GOI ESKOLA POLITEKNIKOA
ESCUELA POLITÉCNICA SUPERIOR
MONDRAGON UNIBERTSITATEA

MILESTONE 0 – SOFTWARE ENGINEERING

LOREDI ALTZIBAR

NAHIA LI GÓMARA

AITOR LANDA

XABIER LANDA

ANDER OLASO

ANE SAJERAS

POPBL5 – TEAM 4

DEGREE OF COMPUTER

ENGINEERING Tutor: GOIURIA

SAGARDUI

Arrasate, January, 22nd 2022

CONTENT

DESCRIPTION

The goal of this document is to describe the software engineering techniques, methods and tools used during the project. Sections should be used as a reference for the content although can be changed if needed.

This document should be saved in the repository so that the expect can review it during the execution of the project.

1. PART 1 PROJECT SETUP

The goal of this section is to describe the setup of methods and tools that we are going to use during the Project.

1.1 DOCUMENTATION

The goal of this section is to explain the documentation is going to use for JavaDoc and the UML Diagrams generation.

1.1.1 JAVADOC

Tags: each structure will have a minimum requirement of tags below

Class:

@File (when calling another file)
@version (adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.)
@Brief (for descriptions)
@Date (creation date)
@Date (last modification date)
@Authors

Function:

@brief
@param
@return

Programming rules:

1. Variable:
   - The names must be significant.
   - For the variable names it is used the lower CamelCase method.

2. Functions:
   - The names must be significant.
   - For the variable names it is used the lower CamelCase method.
   - Tags before the code description

3. Constants:
   - The names must be significant.
   - All in capital letters and _ as space.

4. Classes:
   - The names must be significant.
   - For the variable names it is used the upper CamelCase method.
   - Tags before the code description block:

1.1.2 UML DIAGRAM GENERATION

Configuration for UML Diagrams generation: UML diagrams are configured in the pom.xml where yWorks doclet is include as property. Then, yWorks will be configured as a part of Javadoc plugin.

```
<!-- Properties for the pom-->
<properties>
  <!-- Path to the doclet tool-->
  <yDoc.path>C:\Users\aitor\Desktop\yworks-uml-doclet-3.1-jdk1.8</yDoc.path>
  <!-- JDK version-->
  <jdk.version>1.8</jdk.version>
  <jdk.path>C:\Program Files\Java\jdk1.8.0_191\bin\javac</jdk.path>
  </properties>
<build>
<plugins>
 <!-- Select the jdk -->
 <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <configuration>
          <source>${jdk.version}</source>
          <target>${jdk.version}</target>
          <fork>true</fork>
                <executable>${jdk.path}</executable>
      </configuration>
  </plugin>
  <!-- Generate javadoc documentation with UML diagrams javadoc:javadoc-->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>2.8.1</version>
    <configuration>
      <doclet>ydoc.doclets.YStandard</doclet>
      <docletPath>${yDoc.path}/lib/ydoc.jar:${yDoc.path}/lib/styleed.jar:${yDoc.path}/resources</docletPath>
      <additionalparam>-umlautogen</additionalparam>
    </configuration>
  </plugin>
  <!-- Sonar, static analysis- sonar:sonar-->
    <plugin>
      <groupId>org.sonarsource.scanner.maven</groupId>
      <artifactId>sonar-maven-plugin</artifactId>
      <version>3.6.0.1398</version>
    </plugin>
       </plugins>
  </build>
</project>
```

1.2 VERSION CONTROL

The goal of this section is to explain the repository we are going to use, the branching names and version strategy.

1.2.1 REPOSITORY

We are using one centralized repository in “GitLab”. Each developer will clone that centralized repo to a local repo.

URL: https://gitlab.danz.eus/m2gi13/1920_popbl_5_t4/loglas_project

1.2.2 BRANCHING NAMES

1. **Master:** master
   Is the main branch where we the source code is reflecting the last production-ready state of the product.

   - Name convention: *master*

2. **Hotfix:** hotfix

This branch we are going to use if we have a bug or code smells in the production-ready version to resolve immediately.

- Name convention: *nameHotfix*

3. **Release Branches:** new_realeases

Is going to be the almost the version that reflects the desired state in production. That we are going to make minimal changes, fix bugs. We are going to released to the master and do the main develop in the develop branch.

- Name convention: *nameRelease*

4. **Develop:** development

In this branch we are going to develop all product

- Name convention: *developerNameDevelop*

5. **Feature Branches:** new_features

With this branch we are going to develop future features for the product for the upcoming or a distant future release. Introducing a new feature over the development.

a) newInterface: Introducing new interface
b) newGame: Introducing a new game
c) newFunctionality: Introducing a new functionality (login, personal achievements, favorites tag...)

- Name convention: *interfaceFeature*

1.1.3 MERGE RULES

- Merge between different develop branches are when one functionality end.
- Merge between develop and Master are when have a project new version.
- Merge between Hotfix and Master are when the bug is solution.
- Merge between Feature and Master are when the end new feature.

1.1.4 VERSION STRATEGY

Using tags

- Name convention: vX

X and Y being an integer number.
X = Sprint number

We will create a new tag for every sprint deadline from Release branch, Master branch if possible:

- V0.1 = version for sprint 1
- V0.2 = version for sprint 2
- V0.3 = version for sprint 3
- V1 = final version

1.3 STATIC ANALYSIS

Sonar server configured to perform the static analysis during the development, quality profiles and quality gates selected. o Documentation: a Javadoc template and the project configured to obtain UML diagrams o Continuous integration server. Pipelines defined.

- **URL Sonarqube:** http://loglas.duckdns.org:9000/projects

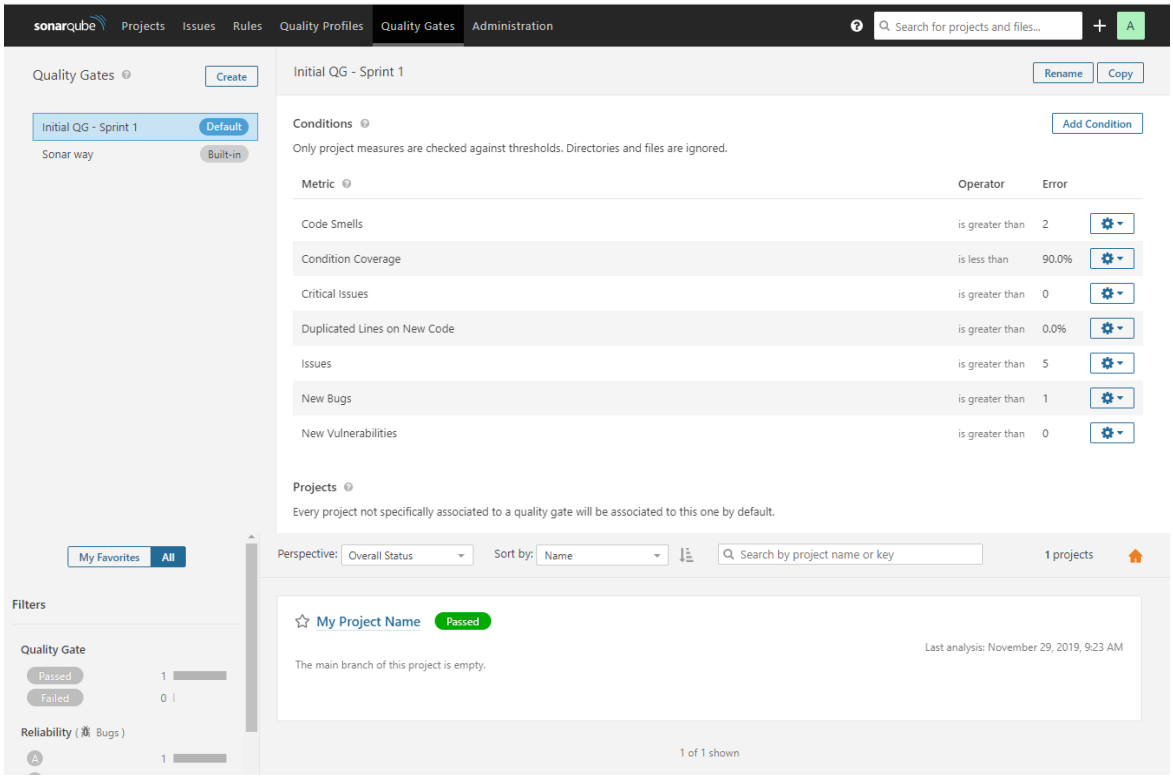
1.3.1 QUALITY PROFILES

We are going to use the default quality profile based on our quality gate.

1.3.2 QUALITY GATE

Quality gate for sprint 1:

- Max. number of issues: 5
- Max. number of critical issues: 0
- Max. number of bugs: 1
- Max. number of code smell: 2
- Max. vulnerability remediation effort: 0
- Duplicated lines (%): 0%



1.4 AUTOMATIC BUILDS

Maven project created with all the steps of an automatic build. Poms with the steps for all the branches.

- Coverage: greater than 90% of the total project.

1.5 CONTINUOS INTEGRATION (JENKINS)

We are using Jenkins for continuous integration and we just figured out a basic set of rules for the first sprint:

1. Grab the code from GitLab repository
2. Run the code as maven and build
3. Deploy

URL Jenkins: http://loglas.duckdns.org:8080/

1.6 ADDITIONAL COMMENTS AND FUTURE LINES:

Initially we are using Docker to make it easier to create, deploy, and run applications by using containers

2. PART 2 PROJECT SETUP

WEEK 1

DOCUMENTATION

The documentation of the code is made with Javadoc and it is set to be generated with the build made with Maven. All the generated files are packed with this document in order to check it.

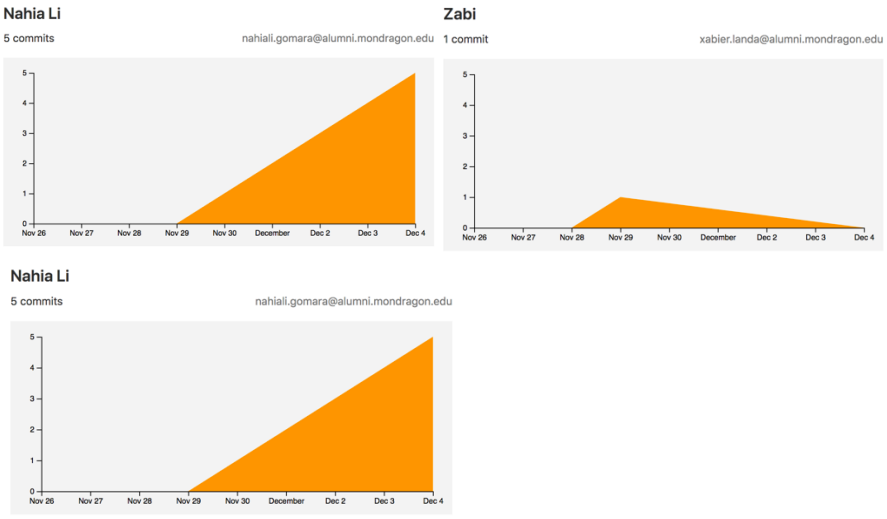
VERSION CONTROL

For the development we are using the following repository: https://gitlab.danz.eus/m2gi13/1920_popbl_5_t4/loglas_project

In which we have made these branches:

- Develop
- springFramework
- SpringHotFix
- Master

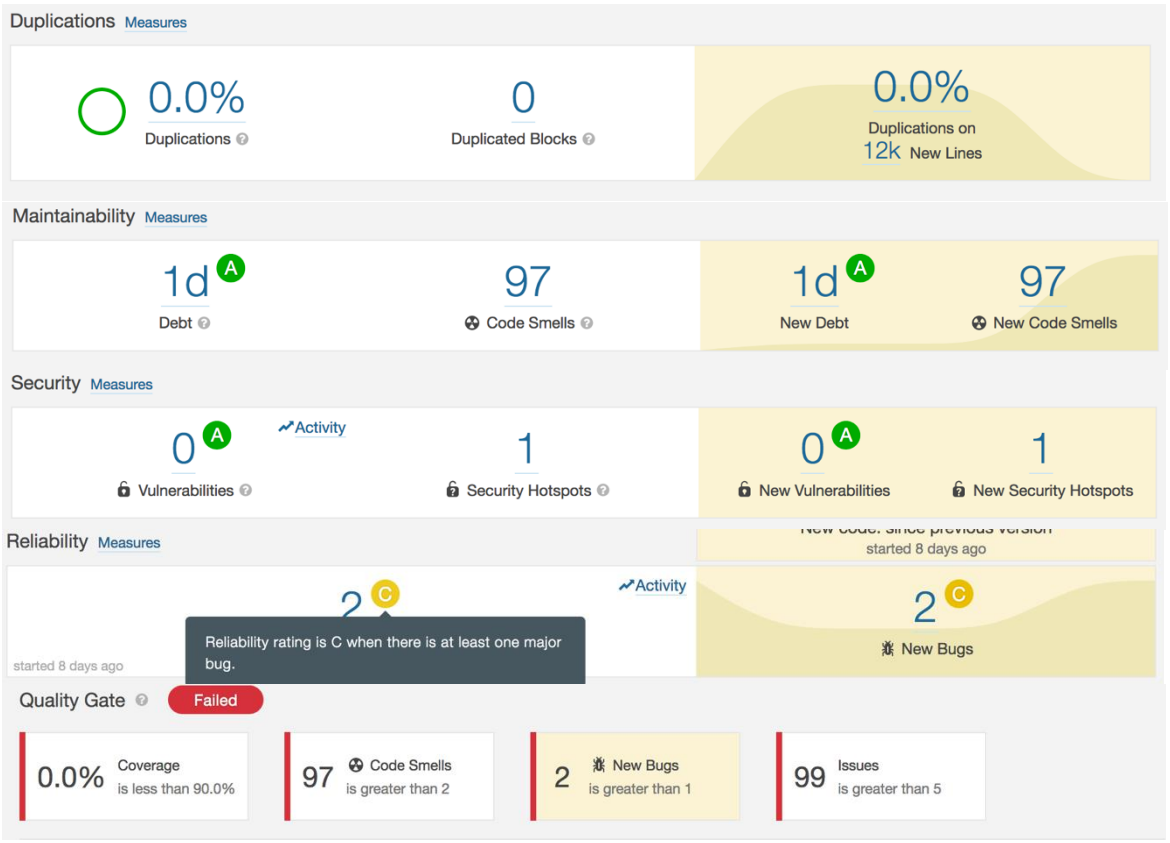
Here are shown the commits made by the developers on the master branch by the developers.



STATIC ANALYSIS

The static analysis is set to make a scan whit sonar each time a developer makes a commit to the repository. Then the information about the quality of the code can be found in our server http://loglas.duckdns.org:9000

But as the server isn't always up, to consume less, each developer can make the static analysis locally on its computer so that it is possible to check the quality of the code that has not been yet implemented.



AUTOMATIC BUILDS

To make the automatic builds with Maven a pom.xml was set and all the dependencies that were needed were packed into it.

Each developer can make its local builds, in order to check if everything is working before doing any change on the repository.

CONTINUOUS INTEGRATION

The continuous integration made with Jenkins in this moment of the development was just a basic webHook to trigger the changes in the repository from our server, but didn't still make any other task such as scanning the code or building and deploying it.

WEEK 2

VERSION CONTROL

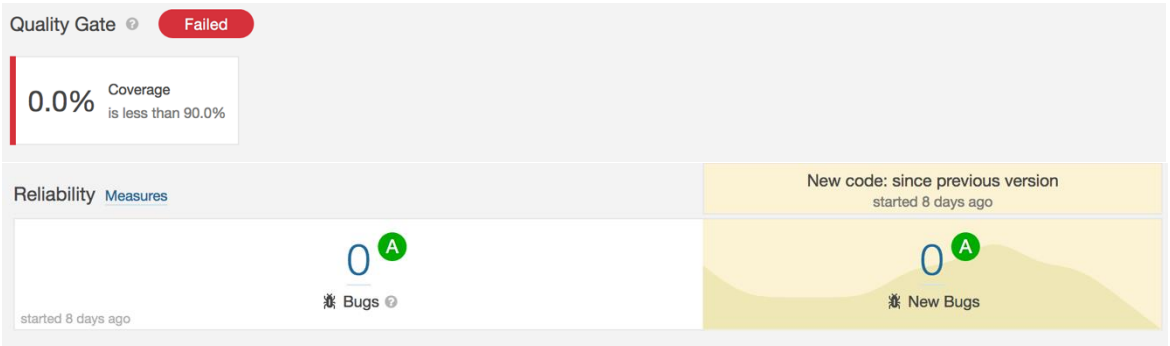
The existing branches of the repository at this point is the following:

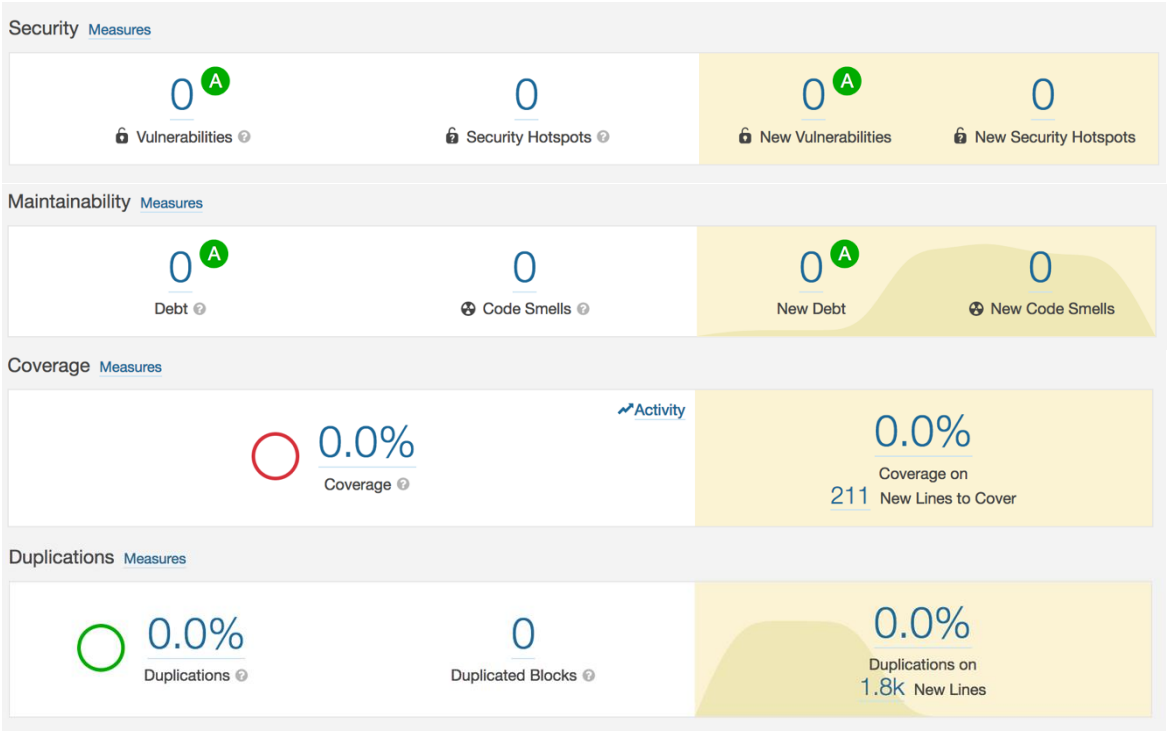
- lobbyFeature
- develop
- springSecurity
- springFramework
- WebSockets
- SpringHotfix
- master

And here are the actual developers commits on the develop branch:

STATIC ANALYSIS

All the code smells and bugs on the develop branch were corrected at this stage of the project.





PROBLEMS: The JQuery library needed to be excluded on the scan because the most of the code smells were triggered by it.

TODO: Implement jacoco to show the coverage of the code, yet it is not implemented.

AUTOMATIC BUILDS

Apart from the builds made by the developers a build was made each time a new version arrived to the repository, when these build is made an executable is generated and the application is deployed on the server.

CONTINUOUS INTEGRATION

In order to automate as much as possible the repetitive things, like building, testing, deploy… a task was generated with Jenkins, it is set to be triggered by a Webhook, caused by a change in the develop branch of the repository, which will lead first to scan the code by Sonarqube and once that the scan is made it will make a maven build and deploy the application in the following link: http://loglas.duckdns.org:8181/

Here we can see two tasks that ran triggered by GitLab, one of the tasks went well and the other one failed on the pipeline.

TODO:

- Different pipelines, one for the develop branch, to scan the quality of the code and do the testing of the project. And the other one to deploy it once per week to the master branch and to the server.
- Generate the Javadoc with the pipeline instead of generating it on the local machines of the developers.

TESTING

The testing is not yet implemented in the project, it is the next step that will be done in order to achieve our coverage goal.

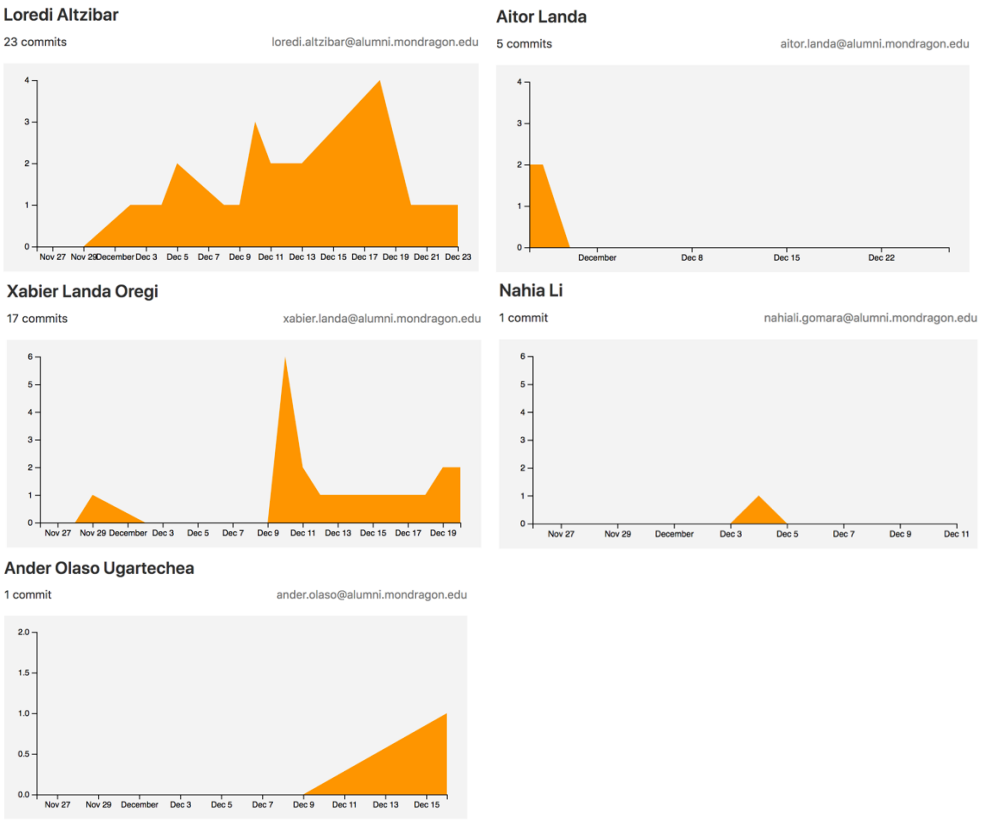
ADDITIONAL COMMENTS & FUTURE LINES

For the week 3 we want to achieve the testing, implement it with Jenkins and use Katalon also in the testing.

WEEK 3

VERSION CONTROL

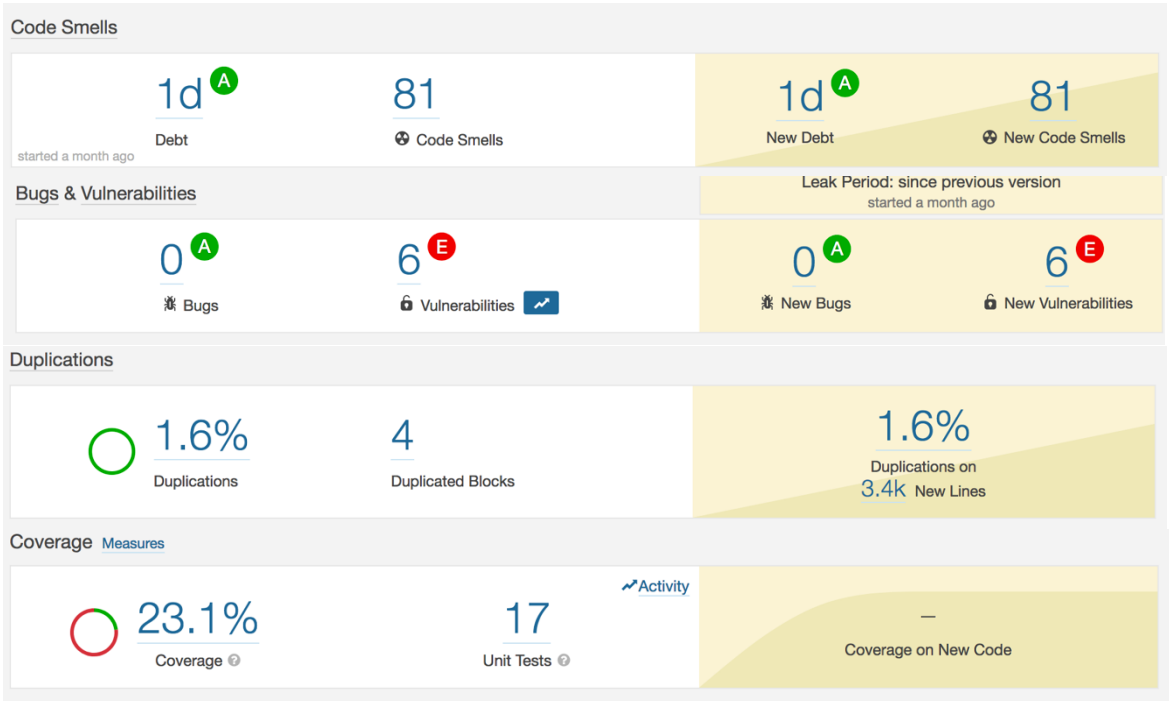
Commits made by the developers until week 3 of the project:



STATIC ANALYSIS

Must of the code smells and vulnerabilities made during this week of the project weren´t fixed.

Most of the vulnerability issues were made by the unit test classes, which needed to use passwords in order to test the functionality of the user creation, login, etc. So those weren´t really vulnerability issues, because just the passwords of the users created for were stores on the code.



At this point we implemented jacoco correctly in order to be able to show our projects coverage on the sonar panel. We can see that the coverage at this point was 23% on the develop branch, so the new functionalities we were developing did not have the unit tests.

AUTOMATIC BUILDS

Apart from the builds made by the developers a build was made each time a new version arrived to the repository, when these build is made an executable is generated and the application is deployed on the server.

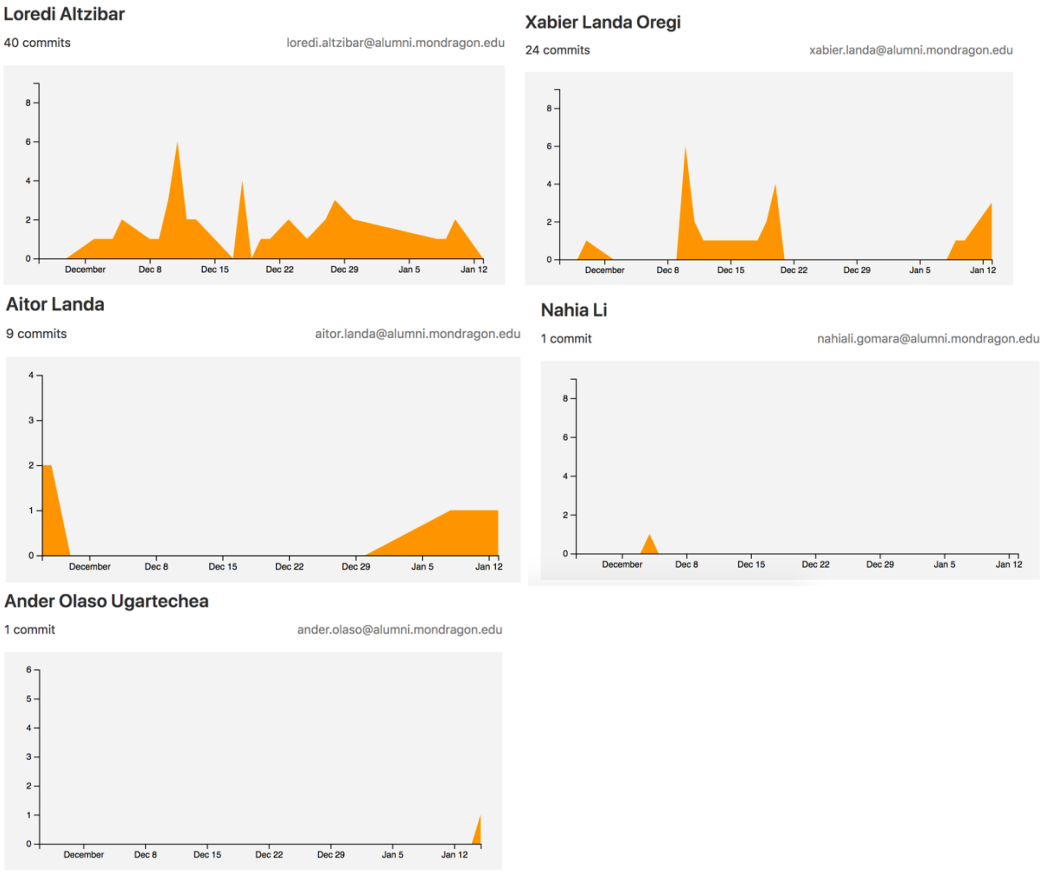
CONTINUOUS INTEGRATION

The continuous integration did not change at this stage as it was well setted.

WEEK 4

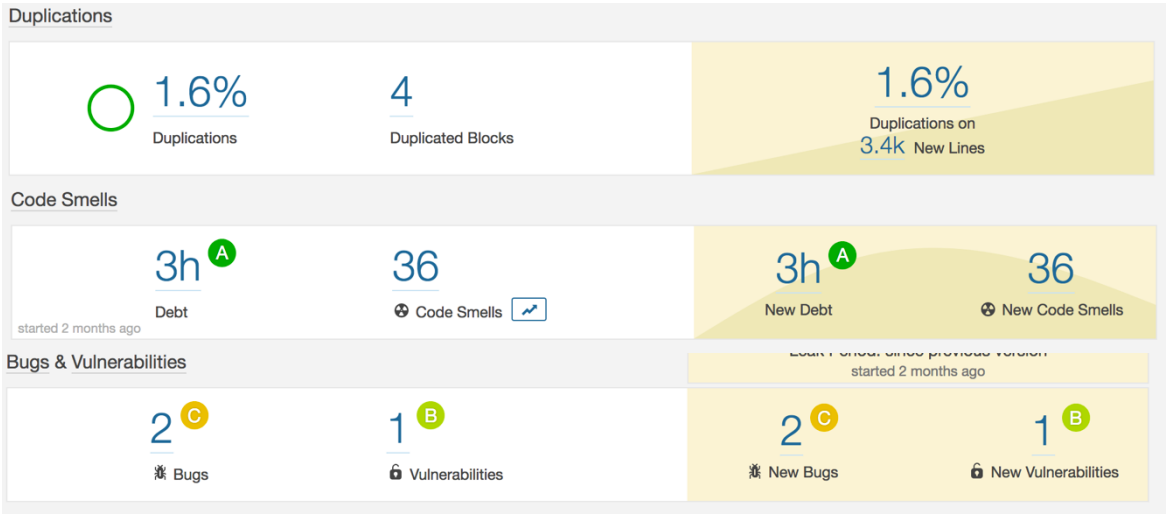
VERSION CONTROL

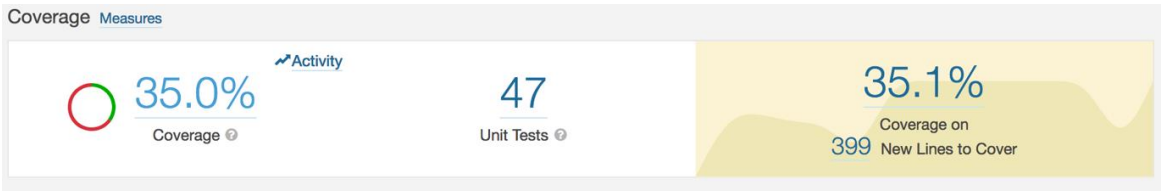
Commits made by the developers until week 4 of the project:



STATIC ANALYSIS

At this point most of the vulnerabilities were corrected and the debt was reduced strictly.





The coverage at this point of the development is of the 35%.

We also started with the black box testing. To do so we installed katalon in the local machines, with the hope to install it on the server as soon as the springboot issue is corrected.

We started doing some tests like testing user creation, valid user login or invalid user login. And we packed them on a test suite, UserTestSuite:

1	Test Cases/createUser
2	Test Cases/testValidUser
3	Test Cases/testInvalidUser

TODO: Implement Katalon in the server and launch it with the other tests.

AUTOMATIC BUILDS

A change has been made with the dependencies, in order to migrate from tomcat to springboot, some problems occurred during this week, in order to launch the application correctly, but it is now solved and can be launched with spring-boot.

CONTINUOUS INTEGRATION

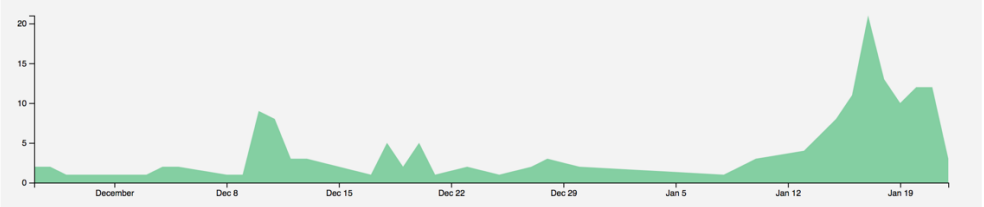
The continuous integration changed at this point in order to execute the application with springboot instead of tomcat. It caused some problems with the dependencies and the code, so we started correcting it with high priority.

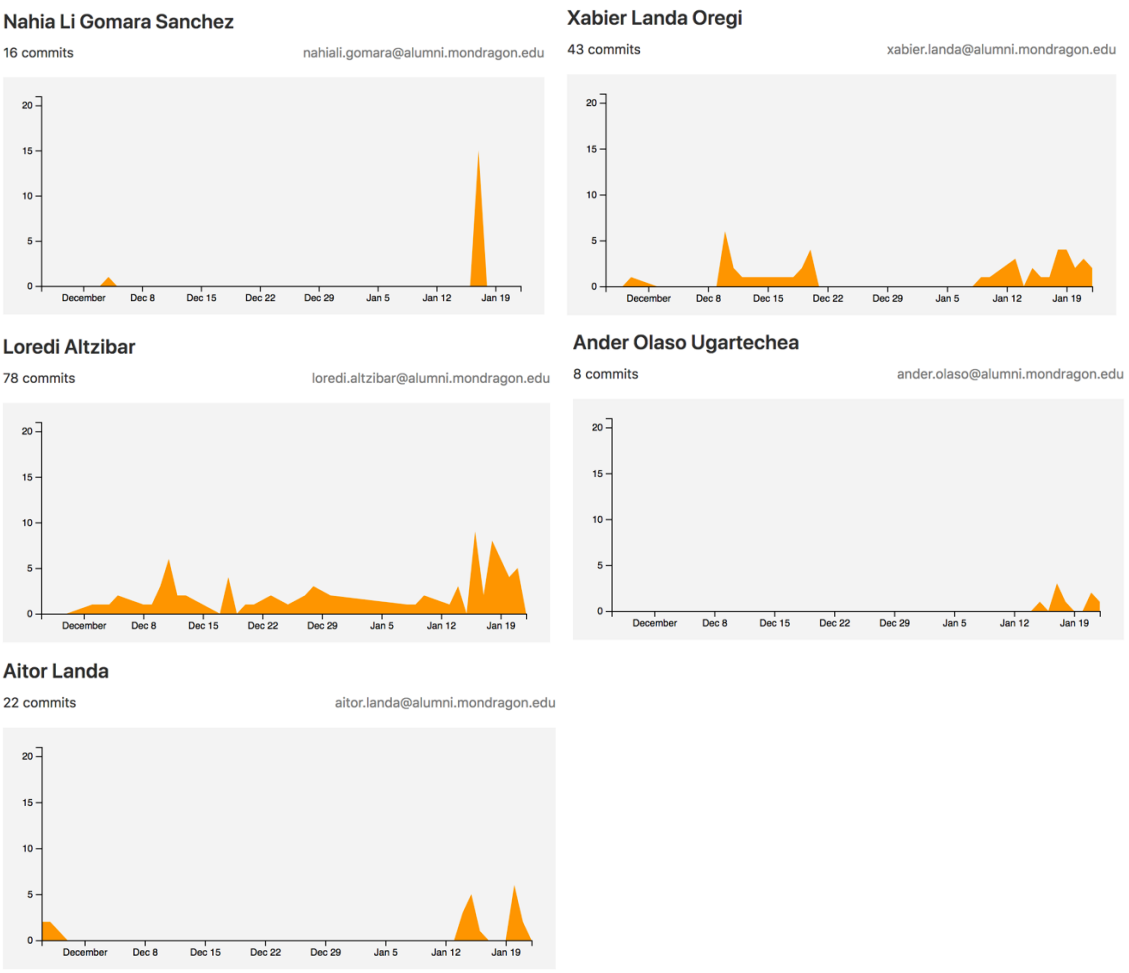
We changed the Jenkins pipeline to execute the application with the embedded tomcat using spring-boot instead of using the tomcat server itself.

3. PART 3 FINAL STATUS

VERSION CONTROL

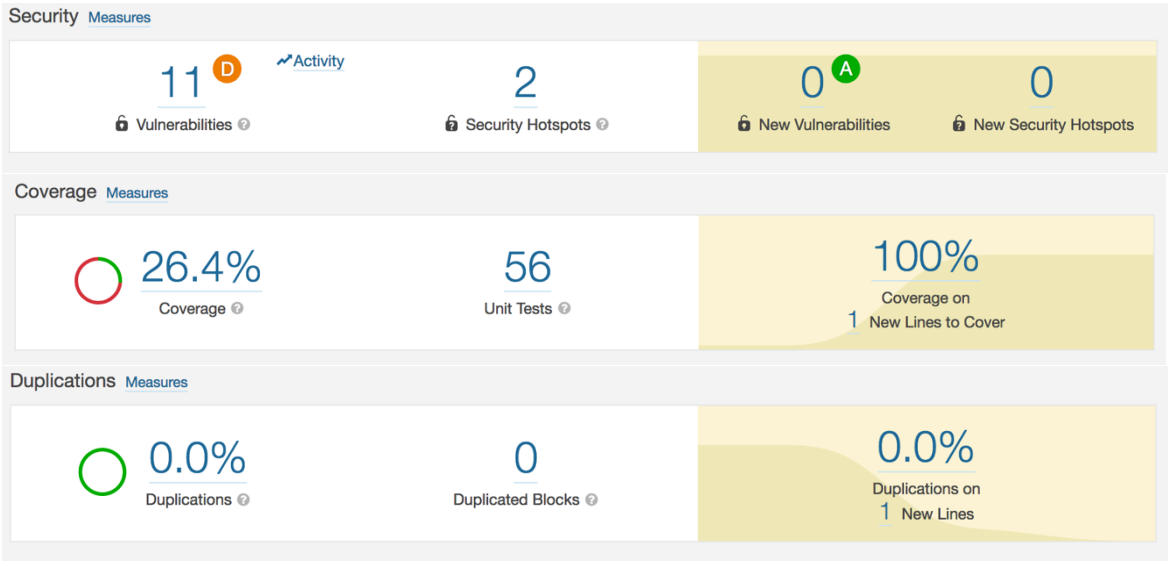
Here is provided the final status of the commits made by the developers to the Gitlab repository:





STATIC ANALYSIS

This is the status of the quality of the code on the last scan made on the project:



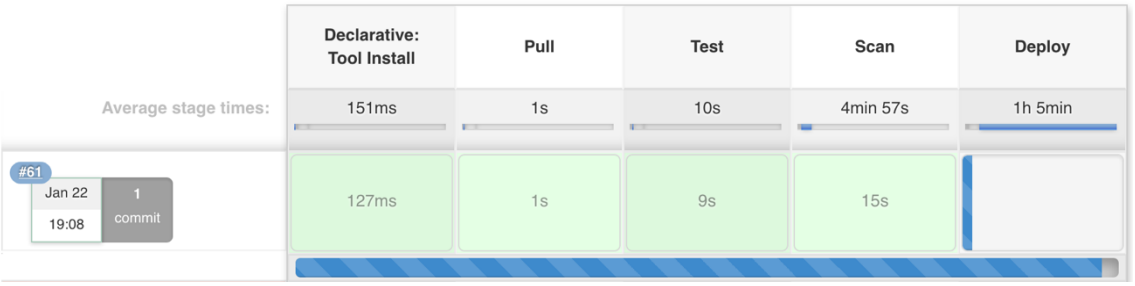
We can see that the quality of the code decreased on the final stage of the project, which happened due to the need of implementing functionalities that weren't implemented before and the lack of time to correct the issues that appeared.

AUTOMATIC BUILDS AND CONTINUOUS INTEGRATION

The automatic builds and continuous integration of the project worked correctly, which helped a lot to the developer team in order to automatize the process of building, testing and scanning the project.

In the following image can be seen a pipeline that worked correctly during all of its steps:

1. The pipeline is triggered when a change is made to the repository
2. It downloads the project from the repository
3. If it succeeds, then it passes all the test cases made with JUnit
4. If there were no errors with the tests it makes the scan to the sonarqube server
5. And finally it launches the application on the server



TESTING

The testing of the project has 2 different parts, one for the unitary tests for the white box testing, and the other one with katalon, to check that the functionalities and navigation are working properly.

The unitary tests are made automatically with Jenkins each time a developer makes a commit to the repository on de develop branch, this is automatized so that each time a change is made it runs the test in order to check that every functionality still works as it is expected.

On the other hand, Katalon is installed on the local machines of the developers, this is due to a failure that happened with the server in which everything was installed, so all the tools needed to be installed again on the server from scratch and it was not possible to install Katalon on the server.

We have made different test suites with Katalon, and each one of the test suites have various tests, which are:

1. UserTestSuite
    a. TestCreateUser
    b. TestInvalidUser
    c. TestValidUser
    d. TestProfile
    e. TestLogout
2. LobbyTestSuite
    a. TestCreateLobby
    b. TestJoinLobby
3. AdminTestSuite
    a. TestAdminLogin
    b. TestAdminPanel
4. LanguageTestSuite
    a. TestLanguageChange

Each one of the tests interacts with the page and buttons and in the end checks if the loaded page is the correct one, or if it stays in the same page, for example if it is testing an invalid user.

REFACTORING

The refactoring of the code is a very important practice, in order not to have bugs, or several errors on in the application that could lead to problems, so it is a good practice to refactor the code and to leave it with the least errors possible.

We have been refactoring the code each week, in order to maintain the code as clean as possible. Except form the last week, in which we have not refactored yet.

CONCLUSIONS

When it comes to the learning we have gained a lot of experience, due to the different tools and techniques that we have used and the problems that we have faced during the development.

We have learnt the importance of the continuous integration in order to automatize repetitive tasks, like testing, and building the project for production or development, it reduces the possibilities of making mistakes in such repetitive tasks and makes the developers not to loose time doing them.

It is also important to maintain the repository with significant names, so that every commit can be identified easily and reduce misunderstanding between them.

The use of the comments in the code is also a very important part, it reduces the time that the developers, other than the one developing that specific functionality, have to

spend to understand de code and reduces the need to explain it to the other developers. It is also important to generate de documentation of the code so that anyone can read it and understand the functioning of the application.

The most important thing is to define all the infrastructure from the very beginning of the project and configure it as soon as possible, before even starting to develop the code so that afterwards the work that needs to be done is much easier, in this case automatic and the development process should flow much better.

FUTURE LINES

On the future it would be a good approach to install Katalon on the server and to launch its tests automatically, along with the unit tests.

We also want to implement another functionality with Jenkins, so that whenever something happens that caused an error on the pipeline, like a test failure or the quality not passing, an email is sent to the developer that has made that commit so that the status is not needed to be checked inside Jenkins.

Another good technique during the development process should be to use a pair programming method, so that each new functionality development has two programmers assigned to it, and develop simultaneously the functionality by one developer and the test cases for it by the other, it should ensure that every functionality has its own test cases and the programming should flow better because the developers could help each others easily.