Institute of Information Systems Engineering

Software Quality Management

2020/2021 - Summer Semester

Software Quality Report University Information System

Carla Cruz 12016293

Yana Peycheva 01634001

Bernhard Kirschner 01425897

August 23, 2021

1 Continuous Integration Concept

In this section our Continuous Integration (CI) Concept is documented. We will start with a comparison of different available CI tools. Based on this comparison we will decide which tool to use and which setup to configure.

1.1 Comparison of Tools

There are many different tools to accomplish CI (examples: Jenkins, Buildbot, TeamCity, Bamboo, Codeship, Gitlab CI). In this section we will compare three successful and well known CI tools based on different properties.

Tool	**free version available?**	**Usability**	**Documentation available?**	**Popularity**
Jenkins	yes	easy	Well documented, lots of experienced users online	broadly used (different kinds of project)
GitLab-CI	yes	easy	Great documentation online, easy to understand and modern looking	only usable for GitLab projects
TeamCity	yes (limited functionality)	medium	Documentation online, a bit confusing	medium

Table 1: Comparison of CI Tools

Subjective information of this table was taken from different comparisons online, we do not have enough experience with all of these tools to objectively compare usability [7] [6]. Overall the tools all seem to have their advantages and drawbacks, but their most important aspect is to be usable in our environment without any additional costs. Also we do not have the time to test all available tools with a test license in a testing environment. Thus we have to make a decision on the limited data available.

Considering the simple documentation, extensive capabilities, the fact that no costs arise and that we do not have the possibilities to host a service, but rather use the available infrastructure, we have decided to use GitLab-CI.

After some research and reading documentation we found our self confirmed in our decision, because according to available documentation it should be possible to use Gitlab-CI with many different tools and technologies.

1.2 Proposed configuration for build jobs

With every commit to the master branch the whole code should get built and tested. Besides the dependency of commits, we also integrate a nightly build to constantly obtain a new build every morning. To build the given project three jobs are necessary:

1. Job to build maven project
2. Job to build Node.js code
3. Job to automatically perform tests

We base our build jobs on docker containers. Thus different docker images are used (i.e. maven:3-jdk-11 and node:12-alpine).

1.3 Setting up GitLab CI

After some research we found that it would be necessary to set up GitLab-CI runners and build pipelines. But since we did not have access to the admin interface of the project, we decided to try on a test instance of GitLab first.

Firstly a GitLab-CI runner had to be set up locally. Then, the configured runner had to be registered and the GitLab-Runner hat to be installed and started. Lastly a .gitlab-ci.yml file was created in the project repository [1].

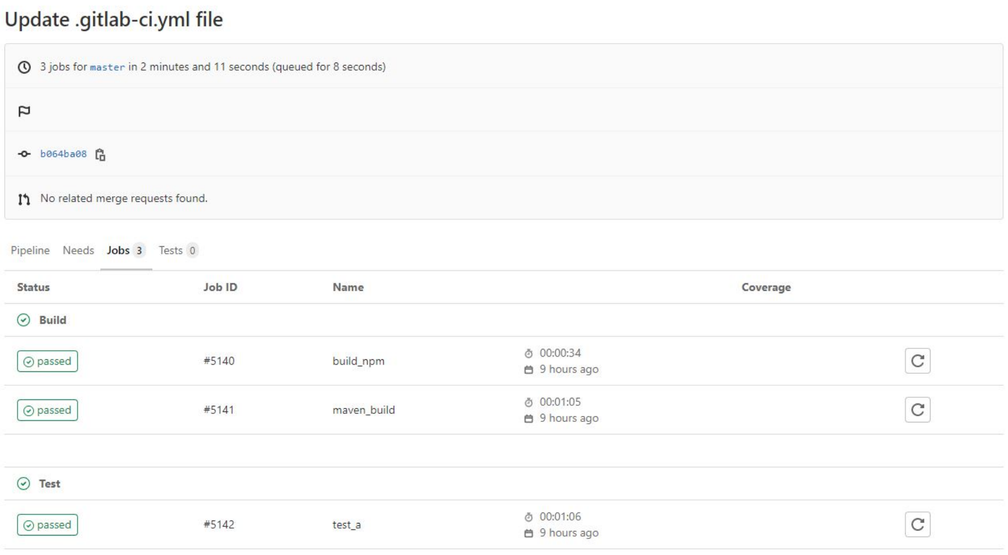
After some tries we realised that the setup of the test instance was not necessary, so we dropped it and relied solely on the provided GitLab instance (peso.inso.tuwien.ac.at), which worked well. Obviously GitLab-Runners are available for the project and we can use them to implement CI pipelines.

To start a pipeline and use CI capabilities in GitLab, a .gitlab-ci.yml file has to be created and placed in the root folder of the repository. Our first draft of the `.gitlab-ci.yml` file configuration looks as follows (changes may apply after this point):

```
stages:
  - build
  - test

image: maven:3-jdk-11

build_npm:
  stage: build
  image: node:12-alpine
  script:
    - cd application/src/main/resources/static
    - npm install

maven_build:
  stage: build
  #image: maven:3-jdk-11
  script: mvn install -B

test_a:
  stage: test
  script:
    - mvn clean test
  needs: ["maven_build"]
```

This setup works well. The build takes two minutes and eleven seconds. All three jobs build successfully (see Figure 2).

[1]All setup steps were performed as described in GitLab Documentation [8] [10]



Update .gitlab-ci.yml file

3 jobs for master in 2 minutes and 11 seconds (queued for 8 seconds)

b064ba08

No related merge requests found.

Status	Job ID	Name		Coverage
Build				
passed	#5140	build_npm	00:00:34 9 hours ago	
passed	#5141	maven_build	00:01:05 9 hours ago	
Test				
passed	#5142	test_a	00:01:06 9 hours ago	

Figure 2: Successfull pipeline build

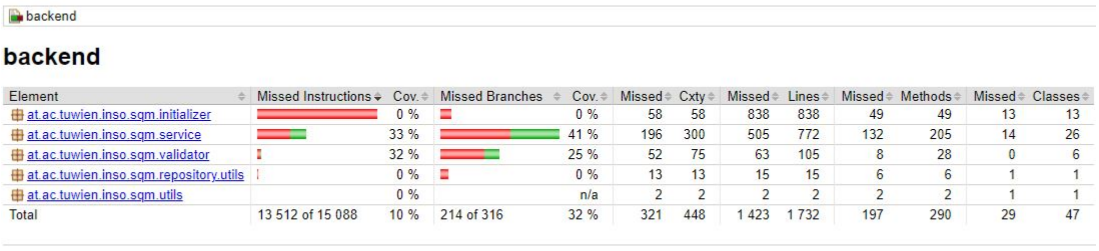
2 Test Coverage

In this section our elaboration of Test Coverage is documented. First we are going to answer project related questions, then we will take care of Tool/ Plugin related questions.

2.1 Project related Questions

2.1.1 Do the coverage results sound reasonable?

The observed coverage results do not sound reasonable. Numbers that low seem to be unlikely and should urgently be tackled by the development and testing team.

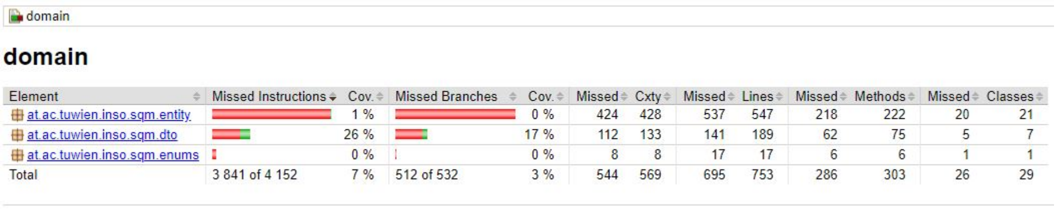


backend

backend

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
at.ac.tuwien.inso.sqm.initializer		0 %		0 %	58	58	838	838	49	49	13	13
at.ac.tuwien.inso.sqm.service		33 %		41 %	196	300	505	772	132	205	14	26
at.ac.tuwien.inso.sqm.validator		32 %		25 %	52	75	63	105	8	28	0	6
at.ac.tuwien.inso.sqm.repository.utils		0 %		0 %	13	13	15	15	6	6	1	1
at.ac.tuwien.inso.sqm.utils		0 %		n/a	2	2	2	2	2	2	1	1
Total	13 512 of 15 088	10 %	214 of 316	32 %	321	448	1 423	1 732	197	290	29	47

Figure 3: Code Coverage from submodule backend



domain

domain

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
at.ac.tuwien.inso.sqm.entity		1 %		0 %	424	428	537	547	218	222	20	21
at.ac.tuwien.inso.sqm.dto		26 %		17 %	112	133	141	189	62	75	5	7
at.ac.tuwien.inso.sqm.enums		0 %		0 %	8	8	17	17	6	6	1	1
Total	3 841 of 4 152	7 %	512 of 532	3 %	544	569	695	753	286	303	26	29

Figure 4: Code Coverage from submodule domain

2.1.2 Is the coverage sufficient for the project?

The observed coverage results are not sufficient to provide any valuable feedback. Usually code coverage of around 80 - 90% is requested [14]. Our observation with Jacoco results in test coverages way below 50%. These numbers are way too low to be considered sufficient. Test coverage that low is not able to guarantee high quality software.

2.1.3 Which parts of the source code are not sufficiently covered by tests?

In this project the highest achieved test coverage percentage is 33% in the package at.ac.tuwien.inso.sqm.service in the module backend. All of the other packages have lower test coverage, some of which have 0 test coverage. This means that the test coverage is insufficient in the entire project and not only in specific parts of it.

2.1.4 Which improvements would you recommend? Which parts of the project would benefit most from improved coverage?

It is highly recommend to implement additional tests in the whole project. This would help find errors and bugs in the code and it would improve the overall quality in the project.

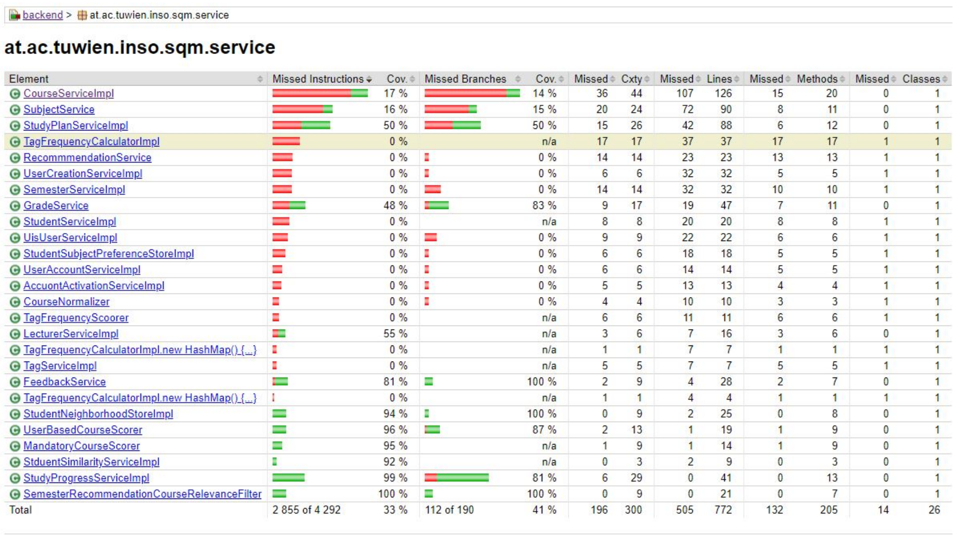
2.2 Tool/Plugin Related Questions

2.2.1 Why did you choose the respective tool(s)/ plugin(s)?

Jacoco is a test coverage tool for Java. One of the reasons for choosing Jacoco is the easy integration with maven. Because of the fact that the project UIS uses for build automation tool maven, we could easily add Jacoco as a plugin in the pom.xml file. Furthermore, Jacoco offers different levels of granularity of the representation of the covered code. This makes it easy for different stakeholders to understand the report, for example a project manager can just have an overview of which modules in the code are better covered, and a tester can go into a more detailed view and check for which code parts exactly tests should be added. Another reason for our team to choose Jacoco is that two of us had experience with this plugin and were satisfied by it.

2.2.2 How is the granularity and structure of the results?

Jacoco provides an overview for the whole project. In this special case a project is a submodule (backend or domain). By clicking on packages in the overview a more detailed insight is provided, which lists details for every class inside a package (see Figure 5).



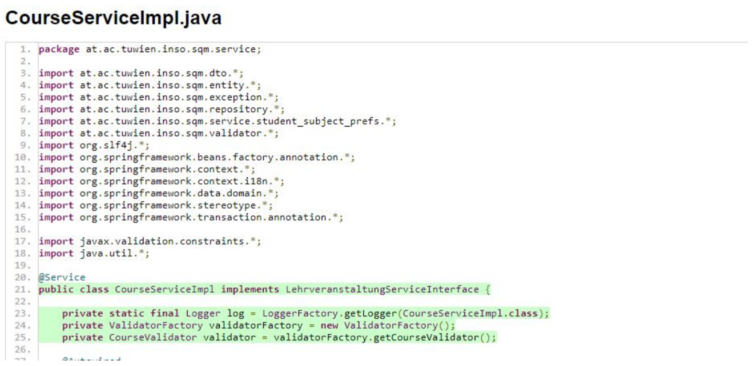
backend > at.ac.tuwien.inso.sqm.service

at.ac.tuwien.inso.sqm.service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CourseServiceImpl		17 %		14 %	36	44	107	126	15	20	0	1
SubjectService		16 %		15 %	20	24	72	90	8	11	0	1
StudyPlanServiceImpl		50 %		50 %	15	26	42	88	6	12	0	1
TagFrequencyCalculatorImpl		0 %		n/a	17	17	37	37	17	17	1	1
RecommmendationService		0 %		0 %	14	14	23	23	13	13	1	1
UserCreationServiceImpl		0 %		0 %	6	6	32	32	5	5	1	1
SemesterServiceImpl		0 %		0 %	14	14	32	32	10	10	1	1
GradeService		48 %		83 %	9	17	19	47	7	11	0	1
StudentServiceImpl		0 %		n/a	8	8	20	20	8	8	1	1
UisUserServiceImpl		0 %		0 %	9	9	22	22	6	6	1	1
StudentSubjectPreferenceStoreImpl		0 %		0 %	6	6	18	18	5	5	1	1
UserAccountServiceImpl		0 %		0 %	6	6	14	14	5	5	1	1
AccuontActivationServiceImpl		0 %		0 %	5	5	13	13	4	4	1	1
CourseNormalizer		0 %		0 %	4	4	10	10	3	3	1	1
TagFrequencyScoorer		0 %		n/a	6	6	11	11	6	6	1	1
LecturerServiceImpl		55 %		n/a	3	6	7	16	3	6	0	1
TagFrequencyCalculatorImpl.new HashMap() {...}		0 %		n/a	1	1	7	7	1	1	1	1
TagServiceImpl		0 %		n/a	5	5	7	7	5	5	1	1
FeedbackService		81 %		100 %	2	9	4	28	2	7	0	1
TagFrequencyCalculatorImpl.new HashMap() {...}		0 %		n/a	1	1	4	4	1	1	1	1
StudentNeighborhoodStoreImpl		94 %		100 %	0	9	2	25	0	8	0	1
UserBasedCourseScorer		96 %		87 %	2	13	1	19	1	9	0	1
MandatoryCourseScorer		95 %		n/a	1	9	1	14	1	9	0	1
StduentSimilarityServiceImpl		92 %		n/a	0	3	2	9	0	3	0	1
StudyProgressServiceImpl		99 %		81 %	6	29	0	41	0	13	0	1
SemesterRecommendationCourseRelevanceFilter		100 %		100 %	0	9	0	21	0	7	0	1
Total	2 855 of 4 292	33 %	112 of 190	41 %	196	300	505	772	132	205	14	26

Figure 5: Code Coverage from Service Package placed in the backend project

For even more details a specific class can be selected. Lastly to find out problems in the implementation a single file can be selected and will be viewed in detail (see Figure 6).



CourseServiceImpl.java

```
package at.ac.tuwien.inso.sqm.service;

import at.ac.tuwien.inso.sqm.dto.*;
import at.ac.tuwien.inso.sqm.entity.*;
import at.ac.tuwien.inso.sqm.exception.*;
import at.ac.tuwien.inso.sqm.repository.*;
import at.ac.tuwien.inso.sqm.service.student_subject_prefs.*;
import at.ac.tuwien.inso.sqm.validator.*;
import org.slf4j.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.context.*;
import org.springframework.context.i18n.*;
import org.springframework.data.domain.*;
import org.springframework.stereotype.*;
import org.springframework.transaction.annotation.*;

import javax.validation.constraints.*;
import java.util.*;

@Service
public class CourseServiceImpl implements LehrveranstaltungServiceInterface {

    private static final Logger log = LoggerFactory.getLogger(CourseServiceImpl.class);
    private ValidatorFactory validatorFactory = new ValidatorFactory();
    private CourseValidator validator = validatorFactory.getCourseValidator();

```

Figure 6: The implementation of CourseServiceImpl.java as depicted by Jacoco's report

In summary the structure of the results is easy to use and navigate. The different levels of granularity offered by Jacoco are helpful to find the exact information each user is looking for.

Jacoco creates reports as HTML files which are easily portable and can be used for further analysis.

2.2.3 Which stakeholders might benefit from the results?

The results have to be analysed and used to improve testing. Thus they benefit most stakeholders (some are only affected indirectly). Obviously test engineers and developers have to implement tests to fulfill the testing coverage which are defined beforehand and thus required by management.

Quality control managers, project managers and product owners benefit from the results because they get an overview of the whole project. They might not need the detailed information a software engineer needs to implement fitting new tests, but they still get a sense of progress similar to a burndown chart.

In summary also the customer benefits from those results. If the project gets tested more thoroughly the probability is higher to get a better product.

2.2.4 Can the results be exported as a report? How does the representation look like?

The provided results from Jacoco are already a report in HTML form. To see the representation see the subsection *How is the granularity and structure of the results?*. The setup of the continuous integration pipeline creates an artifact which can be downloaded to view the results. To do so, please go to the pipeline overview of gitlab and select the dropdown on the right side of the pipeline you want to know more about (see Figure 7).

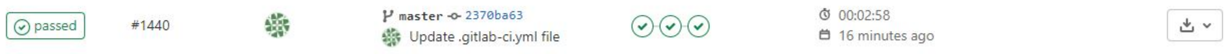


Figure 7: The archive containing Jacoco Reports can be downloaded via the pipeline interface on Gitlab

2.2.5 Do(es) the tool(s)/plugin(s) qualify for further use in this project? Why (not)?

Jacoco qualifies for further use in the project. If the project expands, for example the client wants to add new additional features to the application, they still need to be tested and project test coverage report would be essential for finding errors and bugs in the code. Jacoco can then be easily configured to create coverage reports, additionally, for these new features.

3 SonarQube Integration

Discuss and interpret the general quality state the project is currently in by observing results for technical debt, reliability and security and document your insights in the report.:

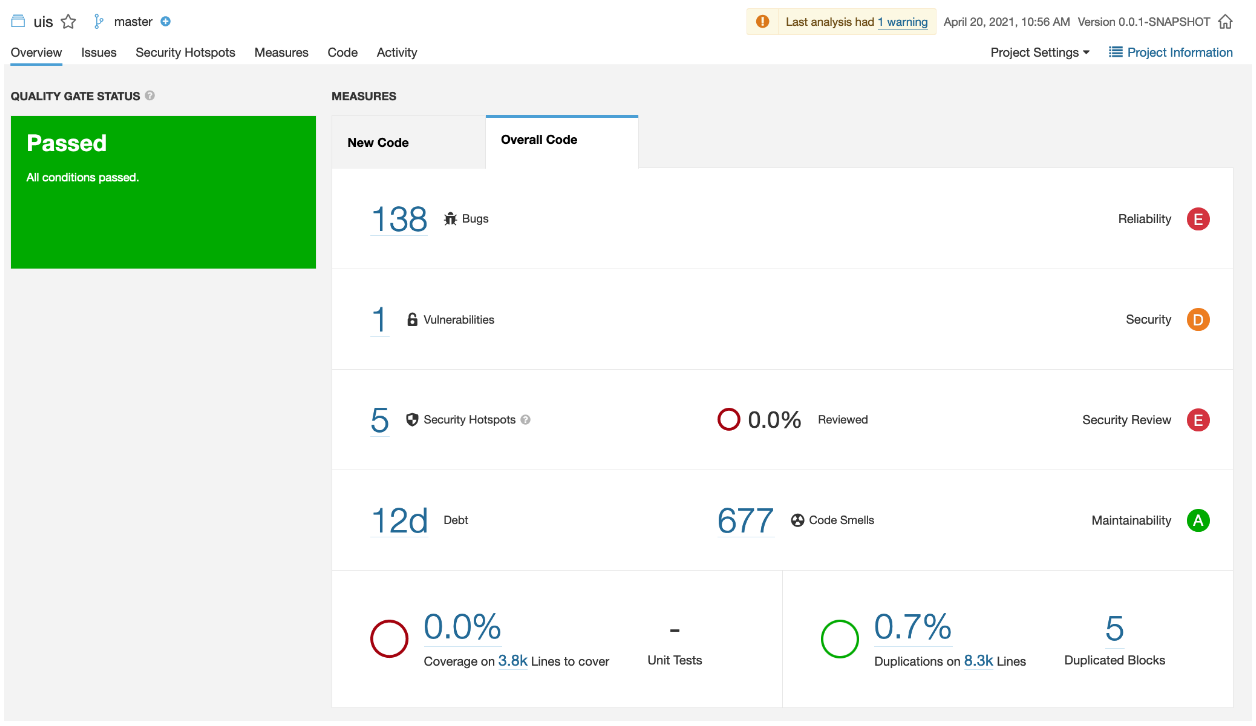
As we know, ISO/IEC 25010 classifies software quality in a set of characteristics, some like reliability, security and maintainability. In this project, according to the general information given by Sonarqube, we can observe that we have a debt of 12d and 677 code smells. Related to the reliability, we have 138 bugs and in the security part, we have 5 security hotspots and 0.0 % Reviewed. Based on this analysis we can realize that the project has a lot of problems, not being the quality that we expect in a project like this.

In this project there are 3 types of issues: vulnerabilities, code smells and bugs. Vulnerabilities are places in the code, which are open to hackers attack. Bugs are code errors, and code smell can have impact on the maintainability of the project. There are categorised using Severity levels: The highest severity level is Blocker, which means there is a high probability that this bug impacts the behaviour of the application. The lower level is Critical, these issues have a critical impact on the application but there not really likely to occur. Severity level Major includes the problems in project which are likely to occur but won't have a big impact on the behaviour of the software. The next level is Minor, the issues which are classified in Minor level are not that probable to occur and may have only minor influence in the programs behaviour. Info is the lowest level of severity, and it's assigned to no bugs or issues, rather they just give some finding or information.

In the project there is one vulnerability. It is classified with Critical severity. The problem is that passwords are stored as plain text. This is an issue, because if an attacker gains access to the database then they will also be able to see all of the passwords. This vulnerability hides a big risk for the project security and it should be fixed immediately.

The bugs (coding errors) in the project are 138 Bugs and from them 3 have severity ranking Blocker and can lead to a runtime exceptions. All of the other Bugs in the code are with Major or lower severity ranking. The total number of code smells is 676 and the ones which are with severity Blockers are 21. From them 16 are missing assertions in the test cases and can be easily fixed.

When fixing the issues in the project, they should be prioritized according to their severity ranking. It is highly recommended to first fix the issues classified with Blocker severity ranking, because they could cause the biggest impact on the application and are most probable to occur.


uis master
Last analysis had 1 warning April 20, 2021, 10:56 AM Version 0.0.1-SNAPSHOT
Overview Issues Security Hotspots Measures Code Activity Project Settings Project Information

QUALITY GATE STATUS
Passed
All conditions passed.

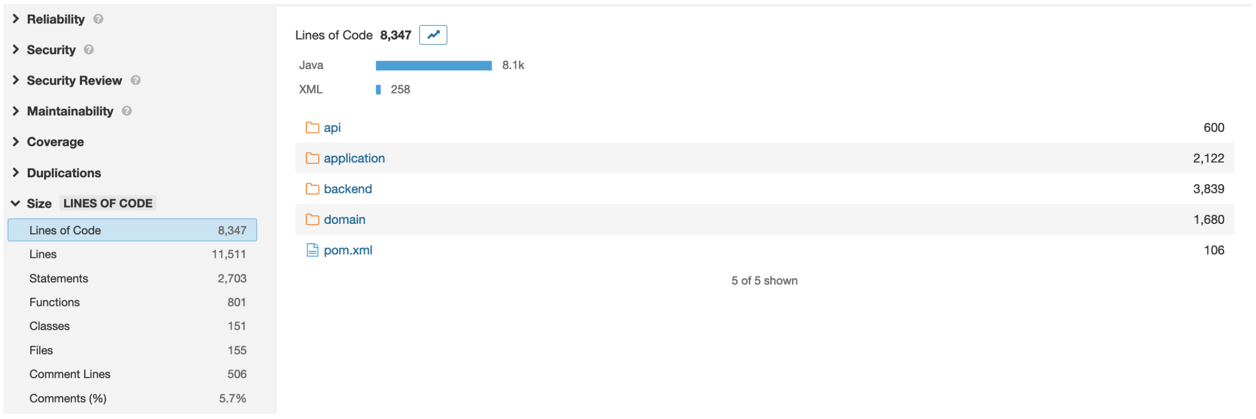
MEASURES
New Code | Overall Code

138 Bugs — Reliability E
1 Vulnerabilities — Security D
5 Security Hotspots — 0.0% Reviewed — Security Review E
12d Debt — 677 Code Smells — Maintainability A
0.0% Coverage on 3.8k Lines to cover — - Unit Tests
0.7% Duplications on 8.3k Lines — 5 Duplicated Blocks

Figure 8: Sonarqube

- Size - 8,347

The size is consider a scope metric. With this parameter we can measure the size of a project, module, class or function This do not consider structure or complexity. In the figure, we can see the LOC is 8,347. It is also possible to measure the total lines and comment lines.


Reliability
Security
Security Review
Maintainability
Coverage
Duplications
Size LINES OF CODE

Lines of Code	8,347
Lines	11,511
Statements	2,703
Functions	801
Classes	151
Files	155
Comment Lines	506
Comments (%)	5.7%

Lines of Code 8,347

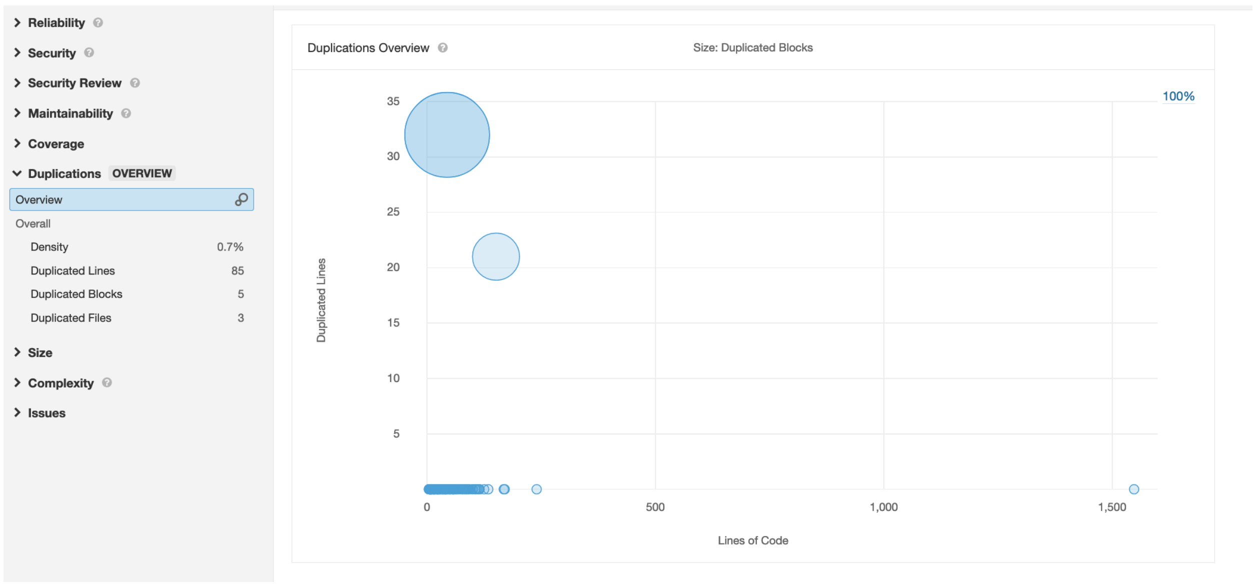
Java	8.1k
XML	258

api	600
application	2,122
backend	3,839
domain	1,680
pom.xml	106

5 of 5 shown

Figure 9: Size - Lines of Code

- Duplications - 0.7 %



Overall

Density	0.7%
Duplicated Lines	85
Duplicated Blocks	5
Duplicated Files	3

Figure 10: Duplications

- Complexity

Using the Complexity measure, we can check how simple or complicated the control flow of the application is. Cyclomatic Complexity measures the minimum number of test cases required for full test coverage. Cognitive Complexity is a measure of how difficult the application is to understand.

Cyclomatic Complexity: 1,207
Cognitive Complexity: 535

With Cyclomatic Complexity number, we realise that are required a lot of test cases for the full test coverage of our application. The Cognitive Complexity is also high, that means that the application is hard to understand.

- Available Plugins

One useful SonarQube plugin is Pitest. It adds additional mutation analyse to our test coverage. When adding Pitest Mutation as quality profile the project analysis with SonarQube shows no bugs. Another interesting plugin is Sonargraph Integration. This tool gives further information about the structural aspects of the project and provides metrics about the cyclic dependencies in the project. There are 2 quality profiles: Sonargraph Integration and SonargraphIntegration (Strict). Both of the quality profiles don't identify any codes smells after the execution of the analysis.

3.1 Which rules are frequently violated?

In total in the project there are 76 rules which are violated. From these ones the 10 most frequently violated rules are the following:

1. String literals should not be duplicated (256 violations). This rule is categorized as a Code smell, it's severity ranking is Critical.
2. BigDecimal(double) should not be used (116 violations). This is rule is categorized as a Bug and its severity ranking is Major.
3. Printf-style formats strings should be used correctly (67 violations). This is a code smell and its severity is Major.

4. Preconditions and logging arguments should not require evaluation (57 violations). This is a code smell with severity Major.
5. Package declaration should match source file directory (28 violations). This is a Code smell with severity Critical.
6. @RequsetMapping method should be public (21 violations). This a Code smell with severity rank Major.
7. Test should include assertions (16 violations). This is a Code smell with severity Blocker.
8. Track uses of "TODO" tags. This is a code smell with INFO severity.
9. Constants names should comply with naming convention. This is code smell with severity Critical.
10. Track uses of "FIXME" tags. This is a code smell with Major severity.

3.2 Are the most frequent violated rules appropriate? Are there false positives?

1. String literals should not be duplicated: This rule is appropriate and it is not a false positive. String literal duplication can be avoided by defining additional constants. If some changes in the duplicates strings occur it would be much easier to change only one constant, than change multiple string literals.
2. BigDecimal(double) should not be used. This rule is appropriate and it is not a false positive. Using BigDecimal(double) can lead to false results, because of floating point imprecision. It can be easily fixed by using BigDecimal.valueOf(double).
3. Printf-style formats strings should be used correctly. This rule is a false positive. Printf-style formats strings are not validated by the compiler, but interpreted at runtime so they can contain errors. The messages even if not checked by compiler don't have any errors. Still the use of parameterized messages is better, read 4.
4. Preconditions and logging arguments should not require evaluation. This code smell can result in performance penalty, because it requires evaluation of the precondition and logging arguments even if it is not needed. For that reason, parameterized messages should be used.
5. Package declaration should match source file directory. This code smell is not checked by the compiler and can affect the maintainability of the project. But this rule is inappropriate, because the packages are correctly defined as relative paths in the project and there is no need to define the full path to the package.
6. @RequsetMapping method should be public. Spring invokes methods via reflections and not visibility, and marking code with private won't be a good way to control how such code is called. This rule is appropriate and it is not a false positive. The private visibility of some methods should be replaced by public.
7. Test should include assertions. This rule is appropriate and it is not a false positive. Without including assertions in the tests the only thing which a test does is check if the executed function executed don't throw an exception. This is insufficient, because the results of the functions also need to be checked. Assertions should be added to the test cases so the result of the tested functions can be actually verified.
8. Track uses of "TODO" tags. This rules is appropriate and it is not a false positive. It often happens that a programmer forgets the TODO tags. A programmer should be aware of these tags and resolve them.
9. Constants names should comply with naming convention. This rules is appropriate and it is not a false positive. The constant names should apply to programmers conventions. Otherwise the readability and accordingly the maintainability of the overall project are worsened.
10. Track uses of "FIXME" tags. This rules is appropriate and it is not a false positive. It often happens that a programmer forgets the FIXME tags. A programmer should be aware of these tags and resolve them.

3.3 Should certain rules be deactivated? If so, define a new quality profile for the project where you customize your rule set.

The following rules should be deactivated in the project:

1. Printf-style formats strings should be used correctly
2. Package declaration should match source file directory should be deactivated.

3.4 What might be the underlying cause for the quality problems?

When developing a bigger project it is common to loose track of the overall quality of the project and with that implement quality problems. Even by thoroughly checking the code in the whole project by programmers, it doesn't exclude missing some bugs and code smells. That's why it's really important to integrate quality tools in the whole life-cycle of the project. The main cause for the quality issues lays in not integrating quality tools and methods in the project.

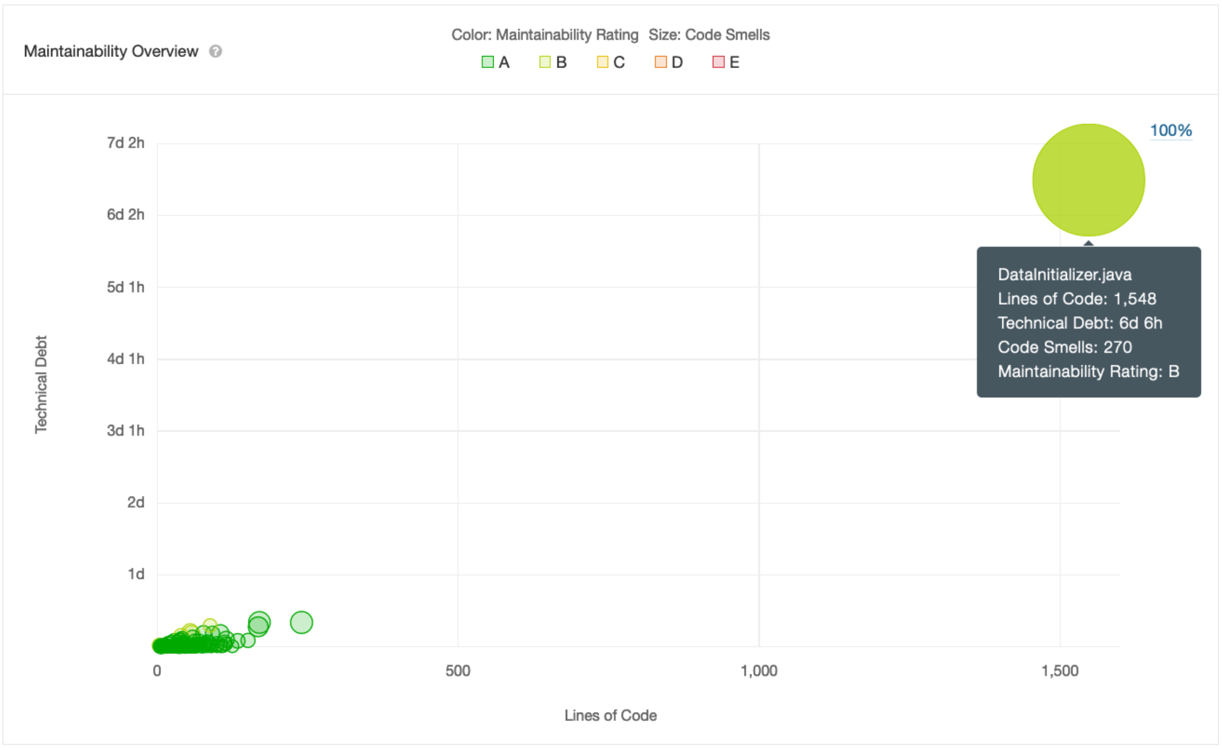
3.5 Which changes/recommendations can be proposed to the project's development team to prevent such problems in the future?

An integration of several tools in the further development of the project:

- checkstyle: This tool would help programmers to fulfil the coding standards and conventions. Before every commit of the programmers the code will be checked and if it doesn't fulfil the with checkstyle defined rules, the code should be fixed. This way it would be possible to prevent implementing further convention and standard problems in the project like some unneeded comment sections, TODO or FIX tags, enum variables conventions and so on.
- Tool for visualizing the code properties, metrics and dependencies , for example SoftVis3D . This would help keep an overview on the how the project complexity develops, track the areas of the project which are more prone to errors and monitor the project metrics overtime.
- When expanding the existing source code and when introducing new features a quality check using Sonarqube is strongly recommended. This would help find the quality issues in time and prevent introducing further technical debt into the project, which would lead to higher overall quality in the project.

3.6 Are there hot spots in certain code modules/packages?

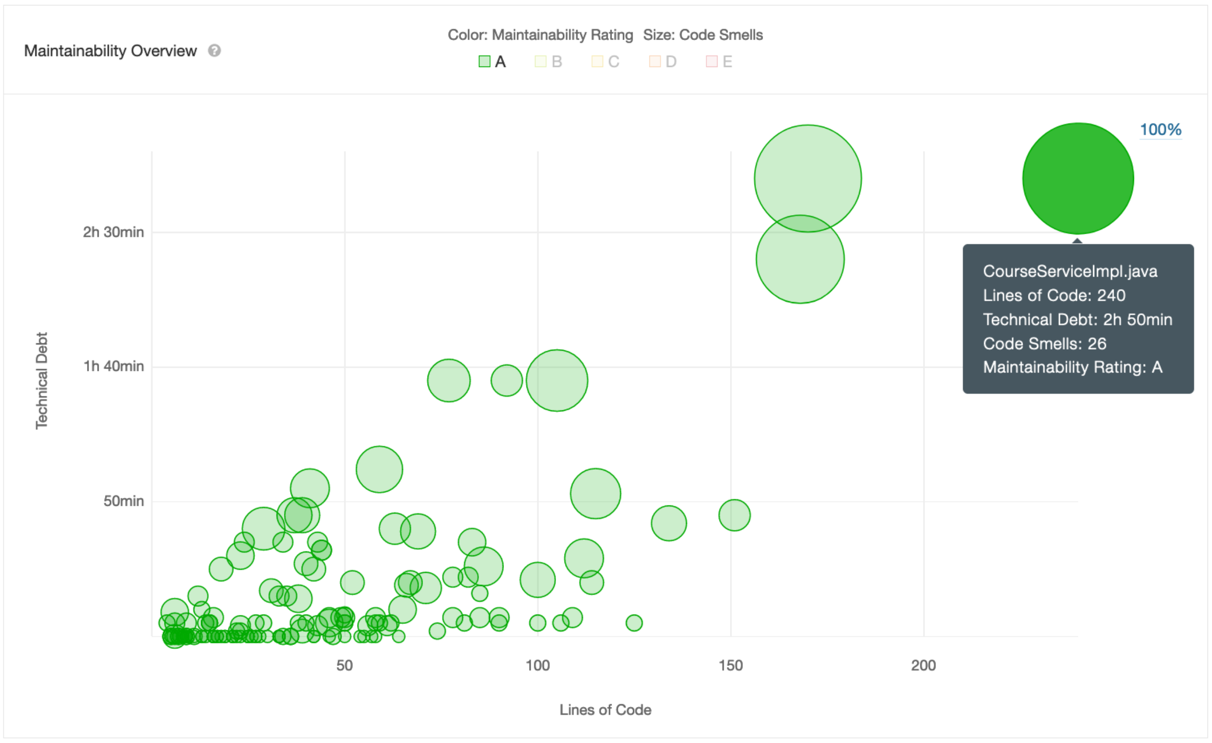
Yes. According to our analysis is possible to see there are hotspots, that means, highlight files that needs attention. The different charts correlate various metrics. In the picture we can see the hotspots in our project:



DataInitializer.java
Lines of Code: 1,548
Technical Debt: 6d 6h
Code Smells: 270
Maintainability Rating: B

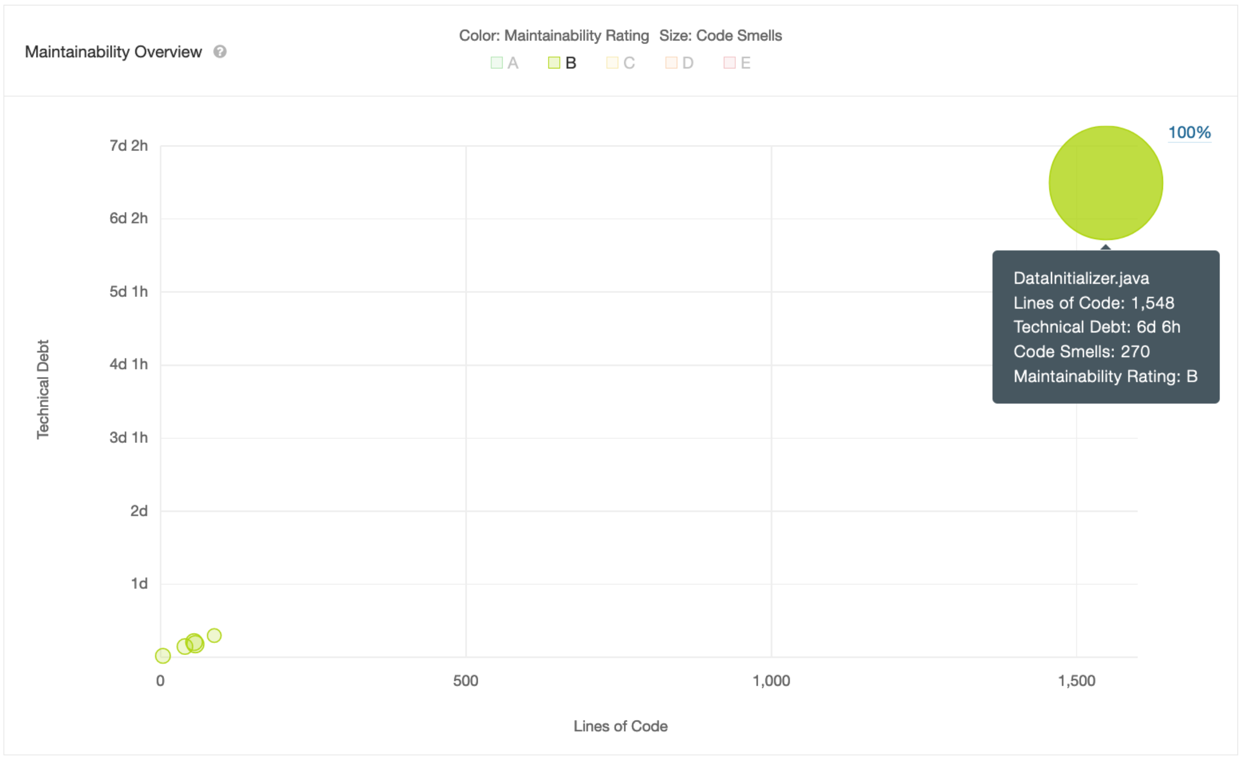
Figure 11: Hotspots

In this analysis, it is possible to filter by the level of maintainability, and in the next pictures we can see with the level A and level B.



CourseServiceImpl.java
Lines of Code: 240
Technical Debt: 2h 50min
Code Smells: 26
Maintainability Rating: A

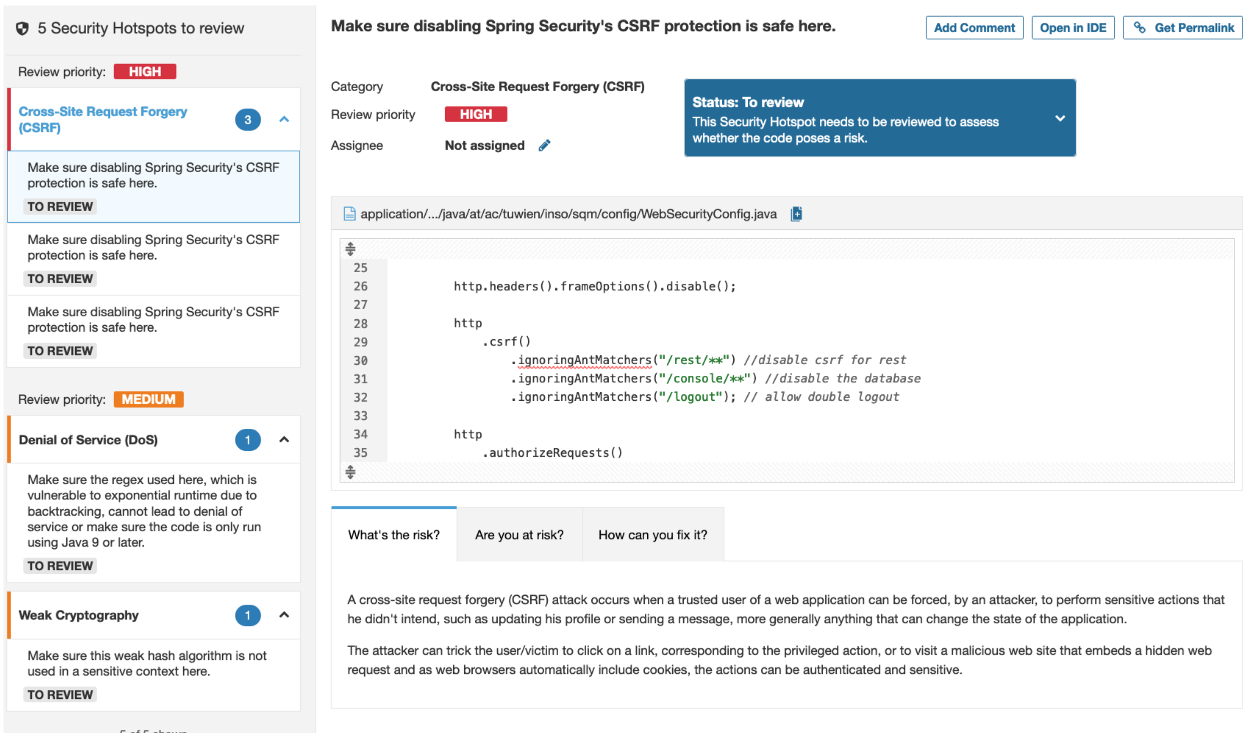
Figure 12: Maintainability Rating: A



DataInitializer.java
Lines of Code: 1,548
Technical Debt: 6d 6h
Code Smells: 270
Maintainability Rating: B

Figure 13: Maintainability Rating: B

In this project is also possible to see that we have security hotspots (5 security hotspots). In which security hotspot we can analyse with more detail what risk is and how to fix it. The Sonarqube also tell us the review priority of which risk, knowing which one is more import to reviwe and solve.



5 Security Hotspots to review

Review priority: HIGH

Cross-Site Request Forgery (CSRF) 3

Make sure disabling Spring Security's CSRF protection is safe here.
TO REVIEW

Make sure disabling Spring Security's CSRF protection is safe here.
TO REVIEW

Make sure disabling Spring Security's CSRF protection is safe here.
TO REVIEW

Review priority: MEDIUM

Denial of Service (DoS) 1

Make sure the regex used here, which is vulnerable to exponential runtime due to backtracking, cannot lead to denial of service or make sure the code is only run using Java 9 or later.
TO REVIEW

Weak Cryptography 1

Make sure this weak hash algorithm is not used in a sensitive context here.
TO REVIEW

Make sure disabling Spring Security's CSRF protection is safe here.

Category	Cross-Site Request Forgery (CSRF)
Review priority	HIGH
Assignee	Not assigned

Status: To review
This Security Hotspot needs to be reviewed to assess whether the code poses a risk.

application/.../java/at/ac/tuwien/inso/sqm/config/WebSecurityConfig.java

```

        http.headers().frameOptions().disable();

        http
            .csrf()
                .ignoringAntMatchers("/rest/**") //disable csrf for rest
                .ignoringAntMatchers("/console/**") //disable the database
                .ignoringAntMatchers("/logout"); // allow double logout

        http
            .authorizeRequests()
```

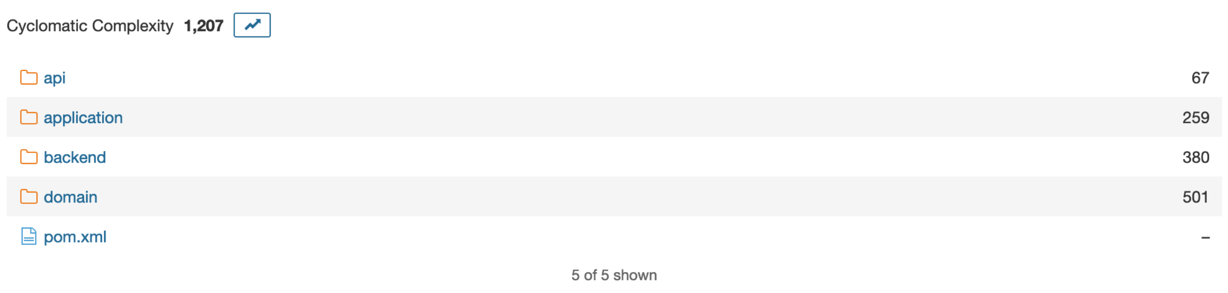
A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn't intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

Figure 14: Security Hotspots

3.7 Which parts of the project have high complexity? Is it desirable/feasible to reduce complexity? Which classes might benefit most?

Analysing the Sonarqube measures, it is possible to observe the complexity measure and this is split in two parts: Cyclomatic Complexity and Cognitive Complexity. In which section, we can check what are the classes or packages with higher complexity:

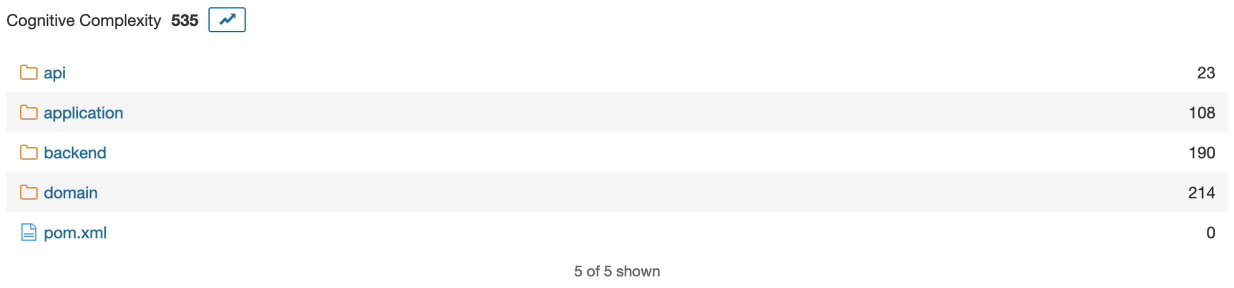


Cyclomatic Complexity **1,207**

api	67
application	259
backend	380
domain	501
pom.xml	–

5 of 5 shown

Figure 15: Cyclomatic Complexity


Cognitive Complexity 535

api	23
application	108
backend	190
domain	214
pom.xml	0

5 of 5 shown

Figure 16: Cognitive Complexity

Based on the pictures we can say the part with highest complexity in the project is the domain part.
To reduce some complexity, we need to start to solve the code smells and this can a lot this measure. Reduce the complexity it is desirable and also feasable if it is an easy task to change the project to achieve this results. The Classes that will probably benifit the most with this reduce are the Classes with the highest level of complexity.

3.8 Is code duplication a problem?

Yes, it is a problem. As the name suggests is a repetition of a line or a block of code in the same file or sometimes in the same local environment. Some people can consider code duplication acceptable but in reality, in fact, creates bigger problems to the software than we could imagine. Code which includes duplicate functionality is more difficult to support, simply because it is longer, and because if it needs updating, there is a danger that one copy of the code will be updated without further checking for the presence of other instances of the same code.
Duplication can usually occur when we have multiple programmers working on different parts at the same time. A code smell is any characteristic in the source code of a program that possibly indicates a deeper problem. The code duplication is consider a code smell so it is consider a problem to our project.

3.9 Is the default quality gate sufficient? If not, which modifications would you suggest?

No, the default quality gate is not sufficient. A Quality Gate is a set of measure-based Boolean conditions. It helps you know immediately whether your project is production-ready. If your current status is not Passed, you'll see which measures caused the problem and the values required to pass. As we know a quality gate:

- Allows to enforce a code quality policy
- Decides whether your project is releaseable (Passed/Failed)
- Status can be pushed and pulled externally
- Default Quality Gate fails if leak period is worse than in the period before
- Quality Gate can be tailored to project‘s needs

As we can see, our quality gate status is passed, however the default quality can be not sufficient for the necessities and goals of our project. Like this, we should apply some conditions that are important for our project and after that see it the project passes that new gate.
With the Quality Gate, you can enforce ratings (reliability, security, security review, and maintainability) based on metrics on overall code and new code. These metrics are part of the default quality gate. Note that, while test code quality impacts your Quality Gate, it's only measured based on the maintainability and reliability metrics. Duplication and security issues aren't measured on test code. [13].
We should create a new Quality Gate with the same parameters of the default quality gate but changing the values that are consider right now. Also choose as a parameter, for example, the number of bugs and

put a maximum value and adjust all this conditions for what we really want to consider in our project. Depending on what we want to observe, out different conditions in ours quality gates.

3.10 Which strategies would you propose to the project's development team to improve quality in the long term?

There are many strategies that could be applied in this project but, what is important to keep in mind are the good practices. According to what we study the best practices are:

1. Integrate SonarQube to the build process
2. Focus on the leak
3. Start with a small set of rules and increase step by step
4. Deactivate rules which have no benefit for the project
5. Deactivate rules which have too high false positive rate
6. Only fix issues that you understand
7. Eliminate false positives to create trust into the system
8. Implement your own rules to automatically track project specific guidelines

4 Checkstyle Integration

For this project the chosen rule-set is sun_checks.xml, which contains rules for coding conventions, best practices and standards for the Java Programming Language. After the execution of checkstyle the report shows that in the project there are in total 5244 in 153 files.

After some research the following rules are not needed for the project and have to be removed from the checkstyle.xml:

- DesignForExtension: this rule makes sense only for library projects, as described in the documentation [4].
- FinalParameters: This rule is too restrictive and not needed. As stated in GitHub from Checkstyle: "we will not use that fanatic validation, extra modifiers pollute a code it is better to use extra validation(Check) that argument is reassigned. But this Check will exists as it was created by community demand."[5]
- NewlineAtEndOfFile: These rule is not really needed. With or without a new line in the end of every file won't result in remarkable difference in the project quality.
- JavaDocVariable: Documentation for every single variable in the project is excessive and not really needed.

And the following rules have to be changed:

- Hiddenfield: This rule does not allow any local variable or parameter which shadows a a field designed in the same class. Having such parameters or variables in the Constructor or Setter is a popular practice in the Java Programming and it doesn't worsen the project quality. So, the rule should be changed to allow Constructors and Setters by setting the options ignoreConstructorParameter, ignoreSetter and setterCanReturnItsClass to true.
- MissingJavadocMethod: These rule is too restrictive and should be changed to allow getters and setters, as well as shorter methods (<20) to not have JavaDoc.

- OperatorWrap: These rule checks the style of how to wrap lines on operators. In these project the used style is such, that the token (operator) has to be at the end of the line. So, the checkstyle rule has to be accordingly changed to check for the option: "eol", which represents the preferred style.
- LineLength: In order to be not too restrictive the allowed limit for maximum line length is set to 100.

When configuring checkstyle it is highly recommended to enable the option for printing the errors in the console. This would make programmers aware of the checkstyle errors. Failing the project for each build is, however, not recommended, because the bugs found with checkstyle don't really affect the application flow. Furthermore, when working under time pressure it would be a burden for programmers to fix each checkstyle error for each time they build the project. However, in order to avoid the case where programmers ignore the checkstyle errors in the console and not fix the code according to the the ruleset it is recommended to add checkstyle as a pre-commit git hook. This means that before adding new piece of code, a programmer should fix all of the checkstyle errors, otherwise the commit will fail. This would allow programmers to implement the required functionality without having to consider for each build the checkstyle errors and then at the same time they will have to fix the checkstyle errors, before committing the new piece of code. This would prevent implementing more violations of the coding conventions in the future and at the same it is not too restrictive.

Another recommended step is configuring the IDE (Eclipse or Intellij) to fix the checkstyle errors (which can be automatically fixed) with reformatting. After configuring the reformatting options in Intellij and changing the rules as described before the number of checkstyle errors are reduced from over 5000 to 269.

Which problems arise when integrating Checkstyle into an existing source base?

When integrating Checkstyle into an existing source base, it is often the case that different programmers have different programming styles. That's why when a group of people works on the same project without specifying any common standards it results in a mixture of those styles, which is harmful for the project maintainability. Furthermore, introducing Checkstyle at later point in the software lifecycle, results in a huge amount of checkstyle errors, which need to be fixed and can cost time. Another difficulty is, when configuring the Checkstyle rules, to keep a balance between not being too restrictful and at the same time improving the projects quality.

Which strategies exist to overcome these problems?

A good strategy is to choose a checkstyle template, which is the most appropriate for the project (in this case sun checkstyle). Afterwards, checkstyle offers methods to easily configure the already existing sun rules and to change them accordingly, so that they fit the best with the already implemented style in the project. Regarding fixing these errors, a big help is the IDE reformatting, which can be configured to work with checkstyle. This method saves a big amount of time and effort, and fixes a big amount of checkstyle errors.

How could such problems be prevented in future projects?

The best case is to define the coding standards and conventions in the planning phase from the software's life cycle. Checkstyle should be integrated, when setting up the project dependencies and the project environment. This would prevent the case, where a big amount of errors have to be fixed at once and would help maintain the software quality.

Did you introduce new problems?

The problem introduced in this case is that programmers are obligated to always keep and stick to the defined standards and programming style, even if they would prefer a different one. So even, when trying to keep the rules not too respectful, there are still limits which should be taken into consideration while developing the project.

5 Development Guidelines

In this section we are going to have a look at the source code in addition to our analysis tools Sonarqube and Checkstyle to find irregularities and document development guidelines which have to be followed afterwards.

5.1 Observed irregularities

To formulate fitting development guidelines we first have to check which irregularities show up so far. Therefor we focus on different aspects of the code (see following subsections).

Documentation irregularities

- Only Interfaces were documented at all.
- Some Interfaces (StudentSimilarityService) do not have JavaDoc yet.
- German and English used in JavaDoc.

Naming irregularities

- Languages mixed up - English and German (e.g. class Lehrveranstaltung)
- Typing errors found (e.g. CuorseRegistration, AdminStudyPlnasController)
- Some Interfaces have 'Interface' in the name, some have an 'I', others do not have any hint on them being an Interface in their name.
- Not all implementations of Interfaces use 'Impl' at the end of their name.
- Not all Dto classes use 'Dto' at the end of their name.
- The logger is called 'logger' and 'log' in different classes.

Logging irregularities

- Observed Log levels: Info, Warn, Debug, Error
- Logging partially done in German
- Wrong usage of logging levels: log.warn used where log.error should be used (e.g. in FeedbackService.java, line 79); log.info sometimes used when log.warn should be used (e.g. in FeedbackService.java, line 90); log.debug sometimes used when log.info should be used (e.g. StudentMyCoursesController.java, line 31).
- log.error is not used as it should be used
- Important system activities are not logged appropriately

Formatting irregularities

- Empty constructor methods provided (e.g. /application/.../controller/student/forms/FeedbackForm).

5.2 Development guidelines

After evaluating all of the irregularities and checking all source code files we came up with the following development guidelines:

Documentation guidelines

- Always provide JavaDoc for Interfaces and all methods in them.
- Always stick to English for JavaDoc
- All return and param values have to be documented

Naming guidelines

- Always stick to meaningful names for variables/ classes/ methods.
- Always stick to English as naming language.
- Always double check for typing errors.
- The name of Interface classes should always end with 'Interface' (e.g. AccountActivationServiceInterface).
- The name of Form classes should always end with 'Form' (e.g. CreateUserForm).
- The name of Controller classes should always end with 'Controller' (e.g. AdminSemesterController).
- The name of an implementation class should always end with 'Impl'.
- The name of any DTO has to end with 'Dto'.

Logging guidelines

- Log output should be written in English.
- Use log.error when an exception is thrown.
- Use log.info when any type of information is given to the user.
- Use log.warn when the user tries to do an illegal action.
- Use log.debug only when you want to give more information for eventual problems with the tool.
- Use log.trace to describe events showing step by step execution of your code that can be ignored during the standard operation, but may be useful during extended debugging sessions.

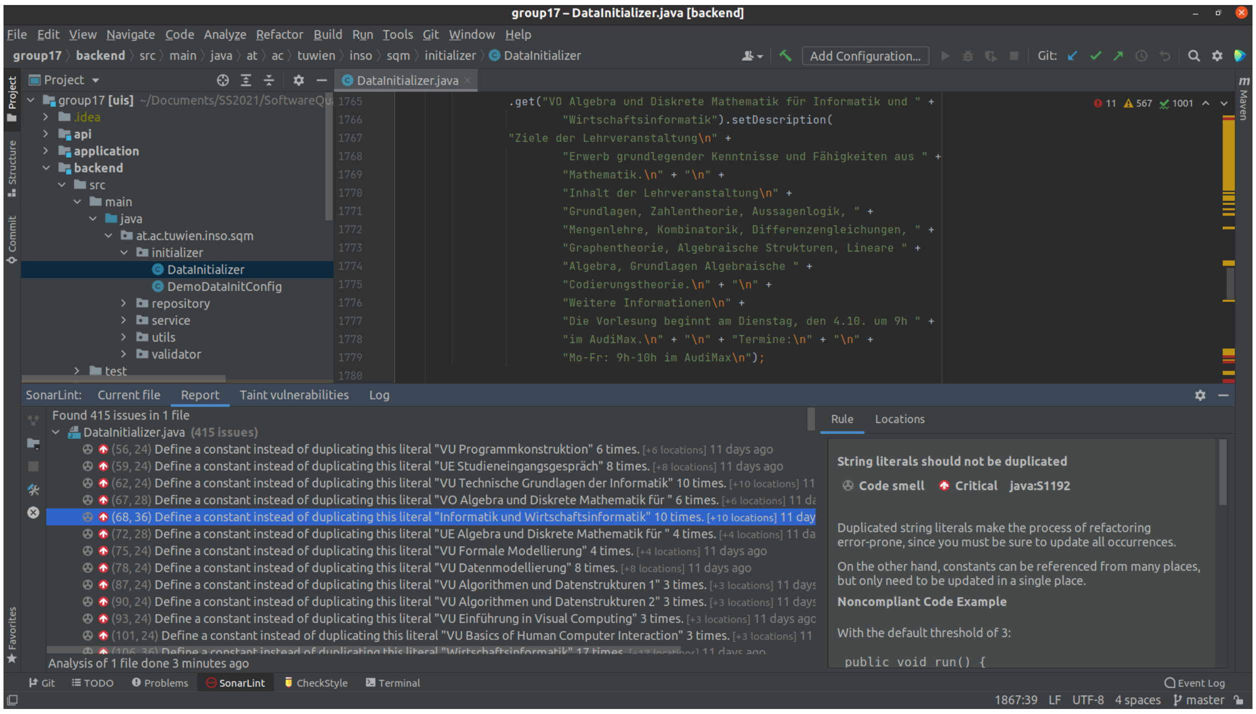
Other guidelines

- Do not provide empty Constructor methods.

6 SonarLint Integration

SonarLint is a free IDE extension that lets the code developers fix coding issues before they exist. Like a spell checker, SonarLint highlights Bugs and Security Vulnerabilities as you write code, with clear remediation guidance so you can fix them before the code is even committed. SonarLint helps all developers write better and safer code. Team focus shifts to what really matters - the quality of new code. Only clean code makes it into the master and your team gets more accomplished as quality improves.

After configured the SonarLint in connected mode (connect it to our SonarQube server), it was possible to analyse the code with the SonarLint. SonarQube is a server where we can host our projects and execute analysis, whereas SonarLint is an agent that allow us to connect with this SonarQube and execute the analysis remotely. In the picture below, it is possible to see of SonarLint being used in one class:


group17 – DataInitializer.java [backend]
File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help
group17 backend src main java at ac tuwien inso sqm initializer DataInitializer
Add Configuration...
Git:
Project
group17 [uis] ~/Documents/SS2021/SoftwareQu
.idea
api
application
backend
src
main
java
at.ac.tuwien.inso.sqm
initializer
DataInitializer
DemoDataInitConfig
repository
service
utils
validator
test
DataInitializer.java
11 567 1001
SonarLint: Current file Report Taint vulnerabilities Log
Rule Locations
Git TODO Problems SonarLint CheckStyle Terminal
Event Log
1867:39 LF UTF-8 4 spaces master

```
                .get("VO Algebra und Diskrete Mathematik für Informatik und " +
                        "Wirtschaftsinformatik").setDescription(
                "Ziele der Lehrveranstaltung\n" +
                        "Erwerb grundlegender Kenntnisse und Fähigkeiten aus " +
                        "Mathematik.\n" + "\n" +
                        "Inhalt der Lehrveranstaltung\n" +
                        "Grundlagen, Zahlentheorie, Aussagenlogik, " +
                        "Mengenlehre, Kombinatorik, Differenzengleichungen, " +
                        "Graphentheorie, Algebraische Strukturen, Lineare " +
                        "Algebra, Grundlagen Algebraische " +
                        "Codierungstheorie.\n" + "\n" +
                        "Weitere Informationen\n" +
                        "Die Vorlesung beginnt am Dienstag, den 4.10. um 9h " +
                        "im AudiMax.\n" + "\n" + "Termine:\n" + "\n" +
                        "Mo-Fr: 9h-10h im AudiMax\n");
```

```
Found 415 issues in 1 file
DataInitializer.java (415 issues)
(56, 24) Define a constant instead of duplicating this literal "VU Programmkonstruktion" 6 times. [+6 locations] 11 days ago
(59, 24) Define a constant instead of duplicating this literal "UE Studieneingangsgespräch" 8 times. [+8 locations] 11 days ago
(62, 24) Define a constant instead of duplicating this literal "VU Technische Grundlagen der Informatik" 10 times. [+10 locations] 11
(67, 28) Define a constant instead of duplicating this literal "VO Algebra und Diskrete Mathematik für " 6 times. [+6 locations] 11 da
(68, 36) Define a constant instead of duplicating this literal "Informatik und Wirtschaftsinformatik" 10 times. [+10 locations] 11 day
(72, 28) Define a constant instead of duplicating this literal "UE Algebra und Diskrete Mathematik für " 4 times. [+4 locations] 11 da
(75, 24) Define a constant instead of duplicating this literal "VU Formale Modellierung" 4 times. [+4 locations] 11 days ago
(78, 24) Define a constant instead of duplicating this literal "VU Datenmodellierung" 8 times. [+8 locations] 11 days ago
(87, 24) Define a constant instead of duplicating this literal "VU Algorithmen und Datenstrukturen 1" 3 times. [+3 locations] 11 days
(90, 24) Define a constant instead of duplicating this literal "VU Algorithmen und Datenstrukturen 2" 3 times. [+3 locations] 11 days
(93, 24) Define a constant instead of duplicating this literal "VU Einführung in Visual Computing" 3 times. [+3 locations] 11 days agc
(101, 24) Define a constant instead of duplicating this literal "VU Basics of Human Computer Interaction" 3 times. [+3 locations] 11
(106, 36) Define a constant instead of duplicating this literal "Wirtschaftsinformatik" 17 times. [+17 locations] 11 days ago
Analysis of 1 file done 3 minutes ago
```

String literals should not be duplicated

Code smell Critical java:S1192

Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.

On the other hand, constants can be referenced from many places, but only need to be updated in a single place.

Noncompliant Code Example

With the default threshold of 3:

```
public void run() {
```

Figure 17: SonarLint Example

The project's development team should integrate SonarLint into daily processes to add value for the quality of the source code and to prevent bad code from being pushed to the repository because with SonarLint we have[12]:

Instant View

SonarLint highlights code issues with markers on open files. It also provides an issues summary table for a selected component in the IDE, including the creation time of the issue.

On-the-fly Detection

SonarLint offers the ability to see problems as you code, like a spell checker for text. This is done by subtly pointing out the issue without distracting the developer so that he can focus on code but is still notified of the issue.

Smart Education

Once SonarLint detects an issue, it also shows the associated documentation to help the developer understand the issue and why it is a problem. It gives a compliant and non compliant code example and shows how to resolve the example issue. Easy, powerful and fun to use: a good way for developers to learn.

Push Notifications

Get real-time code quality notifications in your IDE as you work. Track Quality Gate status (failed, passed, warning) or that an analysis has assigned a new issue to you.

7 Additional Tools - comparison of static code analysis and metrics tools

In this section we write down our observations and notes on a research about different available static analysis and metrics tools. At first we started by checking different static code analysis tools and comparing their functionalities. As a comparison base we used Sonarqube. We came across following popular static code analysis tools:

- Sonarqube
- Synopsis Coverity
- Veracode
- Deepsource (costs from 24$ per month)
- Codacy

Most static code analysis tool offer the same functionality and are for free (the only exception being Deepsource with a base price of 24$ per month). Thus we concluded that another static code analysis tools would not benefit this project.

Our next idea were code review tools. Therefore we checked on the most popular ones:

- Atlassian Crucible
- JetBrains Upsource
- SmartBear Collaborator

These tools allow the development team to review code in teamwork. Therefore, a Quality Manager (or a team of Quality Managers) could be nominated and check the code. In this review process developers can be assigned to fix issues. The big benefit of these tools is the improved formalised review process. All three tools provide mostly the same functionality, but Upsource appears to be the most interesting one for our case as it can be integrated into IntelliJ IDEA as plugin. Sadly, it does not seem to be available as a student version for free. Therefore we dropped the idea of code review tools.

Our next idea was taken from the assignment. Tools which analyse architectural debt could benefit the development team, as Software Architecture could be improved. We will have a look at *Sonargraph* and *ArchUnit*.

Sonargraph [11] does provide free licenses for students and could be installed as a plugin in IntelliJ. This sounds quite promising, but the online documentation is quite confusing and overwhelming. On the other hand Sonargraph seems to be very powerful and can be integrated and combined with most modern tools.

ArchUnit [2] seems to work out of the box as a maven dependency, is free and does provide functionality to test the architecture of a program directly. We decided to give it a try. In the following subsection we will document our observations.

7.1 Trying ArchUnit in our project environment

We started by integrating the following dependency in the projects main pom file and the application's pom file.

```
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit</artifactId>
    <version>0.18.0</version>
    <scope>test</scope>
</dependency>
```

Now ArchUnit is available to the project and can be used to write new test cases or expand existing ones. As a next step we tried to implement a test case with ArchUnit to understand the fundamentals of this dependency. This test case should check the architecture of the project for cyclic dependencies. The created test case is placed in /application/src/test/java/at.ac.tuwien.inso.sqm/ and is called *CyclicDependencyRulesTest*. The test case is taken from the examples page from ArchUnit [3]. We modified it to check the application module from this project.

After successfully altering the test case with ArchUnit we can say that this tool should be used for further development in this project. Software Testing engineers have to adapt to ArchUnit and use it to update existing tests and create new ones to guarantee high code quality throughout the development process.

8 Merge Requests and Review Checklist

After understanding the concept of merge requests and protected branches, we analysed how these concepts can be applied in a meaningful way to establish a review workflow. This should prevent developers from pushing unreviewed branches. We found the answers to following questions.

Which branching strategy do you suggest to introduce? Which branches need review before being accepted for merge?

In this project it is suggested branches with GitLab flow because GitHub flow has only feature branches and a main branch, which is insufficient for the project. Additionally to a main and a feature branch, there should be also release branches.

Working with release branches it is necessary if it will be released software to the outside world. It is necessary to create stable branches using the main branch as a starting point, and branch as late as possible. When this is done, it is minimize the length of time during which we have to apply bug fixes to multiple branches. After announcing a release branch, we only add serious bug fixes to the branch.

Furthermore, every feature branch needs review before being accepted for merge. How this should be done and to whom it should be assigned is explained in the next points.

Which points need to be verified by a reviewer? Define a checklist that each reviewer can work through. Revisit the development guidelines and common quality problems you identified during the previous labs and try to address them in particular.

Every change that is done should be reviewed thoroughly to avoid future problems in the project work. However, we can define a checklist of the most important things to verify but we consider that every detail is important.

- We should verify if the documentation is correct for the functions we had defined and that we are reviewing [ex: always stick to the same language, if describes correctly what is done, all the values documented]
- If the variables have meaningful and intuitive names
- The names of the new classes and functions respects the pattern of the whole project
- Verify if the changes that are being done to the project are useful and they make sense
- Verify if the new code still passes the test of the application
- The code that is being reviewed does not have bugs and still working
- Verify if there is enough Test Coverage after the changes in the code
- The project work should still be functional after the changes and respect all the parameters

Which naming conventions should feature branches follow?

The feature branches should be names as follows: feature-<name>where <name>should be a meaningful name describing the functionality which is developed in the feature branch. For separators use hyphens.

Who (e.g. which group of developers) is qualified to accept a merge request? Which skills and experience level should they have?

Before creating a merge request for the new piece of code, the code should have a sufficient percentage of test coverage, as well as have as good overall quality(checkstyle).

Teams should include people with different level of expertise: junior, middle, and senior. Seniors should have at least 5 years of experience in software development with the used technologies. Merge requests from Junior and Middle software developers should be reviewed and accepted by the Seniors. Seniors should review each others code. A good method for the Seniors group would be to split in pairs, thus reviewing each others code. In case there are no Seniors in a team (which is not recommended) the request should be reviewed and accepted by all team members. In case of insufficient human resources, the team should be split in pairs, and as described before review each others code.

How should be proceeded in case a merge request is declined? Consider how the organizational workflow should look like in this case.

If a review is declined, it would be the best to the person who declined the request to write comments, for example git offers such an option on behalf of the merge requests, or communicate it with the team member who initiated the merge request and explain why the merge request is declined and what exactly should be improved. In the case where the person that made the MR is not able to fix the code, they should ask for help other team members, for example a Senior software developer. Afterwards the whole process should start again as described in the previous point: test coverage, quality and afterwards a sending a new merge request.

9 Knowledge Management

In this section we will compare different Knowledge Management systems and propose a structure which should be used for the rest of the project lifetime. Besides that we will introduce a prototype of the wiki software of choice and create some content as example.

9.1 Comparing available tools

There are many different wiki solutions available. Some of them are proprietary, others are open source and available for free. We will have a look at some of the most common ones (including one proprietary solution). We found the following Wiki software solutions to be very common and will compare them (although many more open source solutions exist - e.g. DokuWiki, TikiWiki, etc.) [1]:

The easiest way to get a knowledge base in our setup is using the built in wiki solution from **GitLab**. The infrastructure already exists, and the wiki is treated as a seperate repository to enable versioning and collaboration. One drawback from the GitLab wiki is that it does not provide a WYSIWYG ('What you see is what you get') editor, but it is simple to use and offers multiple markup languages (e.g. Markdown).

Confluence is available as on-premise or cloud based software. It offers many plugins which enable additional functionalities like drawing diagrams, access databases directly, link Confluence to a Git repository and many more. Also collaboration is very easily possible with Confluence, as it provides great user management and simultaneous editing of pages. Its main drawback is the price. But in a real working environment a free test license can be obtained to test and compare Wiki solutions simultaneously - so it should be considered when collaboration is more important than the price.

MediaWiki is the Wiki platform used by Wikipedia. It provides lots of functionality, is very powerful and well known. On the other hand it is only available on-premise, no support is provided and it is not

Tool	**Free version available?**	**Advantage(s)**	**Drawback(s)**
GitLab Wiki	Yes	Already set up with the project	No WYSIWYG editor
Confluence	Yes (up to 10 users)	Very powerful, Cloud version available, WYSIWYG editor, many plugins available	Only free for very small teams
MediaWiki	Yes	Very powerful, many plugins available, very customizable	Not as user friendly as other solutions

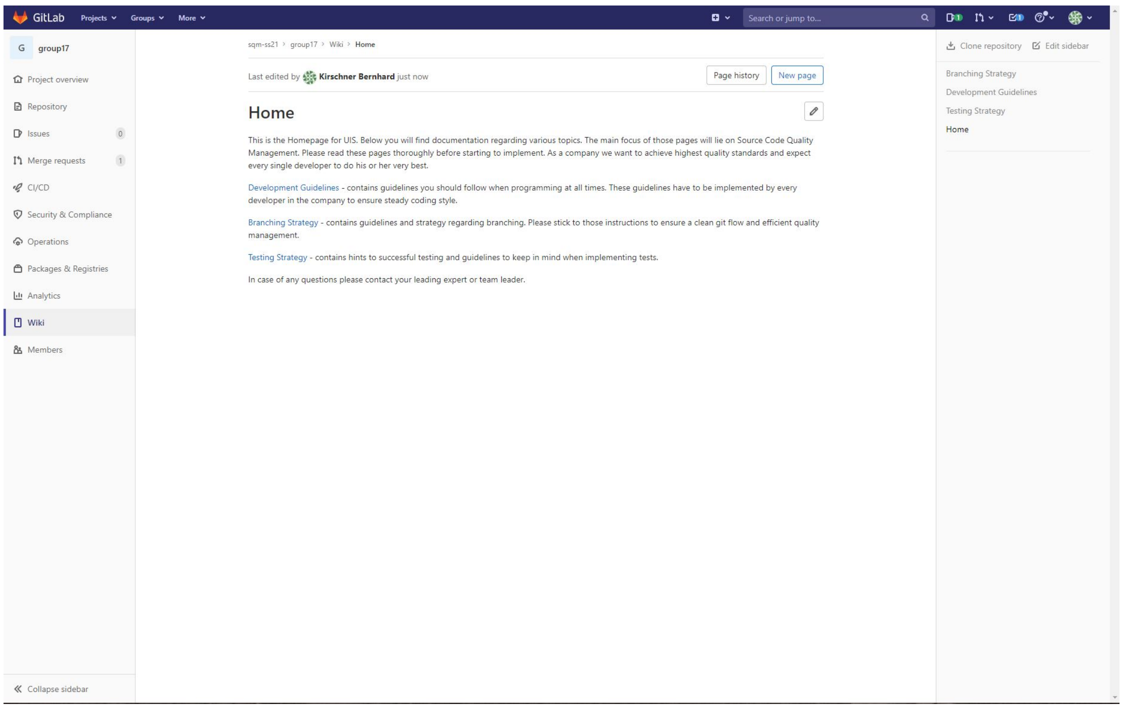
Table 2: Comparison of Wiki Tools

possible to restrict access to pages (no user management). MediaWiki is also designed for high traffic, it may be not as efficient for smaller projects [9].

After comparing one open source, one proprietary and the built in version we decided to use the built in solution for this project. GitLab's Wiki is not very powerful, but is sufficient for this small project, as it is only required to document some guidelines, templates and references. If the requirements increase and Knowledge Management gets more important, a more competent solution must be chosen - MediaWiki is very powerful, but Confluence is more user friendly and offers support because it is proprietary software.

9.2 Creating a baseline of documentation

We have decided to put Development Guidelines (according to section 5), Branching Strategies (according to section 8) and Testing Strategies (according to section 2) in the wiki. Of course anything can be documented there to ensure a uniform style and high code quality throughout the project.



Last edited by Kirschner Bernhard just now

Home

This is the Homepage for UIS. Below you will find documentation regarding various topics. The main focus of those pages will lie on Source Code Quality Management. Please read these pages thoroughly before starting to implement. As a company we want to achieve highest quality standards and expect every single developer to do his or her very best.

Development Guidelines - contains guidelines you should follow when programming at all times. These guidelines have to be implemented by every developer in the company to ensure steady coding style.

Branching Strategy - contains guidelines and strategy regarding branching. Please stick to those instructions to ensure a clean git flow and efficient quality management.

Testing Strategy - contains hints to successful testing and guidelines to keep in mind when implementing tests.

In case of any questions please contact your leading expert or team leader.

Figure 18: Wiki Homepage

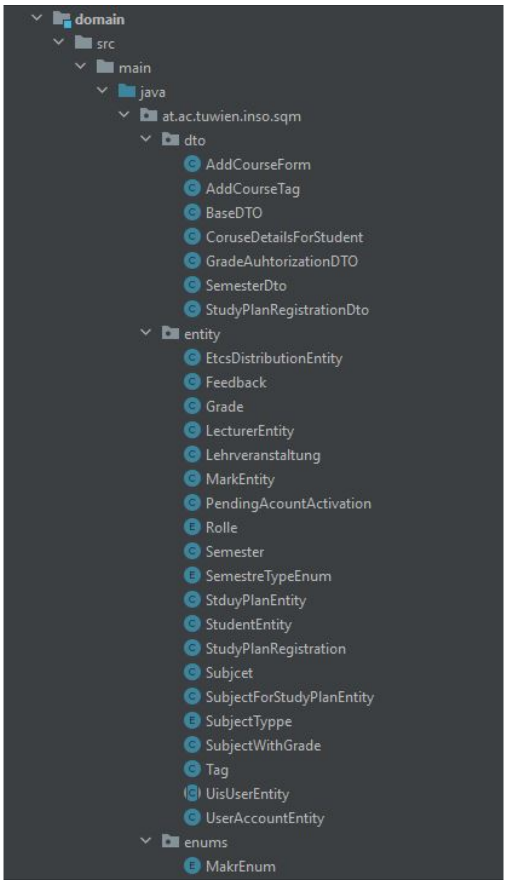
The linked sub-pages are created in the teams wiki[2]. Besides the created pages more content could be uploaded regarding the used tools in the project, frequetly asked questions, basic help or maybe some pages which provide links for inexpirienced developers regarding Spring, Maven, or basically any other project related technology.

10 Quality Improvement

In this section we document some improvements we programmatically performed on the project to increase code quality and provide some examples. All of those changes are just a snippet were not implemented throughout the whole project. This task has to be performed by the developers of the project in the upcoming weeks - we only provide examples and best practices.

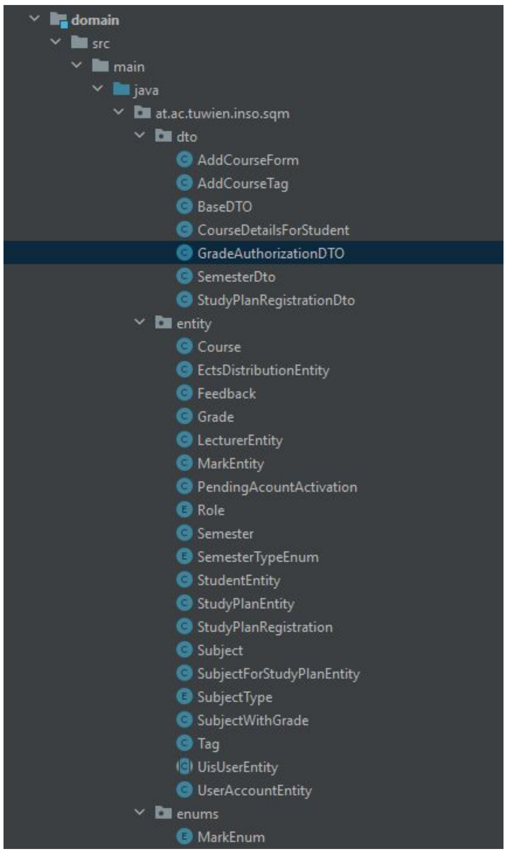
10.1 Fixing typing errors and unifying language on Class level

We fixed typing errors in class names and translated German classes to English. You can find the comparison on the following figures:



```
domain
  src
    main
      java
        at.ac.tuwien.inso.sqm
          dto
            AddCourseForm
            AddCourseTag
            BaseDTO
            CoruseDetailsForStudent
            GradeAuhtorizationDTO
            SemesterDto
            StudyPlanRegistrationDto
          entity
            EtcsDistributionEntity
            Feedback
            Grade
            LecturerEntity
            Lehrveranstaltung
            MarkEntity
            PendingAcountActivation
            Rolle
            Semester
            SemestreTypeEnum
            StduyPlanEntity
            StudentEntity
            StudyPlanRegistration
            Subjcet
            SubjectForStudyPlanEntity
            SubjectTyppe
            SubjectWithGrade
            Tag
            UisUserEntity
            UserAccountEntity
          enums
            MakrEnum
```

Figure 19: Before



```
domain
  src
    main
      java
        at.ac.tuwien.inso.sqm
          dto
            AddCourseForm
            AddCourseTag
            BaseDTO
            CourseDetailsForStudent
            GradeAuthorizationDTO
            SemesterDto
            StudyPlanRegistrationDto
          entity
            Course
            EctsDistributionEntity
            Feedback
            Grade
            LecturerEntity
            MarkEntity
            PendingAcountActivation
            Role
            Semester
            SemesterTypeEnum
            StudentEntity
            StudyPlanEntity
            StudyPlanRegistration
            Subject
            SubjectForStudyPlanEntity
            SubjectType
            SubjectWithGrade
            Tag
            UisUserEntity
            UserAccountEntity
          enums
            MarkEnum
```

Figure 20: After

As you can see in figure 19, many classes had typing errors in the name or were even German although the rest of the project is clearly English. We have fixed this as an example for the sub-project domain. Of course this would have to be done throughout the whole project. Including methods, comments and variables.

[2]`https://peso.inso.tuwien.ac.at/repo/sqm-ss21/group17/-/wikis/home`

10.2 Fixing JavaDoc language

In some cases JavaDoc was provided in German. This is not feasible as we are dealing with an English project with international participants. We exemplary fixed api/src/main/java/at.../service/SemesterServiceInterface to a correct JavaDoc. Before it looked like this:

```
/**
 * Erstellt ein neues Semester.
 * <p>
 * Benutzer muss als Admin authentifiziert werden.
 *
 * @param semester das neue zu erstellende Semester
 * @return das erstellte Semester
 */
@PreAuthorize("hasRole('ADMIN')")
SemesterDto create(SemesterDto semester);
```

Afterwards all methods are documented in English:

```
/**
 * Creates a new semester.
 * <p>
 * User has to be authenticated as admin.
 *
 * @param semester which has to be created.
 * @return the created semester
 */
@PreAuthorize("hasRole('ADMIN')")
SemesterDto create(SemesterDto semester);
```

Besides these language problems, some classes and Interfaces do not contain JavaDoc at all. This is not feasible either. Documentation should be improved throughout the project.

10.3 Checkstyle

In order to add the pre-commit git hook mentioned in the chapter Checkstyle all of the checkstyle errors are fixed. This include adding JavaDoc comments, changing the names of variables, methods and classes, and etc. So the checkstyle errors are reduced to 0.

10.4 Testing

There are two main issues with the testing in the project. The first problem is insufficient test coverage. This can be easily improved by adding additional tests and running the tool Jacoco. The tool would then visualise additionally what tests are missing and for which parts of the projects more tests should be included. The second issue regarding testing in the project is the lack of test assertions in some of the authentication tests. The goal of these tests is to access resource which doesn't require any authentication without an exception being thrown. That's why in this case an assertion is missing, like shown in the following example:

```
 @Test
    public void getForValidationTestNotAuthenticated() {
        gradeService.getForValidation("1");
    }
```

To make it explicit and clarify that no exception in this test should be thrown the test should be changed accordingly:

```
 @Test(expected = Test.None.class) // make explicit that is should not throw an Exception
    public void getForValidationTestNotAuthenticated() {
        gradeService.getForValidation("1");
    }
```

10.5 Improving Logging

We have improved logging exemplarily in backend/src/main/java/at.../service/FeedbackService.java. As an example you can see how it was implemented before:

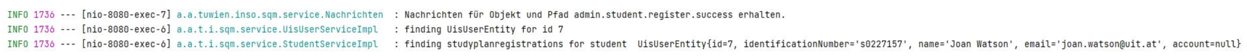
```
private void guardSingleFeedback(Feedback feedback) {
    LOGGER.info(
        "guading single feedback, if no warn log line follows its " +
            "fine.");
    if (feedbackRepository.exists(feedback)) {
        LOGGER.warn(
            "Giving feedback multiple times for the same course is " +
            "not allowed");
        throw new ActionNotAllowedException(
            "Giving feedback multiple times for the same course is " +
            "not allowed");
        }
    }
```

The log levels are obviously not chosen correctly. Log level "warn" should not be used if afterwards an Exception will be thrown. In such cases an "error" level should be used. You can see it in following example:

```
private void guardSingleFeedback(Feedback feedback) {
    LOGGER.info("Guarding single feedback, if no error log line follows its fine.");

    if (feedbackRepository.exists(feedback)) {
        LOGGER.error("Giving feedback multiple times for the same course is not allowed");
        throw new ActionNotAllowedException("Giving feedback multiple times for the same
        course is not allowed");
    }
}
```

This way it is much clearer that an error occurred. Every thrown exception should be logged with an appropriate logging error. Besides that, logging needs to be introduced thoroughly and consistently throughout the project.

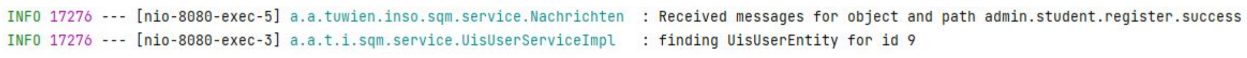
Another example we implemented regarding logging was in api/src/main/java/at.../service/Nachrichten (of course this class should be renamed here, but this is not the focus of this subsection). This German named class also contained German logging (see figure 21).



```
INFO 1736 --- [nio-8080-exec-7] a.a.tuwien.inso.sqm.service.Nachrichten  : Nachrichten für Objekt und Pfad admin.student.register.success erhalten.
INFO 1736 --- [nio-8080-exec-6] a.a.t.i.sqm.service.UisUserServiceImpl   : finding UisUserEntity for id 7
INFO 1736 --- [nio-8080-exec-6] a.a.t.i.sqm.service.StudentServiceImpl   : finding studyplanregistrations for student  UisUserEntity{id=7, identificationNumber='s0227157', name='Joan Watson', email='joan.watson@uit.at', account=null}
```

Figure 21: German logging in an English project is not acceptable

So we corrected this log output to an English equivalent, as you can see in figure 22:



```
INFO 17276 --- [nio-8080-exec-5] a.a.tuwien.inso.sqm.service.Nachrichten  : Received messages for object and path admin.student.register.success
INFO 17276 --- [nio-8080-exec-3] a.a.t.i.sqm.service.UisUserServiceImpl   : finding UisUserEntity for id 9
```

Figure 22: Fixed the German output

11 Conclusion

In software engineering, the Software Quality metrics help to understand better how reliable, safe and secure the code is most likely to be. Developing and tracking software quality metrics can be costly and time-consuming, and with this projects work, we got some insights how hard this task can be. With the help of some tools at every stage of our project work, we provided answers, solved some problems and improved the general quality of the project.

In the first stage, we needed to give an overview about the quality of the existing project and further identify the potential areas for improvement. We have introduced Continuous Integration with Gitlab CI, Jacoco and Sonarqube, being possible to automate testing and building the project, analyze the Test Coverage and to interpret and discuss the general quality state of the project.

In a second instance, using our previous work, we could define common coding standards and development guidelines to stabilize the quality of the source code in the long term and, like this, provide guidelines for the project's development team. Additionally we introduced Checkstyle to the project's technology portfolio to keep code quality consistent. It was a laborious process that required lots of details, but after all the effort, the group believes it has responded to what was asked in the best way.

In the last part, it was possible to improve the quality of the existing source code after all the research and work carried out previously. We took into account all the necessary processes to be done and to achieve success in this stage. Additionally we added guidelines on the recommended branching strategy and introduced a knowledge management system to persistently keep information.

For future work, if the development team wants to extend the project, they need to consider all the processes to be carried out, especially before the beginning. They need to define the tasks well, taking into account the metrics to keep the software quality. To keep this in mind and to help the team to achieve good results, a wiki was created in the repository to help with this process.

References

[1] 8 best wiki software free and open source. https://www.codefear.com/scripts/best-wiki-software/.

[2] Archunit. https://www.archunit.org/.

[3] Archunit test to test for cyclic dependencies. https://github.com/TNG/ArchUnit-Examples/blob/main/example-junit5/src/test/java/com/tngtech/archunit/exampletest/junit5/CyclicDependencyRulesTest.java.

[4] Checkstyle designforextension. https://checkstyle.sourceforge.io/config_design.html.

[5] Checkstyle finalparameters. https://github.com/checkstyle/checkstyle/blob/master/config/checkstyle_checks.xml.

[6] Ci/cd tools comparison: Jenkins, gitlab ci, buildbot, drone, and concourse. https://www.digitalocean.com/community/tutorials/ci-cd-tools-comparison-jenkins-gitlab-ci-buildbot-drone-and-concourse.

[7] Comparison of most popular continuous integration tools: Jenkins, teamcity, bamboo, travis ci and more. https://www.altexsoft.com/blog/engineering/comparison-of-most-popular-continuous-integration-tools-jenkins-teamcity-bamboo-travis-ci-and-mor

[8] Install gitlab runner on windows — gitlab. https://docs.gitlab.com/runner/install/windows.html.

[9] Manual: What is mediawiki? https://www.mediawiki.org/wiki/Manual:What_is_MediaWiki%3F.

[10] Registering runners — gitlab. https://docs.gitlab.com/runner/register/index.html.

[11] Sonargraph plugin for intellij. https://www.hello2morrow.com/products/sonargraph/.

[12] Sonarlint. https://www.sonarsource.com/products/sonarlint.

[13] Sonarqube documentation. https://docs.sonarqube.org/latest/user-guide/quality-gates/.

[14] Christian R Prause, Jürgen Werner, Kay Hornig, Sascha Bosecker, and Marco Kuhrmann. Is 100% test coverage a reasonable requirement? lessons learned from a space software project. In *International Conference on Product-Focused Software Process Improvement*, pages 351–367. Springer, 2017.