effectiveness of the unit tests by introducing small changes to the codebase.

```
stage('Run Unit Tests') {
     parallel {
         stage('Mutation Tests') {
             steps {
                 script {
                     if (isUnix()) {
                         sh 'mvn test -Dtest=pt.psoft.g1.psoftg1.unitTests.mutationTests.**.*T
                     } else {
                         bat 'mvn test -Dtest=pt.psoft.g1.psoftg1.unitTests.mutationTests.**.*
                     }
                 }
             }
         }
         stage('Opaque and Transparent Tests') {
             steps {
                 script {
                     if (isUnix()) {
                         sh 'mvn test -Dtest=pt.psoft.g1.psoftg1.unitTests.opaqueAndTransparen
                     } else {
                         bat 'mvn test -Dtest=pt.psoft.g1.psoftg1.unitTests.opaqueAndTranspare
                     }
                 }
             }
         }
     }
 }
```

This parallel setup allows to run different unit tests categories concurrently, reducing the pipeline's runtime as will be demonstrated in another section.

ii. **Integration Tests** Integration tests guarant interaction between various component in the system, mainly controllers, services and repositories. The following stage will be reponsible for the integration tests run:

```
stage('Integration Testing') {
    parallel {
        stage('Controllers Testing') {
            steps {
                script {
                    if (isUnix()) {
                        sh 'mvn verify -Dtest=pt.psoft.g1.psoftg1.integrationTests.controllers
                    } else {
                        bat 'mvn verify -Dtest=pt.psoft.g1.psoftg1.integrationTests.controller
                    }
                }
            }
        }
        stage('Services Testing') {
            steps {
                script {
                    if (isUnix()) {
                        sh 'mvn verify -Dtest=pt.psoft.g1.psoftg1.integrationTests.services.**
                    } else {
                        bat 'mvn verify -Dtest=pt.psoft.g1.psoftg1.integrationTests.services.*
                    }
                }
            }
        }
        stage('Repository Testing') {
            steps {
                script {
                    if (isUnix()) {
                        sh 'mvn verify -Dtest=pt.psoft.g1.psoftg1.integrationTests.repository.
                    } else {
                        bat 'mvn verify -Dtest=pt.psoft.g1.psoftg1.integrationTests.repository
                    }
                }
            }
        }
    }
}
```

While running integration tests in parallel the pipeline gets optimized, reducing testing time as demonstrated also in another section.

iii. Coverage report Code coverage reports give visually details about the percentage of code tested. For this, we used JaCoCo to generate this report.

```
stage('Publish JaCoCo Report') {
    steps {
        publishHTML([
            reportDir: 'target/site/jacoco',
            reportFiles: 'index.html',
            reportName: 'JaCoCo Report',
            keepAll: true,
            allowMissing: false,
            alwaysLinkToLastBuild: true
        ])
    }
}
```

It will generate a index.html file with the full report saving it in the directory *target/site/jacoco*.

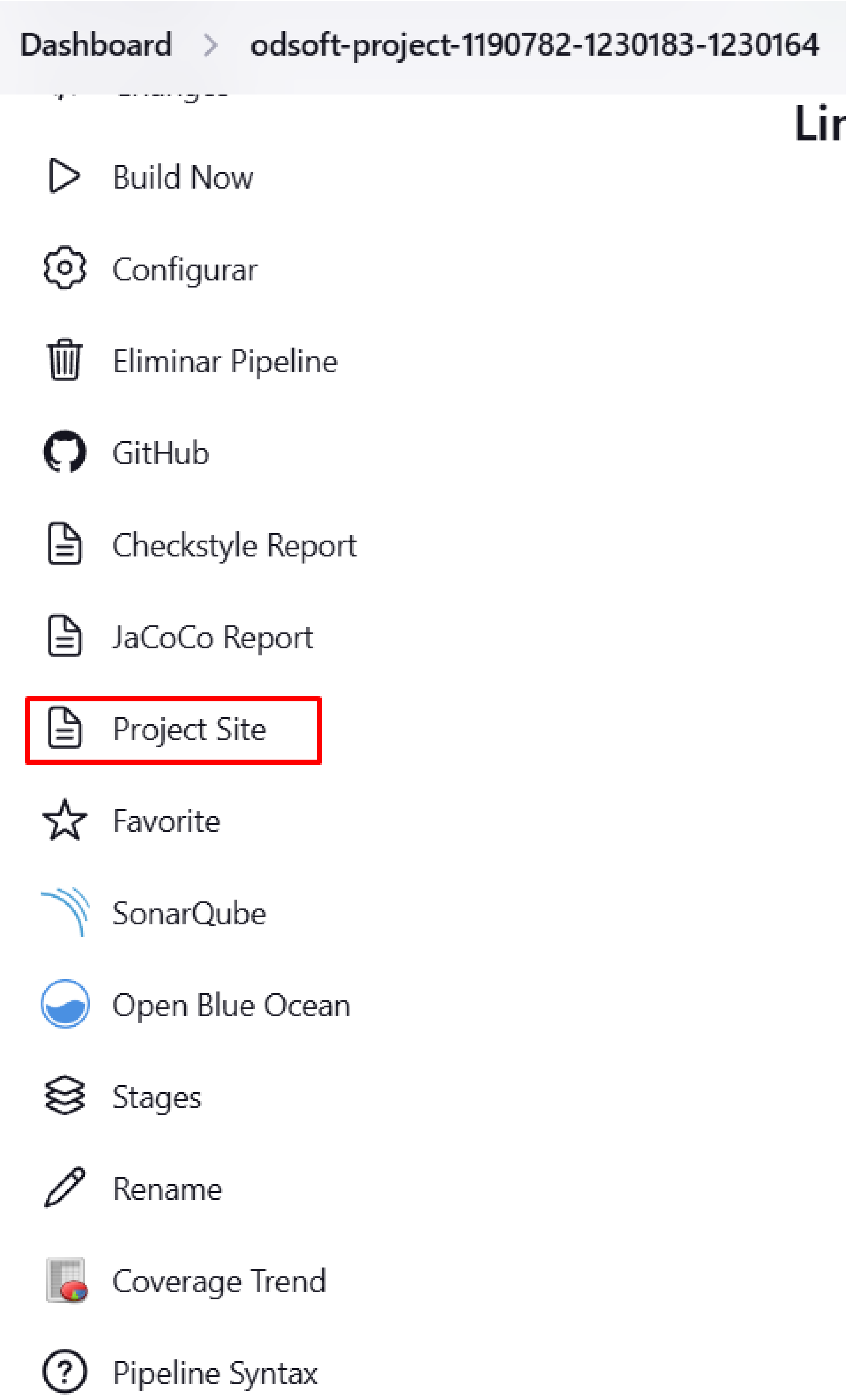
5. **Report Results** This section is responsible for generating and publish comprehensive reports that provide insights about code qualit, testing coverage and many other metrics. The main goal is to create a detailed summary about the pipeline execution.
   i. The first part of this stage is the **mvn site** command that will generate a project site that includes various reports, for example unit test summaries, code analysis and documentation generated during the build execution.

```
stage('Report Results') {
        steps {
            script {
                if (isUnix()) {
                    sh 'mvn site'
                } else {
                    bat 'mvn site'
                }
            }
        }
    }
```

ii. After the site report generation, we used Jenkins publish html plugin to make these reports available in the left panel.

```
stage('Publish Site Report') {
        steps {
            publishHTML([
                reportDir: 'target/site',
                reportFiles: 'index.html',
                reportName: 'Project Site',
                keepAll: true,
                allowMissing: false,
                alwaysLinkToLastBuild: true
            ])
        }
    }
```

This will generate the index.html and move it to the *target/site* directory.



6. **Deployment**

The deployment stage guarantees that the latest application built is automatically transfered and made accessible in both local and remote environments.

~~~
```groovy
stage('Deploy Local') {
            steps {
                  script {
~~~

~~~
                if (isUnix()) {
                    withCredentials([sshUserPrivateKey(credentialsId: 'TOKEN_SSH_ID', keyVariable: 'SSH
                        sh 'scp -o StrictHostKeyChecking=no -i $SSH_KEY target/psoft-g1-0.0.1-SNAPSHOT.
                    }
                } else {
                    bat 'copy target\\psoft-g1-0.0.1-SNAPSHOT.jar C:\\deploy'
                }
            }
        }
    }
```
~~~

1. Local deployment

```
The .jar file generated in the build stage is copied to a specific directory. In this case it will copy

2. Remote deployment to ISEP Server

It will simulate a production environment in the remote server. The unix _scp_ command is used to secur
This command inclued an authentication token stored and managed by Jenkins. We had to configure this to
```

Configurations and Tools Used

During the pipeline, we had to differ from windows and linux system because the both systems don't use the same terminologies, for example bat (Windows) and sh (Linux) ensuring cross-platform compatibility.

- **Jenkins**: Is a popular open-source automation server used for building, testing and deploying software projects. It enables continuous integration (CI) and continuous delivery (CD) by automating stages in the development lifecycle. Jenkins uses “pipelines” defined as code (usually in Jenkinsfile), which allows us to automate build, test, analysis and deployment tasks efficiently.
- **SonarQube**: Is a code quality analysis tool that evaluates code to identify potential bugs, vulnerabilities and code smells. It provides a continuous inspection of codebases, giving feedback on quality metrics just like complexity, duplications and code coverage. Using SonarQube in the pipeline, we can guarantee code meets high standards before deployment, helping to minimize potential issues and enforce best practices. SonarQube also integrates with Jenkins, automatically analyzing code each time it is committed and providing detailed reports on areas for improvement.
- **JaCoCo**: (Java Code Coverage) is a tool that measures code coverage by determining which lines of code are tested and which are not. It helps identify gaps in test cases by providing a comprehensive view of test coverage, which is critical for assessing the effectiveness of unit tests. Integrated into our Jenkins pipeline, JaCoCo generates reports each time tests are run, helping us maintain adequate coverage and guiding future test development for improved code reliability.

These tools combined create a robust and efficient CI/CD pipeline, providing a smooth process from code changes to automated deployment and verification, while ensuring high code quality and reliability.

Technical Decisions and Justifications

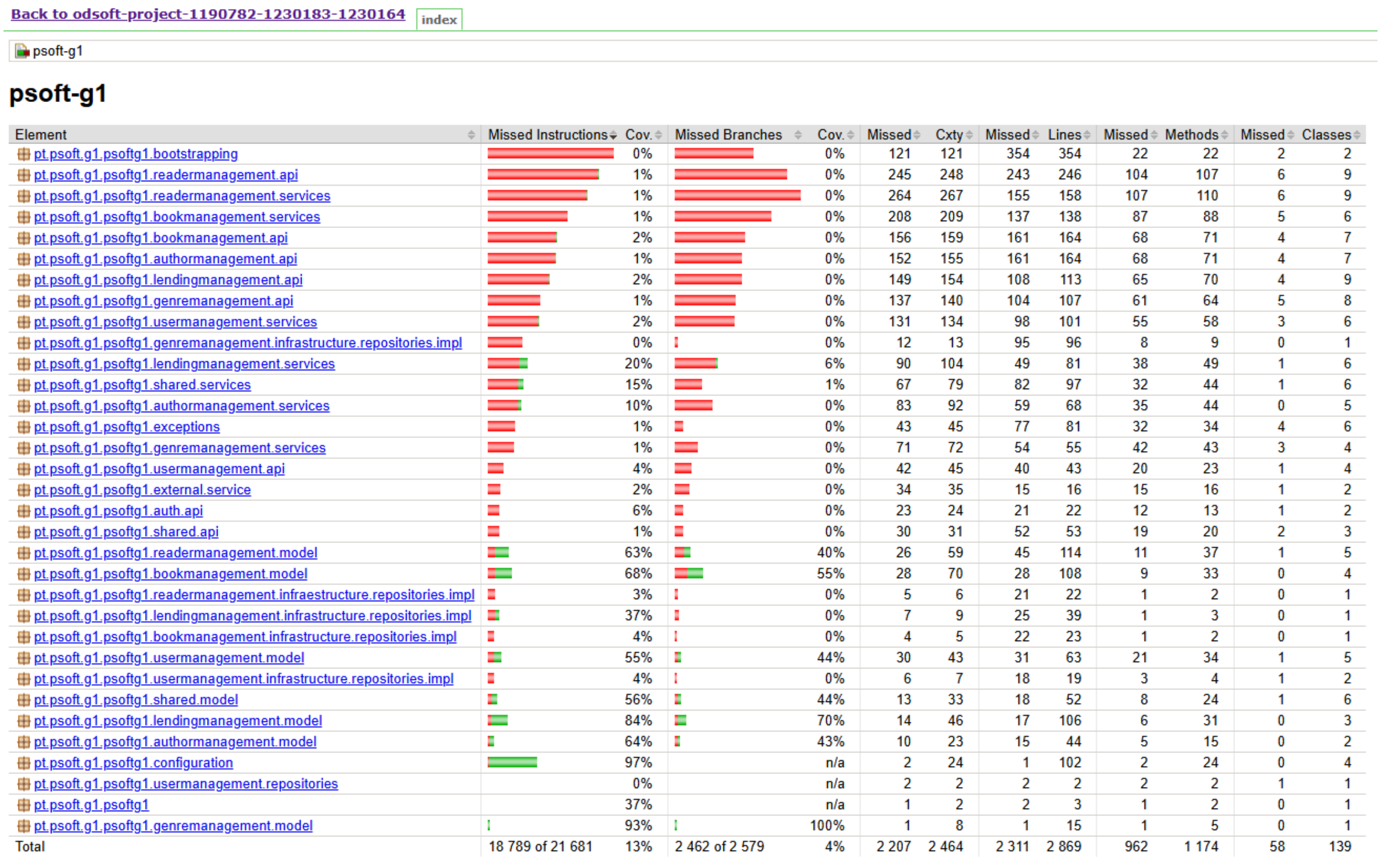
- **Test Automation**: To optimize the pipeline's runtime and ensure rapid feedback, we configured the tests to run in parallel, categorizing them by unit tests and integration tests. By parallelizing these test stages, we reduced overall testing time and enhanced efficiency, enabling faster identification of issues at different levels of code integration and functionality. This approach minimizes bottlenecks, ensures timely detection of issues and maximizes resource utilization.
- **Deployment Environment**: Configured Jenkins to deploy both locally and remotely at ISEP Servers, ensuring flexibility and redundancy.
- **Coverage and Quality**: Integrating SonarQube for automated static code analysis, we ensured a consistent check on code quality, focusing on maintainability, complexity, duplication and potential vulnerabilities. This tool provided actionable insights for continuous improvement, allowing us to set and track quality standards with each commit. Additionally, using JaCoCo for coverage tracking offered a clear view of testing effectiveness, helping us to maintain high-quality code and prioritize additional testing where coverage was lacking.

Results and Analysis

Below is a summary of test results and performance analysis observed during the project:

1. **Test Coverage**: JaCoCo reports indicated satisfactory code coverage, with an increase to 16% due to integration and unit testing. **Note** that this **is not** the final result because the report is being done at the same time as tests are being developed. The final number is supposed to be much higher.

These two reports were taken when we started the project.

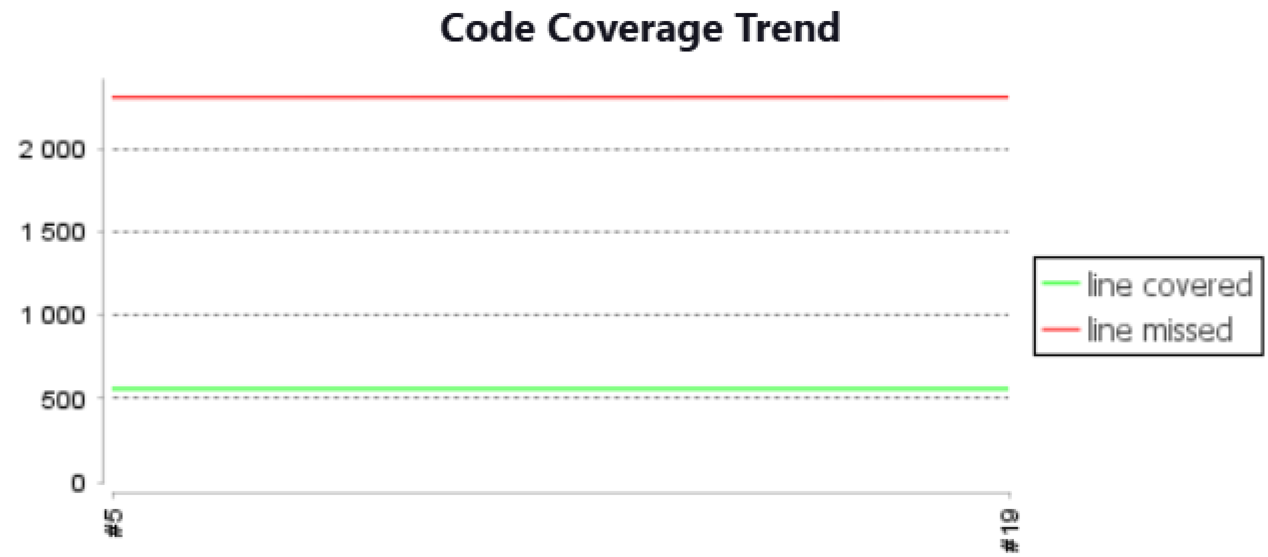


Back to odsoft-project-1190782-1230183-1230164 index

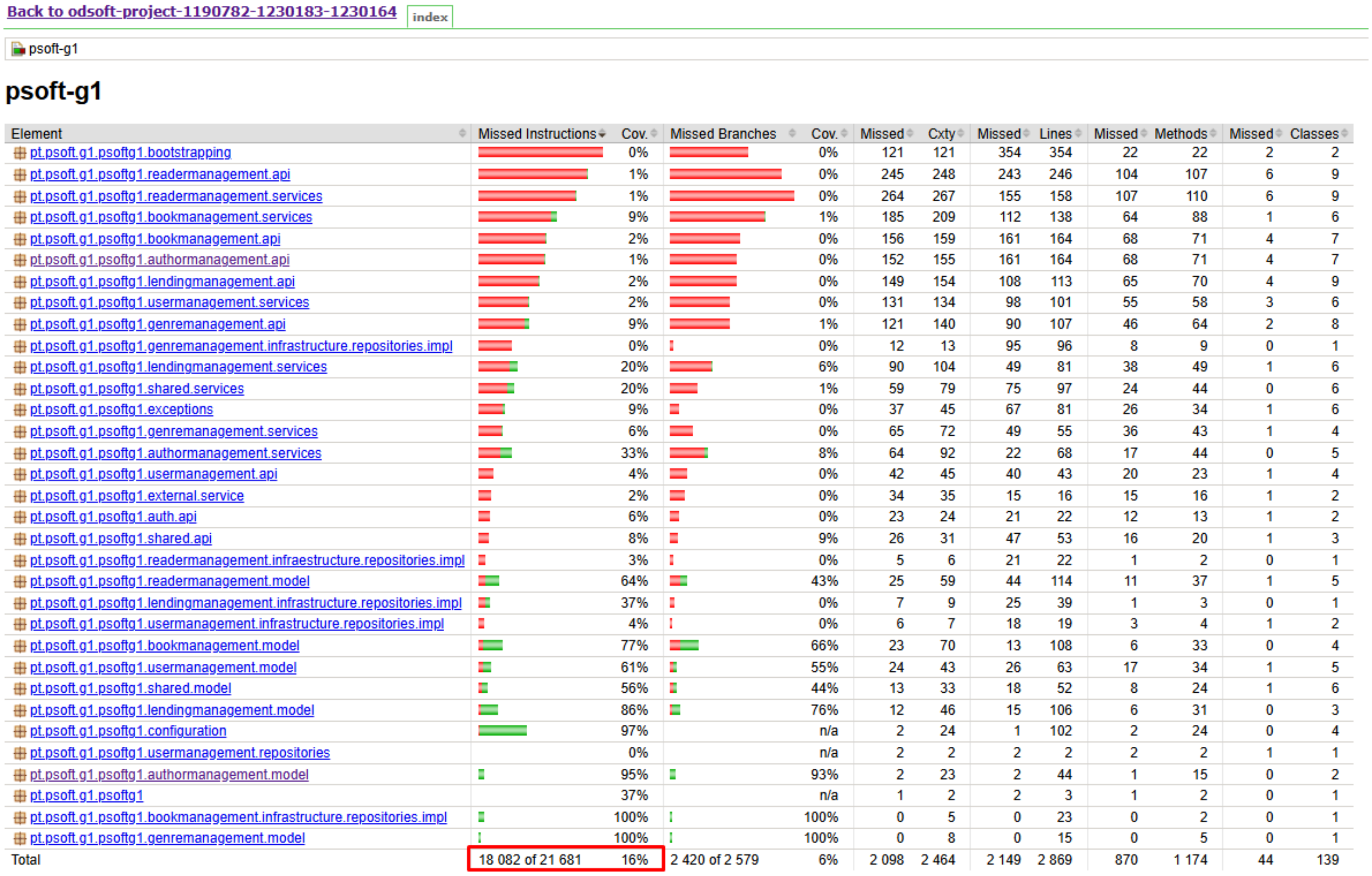
psoft-g1

psoft-g1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
pt.psoft.g1.psoftg1.bootstrapping		0%		0%	121	121	354	354	22	22	2	2
pt.psoft.g1.psoftg1.readermanagement.api		1%		0%	245	248	243	246	104	107	6	9
pt.psoft.g1.psoftg1.readermanagement.services		1%		0%	264	267	155	158	107	110	6	9
pt.psoft.g1.psoftg1.bookmanagement.services		1%		0%	208	209	137	138	87	88	5	6
pt.psoft.g1.psoftg1.bookmanagement.api		2%		0%	156	159	161	164	68	71	4	7
pt.psoft.g1.psoftg1.authormanagement.api		1%		0%	152	155	161	164	68	71	4	7
pt.psoft.g1.psoftg1.lendingmanagement.api		2%		0%	149	154	108	113	65	70	4	9
pt.psoft.g1.psoftg1.genremanagement.api		1%		0%	137	140	104	107	61	64	5	8
pt.psoft.g1.psoftg1.usermanagement.services		2%		0%	131	134	98	101	55	58	3	6
pt.psoft.g1.psoftg1.genremanagement.infrastructure.repositories.impl		0%		0%	12	13	95	96	8	9	0	1
pt.psoft.g1.psoftg1.lendingmanagement.services		20%		6%	90	104	49	81	38	49	1	6
pt.psoft.g1.psoftg1.shared.services		15%		1%	67	79	82	97	32	44	1	6
pt.psoft.g1.psoftg1.authormanagement.services		10%		0%	83	92	59	68	35	44	0	5
pt.psoft.g1.psoftg1.exceptions		1%		0%	43	45	77	81	32	34	4	6
pt.psoft.g1.psoftg1.genremanagement.services		1%		0%	71	72	54	55	42	43	3	4
pt.psoft.g1.psoftg1.usermanagement.api		4%		0%	42	45	40	43	20	23	1	4
pt.psoft.g1.psoftg1.external.service		2%		0%	34	35	15	16	15	16	1	2
pt.psoft.g1.psoftg1.auth.api		6%		0%	23	24	21	22	12	13	1	2
pt.psoft.g1.psoftg1.shared.api		1%		0%	30	31	52	53	19	20	2	3
pt.psoft.g1.psoftg1.readermanagement.model		63%		40%	26	59	45	114	11	37	1	5
pt.psoft.g1.psoftg1.bookmanagement.model		68%		55%	28	70	28	108	9	33	0	4
pt.psoft.g1.psoftg1.readermanagement.infraestructure.repositories.impl		3%		0%	5	6	21	22	1	2	0	1
pt.psoft.g1.psoftg1.lendingmanagement.infrastructure.repositories.impl		37%		0%	7	9	25	39	1	3	0	1
pt.psoft.g1.psoftg1.bookmanagement.infrastructure.repositories.impl		4%		0%	4	5	22	23	1	2	0	1
pt.psoft.g1.psoftg1.usermanagement.model		55%		44%	30	43	31	63	21	34	1	5
pt.psoft.g1.psoftg1.usermanagement.infrastructure.repositories.impl		4%		0%	6	7	18	19	3	4	1	2
pt.psoft.g1.psoftg1.shared.model		56%		44%	13	33	18	52	8	24	1	6
pt.psoft.g1.psoftg1.lendingmanagement.model		84%		70%	14	46	17	106	6	31	0	3
pt.psoft.g1.psoftg1.authormanagement.model		64%		43%	10	23	15	44	5	15	0	2
pt.psoft.g1.psoftg1.configuration		97%		n/a	2	24	1	102	2	24	0	4
pt.psoft.g1.psoftg1.usermanagement.repositories		0%		n/a	2	2	2	2	2	2	1	1
pt.psoft.g1.psoftg1		37%		n/a	1	2	2	3	1	2	0	1
pt.psoft.g1.psoftg1.genremanagement.model		93%		100%	1	8	1	15	1	5	0	1
Total	18 789 of 21 681	13%	2 462 of 2 579	4%	2 207	2 464	2 311	2 869	962	1 174	58	139



We can see that the Code Coverage Line is getting higher while the lines missed are decreasing:

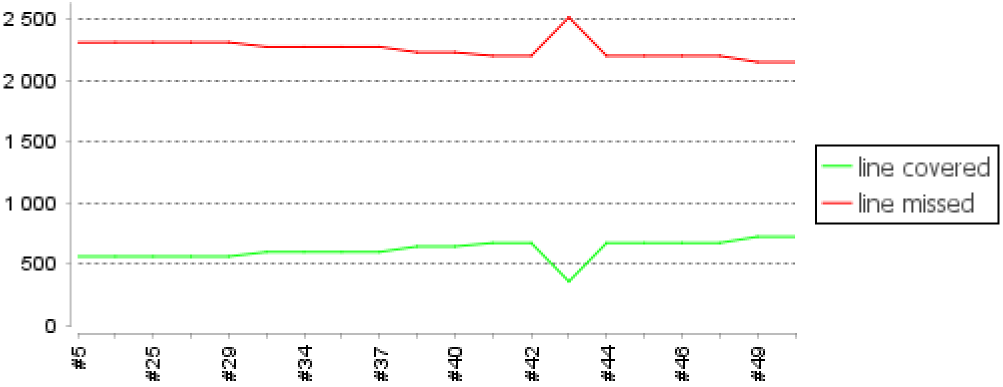


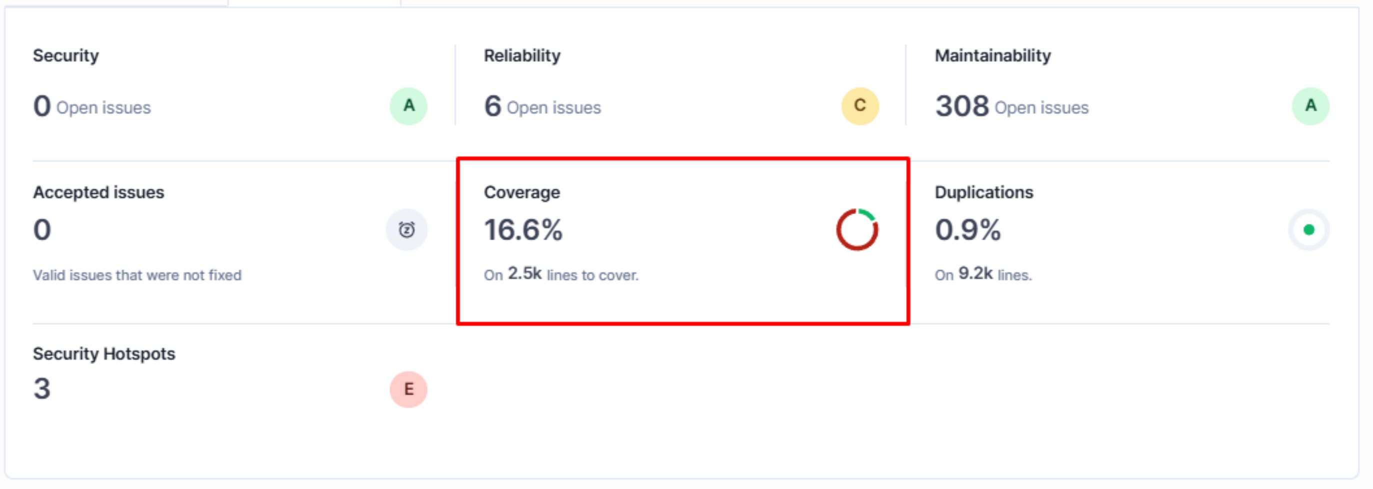
Back to odsoft-project-1190782-1230183-1230164 index

psoft-g1

psoft-g1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
pt.psoft.g1.psoftg1.bootstrapping		0%		0%	121	121	354	354	22	22	2	2
pt.psoft.g1.psoftg1.readermanagement.api		1%		0%	245	248	243	246	104	107	6	9
pt.psoft.g1.psoftg1.readermanagement.services		1%		0%	264	267	155	158	107	110	6	9
pt.psoft.g1.psoftg1.bookmanagement.services		9%		1%	185	209	112	138	64	88	1	6
pt.psoft.g1.psoftg1.bookmanagement.api		2%		0%	156	159	161	164	68	71	4	7
pt.psoft.g1.psoftg1.authormanagement.api		1%		0%	152	155	161	164	68	71	4	7
pt.psoft.g1.psoftg1.lendingmanagement.api		2%		0%	149	154	108	113	65	70	4	9
pt.psoft.g1.psoftg1.usermanagement.services		2%		0%	131	134	98	101	55	58	3	6
pt.psoft.g1.psoftg1.genremanagement.api		9%		1%	121	140	90	107	46	64	2	8
pt.psoft.g1.psoftg1.genremanagement.infrastructure.repositories.impl		0%		0%	12	13	95	96	8	9	0	1
pt.psoft.g1.psoftg1.lendingmanagement.services		20%		6%	90	104	49	81	38	49	1	6
pt.psoft.g1.psoftg1.shared.services		20%		1%	59	79	75	97	24	44	0	6
pt.psoft.g1.psoftg1.exceptions		9%		0%	37	45	67	81	26	34	1	6
pt.psoft.g1.psoftg1.genremanagement.services		6%		0%	65	72	49	55	36	43	1	4
pt.psoft.g1.psoftg1.authormanagement.services		33%		8%	64	92	22	68	17	44	0	5
pt.psoft.g1.psoftg1.usermanagement.api		4%		0%	42	45	40	43	20	23	1	4
pt.psoft.g1.psoftg1.external.service		2%		0%	34	35	15	16	15	16	1	2
pt.psoft.g1.psoftg1.auth.api		6%		0%	23	24	21	22	12	13	1	2
pt.psoft.g1.psoftg1.shared.api		8%		9%	26	31	47	53	16	20	1	3
pt.psoft.g1.psoftg1.readermanagement.infraestructure.repositories.impl		3%		0%	5	6	21	22	1	2	0	1
pt.psoft.g1.psoftg1.readermanagement.model		64%		43%	25	59	44	114	11	37	1	5
pt.psoft.g1.psoftg1.lendingmanagement.infrastructure.repositories.impl		37%		0%	7	9	25	39	1	3	0	1
pt.psoft.g1.psoftg1.usermanagement.infrastructure.repositories.impl		4%		0%	6	7	18	19	3	4	1	2
pt.psoft.g1.psoftg1.bookmanagement.model		77%		66%	23	70	13	108	6	33	0	4
pt.psoft.g1.psoftg1.usermanagement.model		61%		55%	24	43	26	63	17	34	1	5
pt.psoft.g1.psoftg1.shared.model		56%		44%	13	33	18	52	8	24	1	6
pt.psoft.g1.psoftg1.lendingmanagement.model		86%		76%	12	46	15	106	6	31	0	3
pt.psoft.g1.psoftg1.configuration		97%		n/a	2	24	1	102	2	24	0	4
pt.psoft.g1.psoftg1.usermanagement.repositories		0%		n/a	2	2	2	2	2	2	1	1
pt.psoft.g1.psoftg1.authormanagement.model		95%		93%	2	23	2	44	1	15	0	2
pt.psoft.g1.psoftg1		37%		n/a	1	2	2	3	1	2	0	1
pt.psoft.g1.psoftg1.bookmanagement.infrastructure.repositories.impl		100%		100%	0	5	0	23	0	2	0	1
pt.psoft.g1.psoftg1.genremanagement.model		100%		100%	0	8	0	15	0	5	0	1
Total	18 082 of 21 681	16%	2 420 of 2 579	6%	2 098	2 464	2 149	2 869	870	1 174	44	139





Security

0 Open issues

A

Reliability

6 Open issues

C

Maintainability

308 Open issues

A

Accepted issues

0

Valid issues that were not fixed

Coverage

16.6%

On 2.5k lines to cover.

Duplications

0.9%

On 9.2k lines.

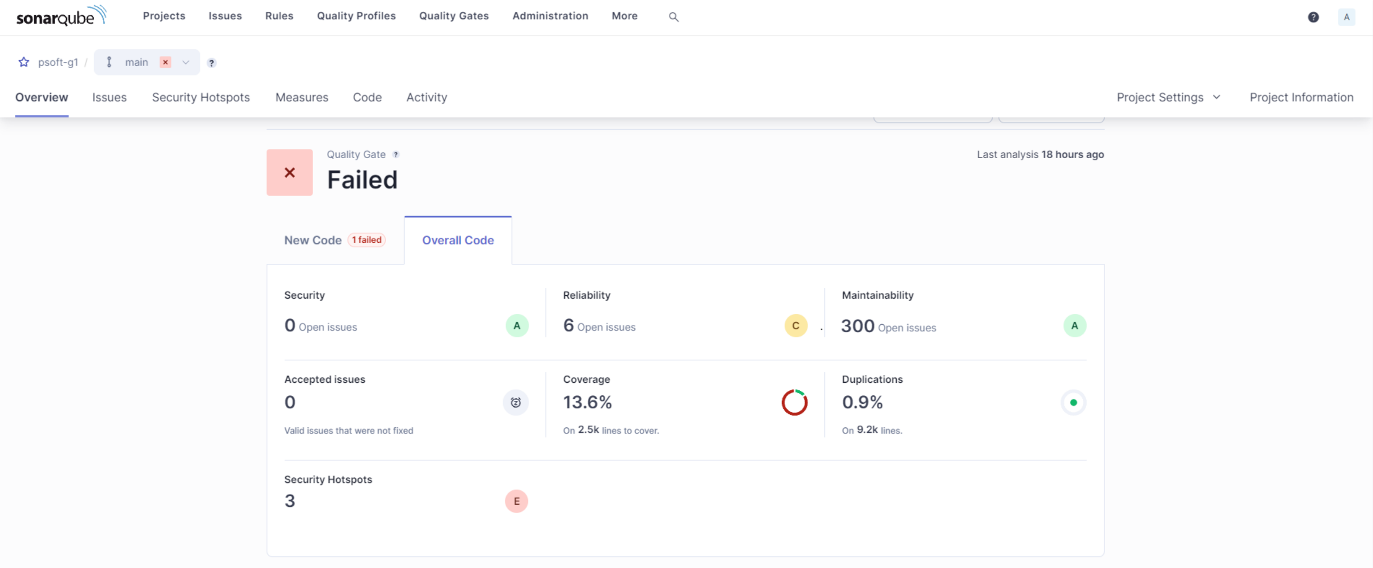
Security Hotspots

3

E

After improving and developing the tests, we can confirm that we improved the Code coverage by looking again at these both graphs.

2. **Code Quality**: SonarQube analysis highlighted improvements in code complexity and duplication.



Quality Gate

Failed

Last analysis 18 hours ago

New Code 1 failed | Overall Code

Security
0 Open issues A

Reliability
6 Open issues C

Maintainability
300 Open issues A

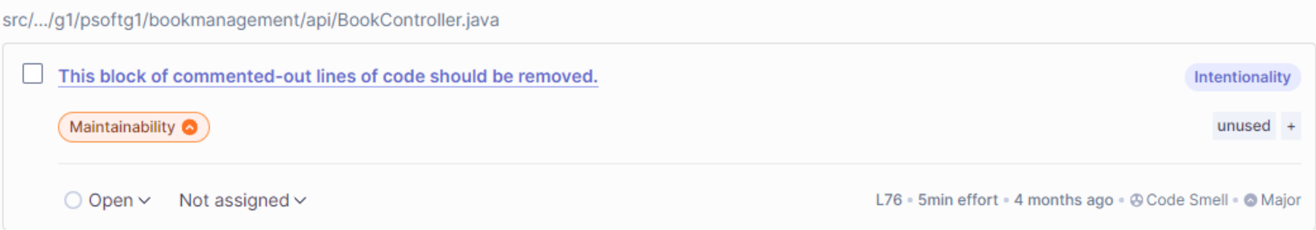
Accepted issues
0
Valid issues that were not fixed

Coverage
13.6%
On 2.5k lines to cover.

Duplications
0.9%
On 9.2k lines.

Security Hotspots
3 E

For example, if we open the maintainability warnings/erros we can find the following:

1. 

src/.../g1/psoftg1/bookmanagement/api/BookController.java

☐ This block of commented-out lines of code should be removed.

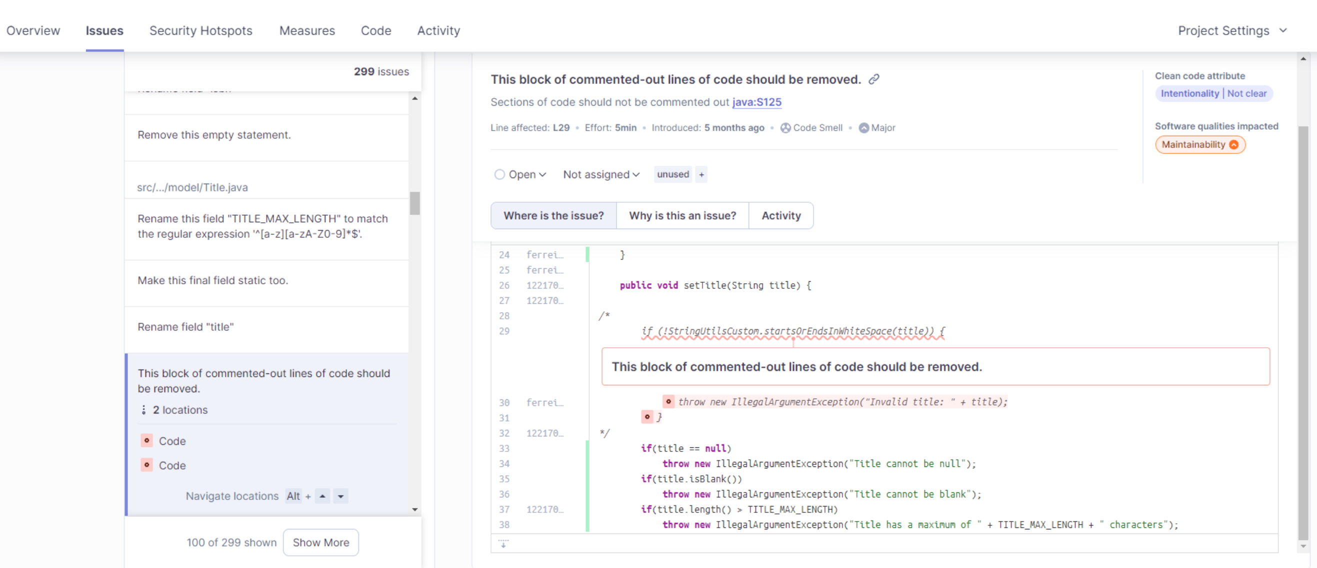
Intentionality

Maintainability

unused +

○ Open Not assigned

L76 • 5min effort • 4 months ago • Code Smell • Major

2. 

299 issues

Remove this empty statement.

src/.../model/Title.java

Rename this field "TITLE_MAX_LENGTH" to match the regular expression '^[a-z][a-zA-Z0-9]*$'.

Make this final field static too.

Rename field "title"

This block of commented-out lines of code should be removed.
2 locations
Code
Code
Navigate locations Alt + ▲ ▼

100 of 299 shown Show More

This block of commented-out lines of code should be removed.

Sections of code should not be commented out java:S125

Line affected: L29 • Effort: 5min • Introduced: 5 months ago • Code Smell • Major

○ Open Not assigned unused +

Clean code attribute
Intentionality | Not clear

Software qualities impacted
Maintainability

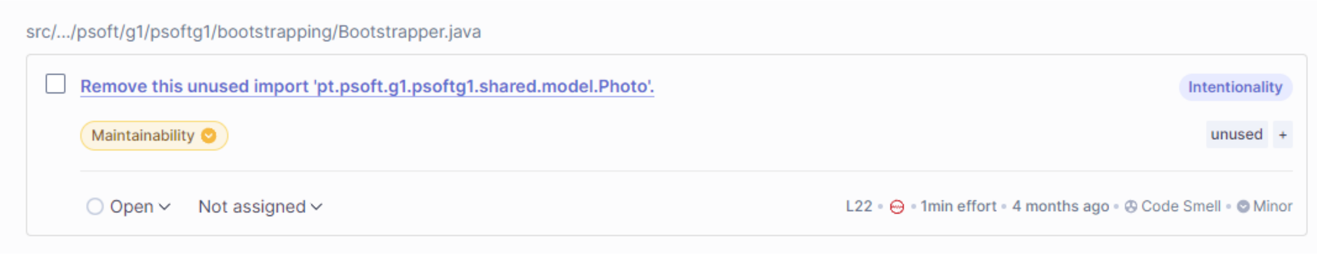
Where is the issue? | Why is this an issue? | Activity

```
        }

    public void setTitle(String title) {
  /*
        if (!StringUtilsCustom.startsOrEndsInWhiteSpace(title)) {
```

This block of commented-out lines of code should be removed.

```
            throw new IllegalArgumentException("Invalid title: " + title);
        }
  */
        if(title == null)
            throw new IllegalArgumentException("Title cannot be null");
        if(title.isBlank())
            throw new IllegalArgumentException("Title cannot be blank");
        if(title.length() > TITLE_MAX_LENGTH)
            throw new IllegalArgumentException("Title has a maximum of " + TITLE_MAX_LENGTH + " characters");
```

3. 

src/.../psoft/g1/psoftg1/bootstrapping/Bootstrapper.java

☐ Remove this unused import 'pt.psoft.g1.psoftg1.shared.model.Photo'.

Intentionality

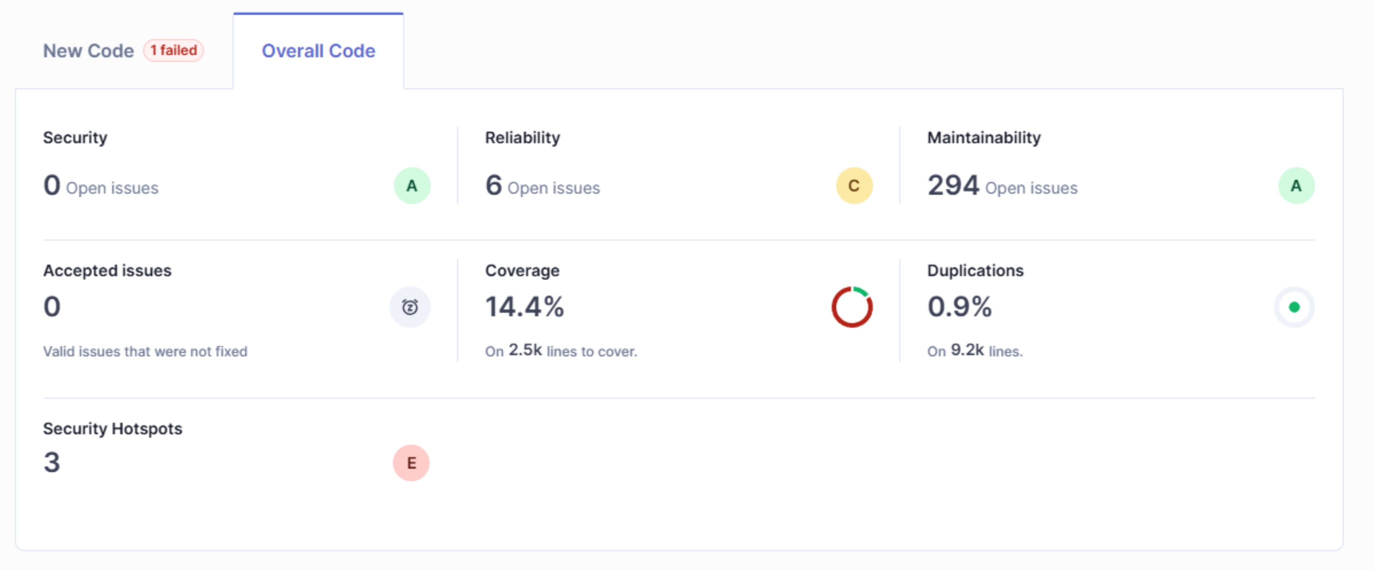
Maintainability

unused +

○ Open Not assigned

L22 • 1min effort • 4 months ago • Code Smell • Minor

We solve this and more problems and now we can see that the maintainability dropped from 300 to 294 open issues. This confirms that the SonarQube is well prepared to analyze the code and new updates.



Security
0 Open issues A

Reliability
6 Open issues C

Maintainability
294 Open issues A

Accepted issues
0
Valid issues that were not fixed

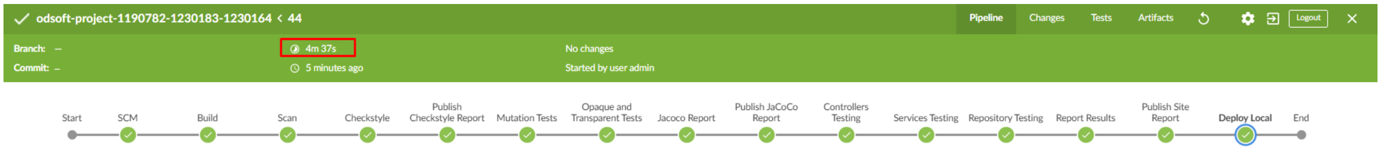
Coverage
14.4%
On 2.5k lines to cover.

Duplications
0.9%
On 9.2k lines.

Security Hotspots
3 E

3. **Pipeline Performance**: The pipeline runtime was optimized by X% throughout the project, reducing build and test parallel execution times.

This was the time without using parallelism (4min 37s)

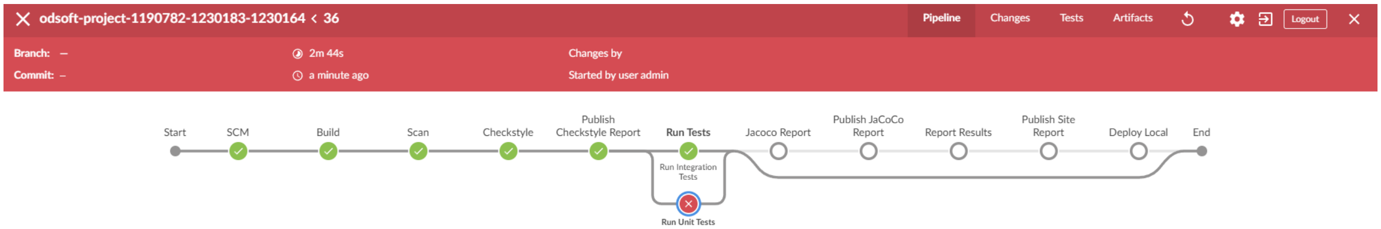


✓ odsoft-project-1190782-1230183-1230164 < 44

Branch: –
Commit: –
4m 37s
5 minutes ago
No changes
Started by user admin

- Build : (48s)
- Scan : (29s)
- CheckStyle : (6s)
- Run Unit Tests : Mutation + Opaque and Transparent (38s)
- Integration Tests : (1m 27s)

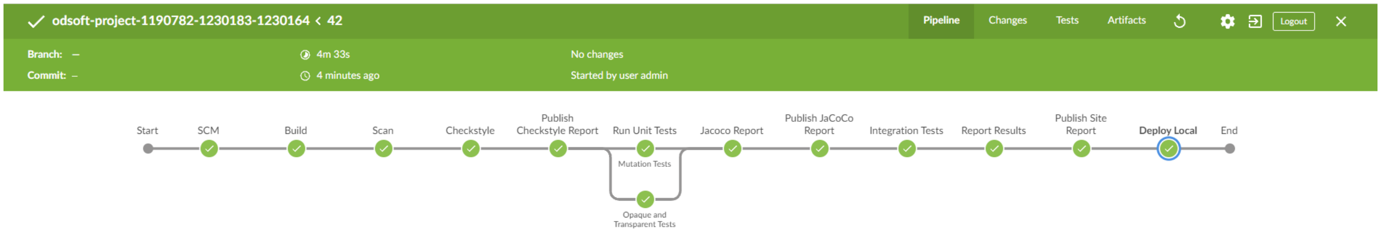
First we tried to run the unit and integration tests in parallel, but the files for jacoco were still being used by integration.



✕ odsoft-project-1190782-1230183-1230164 < 36

Branch: –
Commit: –
2m 44s
a minute ago
Changes by
Started by user admin

Run Unit Test in parallel:



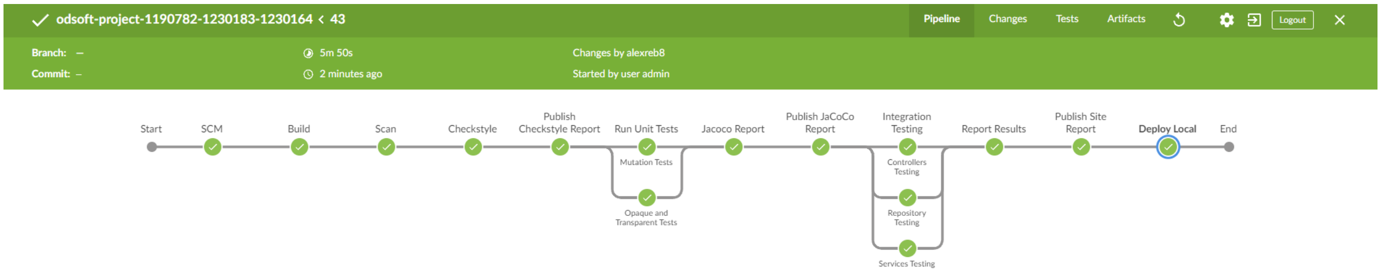
odsoft-project-1190782-1230183-1230164 < 42

Branch: –
Commit: –
4m 33s
4 minutes ago
No changes
Started by user admin

- Run Unit Tests : Mutation + Opaque and Transparent (25s)

With this we confirm that the Unit Tests are faster in parallel

Run Integration Tests in parallel:



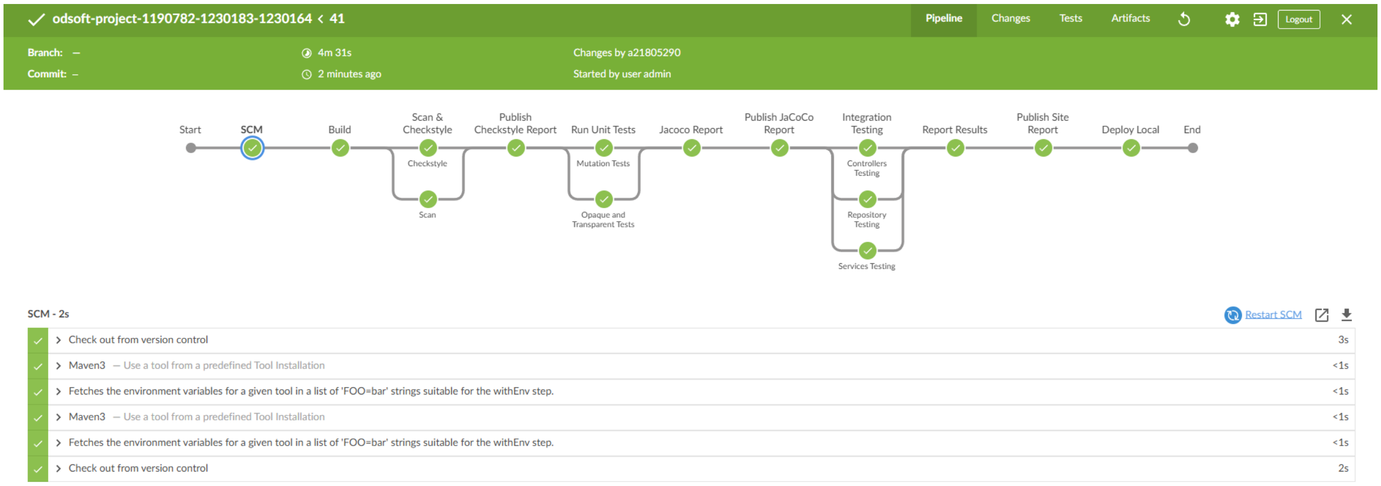
odsoft-project-1190782-1230183-1230164 < 43

Branch: –
Commit: –
5m 50s
2 minutes ago
Changes by alexreb8
Started by user admin

- Integration Tests : Controller + Service + Repo (1m 20s)

With this, we confirm that the integration tests are faster in parallel.

We run **scan and checkstyle** in parallel to get more performance and also because they are compatible with running in parallel.



odsoft-project-1190782-1230183-1230164 < 41

Branch: –
Commit: –
4m 31s
2 minutes ago
Changes by a21805290
Started by user admin

```
SCM - 2s
✓ > Check out from version control                                                                  3s
✓ > Maven3  — Use a tool from a predefined Tool Installation                                       <1s
✓ > Fetches the environment variables for a given tool in a list of 'FOO=bar' strings suitable for the withEnv step.   <1s
✓ > Maven3  — Use a tool from a predefined Tool Installation                                       <1s
✓ > Fetches the environment variables for a given tool in a list of 'FOO=bar' strings suitable for the withEnv step.   <1s
✓ > Check out from version control                                                                  2s
```

- Scan and Checkstyle : (31s)

The final time is 4m 31s. With this we can see that this is the best way we can use the pipeline.

Conclusion

This project underscored the value of a better structured CI/CD process in organization of code quality, its efficiency and its reliability for the library management system. With the use of Jenkins, SonarQube, Checkstyle, and JaCoCo, we implemented an automated mechanism that made the processes of code examining and testing as well as code deployment easier.

The ability to conduct tests along with some static analysis concurrently was critical for the purposes of enhancing the execution times of the entire pipeline without compromising on the exhaustive coverage of validation. The use of SonarQube and Checkstyle further ensured best coding practices were adopted, as well as highlighting areas that need updates and improvement during the build process. JaCoCo offered valuable perspective on the level of test coverage achieved and assisted in identifying areas that needed more tests.

Deployment was automated for remote and local (on-site) environments in order to maintain uniformity between developments as well as productions which strengthened the faith taken upon the system’s reliability. Last but not least, this CI/CD pipeline ensured faster turnaround times whilst enhancing system visibility and control, which have set the stage for further advancement and future growth of the system.

After analysing the results, some improvements for the future have been acknowledged:

- A trigger can be implemented in the pipeline so that it is run when some push is made to an indicated branch and create other pipelines to for example only do build and analysis of the code without doing deploy depending on whether you are doing development of new code and do not want to affect the current version of the system.
- With the addition of new tests we have seen an increase in maintainability problems, so when new tests are implemented more attention should be paid to this and if possible correct those that have already been created.

Contributions

- 1190782 - José Soares
- 1230164 - Alexandre Monteiro
- 1230183 - Fernando Castro