
Software Quality Report

Cordula Eggerth (00750881)

Armin Hasler (00725841)

Franz Kienegger (01426716)

VU Software Quality Management

Sommersemester 2019

1.1 Continuous Integration (CI)

Marktanalyse zu CI-Tools und -Lösungen:

Eine gute CI-Lösung für das vorliegende Softwareprojekt sollte verschiedene Kriterien erfüllen. Die in der untenstehenden **Kriterienmatrix** angeführten Kriterien wurden für jedes der vier CI-Tools analysiert und beurteilt:

<i>Kriterien / Tools</i>	<i>Bamboo</i>	<i>Travis CI</i>	<i>TeamCity</i>	<i>Jenkins</i>
Open Source	Nein	Ja	Nein	Ja
Popularität ¹	216 Stacks	3.730 Stacks	557 Stacks	10.200 Stacks
Plugins	Integration in den Atlassian Toolstack (Jira, Confluence, Bitbucket)	-	362 ²	1.612 ³
Support	Kommerzieller Support durch Atlassian. Umfangreich aus der Community	Distributed CI Service für Projekte, die auf Github vorhanden sind. Support durch Community (aber nicht so umfangreich)	Support abhängig vom gewählten Paket	Umfangreich aus der Community. Vielzahl an Plugins verfügbar, um Projekt individuell anzupassen
IDE Integration	Eclipse, IntelliJ IDEA, Visual Studio	Eclipse, IntelliJ IDEA, NetBeans	Eclipse, IntelliJ IDEA, Visual Studio, RubyMine, PyCharm, PhpStorm, WebStorm	Eclipse, IntelliJ IDEA, NetBeans
Preis	Abhängig von den Remote Agents, ab 1 Remote Agent: 1.100 \$ ⁴	Kostenlos für Open-Source-Projekte, aber kostenpflichtig (teuer) ⁵ für	Kostenlos für 100 Build Configurations und 3 Build Agents,	Komplett kostenlos

¹ <https://stackshare.io/continuous-integration> (10.04.2019)

² <https://plugins.jetbrains.com/teamcity> (11.04.2019)

³ <https://plugins.jenkins.io/> (11.04.2019)

⁴ <https://de.atlassian.com/software/bamboo/pricing> (13.04.2019)

⁵ <https://www.educba.com/jenkins-vs-travis-ci/> (27.04.2019)

		private Projekte	ansonsten ab 299€	
--	--	------------------	-------------------	--

Auswahl eines CI-Tools für das Projekt:

Basierend auf der Analyse und Beurteilung haben wir uns für das Tool **Jenkins** entschieden, da es komplett kostenlos ist, und auch in zahlreichen kommerziellen Projekten Verwendung findet. Außerdem bietet es kostenlos zahlreiche hilfreiche Plugins an, wie beispielsweise JaCoCo zur Coverage-Analyse, SonarQube Scanner zur statischen Code-Analyse und Inspektion (für weitere Plugins siehe: <https://plugins.jenkins.io/>). Mittlerweile umfasst das Angebot von Jenkins über 1400 Plugins.⁶ Durch dieses umfassende Angebot von Plugins kann das Jenkins-CI-System je nach Projekt sehr gut an die eigenen Bedürfnisse angepasst werden. Jenkins bietet sowohl für Code aus einem File-System als auch für Code Repositories aus Git die entsprechenden Schnittstellen, um diese in die CI-Umgebung einzubinden, wobei in der vorliegenden Ausarbeitung beide Möglichkeiten umgesetzt wurden. Die Hauptansicht von Jenkins ist gut organisiert und zeigt übersichtlich die Projekte, für die Builds durchgeführt wurden inklusive Build-Status und -Zeiten.

Informationen zur Konfiguration und Ausführung des CI-Systems:

ALLGEMEINES

Im GitLab befindet sich im Order `docker_configuration` das `docker-compose.yml` File (siehe https://peso.inso.tuwien.ac.at/repo/sqm-ss19/group11/tree/master/docker_configuration), das benötigt wird, um Docker für Lab1 zu konfigurieren:

version: '3.5'

services:

postgres:

```

  container_name: sonar-postgres
  image: postgres
  environment:
    POSTGRES_USER: sonar
    POSTGRES_PASSWORD: sonar
    PGDATA: /data/postgres
  volumes:
    - postgres:/data/postgres
  ports:
    - "5432:5432"
  networks:
    - sonarnet
  restart: always

```

sonarqube:

```

  container_name: sonarqube
  image: sonarqube:latest
  environment:

```

⁶ <https://caylent.com/jenkins-plugins/>

```
SONARQUBE_JDBC_USERNAME: sonar
SONARQUBE_JDBC_PASSWORD: sonar
SONARQUBE_JDBC_URL: jdbc:postgresql://sonar-postgres:5432/sonar
ports:
  - "9001:9000"
networks:
  - sonarnet
restart: always
```

jenkins:

```
container_name: jenkins
image: jenkins/jenkins:latest
ports:
  - "8081:8080"
  - "50000:50000"
networks:
  - sonarnet
restart: always
volumes:
  - C:\Users\Coala\Desktop\Docker\jenkinsVolume:/var/jenkins_home
```

networks:

```
sonarnet:
  name: sonarnet
```

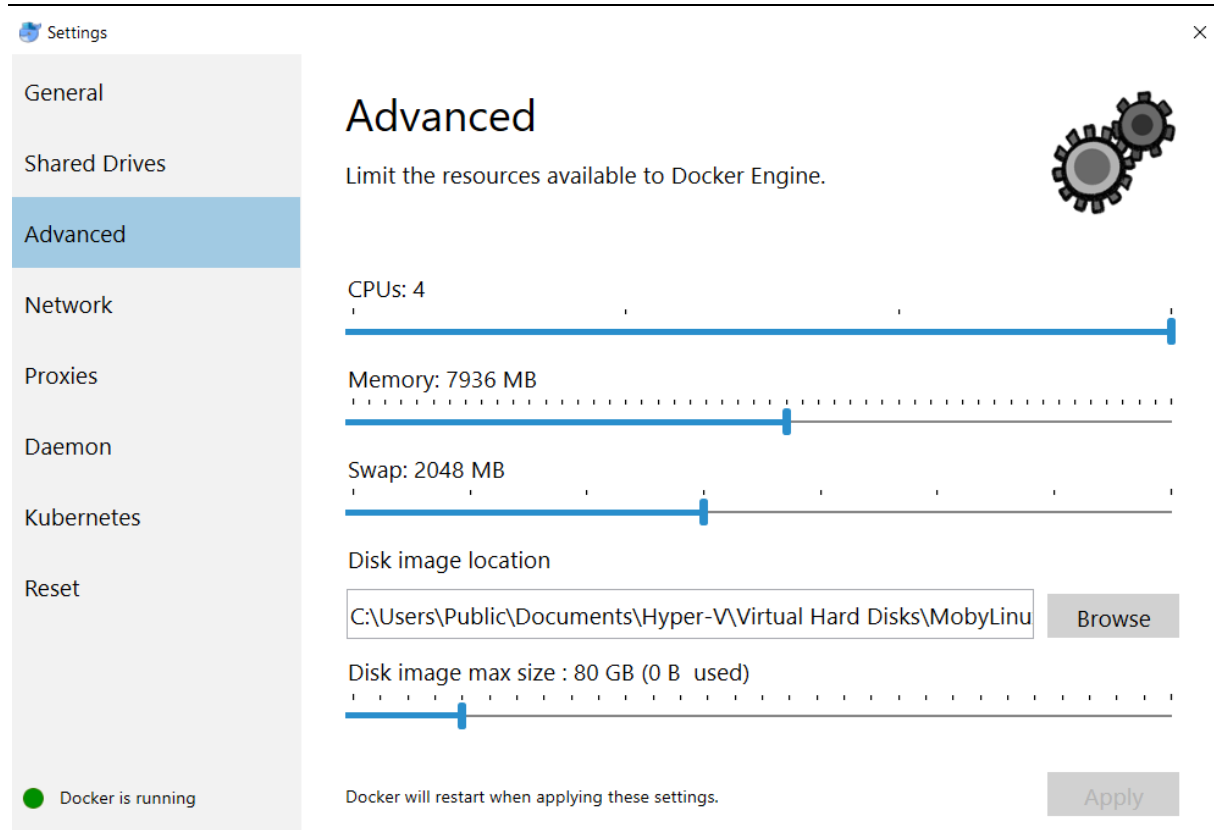
volumes:

```
postgres:
sonarqube:
```

Als Services werden postgres (was von sonarqube gebraucht wird), sonarqube und jenkins angelegt.

Die Pfadangabe für jenkins volumes (siehe obenstehende **blau** gefärbte Zeile) muss zunächst an den Computer angepasst werden, auf dem das Projekt ausgeführt wird.

In den Settings > Advanced von Docker selbst sollten zu Beginn folgende Einstellungen gemacht werden für CPUs, Memory, Swap und Disk Image Max Size, da sonst Docker Container während der Ausführung wegen Überbelastung automatisch gestoppt werden können:

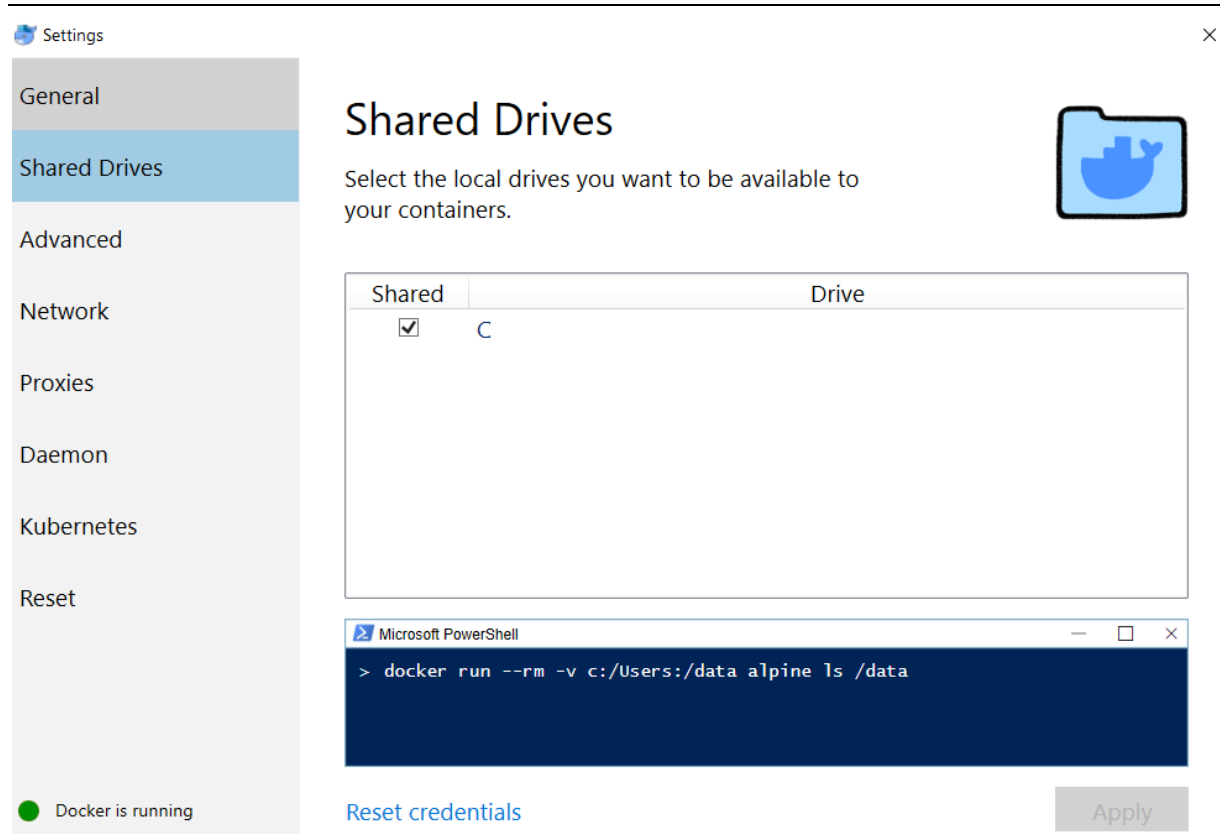


Wieviel CPU, Memory und Ähnliches die Docker-Container brauchen, kann mittels Befehl `docker stats` im Docker überprüft werden und sieht sodann in etwa wie folgender Screenshot aus (Beenden durch „Strg+C“ möglich):

Eingabeaufforderung - docker stats

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
5d410aa2426b	sonar-postgres	0.19%	15.56MiB / 7.522GiB	0.20%	24.7MB / 41.4MB	11.6MB / 54MB	11
2f26f4cdc0b1	jenkins	0.18%	1.037GiB / 7.522GiB	13.78%	1.17MB / 2.15MB	1.01MB / 4.62MB	50
8a8a44471b6e	sonarqube	2.06%	1.894GiB / 7.522GiB	25.18%	43.1MB / 26.3MB	2.32MB / 2.54MB	186

Im Bereich Settings > Shared Drives sollte das Laufwerk, auf dem Docker ausgeführt wird, angegeben werden. (Es kann sein, dass es dazu notwendig ist beim erstmaligen Teilen, alle Antivirenprogramme, Internet Security u. Ä. kurzfristig auszuschalten.).



Danach können die Docker-Container mittels dem Befehl `docker-compose up`⁷ oder `docker-compose up -d`⁸ gestartet werden.

Der Zugriff auf Jenkins erfolgt über <http://localhost:8081/> und auf SonarQube über <http://localhost:9001/projects>, wie in der yml-Datei angegeben. Beim erstmaligen Aufrufen müssen Passwörter und Usernamen gewählt werden.

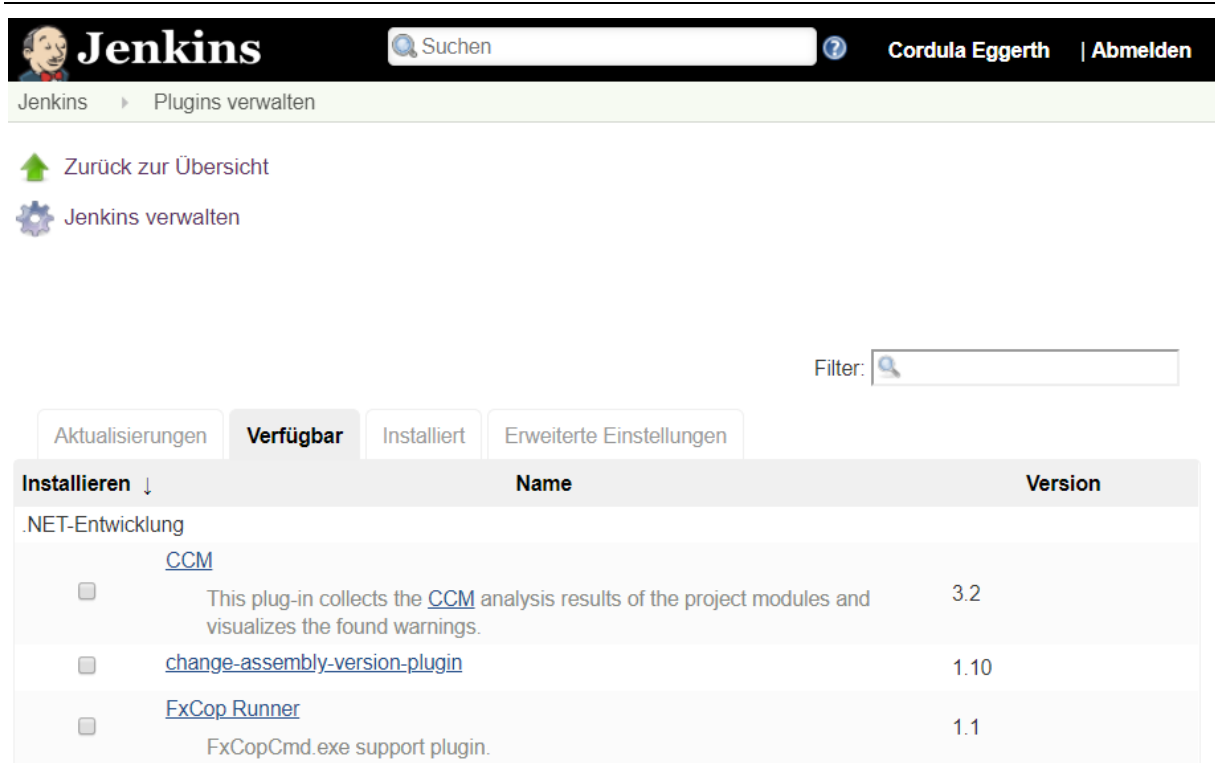
JENKINS KONFIGURATION UND INSTALLATION

(localhost:8081 wie in yml-Datei angegeben)

Über Jenkins verwalten > Plugins verwalten > Tab „verfügbar“ sollten die Plugins File System SCM, JaCoCo und SonarQube Scanner ausgewählt und installiert werden (siehe Screenshot unten). Die Plugins können über den „Filter“ gesucht werden.

⁷ Anmerkung: mit Log.

⁸ Anmerkung: ohne Log.

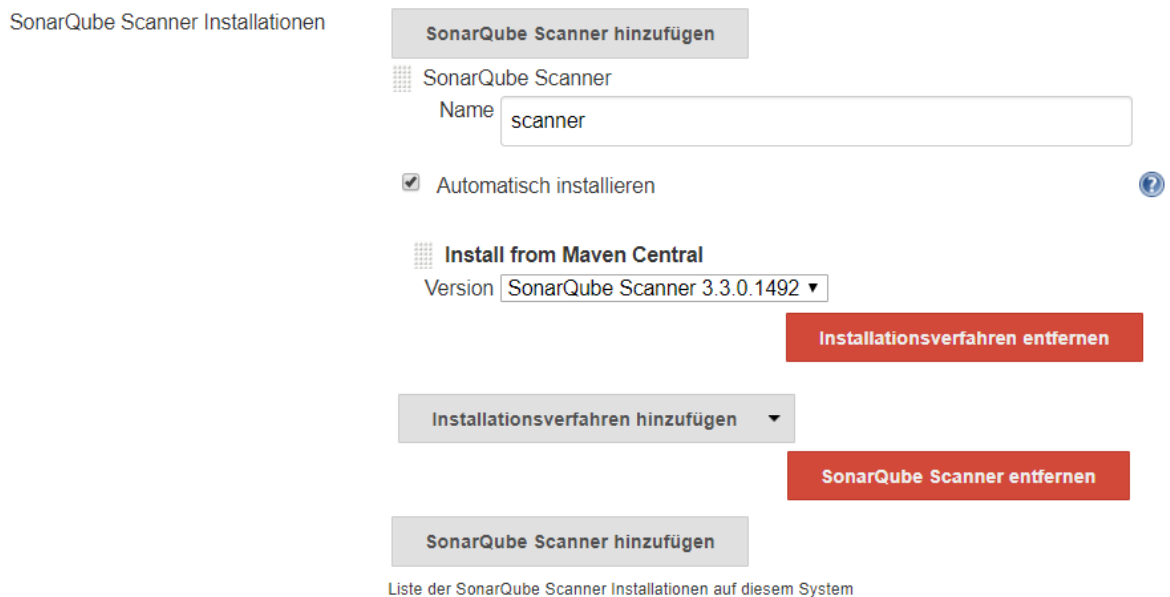


The screenshot shows the Jenkins web interface. At the top, there's a header with the Jenkins logo, a search bar, and user information (Cordula Eggerth | Abmelden). Below the header, there's a navigation bar with 'Jenkins' and 'Plugins verwalten'. The main content area has a 'Zurück zur Übersicht' link and a 'Jenkins verwalten' link. A filter input field is present. Below the filter, there are tabs for 'Aktualisierungen', 'Verfügbar', 'Installiert', and 'Erweiterte Einstellungen'. The 'Verfügbar' tab is selected, showing a table of available plugins. The table has columns for 'Name' and 'Version'. The plugins listed are: .NET-Entwicklung, CCM (version 3.2), change-assembly-version-plugin (version 1.10), and FxCop Runner (version 1.1). Each plugin has a checkbox to select it.

Name	Version
.NET-Entwicklung	
CCM This plug-in collects the CCM analysis results of the project modules and visualizes the found warnings.	3.2
change-assembly-version-plugin	1.10
FxCop Runner FxCopCmd.exe support plugin.	1.1

Über Jenkins verwalten > Konfiguration der Hilfsprogramme sollten Maven und SonarQube Scanner installiert werden, so wie untenstehend in den Screenshots gezeigt:

SonarQube Scanner



The screenshot shows the 'SonarQube Scanner' configuration page. It has a section 'SonarQube Scanner Installationen'. Under this section, there's a button 'SonarQube Scanner hinzufügen'. Below this, there's a list of installed scanners. One scanner is listed with the name 'scanner'. There's a checkbox 'Automatisch installieren' which is checked. Below this, there's a section 'Install from Maven Central' with a dropdown menu for 'Version' set to 'SonarQube Scanner 3.3.0.1492'. There are buttons 'Installationsverfahren entfernen' and 'SonarQube Scanner entfernen'. At the bottom, there's a button 'Installationsverfahren hinzufügen' and another 'SonarQube Scanner hinzufügen' button. The page ends with the text 'Liste der SonarQube Scanner Installationen auf diesem System'.


Maven

Maven Installationen

Maven hinzufügen

Maven

Name

☒ Automatisch installieren 

Installiere von Apache

Version

Installationsverfahren entfernen

Installationsverfahren hinzufügen

Maven entfernen

Maven hinzufügen

Liste der Maven Installationen auf diesem System

Über Jenkins verwalten > System konfigurieren sollten die SonarQube Server Konfiguration mittels generiertem Token und Server URL gemacht werden:

SonarQube servers

Environment variables

☐ Enable injection of SonarQube server configuration as build environment variables

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

SonarQube Installationen

Name

Server URL

Voreinstellung ist http://localhost:9000

Server authentication token

SonarQube authentication token. Mandatory when anonymous access is disabled.

Erweitert...

Delete SonarQube

Add SonarQube

Liste der SonarQube Installationen

Der notwendige Token kann über den SonarQube-Account (localhost: 9001) via My Account > Security generiert werden:

7

The screenshot shows the SonarQube Administration interface. At the top, there is a navigation bar with 'files', 'Quality Gates', and 'Administration'. A search bar and a user profile icon 'A' are also present. The user profile dropdown menu is open, showing 'Administrator', 'My Account', and 'Log out'. The main content area is titled 'Tokens' and contains a text block explaining the purpose of tokens. Below this is a 'Generate Tokens' section with a text input field 'Enter Token Name' and a 'Generate' button. A table lists existing tokens with columns 'Name', 'Last use', and 'Created'. One token named 'jenkins' is listed with a 'Revoke' button.

Name	Last use	Created
jenkins	2 hours ago	April 21, 2019

SONARQUBE KONFIGURATION UND INSTALLATION

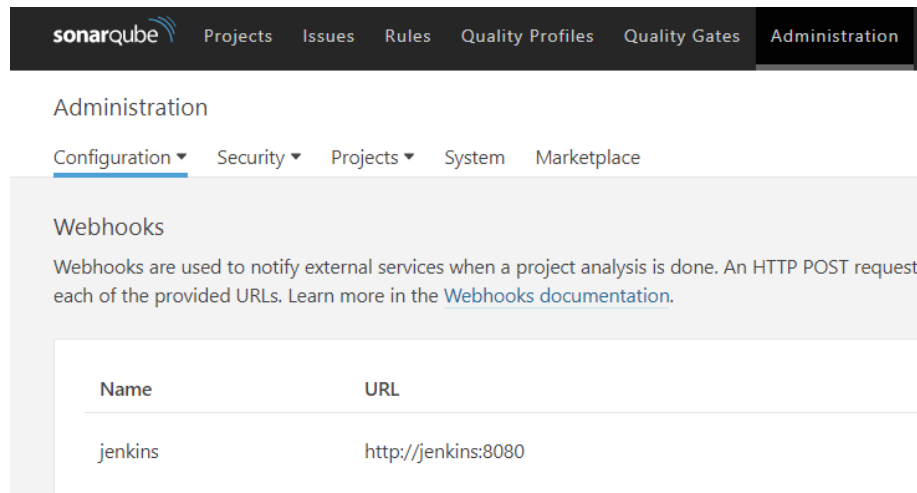
(localhost:9001 wie in yaml-Datei angegeben, mit Default-Username „admin“ und Default-Passwort „admin“)

Über Administration > Marketplace > Updates Only sollten die vorhandenen Updates (wichtig: Sonar Java) durchgeführt werden:

The screenshot shows the SonarQube Marketplace page. The top navigation bar includes the SonarQube logo and links to 'Projects', 'Issues', 'Rules', 'Quality Profiles', 'Quality Gates', and 'Administration'. The 'Administration' section is expanded, showing 'Configuration', 'Security', 'Projects', 'System', and 'Marketplace'. The 'Marketplace' section displays a message: 'You are currently running a Community Edition. Discover more features with SonarSource Editions:'. Below this, there are two cards for 'Developer Edition' and 'Enterprise Edition', each with a list of features and a link to 'Ask for more information'. At the bottom, there is a 'Plugins' section with tabs for 'All', 'Installed', and 'Updates Only', and a search bar.

Edition	Features
Developer Edition	<ul style="list-style-type: none">Branch and Pull Requests analysisAnalysis of additional languages: C/C++, Objective-C, PL/SQL, ABAP, TSQL, SwiftDetection of security vulnerabilitiesSonarLint notifications
Enterprise Edition	<ul style="list-style-type: none">Portfolio managementExecutive reportingAnalysis of additional languages: Java, JavaScript, PHP, Python, Ruby, Swift, VB6Parallel processing of analysisProject transfer

Über Administration > Configuration > Webhooks sollten der/die Webhook/s für Jenkins⁹ angegeben werden:



Falls der Security-Token nicht schon, wie oben angemerkt generiert wurde, sollte dies über My Account > Security > Generate Token gemacht werden.

SPEZIFISCHE KONFIGURATION FÜR DAS PROJEKT AUS LAB1:

In dem oben bereits als „Docker Volume“ gesetzten Ordner einen Ordner erstellen z.B. mit Name „applications“, in den das Lab1-Projekt, das mit Jenkins ausgeführt werden soll, hinkopiert wird.

Zum Projekt wurde die Properties-Datei `sonar-project.properties` hinzugefügt (im „applications“ Ordner im Docker Volume Ordner), damit SonarQube verwendet werden kann:

```
sonar.projectName=UIS
sonar.projectVersion=0.0.1-SNAPSHOT
sonar.projectDescription=University Information System
sonar.projectKey=UIS:0.0.1-SNAPSHOT
```

```
sonar.modules=api,application,backend,domain
```

```
api.sonar.projectBaseDir=api/
api.sonar.projectName=api
api.sonar.sources=src/main/java/
api.sonar.java.binaries=target/classes
api.sonar.jacoco.reportPaths=target/jacoco.exec
```

```
application.sonar.projectBaseDir=application/
application.sonar.projectName=application
application.sonar.sources=src/main/java/
application.sonar.java.binaries=target/classes
application.sonar.jacoco.reportPaths=target/jacoco.exec
```

```
backend.sonar.projectBaseDir=backend/
```

⁹ Anmerkung: verwende die Portnummer des Docker Containers.

```
backend.sonar.projectName=backend
backend.sonar.sources=src/main/java/
backend.sonar.java.binaries=target/classes
backend.sonar.jacoco.reportPaths=target/jacoco.exec
```

```
domain.sonar.projectBaseDir=domain/
domain.sonar.projectName=domain
domain.sonar.sources=src/main/java/
domain.sonar.java.binaries=target/classes
domain.sonar.jacoco.reportPaths=target/jacoco.exec
```

Die Properties-Datei wurde auch im GitLab gepusht: <https://peso.inso.tuwien.ac.at/repo/sqm-ss19/group11/tree/master>. SonarQube wird pro Modul ausgeführt.

Zur Analyse der Test Coverage wird JaCoCo verwendet. Dafür müssen das JaCoCo-Maven-Plugin und das Surefire-Maven-Plugin in dem pom.xml sowie die Anweisungen für die Erstellung des JaCoCo Coverage Reports, das sich auf das Gesamtprojekt bezieht, hinzugefügt werden:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <argLine>-Duser.language=en</argLine>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.3</version>

      <executions>
        <execution>
          <id>agent</id>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>jacoco-report</id>
          <phase>test</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.version>0.0.1-SNAPSHOT</project.version>

  <sonar.java.coveragePlugin>jacoco</sonar.java.coveragePlugin>
  <sonar.dynamicAnalysis>reuseReports</sonar.dynamicAnalysis>
  <sonar.jacoco.reportPath>${project.basedir}/../target/jacoco.exec</sonar.jacoco.reportPath>
  <sonar.language>java</sonar.language>
</properties>
```

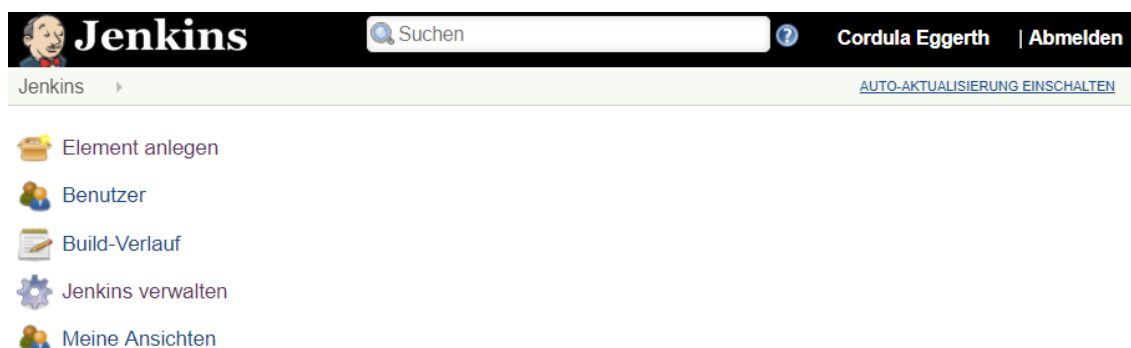
Außerdem muss das Surefire-Maven-Plugin¹⁰ in den pom.xml Files der einzelnen Module jeweils hinzugefügt werden:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.18.1</version>
      <configuration>
        <argLine>${argLine}</argLine>
      </configuration>
    </plugin>
  </plugins>
</build>
```

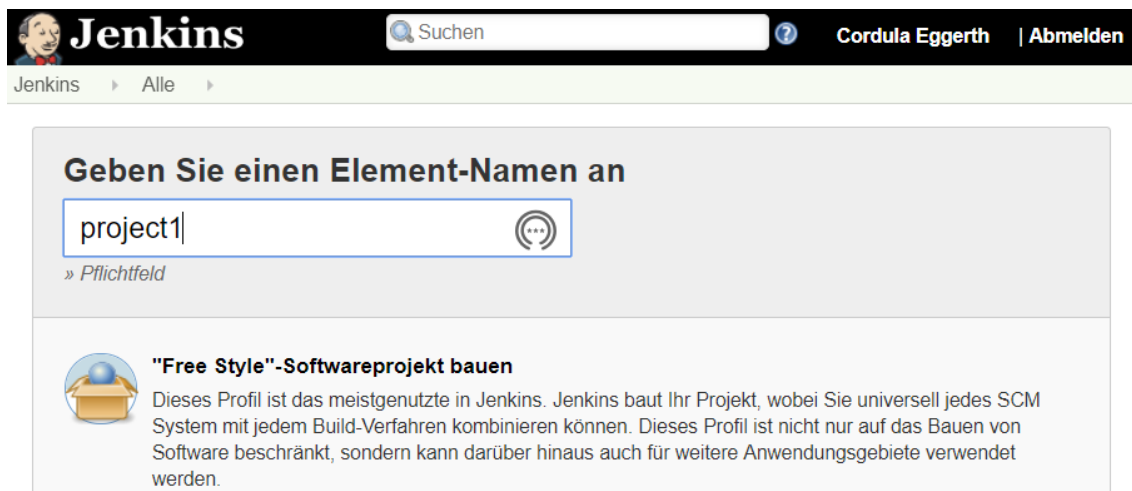
Die Änderungen in den pom.xml Files wurden auch auf GitLab gepusht.

PROJEKT IM JENKINS ANLEGEN, EINSTELLEN UND AUSFÜHREN („Build“): (localhost:8081)

Element anlegen > Name eingeben > Free Style Softwareprojekt bauen:



¹⁰ Anmerkung: Es wird hier mit Version 2.18.1 gearbeitet, da die neuere Version einige Male unvorhergesehen zu Build Failure führte (wegen EOFException). Für genaue Problematik siehe: <https://github.com/jacoco/jacoco/issues/546> bzw. <https://stackoverflow.com/questions/39634478/maven-jacoco-configuration-for-multi-module-projects?fbclid=IwAR2Wi8NKCICUY97w0i9CCPO3mpkqu6hrOESH9umuGLayHz8Rit4mKcux6Jg>.



Jenkins

Suchen

Cordula Eggerth | Abmelden

Jenkins > Alle >

Geben Sie einen Element-Namen an

project1

» Pflichtfeld

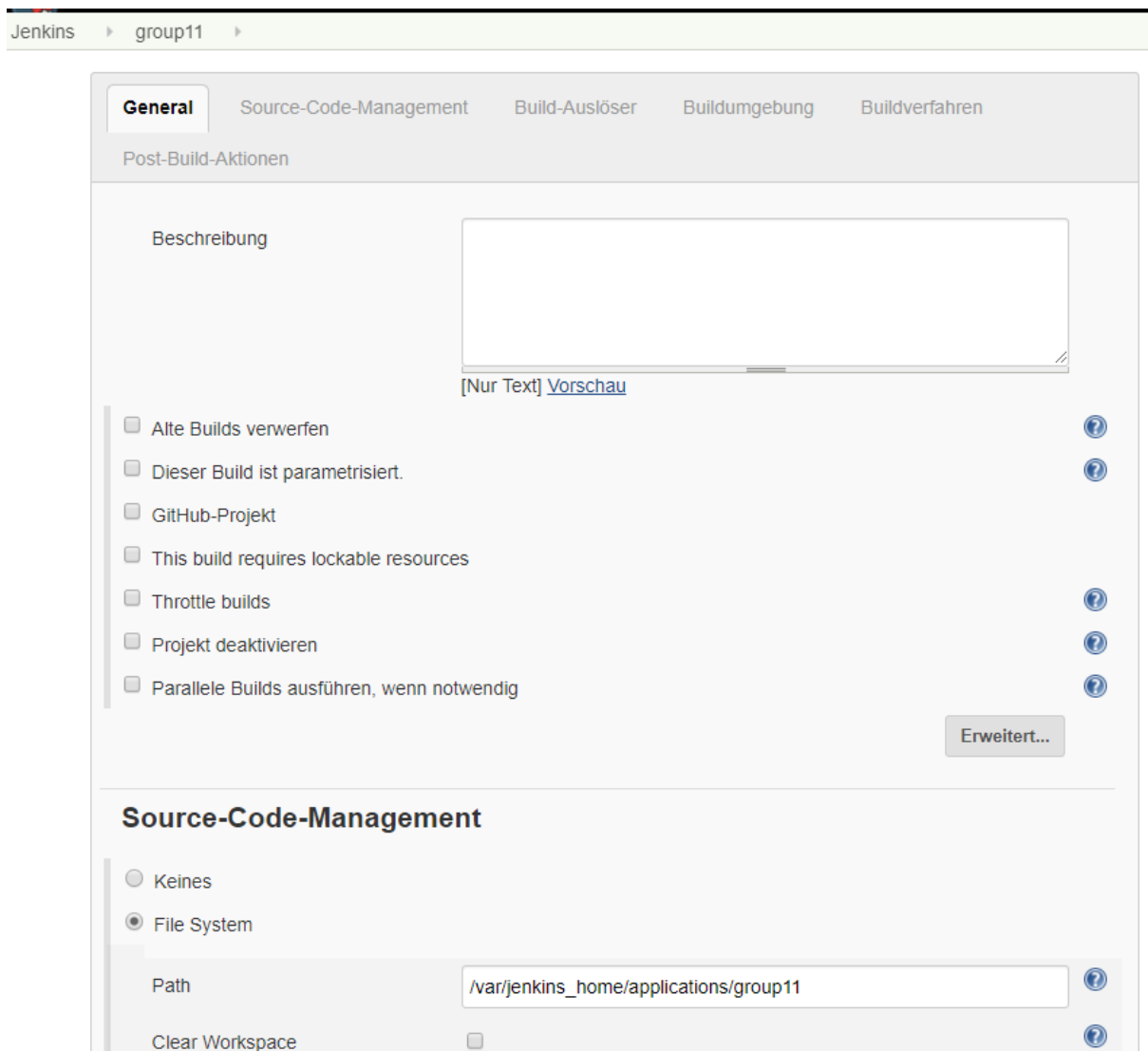
"Free Style"-Softwareprojekt bauen

Dieses Profil ist das meistgenutzte in Jenkins. Jenkins baut Ihr Projekt, wobei Sie universell jedes SCM System mit jedem Build-Verfahren kombinieren können. Dieses Profil ist nicht nur auf das Bauen von Software beschränkt, sondern kann darüber hinaus auch für weitere Anwendungsgebiete verwendet werden.

Option „File SCM“:

Dann müssen Informationen zum File System und Path gegeben werden im Bereich General.

Als Path sollte der Docker Volume Path (z.B: hier `/var/jenkins_home/applications/group11`) gefolgt vom Pfad zum Projekt im „applications“ Ordner angegeben werden.



Jenkins > group11 >

General

Source-Code-Management Build-Auslöser Buildumgebung Buildverfahren

Post-Build-Aktionen

Beschreibung

[Nur Text] [Vorschau](#)

- ☐ Alte Builds verwerfen
- ☐ Dieser Build ist parametrisiert.
- ☐ GitHub-Projekt
- ☐ This build requires lockable resources
- ☐ Throttle builds
- ☐ Projekt deaktivieren
- ☐ Parallele Builds ausführen, wenn notwendig

Erweitert...

Source-Code-Management

- ☐ Keines
- ☒ File System

Path `/var/jenkins_home/applications/group11`

Clear Workspace ☐

Im Bereich Buildverfahren sollte Maven Goals aufrufen gewählt werden, und dann dort die Maven-Version¹¹ eingetragen werden und als Goals das jeweilige Maven-Kommando (z.B. test oder clean test) eingegeben werden:

The image consists of two screenshots of a web-based configuration interface for build processes.

The top screenshot shows the 'Buildverfahren' (Build Process) section. A dropdown menu is open for 'Build-Schritt hinzufügen' (Add Build Step). The menu options are: 'Ant aufrufen', 'Execute SonarQube Scanner', 'Gradle ausführen', 'Maven Goals aufrufen' (highlighted), 'Run with timeout', 'Set build status to "pending" on GitHub commit', 'Shell ausführen', 'SonarScanner for MSBuild - Begin Analysis', 'SonarScanner for MSBuild - End Analysis', and 'Windows Batch-Datei ausführen'.

The bottom screenshot shows the 'Maven Goals aufrufen' configuration form. It has a title bar with a red 'X' and a help icon. The form contains two input fields: 'Maven-Version' with the value 'maven3' and 'Goals' with the value 'test'. There is a small dropdown arrow to the right of the 'Goals' field. Below the fields is a button labeled 'Erweitert...'. At the bottom of the form is a button labeled 'Build-Schritt hinzufügen'.

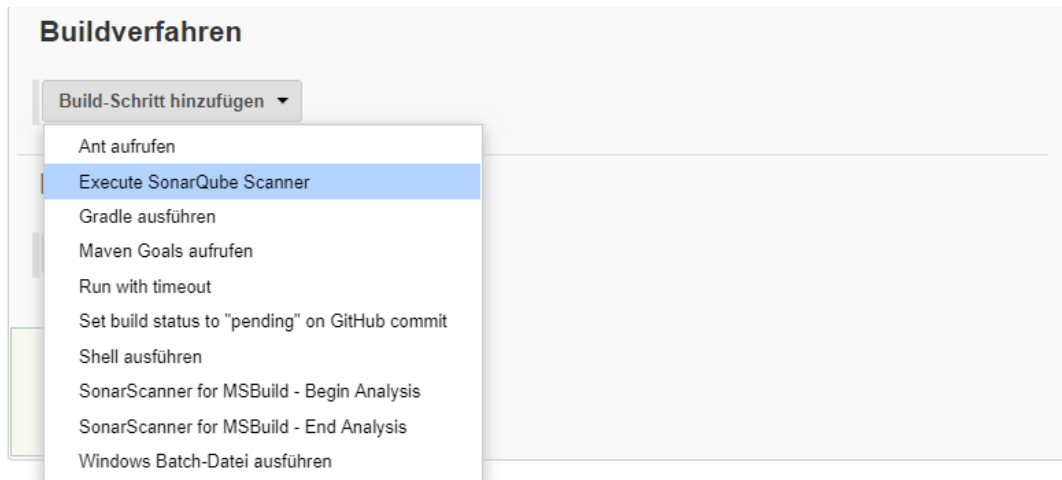
Option „Git“:

Im Bereich Source-Code-Management wird die Option Git ausgewählt und das Repository <https://peso.inso.tuwien.ac.at/repo/sqm-ss19/group11> wird verlinkt.

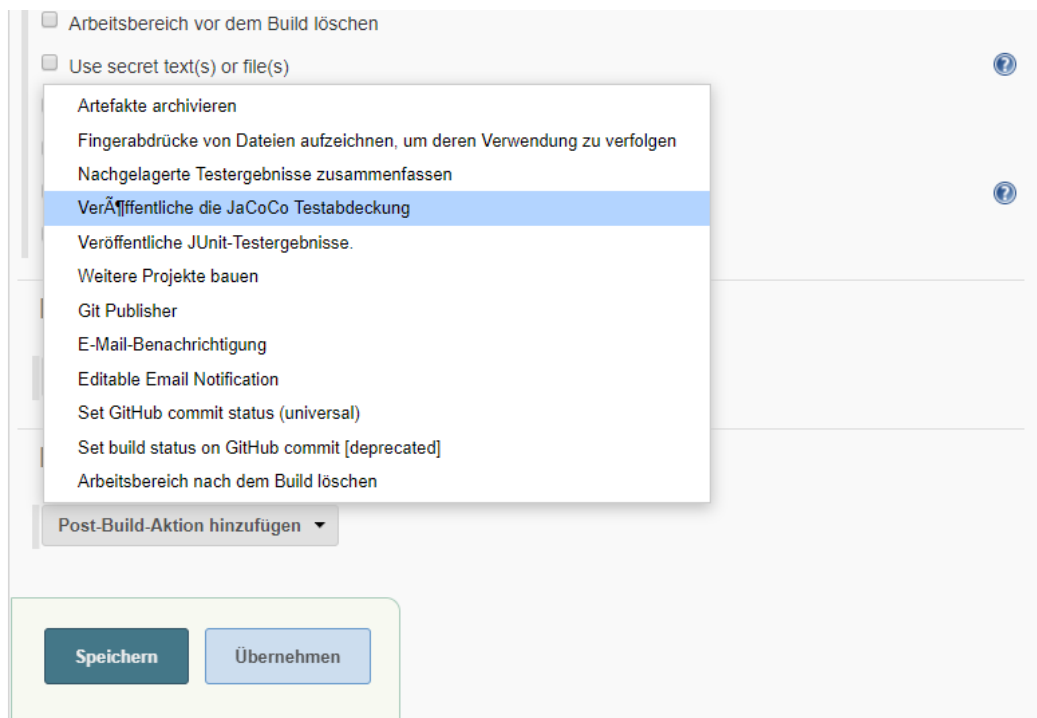
In Maven Goals (in Buildverfahren) kann clean install eingegeben werden.

Außerdem sollte in Buildverfahren im Dropdown-Menü Execute SonarQube Scanner ausgewählt werden:

¹¹ Anmerkung: jene Version, die vorher während der Konfiguration angelegt wurde – sie erscheint bereits in der Maske und kann ausgewählt werden.



Im Bereich Post-Build-Aktionen sollte im Dropdown-Menü Veröffentliche die JaCoCo-Testabdeckung ausgewählt werden:



Die genaue Konfiguration kann folgendermaßen aussehen:

Post-Build-Aktionen

Veröffentlichen die JaCoCo Testabdeckung

Path to exec files (e.g.:
)target/.exec, **)jacoco.exec)

Inclusions (e.g.: **/*.class)

Exclusions (e.g.: **/*Test*.class)

**/*.exec

Path to class directories (e.g.: **)target/classDir, **)classes)

**/*.classes

Path to source directories (e.g.:
**)mySourceFiles)

Inclusions (e.g.:
)*.java,/*.groovy,**/*.gs)

Exclusions (e.g.:
generated/**/*.*.java)

**/src/main/java

**/*.java

☐ Disable display of source files for coverage

☐ Change build status according the thresholds

	Instruction	% Branch	% Complexity	% Line	% Method	% Class
	0	0	0	0	0	0
	0	0	0	0	0	0

Hier ausgefüllte Felder in Post-Build-Aktionen > Veröffentliche die JaCoCo Testabdeckung:

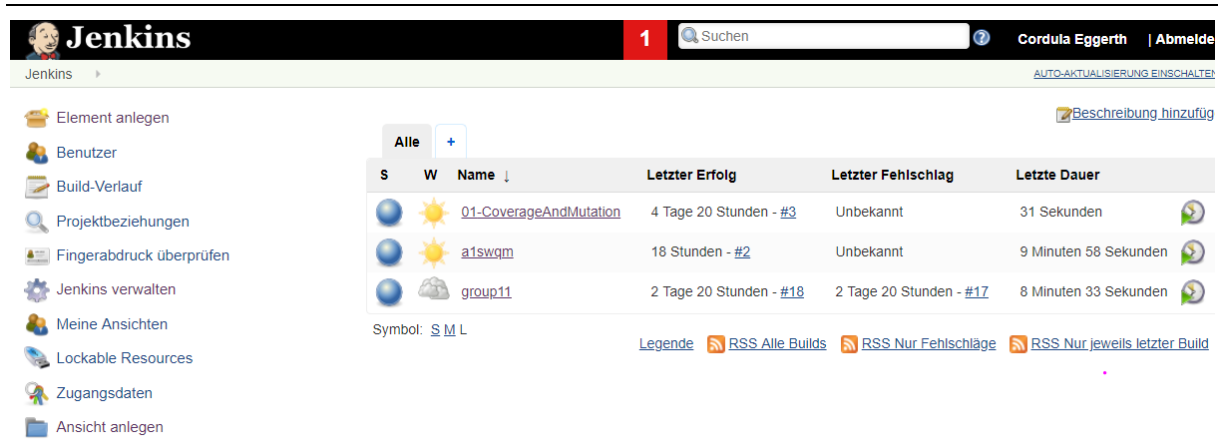
- Path to exec files (e.g.: `**/target/**/*.exec`, `**/jacoco.exec`):
`**/**/*.exec`
- Path to class directories (e.g.: `**/target/classDir`, `**/classes`):
`**/classes`
- Path to source directories (e.g.: `**/mySourceFiles`):
`**/src/main/java`
- Inclusions (e.g.: `**/*.java`, `**/*.groovy`, `**/*.gs`):
`**/*.java`

Die gemachten Änderungen sollten durch Übernehmen bzw. Speichern bestätigt werden, und die Konfiguration bzw. das Set-Up ist nun fertig.

Mittels Jenkins > [gewähltes Projekt] > Konfigurieren können die Einstellungen (falls notwendig) wieder verändert werden.

Sobald das Projekt gebaut wurde, also ein „Build“ durchgeführt wurde, wird dafür ein Eintrag auf der Hauptseite von Jenkins angelegt:

¹² Anmerkung: Die Erstellung des JaCoCo-Reports erfolgt jeweils einzeln pro Modul.



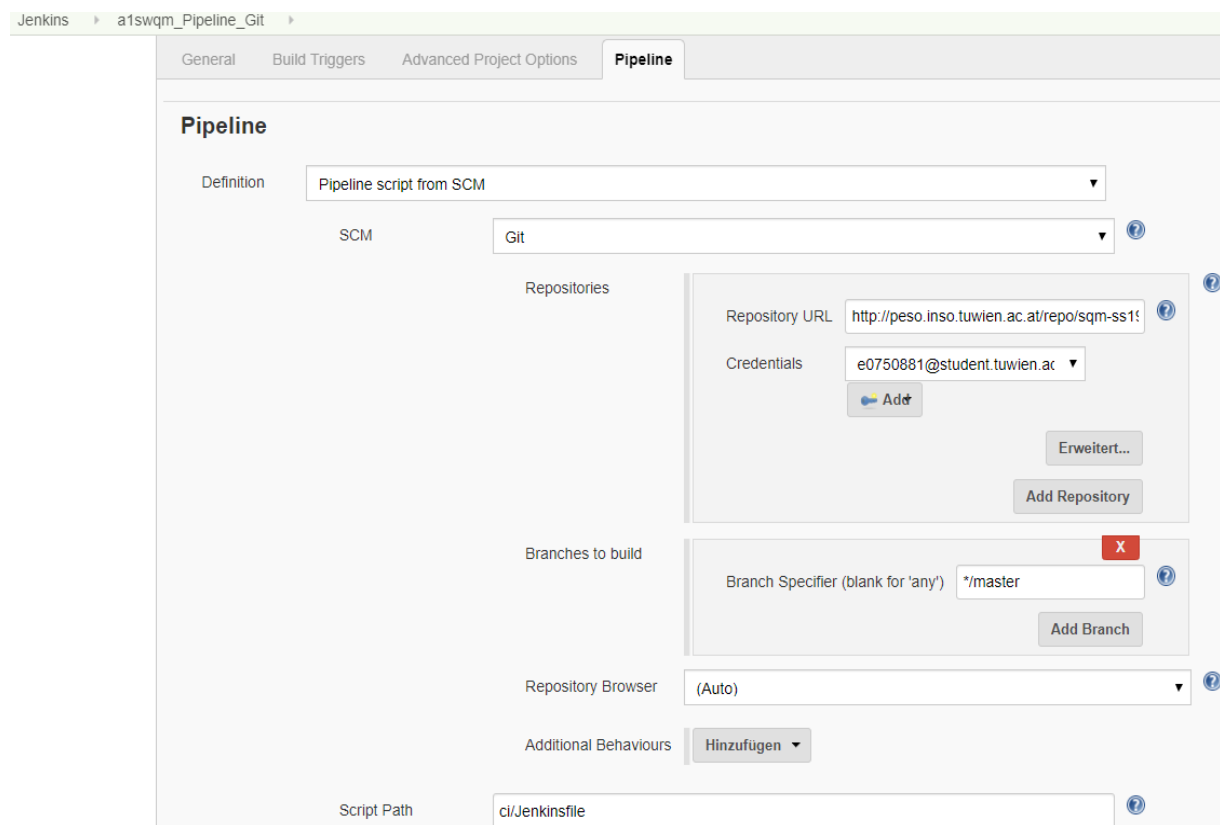
The screenshot shows the Jenkins dashboard with a sidebar on the left containing links like 'Element anlegen', 'Benutzer', 'Build-Verlauf', etc. The main area displays a table of builds. The table has columns for 'S' (Status), 'W' (Weather icon), 'Name', 'Letzter Erfolg', 'Letzter Fehlschlag', and 'Letzte Dauer'. Three builds are listed: '01-CoverageAndMutation', 'a1swqm', and 'group11'. The 'a1swqm' build is highlighted with a sun icon, indicating a successful build.

S	W	Name	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer
		01-CoverageAndMutation	4 Tage 20 Stunden - #3	Unbekannt	31 Sekunden
		a1swqm	18 Stunden - #2	Unbekannt	9 Minuten 58 Sekunden
		group11	2 Tage 20 Stunden - #18	2 Tage 20 Stunden - #17	8 Minuten 33 Sekunden

Sofern die letzten Builds ohne Probleme durchgeführt werden konnten, wird eine Sonne als Symbol angezeigt, falls nicht dann Wolken, die die Build-Stabilität zeigen sollen. Daneben werden auch die Zeit, vor der der letzte Build-Erfolg bzw. -Fehlschlag stattgefunden hat, und die Dauer des letzten Build-Vorgangs angegeben.

Vorschläge für die Build-Jobs und die Ausführung des CI-Systems mittels Pipeline

Bisher wurde jeweils ein Projekt-Build und -Test insgesamt ausgeführt. Nun wird dieser Prozess in eine Reihe von Schritten einer Pipeline aufgeteilt. Anfangs wird in Jenkins dafür Pipeline als Typ ausgewählt und angelegt. Im Bereich Pipeline werden folgende Konfigurationseigenschaften angegeben, um Jenkins mit dem Git-Repository verbinden zu können:



The screenshot shows the Jenkins Pipeline configuration page. The 'Definition' dropdown is set to 'Pipeline script from SCM'. The 'SCM' dropdown is set to 'Git'. The 'Repositories' section shows a 'Repository URL' of 'http://peso.inso.tuwien.ac.at/repo/sqm-ss1' and 'Credentials' set to 'e0750881@student.tuwien.ac'. The 'Branches to build' section shows a 'Branch Specifier (blank for 'any')' of '*/master'. The 'Repository Browser' is set to '(Auto)'. The 'Script Path' is set to 'ci/Jenkinsfile'.

Zur Ausführung der Pipeline wird im zugehörigen Git-Repository ein Ordner (namens `ci`) erstellt, in dem das Jenkinsfile (mit Groovy-Syntax) liegt. Diese Datei enthält die Jobs und Stages, die im Rahmen der Jenkins-Pipeline ausgeführt werden:

```
pipeline {
    agent any

    tools {
        maven "maven3"
    }

    stages{
        stage('Clean Workspace') {
            steps{
                script {
                    cleanWs()
                }
            }
        }

        stage('Checkout') {
            environment {
                MyCredentialsID = "2d83eae8-c340-4166-a74a-16222aa2b895"
                GitlabURL      = "http://peso.inso.tuwien.ac.at/repo/sqm-ss19/group11.git"
            }
            steps {
                checkout([$class: 'GitSCM',
                    branches: [[name: '*/master']],
                    doGenerateSubmoduleConfigurations: false,
                    extensions: [],
                    submoduleCfg: [],
                    userRemoteConfigs: [[credentialsId: MyCredentialsID, url: GitlabURL]]
                ])
            }
        }

        stage('Build and Test') {
            steps {
                sh "mvn clean test"
            }
        }

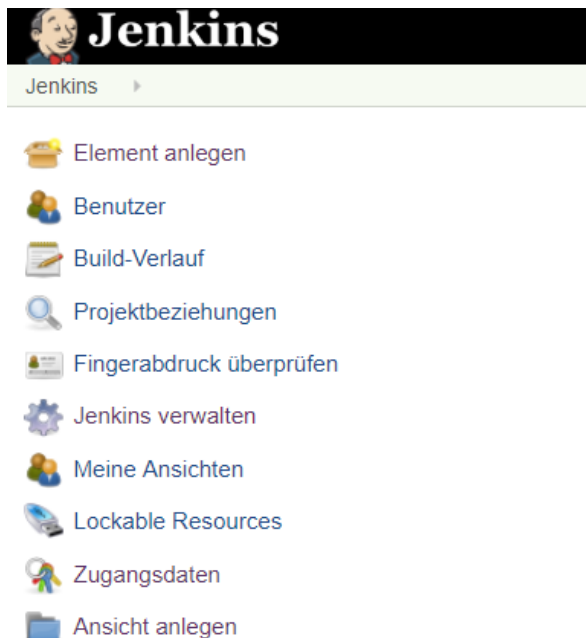
        stage('Code coverage with JaCoCo') {
            steps {
                jacoco()
            }
        }

        stage('Sonarqube-Analysis') {
```

```
environment {  
    scannerHome = tool 'scanner'  
}  
steps {  
    withSonarQubeEnv('sonarqube') {  
        sh "${scannerHome}/bin/sonar-scanner"  
    }  
}  
}  
}
```

Die **Pipeline** besteht im vorliegenden Fall aus fünf Stages (bzw. Schritten), nämlich Clean Workspace, Checkout, Build and Test, Code Coverage with JaCoCo und Sonarqube-Analysis, die nacheinander ausgeführt werden. In der Stage Build and Test wird einmal `mvn clean test` ausgeführt, also es findet Build und Test statt.

Bei den obenstehend **blau hinterlegten** Bereichen sollte darauf geachtet werden, dass der Name der selbst definierten Maven-Version und Sonarqube-Scanner-Version verwendet wird – dieser Name wird direkt aus der Jenkins-Konfiguration übernommen. Im obenstehend **grün hinterlegten** Bereich sollte die eigene Credentials-ID für die Zugangsdaten zu dem Git-Repository angegeben werden – diese ist im allgemeinen Jenkins-Dashboard unter Zugangsdaten zu finden, von denen nur die eigene ID im Groovy-Script in der Stage Checkout hinzugefügt werden sollte (wie anhand des untenstehenden Beispiels gezeigt):



Credentials

T	P	Store ↓	Domain	ID	Name
		Jenkins	(global)	2b856514-9980-4a00-a560-23c5b42dc30b	e0750881@tuwien.ac.at/*****
		Jenkins	(global)	2d83eae8-c340-4166-a74a-16222aa2b895	e0750881@student.tuwien.ac.at/*****

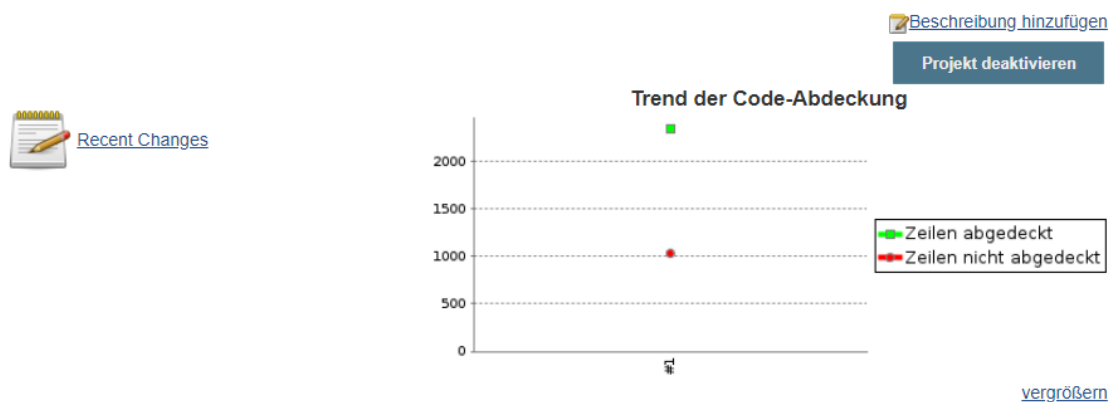
Symbol: [S](#) [M](#) [L](#)

Stores scoped to Jenkins

P	Store ↓	Domains
	Jenkins	(global)

Durch Klick auf **Jetzt bauen** kann die Pipeline gestartet werden. Im Bereich des betrachteten Projekt wird im Jenkins dann ein Bereich angelegt, in dem die einzelnen Phasen durchgeführt werden und Details zur Ausführung der Stages gegeben werden. Das Ergebnis sieht für die Ausführung der vorliegenden Pipeline wie folgt aus:

Pipeline a1swqm_Pipeline_Git



Stage View

	Declarative: Checkout SCM	Declarative: Tool Install	Clean Workspace	Checkout	Build and Test	Code coverage with JaCoCo	Sonarqube-Analysis
Average stage times: (Average full run time: ~5min 2s)	13s	295ms	1s	9s	3min 17s	6s	1min 10s
#1 Apr 28 15:16 No Changes	13s	295ms	1s	9s	3min 17s	6s	1min 10s

SonarQube Quality Gate

UIS **OK**server-side processing: **Success**

1.2 Test Coverage

Es gibt eine Vielzahl an Test Coverage Tools (Cobertura, JaCoCo, OpenClover, usw.), die als Plugin in Jenkins integriert werden können.

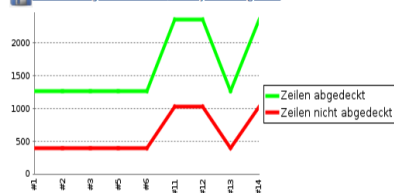
Wir haben uns für **JaCoCo** entschieden, weil es ein häufig verwendetes Open Source Tool ist und es regelmäßige Updates dafür gibt. Die Test Coverage wird für das Projekt insgesamt, jedes Package und jede Klasse ausführlich analysiert – also auf verschiedenen Levels. Außerdem ist leicht erkennbar, an welcher Stelle noch Tests fehlen.

Granularität der Resultate des JaCoCo Coverage Reports:

Die Granularität des JaCoCo Coverage Report ist sehr hoch. Beim Report handelt es sich um einen HTML-Report, der eine Darstellung der Testabdeckung vom Gesamtprojekt bis hinunter auf Klassenebene bietet:

JaCoCo Coverage Report

[Download jacoco.exec binary coverage file](#)



Overall Coverage Summary


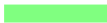

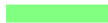








Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
all classes	57% M: 10204 C: 13263	41% M: 646 C: 454	58% M: 593 C: 816	69% M: 1035 C: 2352	81% M: 164 C: 695	95% M: 6 C: 121

Coverage Breakdown by Package

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
at.ac.tuwien.inso.sgm	M: 13 C: 7 35%	M: 0 C: 0 100%	M: 1 C: 2 67%	M: 6 C: 2 25%	M: 1 C: 2 67%	M: 0 C: 1 100%
at.ac.tuwien.inso.sgm.config	M: 0 C: 199 100%	M: 0 C: 0 100%	M: 0 C: 9 100%	M: 0 C: 46 100%	M: 0 C: 9 100%	M: 0 C: 3 100%
at.ac.tuwien.inso.sgm.controller	M: 219 C: 193 47%	M: 26 C: 2 7%	M: 19 C: 15 44%	M: 56 C: 47 46%	M: 6 C: 14 70%	M: 1 C: 4 80%
at.ac.tuwien.inso.sgm.controller.admin	M: 138 C: 869 86%	M: 17 C: 41 71%	M: 15 C: 49 77%	M: 31 C: 171 85%	M: 0 C: 35 100%	M: 0 C: 5 100%
at.ac.tuwien.inso.sgm.controller.admin.forms	M: 127 C: 243 66%	M: 31 C: 7 18%	M: 21 C: 36 63%	M: 15 C: 65 81%	M: 3 C: 35 92%	M: 0 C: 4 100%

Der Report auf Gesamtprojekt-Ebene besteht dabei aus drei Bereichen. Der erste Bereich zeigt die Anzahl der durch die Tests abgedeckten bzw. nicht abgedeckten Zeilen und deren Veränderung über den Buildverlauf. Der zweite Bereich (*Overall Coverage Summary*) gibt einen allgemeinen Überblick über die Testabdeckung des gesamten Projektes. Der dritte Bereich (*Coverage Breakdown by Package*) listet alle Packages des Projektes auf, wobei man hier bis auf Klassenebene hineinnavigieren kann:

UisApplication

Name	instruction	branch	complexity	Zeilen	Methoden
UisApplication()	M: 0 C: 3 100% 	M: 0 C: 0 100%	M: 0 C: 1 100% 	M: 0 C: 1 100% 	M: 0 C: 1 100% 
main(String[])	M: 13 C: 0 0% 	M: 0 C: 0 100%	M: 1 C: 0 0% 	M: 6 C: 0 0% 	M: 1 C: 0 0% 
static {...}	M: 0 C: 4 100% 	M: 0 C: 0 100%	M: 0 C: 1 100% 	M: 0 C: 1 100% 	M: 0 C: 1 100% 

Coverage

```

1: package at.ac.tuwien.inso.sqm;
2:
3: import org.slf4j.Logger;
4: import org.slf4j.LoggerFactory;
5: import org.springframework.boot.SpringApplication;
6: import org.springframework.boot.autoconfigure.SpringBootApplication;
7: import org.springframework.cache.annotation.EnableCaching;
8: import org.springframework.scheduling.annotation.EnableScheduling;
9:
10: @SpringBootApplication
11: @EnableCaching
12: @EnableScheduling
13: public class UisApplication {
14:
15:     private static final Logger logger = LoggerFactory.getLogger(UisApplication.class);
16:
17:     public static void main(String[] args) {
18:         try
19:         {
20:             SpringApplication.run(UisApplication.class, args);
21:         }
22:         catch(Error e)
23:         {
24:             logger.error("An error occurred in the application", e);

```

Nutzen des Reports für verschiedene Stakeholder:

Durch die hohe Granularität des Reports profitieren Entwickler, Tester und Manager alle von den Resultaten. Die Entwickler und Tester können bis auf Klassenebene hineinnavigieren und so schnell die noch nicht abgedeckten Codezeilen identifizieren und die vorhandenen Tests anpassen bzw. neue Tests hinzufügen. Für die Manager ist vor allem die Testabdeckung auf Gesamtprojekt-Ebene interessant. Hier kann man schnell erkennen, ob das vorgegebene Level der Testabdeckung erreicht wurde oder nicht. Der Trend der abgedeckten bzw. nicht abgedeckten Zeilen bietet einen guten Anhaltspunkt darüber, wie sich die Tests über den Zeitverlauf entwickelt haben. Die Manager können so gegebenenfalls schnell einschreiten und die Entwickler und Tester zum vermehrten Schreiben von Tests animieren.

Export und Repräsentation des Reports:

Die Reports können nur als .exec-Datei exportiert und in weiterer Folge beispielsweise in IntelliJ IDEA, Eclipse oder SonarQube importiert werden. Das ist aber kein großes Problem, da der HTML-Report in Jenkins eigentlich ausreichend ist und jederzeit auf die Reports aller Builds zugegriffen werden kann.



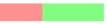



Beurteilung der weiteren Verwendbarkeit des Reports im Projekt:

Unserer Meinung nach ist JaCoCo eine gute Wahl für dieses Projekt, weil es einen feingranularen Report in HTML-Form anbietet, der sowohl von Entwicklern, Testern als auch Managern gleichermaßen gut verwendet werden kann.

Beurteilung der Angemessenheit der Resultate:

Die Resultate der Testabdeckung erscheinen auf den ersten Blick relativ angemessen, sie könnten aber besser sein:

Overall Coverage Summary

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
all classes	57%  M: 10204 C: 13263	41%  M: 646 C: 454	58%  M: 593 C: 816	69%  M: 1035 C: 2352	81%  M: 164 C: 695	95%  M: 6 C: 121

Wenn wir uns die Resultate genauer anschauen, sehen wir, dass auch Klassen mitberücksichtigt werden, die eigentlich nicht getestet werden müssen. Die Klassen `DataInitializer` und `DemoDataInitConfig` im Paket `at.ac.tuwien.inso.sqm.initializer` dienen beispielsweise nur der Initialisierung von Testdaten und müssen dementsprechend nicht getestet werden. Wir haben diese zwei Klassen durch folgende Anpassung in Jenkins aus der Testabdeckung ausgeschlossen:

Veröffentliche die JaCoCo Testabdeckung

Path to exec files (e.g.: `**/target/**/*.exec`, `**/jacoco.exec`)

Inclusions (e.g.: `**/*.class`)

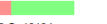
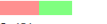
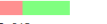
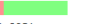


Exclusions (e.g.: `**/*Test*.class`)

`**/target/**/*.exec`

`**/DataInitializer.class, **/DemoDataInitConfig.class`

Ein neuerlicher Build im Jenkins führt im Anschluss daran zu einer Testabdeckung, die sich recht deutlich vom ersten Resultat (im positiven Sinne) unterscheidet:

Overall Coverage Summary

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
all classes	80%  M: 3200 C: 13194	42%  M: 628 C: 454	60%  M: 547 C: 815	81%  M: 543 C: 2351	85%  M: 127 C: 694	96%  M: 5 C: 120

Beurteilung der Coverage-Ergebnisse des betrachteten Projekts:

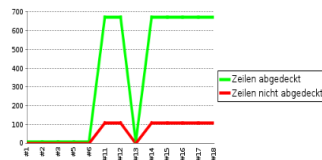
Optimal wäre natürlich eine Test Coverage von 100%, aber unserer Meinung nach ist für dieses Projekt 80% ausreichend, weil es sich nicht um eine sehr sicherheits- und lebenskritische Anwendung handelt, sondern "lediglich" um eine Web-Anwendung für Universitäten. Wenn wir uns die Testabdeckung des Projekts anschauen, sehen wir, dass wir nicht pauschal sagen können, ob die Coverage ausreichend ist oder nicht, sondern davon abhängig ist, welches Kriterium der Testabdeckung wir uns anschauen. Die Testabdeckung bezüglich Klassen (96%), Methoden (85%), Zeilen (81%) und Instruction (80%) ist ausreichend, wohingegen die Testabdeckung bezüglich Branch (42%) und Complexity (60%) nicht ausreichend ist.

Identifikation von Bereichen, die im Source Code nicht ausreichend durch Tests abgedeckt wurden:

Das Domain-Modul (mit den Packages `at.ac.tuwien.inso.sqm.dto`, `at.ac.tuwien.inso.sqm.entity` und `at.ac.tuwien.inso.sqm.enums`) hat eine schlechte Testabdeckung:

at.ac.tuwien.inso.sqm.dto	M: 434 C: 554 56%	M: 94 C: 22 19%	M: 78 C: 55 41%	M: 80 C: 109 58%	M: 30 C: 45 60%	M: 1 C: 6 86%
at.ac.tuwien.inso.sqm.entity	M: 811 C: 2245 73%	M: 270 C: 142 34%	M: 216 C: 212 50%	M: 129 C: 418 76%	M: 36 C: 186 84%	M: 0 C: 21 100%
at.ac.tuwien.inso.sqm.enums	M: 108 C: 0 0%	M: 4 C: 0 0%	M: 8 C: 0 0%	M: 17 C: 0 0%	M: 6 C: 0 0%	M: 1 C: 0 0%

Innerhalb des Pakets `at.ac.tuwien.inso.sqm.service` fällt vor allem die Klasse `CourseServiceImpl` mit einer schlechten Testabdeckung ins Auge:

Package: `at.ac.tuwien.inso.sqm.service`

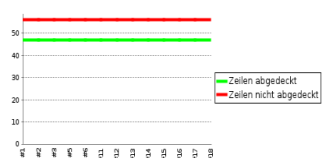
Coverage Summary

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
<code>at.ac.tuwien.inso.sqm.service</code>	M: 733 C: 3610 83%	M: 51 C: 139 73%	M: 65 C: 239 79%	M: 109 C: 670 86%	M: 23 C: 186 89%	M: 0 C: 27 100%

Coverage Breakdown by Source File

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
AccountActivationServiceImpl	M: 0 C: 70 100%	M: 0 C: 2 100%	M: 0 C: 5 100%	M: 0 C: 13 100%	M: 0 C: 4 100%	M: 0 C: 1 100%
CourseNormalizer	M: 20 C: 35 64%	M: 1 C: 1 50%	M: 2 C: 2 50%	M: 4 C: 6 60%	M: 1 C: 2 67%	M: 0 C: 1 100%
CourseServiceImpl	M: 276 C: 479 63%	M: 26 C: 22 46%	M: 20 C: 24 55%	M: 47 C: 79 63%	M: 3 C: 17 85%	M: 0 C: 1 100%
FeedbackService	M: 0 C: 128 100%	M: 0 C: 4 100%	M: 0 C: 9 100%	M: 0 C: 28 100%	M: 0 C: 7 100%	M: 0 C: 1 100%
GradeService	M: 46 C: 221 83%	M: 2 C: 10 83%	M: 4 C: 13 76%	M: 7 C: 40 85%	M: 2 C: 9 82%	M: 0 C: 1 100%
LecturerServiceImpl	M: 0 C: 105 100%	M: 0 C: 0 100%	M: 0 C: 6 100%	M: 0 C: 16 100%	M: 0 C: 6 100%	M: 0 C: 1 100%
MandatoryCourseScorer	M: 0 C: 85 100%	M: 0 C: 0 100%	M: 0 C: 9 100%	M: 0 C: 14 100%	M: 0 C: 9 100%	M: 0 C: 1 100%
Nachrichten	M: 0 C: 47 100%	M: 0 C: 0 100%	M: 0 C: 4 100%	M: 0 C: 7 100%	M: 0 C: 4 100%	M: 0 C: 1 100%
RecommendationService	M: 41 C: 118 74%	M: 0 C: 2 100%	M: 3 C: 11 79%	M: 3 C: 20 87%	M: 3 C: 10 77%	M: 0 C: 1 100%
SemesterRecommendationCourseRelevanceFilter	M: 0 C: 108 100%	M: 0 C: 4 100%	M: 0 C: 9 100%	M: 0 C: 21 100%	M: 0 C: 7 100%	M: 0 C: 1 100%

Im Paket `at.ac.tuwien.inso.sqm.controller` weisen vor allem die Klassen `GlobalExceptionHandler`, `LoginController` und `RootController` eine schlechte Testabdeckung auf:

Package: `at.ac.tuwien.inso.sqm.controller`

Coverage Summary

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
<code>at.ac.tuwien.inso.sqm.controller</code>	M: 219 C: 193 47%	M: 26 C: 2 7%	M: 19 C: 15 44%	M: 56 C: 47 46%	M: 6 C: 14 70%	M: 1 C: 4 80%

Coverage Breakdown by Source File

Name	instruction	branch	complexity	Zeilen	Methoden	Klassen
AccountActivationController	M: 0 C: 75 100%	M: 0 C: 2 100%	M: 0 C: 5 100%	M: 0 C: 21 100%	M: 0 C: 4 100%	M: 0 C: 1 100%
Constants	M: 3 C: 0 0%	M: 0 C: 0 100%	M: 1 C: 0 0%	M: 1 C: 0 0%	M: 1 C: 0 0%	M: 1 C: 0 0%
GlobalExceptionHandler	M: 63 C: 104 62%	M: 0 C: 0 100%	M: 3 C: 6 67%	M: 12 C: 22 65%	M: 3 C: 6 67%	M: 0 C: 1 100%
LoginController	M: 98 C: 7 7%	M: 18 C: 0 0%	M: 10 C: 2 17%	M: 27 C: 2 7%	M: 1 C: 2 67%	M: 0 C: 1 100%
RootController	M: 55 C: 7 11%	M: 8 C: 0 0%	M: 5 C: 2 29%	M: 16 C: 2 11%	M: 1 C: 2 67%	M: 0 C: 1 100%

Was sich durchwegs durch das ganze Projekt zieht, sind fehlende oder unvollständige Tests der `equals()`-Methoden.

<i>Empfehlungen zur Verbesserung der Coverage des Projekts:</i>
--

Wie wir bereits weiter oben gesehen haben, ist vor allem die Testabdeckung bezüglich Branch und Complexity stark verbesserungswürdig. Deshalb sollten die Tests vor allem hinsichtlich dieser Kriterien angepasst bzw. erweitert werden. Bei der schlechten Testabdeckung im Domain-Modul stellt sich die Frage, ob es wirklich notwendig ist, die Getter/Setter der DTOs und Entities zu testen oder ob es nicht besser wäre, das gesamte Modul aus der Berechnung der Testabdeckung auszuschließen. Die Services dagegen stellen eine zentrale Komponente der Anwendung dar und sollten daher ausreichend getestet werden, was bei der Klasse `CourseServiceImpl` beispielsweise nicht der Fall ist. Die Testabdeckung der Klassen `LoginController`, `RootController` und `GlobalExceptionHandler` sollte auch erhöht werden. Außerdem sollten die `equals()`-Methoden vollständig getestet werden.

1.3 SonarQube Integration



Zunächst ist anzumerken, dass das Projekt die vordefinierten Qualitätsstandards erfüllt. Diese Regeln werden von Sonarqube als Default (Sonar way) zur Verfügung gestellt. Die Analyse zeigt, dass das Projekt 134 *Bugs*, 12 *Vulnerabilities* und 436 *Code Smells* aufweist. Dies gibt einen Hinweis auf eine schlampige Programmierung, da die Applikation nur aus ca. 8000 Codezeilen besteht. Als äußerst positiv ist anzumerken, dass die Code Duplikationen unter 1 Prozent liegen. Dies kann jedoch bei genauerer Fokussierung auch auf 0 Prozent reduziert werden.

Priorisierung der Issues:

Um die Reihenfolge der Abarbeitung der einzelnen Sonar Issues in eine Reihenfolge zu bringen, werden sie in folgende Kategorien eingestuft:

- [High]: sollten umgehend im aktuellen Sprint behoben werden
- [Medium]: können im nächsten Sprint eingeplant werden
- [Low]: können vorerst vernachlässigt werden, benötigen jedoch zumeist keinen großen Aufwand

Vulnerabilities (3h 10min):

Im Projekt befinden sich 12 “Vulnerabilities”, welche alle als Minor gekennzeichnet sind. Laut SonarQube beträgt der Aufwand 3h 10min, um diese Fehler zu bereinigen. Jedoch sollten diese Issues als erstes behandelt werden da sie alle eine sehr hohe Priorität aufweisen und zu Anomalien im Code führen können.

- **[High]: "enum" fields should not be publicly mutable [4]**
(Lower the visibility of this setter or remove it altogether.)

Da Enums als Konstanten angesehen werden, sollte kein `public` Zugriff erlaubt werden. Sie sollten nur mittels Konstruktor initialisiert werden.

- **[High]: Class variable fields should not have public accessibility [4]**
(Make `QR_PREFIX` a static final constant or non-public and provide accessors if needed.)

Auf Klassenvariablen sollte mittels Getter/Setter zugegriffen und sollten als `private` deklariert werden.

- **[High]: "public static" fields should be constant [2]**
(Make this "public static `QR_PREFIX`" field final)

Diese Anpassung benötigt keinen Aufwand, da man nur den `final` Typ anfügen muss. Per Definition sollten alle `public static` fields auf `final` gesetzt werden.

- **[Medium]: Mutable fields should not be "public static" [2]**
(Make this member "protected".)

```
public static final Map<Feedback.Type, Double> feedbackWeights = new HashMap<Feedback.Type, Double>() {
```

Make this member "protected". ...

22 days ago ▾ L33 🔗

 Vulnerability ▾  Minor ▾  Open ▾ Not assigned ▾ 15min effort [Comment](#)

 cert, cwe, unpredictable ▾

Dieses Issue kann als “**false positive**” angesehen werden, da es durch die `final` Deklaration nicht modifiziert werden kann, jedoch sollte die Variable dennoch als `private` gekennzeichnet werden.

Bugs (1d 3h):

Die 134 Bugs die sich im Projekt befinden sind alle als Major bzw. Minor gekennzeichnet, jedoch ist die Kategorisierung nicht aussagekräftig. Vor allem die Issues in den Data-Initializern können vernachlässigt werden, da sie nicht für die Produktion benötigt werden. Laut der Analyse benötigt die Behebung ca. 1 Tag.

Sonar Issue	Kategorie
[Major]: "BigDecimal(double)" should not be used [116]	[High]
[Major]: Null pointers should not be dereferenced [1]	[High]
[Major]: Optional value should only be accessed after calling <code>isPresent()</code> [2]	[High]
[Major]: "equals" method overrides should accept "Object" parameters [1]	[Medium]
[Minor]: Double Brace Initialization should not be used [13]	[Low]
Major]: Identical expressions should not be used on both sides of a binary operator [1]	[Low]

Code Smells (9d) :

In der UIS Applikation befinden sich 463 Code Smells. Diese sollten behoben werden, um die Lesbarkeit und das Refactoring des Codes in Zukunft zu erleichtern. Auch können dadurch etwaige Missverständnisse behoben werden und neue Projektmitglieder sich besser in den Code einlesen. Laut der Sonar Analyse beträgt die Dauer 9 Tage, um alle Issues zu beheben.

Sonar Issue	Kategorie
[Critical]: Methods should not be empty [1]	[High]
[Major]: Track uses of "FIXME" tags [15]	[High]
[Blocker]: Methods and field names should not be the same or differ only by capitalization [2]	[High]
[Info]: Track uses of "TODO" tags [16]	[High]
[Critical]: Constant names should comply with a naming convention [15]	[High]
[Minor]: Boolean checks should not be inverted [1]	[High]
[Minor]: Static non-final field names should comply with a naming convention [2]	[High]
[Critical]: Cognitive Complexity of methods should not be too high [2]	[High]
[Major]: Generic exceptions should never be thrown [5]	[High]
[Major]: Local variables should not shadow class fields [6]	[High]
[Minor]: Field names should comply with a naming convention [7]	[Medium]
[Minor]: Empty statements should be removed [9]	[Medium]
[Info]: Deprected Code should be removed [3]	[Medium]
[Minor]: Local variables should not be declared and then immediately returned or thrown [8]	[Medium]
[Minor]: Collection.isEmpty() should be used to test for emptiness [6]	[Medium]
[Minor]: Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" [2]	[Medium]

[Minor]: Catches should be combined [1]	[Medium]
[Major]: Source files should not have any duplicated blocks [3]	[Medium]
[Major]: Deprecated elements should have both the annotation and the Javadoc tag [2]	[Medium]
[Major]: Unused "private" methods should be removed [6]	[Low]
[Major]: "@Override" should be used on overriding and implementing methods [3]	[Low]
[Major]: "entrySet()" should be iterated when both the key and value are needed [9]	[Low]
[Major]: Sections of code should not be commented out [12]	[Low]
[Major]: Throwable and Error should not be caught [1]	[Low]
[Major]: Utility classes should not have public constructors [2]	[Low]
[Major]: Collapsible "if" statements should be merged [1]	[Low]
[Major]: Unused "private" fields should be removed [2]	[Low]
[Major]: Boolean expressions should not be gratuitous [4]	[Low]
[Major]: Only static class initializers should be used [13]	[Low]
[Minor]: Local variable and method parameter names should comply with a naming convention [1]	[Low]
[Critical]: Package declaration should match source file directory [28]	[Low]
[Minor]: Subclasses that add fields should override "equals" [2]	[Low]
[Minor]: The diamond operator (" \diamond ") should be used [2]	[Low]
[Minor]: Null checks should not be used with "instanceof" [1]	[Low]
[Minor]: Lambdas containing only one statement should not nest this statement in a block [1]	[Low]
[Minor]: Private fields only used as local variables in methods should become local variables [3]	[Low]
[Minor]: "throws" declarations should not be superfluous [3]	[Low]
[Minor]: Lambdas should be replaced with method references [4]	[Low]
[Minor]: Unnecessary imports should be removed [2]	[Low]
[Critical]: String literals should not be duplicated [256]	[Low]

Kenngrößen (Measures):

- **Complexity**

Die zyklomatische Komplexität (oder auch McCabe-Metrik genannt) gibt an, wie viele verschiedene Pfade es durch ein betrachtetes Projekt (in der gegebenen Granularitätsstufe) gibt.¹³ Die zyklomatische Komplexität (hier: insgesamt auf Projektebene 1207) sollte so klein wie möglich gehalten werden, da Projekte umso schwieriger verständlich werden, je komplexer sie sind.¹⁴ Außerdem sind komplexe Projekte anfälliger dafür, dass unbeabsichtigt Fehler in den Code eingebaut werden.¹⁵ Die kognitive Komplexität gibt an, wie verständlich der Code für Menschen ist, und wie er von Menschen geistig strukturiert werden würde.¹⁶ Im vorliegenden Projekt sieht man, dass die rein mathematische Kennzahl der zyklomatischen Komplexität über der von Menschen gesehenen Komplexität liegt. Es ist ratsam, die jeweilige Komplexität auch klassenspezifisch zu beurteilen, da diese innerhalb des Projekts variieren kann.

▼ Complexity ?	
Cyclomatic Complexity	1,207
Cognitive Complexity	535

- **Size**

SonarQube bietet die Möglichkeit die Größe der Applikation zu ermitteln, diese wird in den Lines of Code aufgelistet. Die Größe kann mit verschiedenen Parametern Statements/Function/Classes und Comments aufgeschlüsselt werden. Gesamt umfasst das Projekt 11207 Zeilen Code, wobei aber davon nur 8089 Zeilen tatsächlich Code sind. Kommentare umfassen ca. 500 Zeilen.

▼ Size	
Lines of Code	8,089
Lines	11,207
Statements	2,703
Functions	801
Classes	151
Files	150
Comment Lines	494

¹³ https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2010_2011/siw-1011-weging-code-qualitaet.pdf.

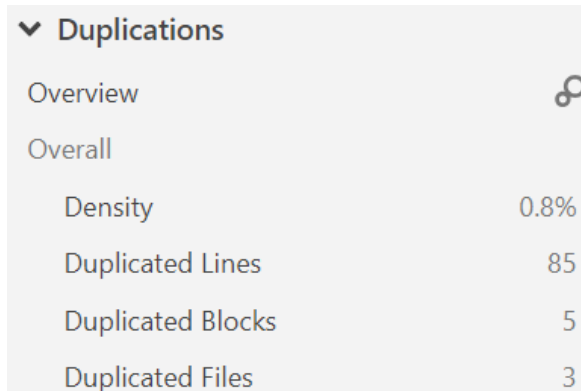
¹⁴ https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2010_2011/siw-1011-weging-code-qualitaet.pdf.

¹⁵ https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2010_2011/siw-1011-weging-code-qualitaet.pdf.


¹⁶ <https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html>.

- **Duplications:**

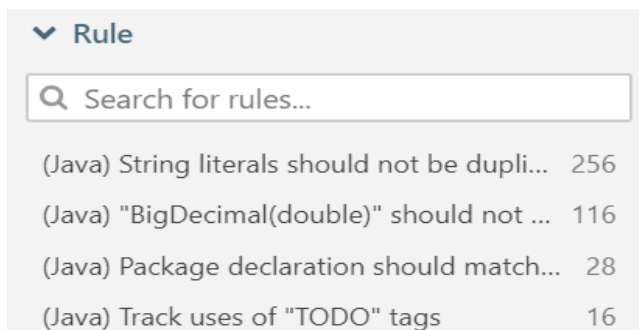
Die UIS Applikation weist nur 0.8% an Duplikationen auf. Dies macht im Projekt 85 Codezeilen aus, was relativ wenig ist. Natürlich sollte aber trotzdem daran gearbeitet werden, das Projekt insgesamt duplikationsfrei zu machen.



The screenshot shows the 'Duplications' overview in SonarQube. It lists the following statistics:

Duplications	
Overview	
Overall	
Density	0.8%
Duplicated Lines	85
Duplicated Blocks	5
Duplicated Files	3

Regeln, die am häufigsten verletzt werden:



The screenshot shows the 'Rule' section in SonarQube with a search bar and a list of rules:

Rule	
<input data-bbox="215 1019 813 1064" type="text" value="Search for rules..."/>	
(Java) String literals should not be dupli...	256
(Java) "BigDecimal(double)" should not ...	116
(Java) Package declaration should match...	28
(Java) Track uses of "TODO" tags	16

Dieser Screenshot zeigt die Regeln, welche am öftesten verletzt wurden. Die Regel “*String literals should not be duplicated*” wurde mit Abstand am öftesten verletzt, dies kann in Zukunft bei Refactoring und Anpassung Probleme bereiten, da jede Stelle angepasst werden muss. Die wohl wichtigste Regel ist jedoch “*BigDecimal(double) should not be used*” dies kann bei Berechnungen zu Rundungsfehlern führen. Laut Sonarqube würde die Behebung dieser beiden Regelverletzungen 7 Tage in Anspruch nehmen.

**Sind die am häufigsten verletzten Regeln angemessen?
Gibt es False Positives?**

Die Regel “*String literals should not be duplicated*” wird am häufigsten verletzt, dies tritt jedoch nur in der Testdaten (Initialisierung) auf. Die Regel bringt durchaus einen Mehrwert, sollte aber nicht auf diese Klasse angewendet werden, und ist daher eine Art False Positive, also ein „falscher Alarm“. Die *BigDecimal*-Regel ist sehr kritisch, da sie zu Berechnungsfehlern führen kann und sollte deshalb nicht deaktiviert werden.

Ein False Positive stellt die Regel “*Mutable fields should not be ‘public static’, (make this member protected)*“ dar, da diese irreführend sein kann. Nichtsdestotrotz sollte die Regel genauer betrachtet werden, um den vorschnellen Ausschluss zu hinterfragen.

Sollten einige von den Default-Regeln deaktiviert/ersetzt werden?

Die Default Regeln von SonarQube sind ein guter Startpunkt, um die Code Qualität enorm zu steigern. Jedoch sollten nicht davor zurückgeschreckt werden, neue Regeln hinzufügen, wenn diese vom Team abgesegnet wurden. Diese können bei Code Reviews gemeinsam erarbeitet werden, um möglichst einen einheitlichen Stil zu garantieren. Durch die Einbeziehung des Projektteams können die Regeln speziell für das genannte Projekt am besten angepasst werden, da diese das meiste Know-How mitbringen.

Mögliche zugrundeliegende Ursache für Qualitätsprobleme:

Das Projekt leidet an folgenden Qualitätsproblemen:

- Einige Sicherheitsverletzungen
- Code Smells, welche die Lesbarkeit stark beeinträchtigen
- Viele TODO-Kommentare die behoben werden müssen

Von den durch SonarQube angemerkten Problemen abgesehen liegen weitere Problemen vor:

- Schlampige Programmierung
- Viele Rechtschreibfehler/Fehler in der Benennung
- Schlechte Doku
- Keine Checkstyles/Linting

Empfohlene Änderungen für das Projektteam, um derartige Probleme in der Zukunft zu vermeiden:

Es wird eine Schulung des Projektteams angeraten, damit der Fokus auf Erkennen und Verbessern der Code-Qualität gelegt wird. Nach jeder Änderung des Codes sollte auf Code Smells und Bugs geachtet werden, da sich der Aufwand stark erhöht, wenn man erst nachdem das gesamte Projekt implementiert wurde, die Qualitätsreports betrachtet. Es sollte möglichst früh das Augenmerk auf Code-Qualität gelegt werden, und der Code sollte in kleinen teilen re-factored werden. Das Quality Gate sollte kontinuierlich nach oben gesetzt werden.

Identifikation von “Bad Quality Hotspots” in bestimmten Code-Modulen/Packages:

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Coverage
📁 backend/src/main/java/at/ac/tuwien/inso/sqm	3,818	130	6	331	20.1%
📁 initializer	1,564	126	0	267	0.0%
📁 repository	258	0	0	3	0.0%
📁 service	1,795	4	6	58	35.9%
📁 utils	9	0	0	2	0.0%
📁 validator	192	0	0	1	33.2%

Im Backend-Folder (bzw. Module) befinden sich die meisten Bugs und Code Smells, diese sollten umgehend minimiert werden. Daher kann das Backend in diesem Fall als ein Hotspot für schlechte Code-Qualität gesehen werden.

Mit Abstand die meisten Bugs befinden sich im Package `at.ac.tuwien.inso.sqm.initializer` in der `DataInitializer` Klasse im Backend. Insgesamt sind dies 126 von 136 Bugs, die dort aufgezeigt wurden. Auch annähernd die Hälfte der Code Smells befinden sich in dieser Klasse (i.e. 267 von 602 Code Smells). Einen anderen Hotspot stellt das Package `at.ac.tuwien.inso.sqm.service` dar, jedoch liegen die 58 Code Smells in diversen Klassen verteilt.

Beurteilung der Komplexität im vorliegenden Projekt und Vorschläge zur Verbesserung:

Die Komplexität der Initialisierung für die Testdaten sollte aufgebrochen werden, da das File aus über 1500 Codezeilen besteht. Daher können nur sehr schwierig unterschiedliche Testszenarien eingespielt werden.

Die Komplexität könnte auch durch den gezielten Einsatz von Generizität reduziert werden, um nicht jede Methode `findOne` für jeden Service wiederholt im Backend implementieren zu müssen.

Beurteilung der Code-Duplikation:

Der Kernpunkt von objektorientierten Sprachen ist die Vererbung, daher sollte man keine Code-Teile kopieren. Dieses Konzept wird als DRY (Don't repeat yourself) bezeichnet. Bei Anpassungen muss jeder Codeteil angepasst werden, bzw. muss jeder Teil auch wiedergefunden werden.

Beurteilung des Default Quality Gate:

Die Sonar-Voreinstellungen für das Java-Ruleset (Sonar-Way) definieren 379 Regeln, die eingehalten werden müssen. Sie bieten eine gute Grundlage, sie sollten aber auch nach einigen Zyklen angepasst werden, um den Bedürfnissen der Entwickler gerecht zu werden.

Die Testdaten-Initialisierung sollte von der Sonar-Analyse nicht überprüft werden, da sie im aktiven Betrieb nicht verwendet wird und damit keinen Mehrwert bringt.

Strategievorschlge an das Projektteam, um die Code-Qualitt langfristig zu steigern:

Es sollte ein Limit von 80 % fr die Testabdeckung definiert werden, dies sollte bei jedem Build berprft werden und gegebenenfalls der Build fehlschlagen.

Weiters sollte vereinbart werden, dass alle Code Smells und Bugs auf ein Mindestma reduziert werden. Es sollte regelmig eine statische Inspektion und Code-Qualittsanalyse durchgefhrt werden, um so frh wie mglich auf Code Smells, Bugs und hnliches aufmerksam zu werden und die Zeit fr das Refactoring gering zu halten. Auch sollte bei der Code-Duplikation eine Modularisierung/Abstraktion eingesetzt werden, um keinen redundanten Code verwalten zu mssen.

Es knnten auch Checkstyle und hnliche Tools eingesetzt werden, die direkt von der Entwicklungsumgebung untersttzt werden, um nicht auf den Jenkins-Build warten zu mssen. Der lokale Build am Rechner kann die Regeln besser und schneller berprfen.

2.1 Checkstyle Integration

Google Checks¹⁷ wurde als Ausgangspunkt für unsere eigene Checkstyle-Datei gewählt, weil diese Checks im Gegensatz zu den Sun Checks kontinuierlich angepasst werden.

Das Maven Checkstyle Plugin wird folgendermaßen in der `pom.xml` (des Gesamtprojekts) hinzugefügt:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>3.0.0</version>
  <configuration>
    <configLocation>checkstyle.xml</configLocation>
    <suppressionsLocation>checkstyle-
      suppressions.xml
    </suppressionsLocation>
    <encoding>UTF-8</encoding>
    <consoleOutput>true</consoleOutput>
    <failsOnError>true</failsOnError>
    <linkXRef>false</linkXRef>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>com.puppcrawl.tools</groupId>
      <artifactId>checkstyle</artifactId>
      <version>8.20</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Als Checkstyle-Version haben wir uns für die aktuellste Version 8.20¹⁸ (Dependency `com.puppcrawl.tools`) entschieden, welche am 28.04.2019 released wurde. Wir haben eine neue Checkstyle-Datei `checkstyle.xml` erstellt und dort den Inhalt der Google Checks¹⁹ als Ausgangspunkt genommen. Außerdem haben wir eine Datei `checkstyle-suppressions.xml` erstellt, in welcher angegebene Klassen von bestimmten Regelüberprüfungen ausgeschlossen werden können. Unserer Meinung nach soll der Build bei Checkstyle-Fehlern fehlschlagen, um eine nachhaltig hohe und einheitliche Codequalität zu gewährleisten. Hierfür haben wir in der Checkstyle-Datei die `severity` (d.h. den Schweregrad der Vorfälle) von `warning` auf `error` geändert.

¹⁷ Siehe: https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml.

¹⁸ http://checkstyle.sourceforge.net/releasenotes.html#Release_8.20

¹⁹ https://github.com/checkstyle/checkstyle/blob/checkstyle-8.20/src/main/resources/google_checks.xml

Es gibt eine Vielzahl an Checkstyle-Regeln²⁰. Insgesamt haben wir versucht, einen sinnvollen Trade-Off zwischen hoher Qualität und nicht zu schwerwiegenden Einschränkungen zu finden. Um dies zu erreichen, haben wir (vor dem ersten Checkstyle-Lauf) folgende *Anpassungen im Vergleich zu den ursprünglichen Google Checks* vorgenommen:

Checkstyle-Regel	Vorgenommene Aktion & Begründung
FileTabCharacter	Entfernt, weil das unserer Meinung nach die Lesbarkeit nicht beeinflusst.
LineLength	Von 100 auf 120 geändert, weil uns eine maximale Zeilenlänge von 100 etwas zu restriktiv erschien.
VariableDeclarationUsageDistance	Entfernt, weil das unserer Meinung nach die Lesbarkeit nicht beeinflusst.
CustomImportOrder	Entfernt, weil das unserer Meinung nach die Lesbarkeit nicht beeinflusst.
JavadocTagContinuationIndentation, SummaryJavadoc, JavadocParagraph, AtclauseOrder, SingleLineJavadoc	Entfernt, weil uns die Überprüfung von Regeln innerhalb von Javadoc-Blöcken zu restriktiv erschien.
CommentsIndentation	Entfernt, weil das unserer Meinung nach die Lesbarkeit nicht beeinflusst.

Nachdem diese Regelanpassungen gemacht wurden, hat das Build-Ergebnis für das Projekt sich wie im Folgenden beschrieben verändert. Der *erste Checkstyle-Lauf* führt zu 6.143 Regelverstößen:

Checkstyle-Regel	Anzahl von Verstößen
AnnotationLocation	1
LeftCurly	2
NeedBraces	106
RightCurly	15
OneStatementPerLine	1
AvoidStarImport	83
Indentation	4.404
JavadocMethod	152

²⁰ <http://checkstyle.sourceforge.net/checks.html>

NonEmptyAtclauseDescription	45
AbbreviationAsWordInName	7
LambdaParameterName	1
MemberName	7
PackageName	14
LineLength	339
EmptyLineSeparator	3
MethodParamPad	2
OperatorWrap	635
ParenPad	1
WhitespaceAround	325

Danach wurden die Checkstyle-Regeln verfeinert, um noch genauere Anpassungen an das vorliegende Projekt zu machen. Anhand des vorgefundenen Zustands wurden dann weitere Anpassungen gemacht, um den passenden Zustand zu finden. **Nach einer Analyse der Regelverletzungen** haben wir somit folgende **Anpassungen** vorgenommen:

Checkstyle-Regel	Aktion
RightCurly	Regel dahingehend angepasst, dass die rechte geschweifte Klammer (}) immer allein in einer Zeile stehen muss, da sich dadurch die Lesbarkeit erhöht.
Indentation	Properties <code>basicOffset</code> , <code>caseIndent</code> und <code>arrayInitIndent</code> von 2 auf 4 geändert, weil das Projekt durchgängig eine Einrückungstiefe von 4 verwendet.
JavadocMethod, NonEmptyAtclauseDescription	Entfernt, weil es unserer Meinung nach nicht das Ziel dieser Übung ist, fehlende Javadoc-Kommentare zu ergänzen. Ansonsten sind diese Regeln eigentlich wichtig und richtig.
AbbreviationAsWordInName	Property <code>allowedAbbreviationLength</code> von 1 auf 3 geändert, um valide Abkürzungen wie beispielsweise DTO zuzulassen.
PackageName	Property <code>format</code> auf <code>^[a-z]+(\.[a-z][a-z0-9_]*)*\$</code> geändert, um „_“ in Paketnamen zu erlauben, wie beispielsweise <code>at.ac.tuwien.inso.sqm.service.study_progress</code> in
OperatorWrap	Property <code>option</code> von <code>NL</code> (new line) auf <code>eol</code> (end of line) geändert, weil der Großteil der Regelverstöße darauf zurückzuführen ist, dass ein „+“ am Ende statt am Beginn der Zeile steht. Wichtig ist bei dieser Regel nicht die gewählte Option, sondern die Einheitlichkeit im gesamten Projekt.

LineLength	Von 120 auf 150 geändert, weil eine Zeilenlänge von 150 bei den heutigen Bildschirmen auch keine Probleme darstellen sollte.
------------	--

Durch die gemachten (und obenstehend beschriebenen) Anpassungen verringert sich die Anzahl der Regelverstöße auf 1.207:

Checkstyle-Regel	Anzahl von Verstößen
AnnotationLocation	1
LeftCurly	2
NeedBraces	106
RightCurly	35
OneStatementPerLine	1
AvoidStarImport	83
Indentation	506
LambdaParameterName	1
MemberName	7
LineLength	131
EmptyLineSeparator	3
MethodParamPad	2
OperatorWrap	3
ParenPad	1
WhitespaceAround	325

Außerdem haben wir in der `checkstyle-suppressions.xml`-Datei folgenden Eintrag hinzugefügt, um die Klasse `DataInitializer` von der Überprüfung der `LineLength`-Regel auszuschließen:

```
<suppressions>
  <suppress checks="LineLength" files="DataInitializer.java"/>
</suppressions>
```

Allein diese Klasse enthält 87 der insgesamt 131 Regelverstöße gegen die maximale Zeilenlänge von 150. Da es sich hier nur um eine Klasse zur Erstellung von Testdaten handelt, ist unserer Meinung nach hier eine Überprüfung der Zeilenlänge nicht unbedingt notwendig.

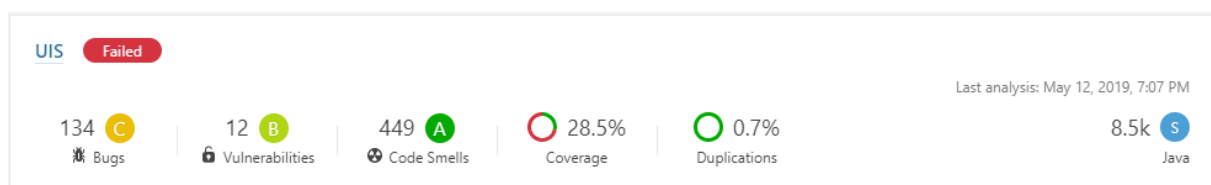
Zur Integration von Checkstyle in eine existierende Code-Basis:

Das nachträgliche Integrieren von Checkstyle in ein bestehendes Projekt führt zu dem Problem, dass je nach Fortschritt des Projekts bereits sehr viele Regelverstöße enthalten sein können. Es müssen dann zuerst alle bestehenden Verstöße behoben werden, bevor Checkstyle vollständig integriert werden kann.

Diese Probleme kann man dadurch umgehen, indem man die Regeln nicht alle auf einmal, sondern erst schrittweise aktiviert. Erst wenn das Projekt mit einer Regel erfolgreich durchbaut, sollte die nächste aktiviert werden. So kann man die Regelverstöße schrittweise und in geordneter Art und Weise abarbeiten und beheben.

Man sollte aber idealerweise Checkstyle schon von Projektbeginn an integrieren und so konfigurieren, dass der Build bei Regelverstößen fehlschlägt. So zwingt man alle Entwickler, die Regeln einzuhalten. Wenn der Build bei Regelverstößen nur eine Warnung ausgibt, kann das dazu führen, dass diese niemand wirklich ernst nimmt und sie somit mit der Zeit untergehen und in Vergessenheit geraten.

Zur Qualitätsverbesserung hinsichtlich des Projekts infolge der gemachten Änderungen:



Die Qualität des Projekts hat sich nicht nennenswert verbessert, wie aus dem Screenshot oben von Sonarqube ersichtlich ist. Das einzige, das sich verbessert hat, sind die Code Smells, welche sich von ursprünglich 463 auf 449 verringert haben, nachdem Checkstyle integriert wurde. Die fehlende Verbesserung der Qualität lässt sich dadurch erklären, dass die Integration von Checkstyle nur der Überprüfung und Vereinheitlichung des Programmierstils dient. Weitere Probleme sind dadurch im Großen und Ganzen in der Code-Basis nicht aufgekommen.

2.2 Development Guidelines

Die Integration von Entwicklungsrichtlinien hat einen großen Vorteil während der Entwicklung, es kann dadurch ein möglichst einheitlicher und lesbarer Code gewährleistet werden. Diese Regeln (bzw. Development Guidelines) können spezifische Vorgaben sein, die sich nicht vollständig oder gar nicht durch SonarQube/SonarLint überprüfen lassen. So können beispielsweise Vorgaben für Namensgebungen, Sprachwahl, Dokumentation (e.g. Javadoc), Logging und die korrekte Verwendung von Patterns und Organisation der Packages/Klassen/Interfaces und Ähnliches definiert werden.

Dokumentation:

Welche Gruppen von Klassen/Methoden sollten mit Javadoc ausgestattet sein?

Die Einführung von Dokumentation für Klassen und Methoden sollte nicht das vorrangige Ziel sein, da diese auch Instandhaltungskosten und Zeit verbraucht. Nach Möglichkeit sollten die Klassen und deren Methoden (insbesondere bei nicht-öffentlichen Code-Teilen) so geschnitten sein, dass durch die Namensgebung die Funktionsweise sofort ersichtlich ist. Jedoch sollte bei REST-Schnittstellen bzw. generell Schnittstellen zu anderen (externen) Applikationen eine Dokumentation der Methoden in einem standardisierten Format gewährleistet werden, da diese zumeist von anderen Projektteams (intern oder extern) auch benötigt werden.²¹ Für die Dokumentation der öffentlichen API des Codes eignet sich Javadoc sehr gut, wie es auch Robert C. Martin im Buch „Clean Code“ beschrieb.²² Für die öffentliche Schnittstelle sollten alle Klassen, Interfaces, Konstruktoren, Methoden und Felder, die als `public` oder `protected` gekennzeichnet sind, mit Javadoc dokumentiert werden.²³ Davon ausgenommen sind die Getter- und Setter-Methoden, deren Funktionsweise allgemein bekannt ist.²⁴ In nicht zur öffentlichen Schnittstelle zählenden Klassen/Methoden sollte Javadoc nicht eingesetzt werden, da diese Art von Dokumentation nur zusätzlichen Aufwand in Form von Arbeitsstunden und Kosten mit sich bringt, und somit vom eigentlichen Software-Engineering-Projekt „ablenkt“. ²⁵

Falls das Software-Engineering-Projekt, wie im vorliegenden Fall, aber schon weit vorangeschritten ist, und diese Empfehlungen nicht von Beginn an beachtet wurden, wäre es unmittelbar am wichtigsten, bei am häufigsten öffentlich benötigten Programmteilen und bei Exceptions nachzuschärfen zu beginnen, und danach kontinuierlich die Javadoc-Kommentare zu erweitern. Bei neu hinzukommenden Code-Teilen sollten die Javadoc-Kommentare – falls diese dafür notwendig sind – sofort miteingearbeitet werden.

²¹ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²² Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²³ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²⁴ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²⁵ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

Die Javadoc-Kommentare sollten außerdem an Stellen, an denen tatsächlicher Produktionscode geschrieben wurde, eingesetzt werden, und nicht in auskommentierter Form als „veraltete“ Zeilen im Projekt verbleiben. Robert C. Martin empfiehlt beispielsweise, bereits auskommentierte Zeilen, strikt zu löschen, da diese ohnehin nicht von großer Wichtigkeit sein können, sobald sie auskommentiert werden mussten, und neue Betrachter des Codes nur dazu verleitet werden, diese auskommentierten Zeilen ebenfalls stehen zu lassen. Ein schlechtes Beispiel hierfür wäre im vorliegenden Projekt in `FeedbackIService.java` (im Modul API):

```
//      /**
//      * Checks if the given student has already rated this course.
//      * @param feedback should not be null
//      * @return true, if the student has already given feedback for this
//      * course, else false
//      */
//      @PreAuthorize("isAuthenticated()")
//      boolean exists(Feedback feedback);
```

Hinsichtlich der Javadoc-Ausformulierung selbst enthält das folgende Beispiel zwar eine verständliche Beschreibung, ist aber voll mit Rechtschreibfehlern und der Javadoc-Parameter `@throws` wurde nicht miteinbezogen, obwohl die Methode eine Exception werfen kann.

```
/**
 * returns a list of lectuerers that are responsible for a subject and
 * match the search string. search startegy should be string.contains.
 *
 * may throw SubjectNotFoundException if no subject was found
 *
 * @param subjectId should not be null and not <1.
 * @param search will be changed to "" if is null.
 * @return a maximum of 10 lecturers
 */
@PreAuthorize("hasRole('ADMIN')")
List<LecturerEntity> getAvailableLecturersForSubject(Long subjectId, String
search);
```

Zwei Beispiele, in denen Javadoc richtig eingesetzt wurde, sind:

```
/**
 * gibt einen Tag anhand seines Namen zurück
 * kann nur verwendet werden, wenn ein Benutzer authentifiziert wird
 *
 * @param name
 * @return ein {@link Tag} Objekt
 */
@PreAuthorize("isAuthenticated()")
Tag findByName(String name);

/**
 *
 * @param id should not be null and not <1
 * @return UisUserEntity
 * @throws BusinessException
 */
```

```

    * returns an {@link UisUserEntity} with the provided id. if no user can be
    found
    * can only be used by ADMINS. if no user is found a {@link
    BusinessExceptionNotFoundException} will be thrown
    */
    @PreAuthorize("hasRole('ADMIN')")
    UisUserEntity findOne(long id) throws BusinessException;

```

Namensgebung (& Rechtschreibung):

Was ist die bevorzugte Projektsprache (für Kommentare, Variablen, Klassen)?

Im Projekt UIS ist die verwendete Sprache hauptsächlich Englisch für Klassen, Methoden und Kommentare. In einigen Fällen sind die Kommentare jedoch auf Deutsch verfasst, die Klassennamen und/oder Variablennamen sind auf Deutsch. Das Problem hierbei ist, dass es verwirrend wirkt für Mitglieder des Projektteams und Außenstehende, die z.B. kein Deutsch können, und somit einige Code-Teile und Kommentare nicht nahtlos verstehen können. Außerdem fällt es selbst Personen, die beide Sprachen können, nicht leicht, sich in dieser sprachlichen Mischung zu orientieren bzw. gezielt Programmteile zu suchen. Wir empfehlen daher, dass für das Projekt insgesamt Englisch als Projektsprache festgelegt wird, und alle Benennungen (sei es für Kommentare, Javadoc oder tatsächlichen Code) einheitlich auf Englisch angepasst werden. Denn dadurch kann der Code bzw. die API einem größeren Kreis an Personen zugänglich gemacht werden.

Zusätzlich zur Problematik der Sprachwahl haben sich sehr viele Rechtschreibfehler – sowohl auf Englisch als auch auf Deutsch – in das Projekt eingeschlichen, die bisher scheinbar noch nicht behoben wurden. Diese sollten allen voran behoben werden, da sonst Unklarheiten entstehen könnten, und es bezüglich Qualität und Genauigkeit keinen guten Eindruck auf einen Leser macht, wenn schon auf den ersten Blick auffällige Rechtschreibfehler im Projekt vorhanden sind.

Untenstehend befindet sich beispielhaft Code aus dem Modul API, in dem Englisch und Deutsch innerhalb einer Klasse vermischt wurden, und in dem sich weiters Rechtschreibfehler befinden:

```

/* returns a list of courses for a given subject and the current semester
 * user needs to be authenticated
 *
 * @param subject
 * @return
 */
@PreAuthorize("isAuthenticated()")
List<Lehrveranstaltung> findCoursesForSubjectAndCurrentSemester(Subject
subject);

/**
 * dismisses a course for a student (used as feedback for machine learning)
 * the user should be of role student
 *
 * @param student should not be null
 * @param courseId should not be null and not <1
 */
@PreAuthorize("hasRole('ROLE_STUDENT')")

```

```
void dismissCourse(StudentEntity student, Long courseId);

/**
 * Meldet einen Studenten von der Lehrveranstaltung ab.
 * Benutzer muss authentifiziert sein.
 * <p>
 * Kann eine BusinessException werfen, wenn die
 * Lehrveranstaltung nicht existiert.
 *
 * @param student           sollte nicht null sein
 * @param lehrveranstaltungsID sollte nicht null sein und nicht <1
 * @return die Lehrveranstaltung ohne den abgemeldeten Studenten
 */
@PreAuthorize("isAuthenticated()")
Lehrveranstaltung studentVonLehrveranstaltungAbmelden(StudentEntity
student, Long lehrveranstaltungsID);
```

Ein Beispiel für die Vermischung von Englisch und Deutsch bei der Benennung von Variablen und Klassen:

```
@Autowired
private Nachrichten messages;
```

Untenstehend ein Beispiel für auffällige Rechtschreibfehler bei der Benennung der Klassen im Modul domain.

```
entity
├── EtcsDistributionEntity
├── Feedback
├── Grade
├── LecturerEntity
├── Lehrveranstaltung
├── MarkEntity
├── PendingAcountActivation
├── Rolle
├── Semester
├── SemestreTypeEnum
├── StdudyPlanEntity
├── StudentEntity
├── StudyPlanRegistration
├── Subjct
├── SubjectForStudyPlanEntity
├── SubjectType
├── SubjectWithGrade
├── Tag
├── UisUserEntity
└── UserAccountEntity
```

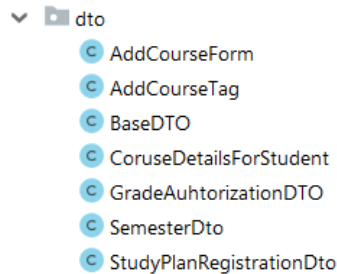
Namensgebung:

Welche einheitlichen Muster sollten für bestimmte Gruppen von Artefakten (Interfaces, DTO, IDs etc.) verwendet werden?

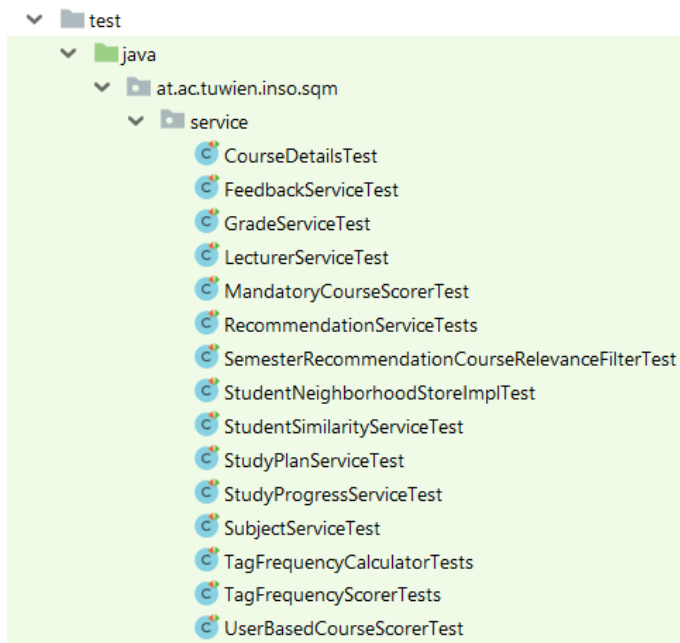
In Bezug auf das obige Beispiel des Package `entity` ist an dieser Stelle nicht klar, warum einige Klassen mit dem Anhang `Entity` benannt wurden, und andere wiederum nicht. Wir

konnten dahinter kein plausibles Muster erkennen. Es sollte eigentlich einheitlich gehandhabt werden, und entweder nie der Anhang verwendet werden oder immer – aber auf keinen Fall eine Mischung, da diese nur zu steigender Verwirrung führt.

Im Package `dto` findet sich diese Problematik ebenfalls wieder, da einige Klassen mit dem Anhang `DTO` bzw. `Dto` geschrieben wurden, andere aber nicht. Wir empfehlen hier, einheitlich das Suffix `DTO` zu verwenden.



Ebenfalls unklar war die Wahl der Benennung in den `test` Packages (wie der untenstehende Screenshot zeigt), da einige Test-Klassen mit dem Suffix `Test`, und andere wieder mit `Tests` geschrieben wurden. Wir empfehlen, einheitlich zu bleiben und das Suffix `Test` zu nehmen.



Im Modul API wurden die Bezeichnungen für Interfaces nicht einheitlich gehandhabt. Wir empfehlen, dieses Muster (z.B. `LecturerService`, `UserAccountService`) für Interfaces zu verwenden, anstatt verschiedene Muster wie etwa `FeedbackIService`, `LehrveranstaltungServiceInterface` und `LecturerService` zu mischen, weswegen aus der Namensgebung nicht mehr klar ist, welches davon tatsächlich ein Interface ist und welches nicht. Für die Klassen, die die Interfaces implementieren, sollte der Suffix `Impl` an den Interface-Namen angehängt werden, z.B. `LecturerServiceImpl` für die Implementierungsklasse zum Interface `LecturerService`.

In Bezug auf allgemein verwendete Variablennamen wie zum Beispiel für Identifier sollte sich das Projektteam auf eine Version einigen, die dann durch das Projekt hindurch und über

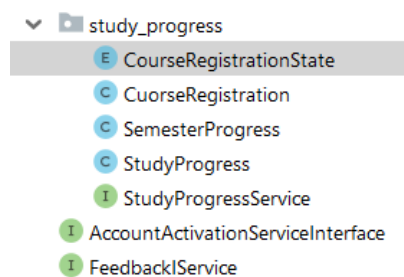
alle Projektmitglieder hinweg verwendet wird. Im Fall von IDs kommt diese auch in der Version `Id` und `ID` vor. Dies sollte vereinheitlicht werden, indem man am Anfang des Projekts idealerweise eine gemeinsame Notation für derartig allgemeine Ausdrücke definiert. Im vorliegenden Projekt ist dies nur nach und nach möglich, da es schon weiter vorangeschritten ist, sollte aber trotzdem noch so gut wie möglich umgesetzt werden.

Öffentliche Konstanten sollten einheitlich vollkommen mit Großbuchstaben geschrieben werden, und falls die Konstante aus mehreren Wörtern bestehen, sollten diese mit „_“ verbunden werden. Die Konventionen für öffentliche Konstanten wurden im Projekt bisher gut eingehalten.

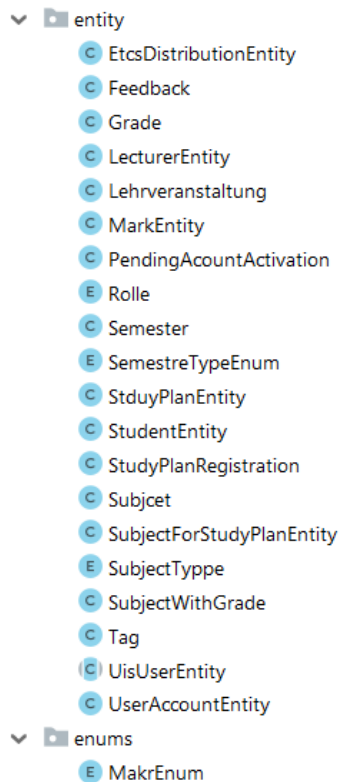
Im Bereich der Enums ist es in einigen Packages nicht klar, dass der Enum an jener Stelle zu finden ist. Ein Beispiel wäre dafür (im Modul `API`):

```
package at.ac.tuwien.inso.sqm.service.study_progress; //FIXME package
naming convention?!

public enum CourseRegistrationState {
    in_progress, needs_feedback, needs_grade, complete_ok, complete_not_ok
}
```



Im Gegensatz dazu wurde im Modul `domain` ein eigenes Package nur für Enums angelegt, in dem sich nur ein Enum befindet. Im danebenliegenden Package sind allerdings wieder Klassen mit Enums vermischt worden und es wurde kein eigenes Package dafür angelegt, wie es untenstehend im Screenshot zu sehen ist. Somit empfehlen wir, eine klare Einteilung zu treffen, und auch die Namensgebung und Platzierung der Enums zu überdenken, sodass keine organisatorischen Ungereimtheiten auftreten.



Namensgebung:

Haben die Klassen/Methoden/Variablen/Parameter Namen, die ihre tatsächliche Bedeutung widerspiegeln?

Hinsichtlich der Semantik sollte für Variablen und Klassen ein Name gegeben werden, der ihre tatsächliche Bedeutung bestmöglich widerspiegelt.²⁶ Es sollte bereits aus dem Name klar hervorgehen, was der Zweck der betrachteten Klasse bzw. Variable ist. Sofern diese nicht allgemein bekannt sind, sollten dafür keine Abkürzungen verwendet werden.

Die Groß- bzw. Kleinschreibung von Variablen-, Interface- und Klassennamen sollten einheitlich gehandhabt werden.²⁷ Variablennamen sollten mit einem Kleinbuchstabe beginnen, und bei vorhandenen weiteren Teilwörter mit einem Großbuchstabe beginnen.²⁸ Klassen- und Interfacenamen sollten mit einem Großbuchstabe beginnen.²⁹ Interfaces empfehlen wir nach dem oben angeführten Schema einheitlich zu benennen.

Gute Beispiele für die Namensgebung:

- Gut, weil Variablenname klar:

```
@Autowired
private UserAccountService userAccountService;
```

²⁶ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²⁷ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²⁸ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

²⁹ Robert C. Martin (2009): Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code. Übersetzt aus dem Amerikanischen von Reinhard Engel. Mitp Verlags GmbH.

```
@Autowired
private LecturerRepository lecturerRepository;
```

- Gut, weil die Parameternamen klar sind:
SemestreTypeEnum(String name, **int** startMonth, **int** startDay)

Schlechte Beispiele für die Namensgebung:

- Schlecht, weil zu viele und unkenntliche Abkürzungen verwendet werden:

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("lang");
    return lci;
}
```

- Schlecht, weil die Bedeutung des Parameternamens unklar ist:

```
public boolean equals(AddCourseTag o)
```

Logging:

Welche Log-Levels gibt es? Wann sollten sie angewandt werden?

Im untersuchten Projekt werden die Application-Logging-Frameworks SLF4J und Log4J verwendet. Diese verwenden bestimmte Log-Levels, um Informationen, die durch die Anwendung generiert werden, nach ihrer Dringlichkeit zu kategorisieren und organisieren.³⁰ Dies ermöglicht ein Filtern der Log-Events.

Generell umfasst Logging folgende Levels³¹ für verschiedene Granularitätsstufen:³²

Log-Level	Beschreibung
ALL	Alle Log-Levels (inkl. „custom“)
DEBUG	Für detaillierte Informationsevents, die für diagnostische Zwecke nützlich sind (aus Developer-Sicht)
INFO	Informative Nachrichten, die eher auf weniger detailliertem Niveau das normale Verhalten der Anwendung zeigen. Sie bieten eine Art Anhaltspunkt über die Abläufe in der Anwendung
WARN	Möglicherweise fehlerhafte Situationen (z.B. Verwendung von veralteter („deprecated“))

³⁰ <https://dzone.com/articles/logging-levels-what-they-are-and-how-they-help-you>

³¹ Anmerkung: zusätzlich können individuelle („custom“) Levels definiert werden.

³² Anmerkung: Quellen beziehen sich auch auf die Tabelle:

https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm; <https://dzone.com/articles/logging-levels-what-they-are-and-how-they-help-you>; <https://stackoverflow.com/questions/5817738/how-to-use-log-levels-in-java>; <https://wiki.base22.com/btg/java-logging-standards-and-guidelines-2361.html>

	Syntax)
ERROR	Fehlerhafte Situationen, in denen die Anwendung aber eventuell weiterlaufen kann (z.B. eine nicht verfügbare Datenbankverbindung oder kein Zugriff auf eine angefragte Datei)
FATAL	Schwerwiegende Fehlersituationen, die (wahrscheinlich) zum Abbruch der Anwendung führen
OFF	Dient zum Ausschalten des Logging – es wird also gar nichts geloggt
TRACE	Detailliertere Informationsevents als bei DEBUG – kann zu Performanceeinbußen führen, da TRACE sehr viele Ressourcen benötigt

Die rangbezogene Reihenfolge der Log-Levels ist vom geringsten bis zum höchsten Rang nach ihrer Dringlichkeit:

ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.³³

Log4J bietet die Level FATAL, ERROR, WARN, INFO, DEBUG und TRACE an, wovon TRACE und FATAL aber im vorliegenden Projekt nicht verwendet werden. Von den verbleibenden Log-Level ist das meist verwendete im Projekt INFO. SLF4J bietet die in Log4J vorhandenen Level außer FATAL an.³⁴

Beispiele aus dem vorliegenden Projekt:

Log-Level INFO:

In diesem Beispiel werden informative Nachrichten geloggt, die den gewöhnlichen Programmablauf mitschreiben. Für diesen Zweck ist das Log-Level INFO ausreichend.

```
public String get(String path) {
    log.info("Nachrichten für den Pfad " + path + " erhalten.");
    return messageSource.getMessage(path, null, LOCALE);
}

public String msg(String path, Object... args) {
    log.info("Nachrichten für Objekt und Pfad " + path + " erhalten.");
    return messageSource.getMessage(path, args,
        LocaleContextHolder.getLocale());
}
```

Log-Level ERROR:

Hier wird die Meldung auf dem Level ERROR geloggt, da bei Eintreten des Ereignisses ein schwerer Fehler passiert, und die ganze UIS-Anwendung sogar komplett abstürzen könnte.

³³ https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm

³⁴ Anmerkung: Quelle bezieht sich auf den gesamten Absatz:
<https://stackoverflow.com/questions/5817738/how-to-use-log-levels-in-java>

```

private static final Logger logger =
    LoggerFactory.getLogger(UisApplication.class);

public static void main(String[] args) {
    try {
        SpringApplication.run(UisApplication.class, args);
    }
    catch (Error e) {
        logger.error("An error occurred in the application", e);
        throw e;
    }
}

```

Logging:

Werden die wichtigen Aktivitäten im Projekt angemessen geloggt? Ist Logging ausreichend implementiert, sodass Anwendungsfehler ("Faults") zurückverfolgt werden können?

Wie im obigen Beispiel angesprochen, zeigte sich, dass an den betrachteten Stellen die Log-Levels unserer Meinung nach in adäquater Art und Weise angewandt wurden. Zur Dokumentation und zum Mitschreiben des normalen Programmverlaufs wurde das Log-Level INFO genutzt. Dies ist inhaltlich angemessen, und wird auch an weiteren Stellen im Code korrekt angewandt. Allerdings gibt es sehr viele Stellen, an denen INFO geloggt wird. Es kann sein, dass es bei einer sehr großen Anzahl an Logging-Aktivitäten zu Performance-Einbußen kommt. Im vorliegenden Projekt müsste man dies für verschiedene Anzahlen von Log-Schritten auf Level INFO noch vergleichen, um darüber genau urteilen zu können.

Wie auch das obige Beispiel zeigte, wurde auch das Log-Level ERROR angemessen verwendet. Es wird tatsächlich im Projekt nur dort eingesetzt, wo schwerwiegende Fehler auftreten, die das Programm zum Absturz bringen könnten.

Das untenstehende Beispiel veranschaulicht die Verwendung des Log-Levels WARN. Wir beurteilen die Verwendung im Projekt generell als angemessen, da das Level WARN für Situationen verwendet wurde, in denen Zustände entstehen, die möglicherweise fehlerhaft sind, aber noch nicht direkt Fehler sind oder zum Absturz führen können. Im vorliegenden Beispiel ist dies beispielweise der Zustand, dass der Account nicht aktiviert werden konnte, da kein Aktivierungscode gefunden werden konnte.

```

private static final Logger log =
    LoggerFactory.getLogger(GlobalExceptionHandler.class);

@Autowired
private AccountActivationServiceInterface accountActivationService;

@GetMapping
public String accountActivationView(@PathVariable String activationCode,
                                    Model model,
                                    AccountActivationForm
    accountActivationForm) {

    try {
        UisUserEntity user =
            accountActivationService.findOne(activationCode).getForUser();
        model.addAttribute("user", user);
    }
}

```

```
    }  
    catch (BusinessObjectNotFoundException ex) {  
        log.warn("Account activation failed", ex);  
        throw new UserFacingException("error.activation_code.notfound");  
    }  
  
    return "account-activation";  
}
```

Auch im Bereich des Exception-Handlings wurde vieles auf dem Level WARN geloggt, wie zum Beispiel bei Vorliegen von unpassenden Typen („type mismatches“), in deren Falle eine `TypeMismatchException` geworfen wird.

```
@ExceptionHandler(TypeMismatchException.class)  
@ResponseStatus(value = HttpStatus.BAD_REQUEST)  
public ModelAndView handleTypeMismatchExceptions(HttpServletRequest request,  
    TypeMismatchException ex) {  
    logger.warn("TypeMismatchRequest: " + request.getRequestURL(), ex);  
    ModelAndView mav = new ModelAndView();  
    mav.setViewName("error");  
    return mav;  
}
```

Des Weiteren wurde das Level DEBUG verwendet, was aber eher selten vorkam im Vergleich zum Level INFO. Ein Beispiel dafür wäre die Implementierung des Hinzufüge-, Änderungs- und Löschvorgangs der studierendenspezifischen Fachpräferenzen. Für diesen Vorgang wurde DEBUG wahrscheinlich ausgewählt, um für die Developer eine genaue Verfolgung der Aktivitäten mitzuschreiben, um sicherzustellen, dass an dieser kritischen Stelle alle Vorgänge richtig durchgeführt werden, und potenzielle Fehler leichter identifizieren zu können. Der untenstehende beispielhafte Code bezieht sich auf den Vorgang des Hinzufügens und Speicherns:

```
@Override  
public void studentRegisteredCourse(StudentEntity student,  
    Lehrveranstaltung course) {  
    StudentSubjectPreference preference = new  
        StudentSubjectPreference(student.getId(), course.getSubject().getId(),  
        REGISTER_PREF_VALUE);  
  
    log.debug("Storing student subject preference due to course  
registration: " + preference);  
  
    preferenceRepository.insert(preference);  
}
```

Development Guidelines für das Projektteam

Zusammenfassend werden basierend auf der obigen Ausarbeitung für das Projektteam folgende Guidelines empfohlen:

- ***Verwendung von Javadoc:***

Javadoc-Kommentare sollten die öffentlichen Schnittstellen/API (Klassen, Methoden, Interfaces) des Projekts dokumentieren, und in einer standardisierten Form geschrieben werden. Es sollten `@param`, `@throws`, `@return` einheitlich verwendet werden und die Javadoc sollte auf Englisch verfasst werden.

- ***Umgang mit auskommentiertem Code:***

An Stellen mit auskommentiertem Code sollte hinterfragt werden, ob dieser überhaupt noch notwendig ist, und dieser idealerweise entweder tatsächlich in das Projekt wieder eingebaut werden oder sonst ganz fallen gelassen werden.

- ***Rechtschreibung:***

Die Rechtschreibung ist zwar nur ein beiläufiger Faktor. Kommentare und die Namen von Variablen/Interfaces/Klassen sollten aber dennoch unbedingt Korrektur gelesen werden. Es kann sonst sein, dass die Suche von bestimmten Programmteilen aufgrund der Rechtschreibfehler erschwert wird, oder Außenstehende aufgrund der Rechtschreibfehler auf eine schlechte Qualität des Projekts schließen würden.

- ***Wahl der Sprache für Programmierung (Variablen, Interfaces, Klassen, Enums, Methoden, Parameter) und Dokumentation (Javadoc, Kommentare):***

Die Sprache für Programmierung sollte einheitlich gehalten werden. Entweder man entscheidet sich für Englisch oder für Deutsch im vorliegenden Fall – aber von einer Mischung ist abzuraten. Wir empfehlen generell, Englisch zu verwenden, da es dann leichter ist, international Projektmitglieder und Stakeholder ohne großen zusätzlichen Übersetzungsaufwand in das Projekt miteinzubeziehen.

- ***Namensgebung:***

Die Namensgebung aller im Projekt verwendeten Bestandteile sollte ihre tatsächliche Bedeutung im Projekt klar darstellen. Es sollten selbsterklärende Namen verwendet werden. Außerdem sollten die oben genannten Konventionen für Variablen-/Klassen-/Interfacenamen eingehalten werden, um ein einheitliches Bild des Codes zu gewährleisten.

- ***Logging:***

Je nach angestrebtem Zweck sollte das passende Log-Level verwendet werden:

- ***DEBUG:*** für detaillierte Informationen zum Zweck der Diagnose (aus Sicht der Developer)
- ***INFO:*** für detaillierte Informationen, die zum Nachvollziehen des (gewöhnlichen) Programmverlaufs dienen
- ***WARN:*** für möglicherweise fehlerhafte Situationen, die aber noch nicht unmittelbar zu einem Versagen der Anwendung führen
- ***ERROR:*** für fehlerhafte Situationen, in denen die Anwendung aber eventuell weiterlaufen kann

- *FATAL*: für schwerwiegende Fehlersituationen, die (wahrscheinlich) zum Abbruch/Versagen der Anwendung führen
- *TRACE*: für noch detailliertere Informationen als bei *DEBUG*, um gezielt Programmabschnitte nachzuvollziehen – dies kann aber zu Performanceeinbußen führen, da für *TRACE* ein hoher Ressourcenaufwand notwendig ist! Daher sollte mit *TRACE* sparsam umgegangen werden und/oder dies nur zeitweise kurzfristig verwendet werden.

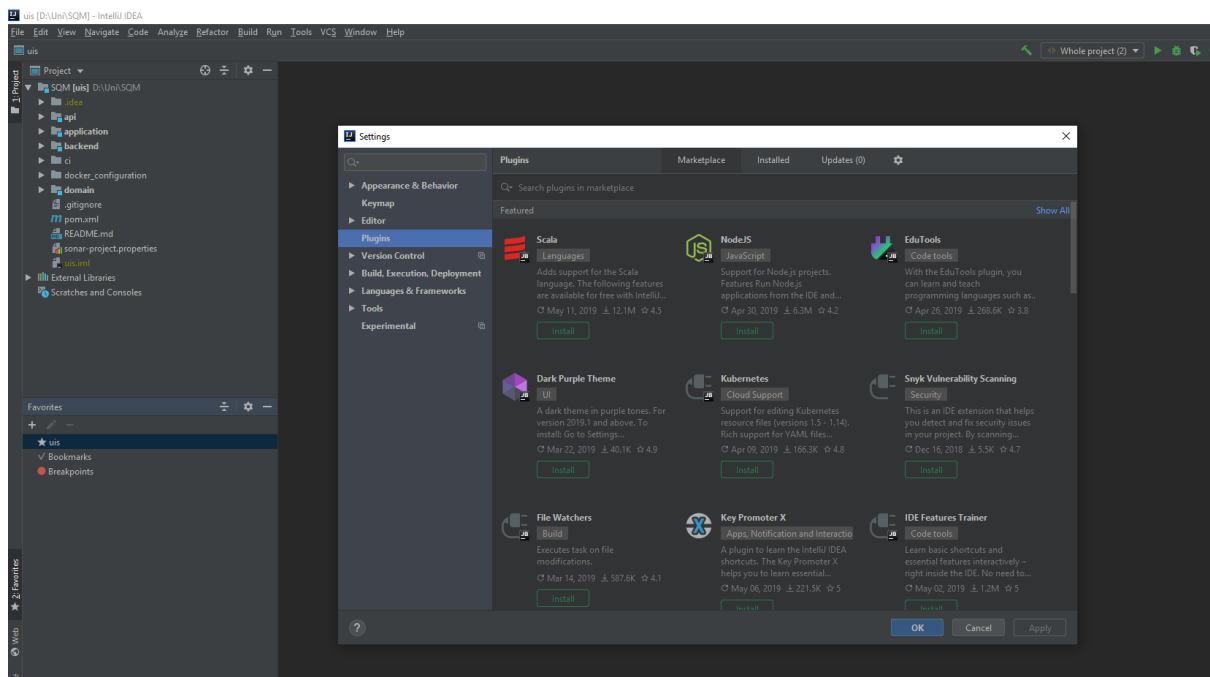
2.3 SonarLint Integration

Im folgenden Abschnitt wird dargestellt, wie SonarLint in die Entwicklungsumgebung IntelliJ eingebunden werden kann. Dies ermöglicht es, eine statische Code-Analyse durchzuführen und unterstützt die Entwickler dabei, da die Issues direkt in der IDE angezeigt werden. Dadurch wird ein einheitlicher Code-Stil unterstützt. Die Qualitätsregeln können dabei direkt aus dem SonarQube-Server gelesen werden.

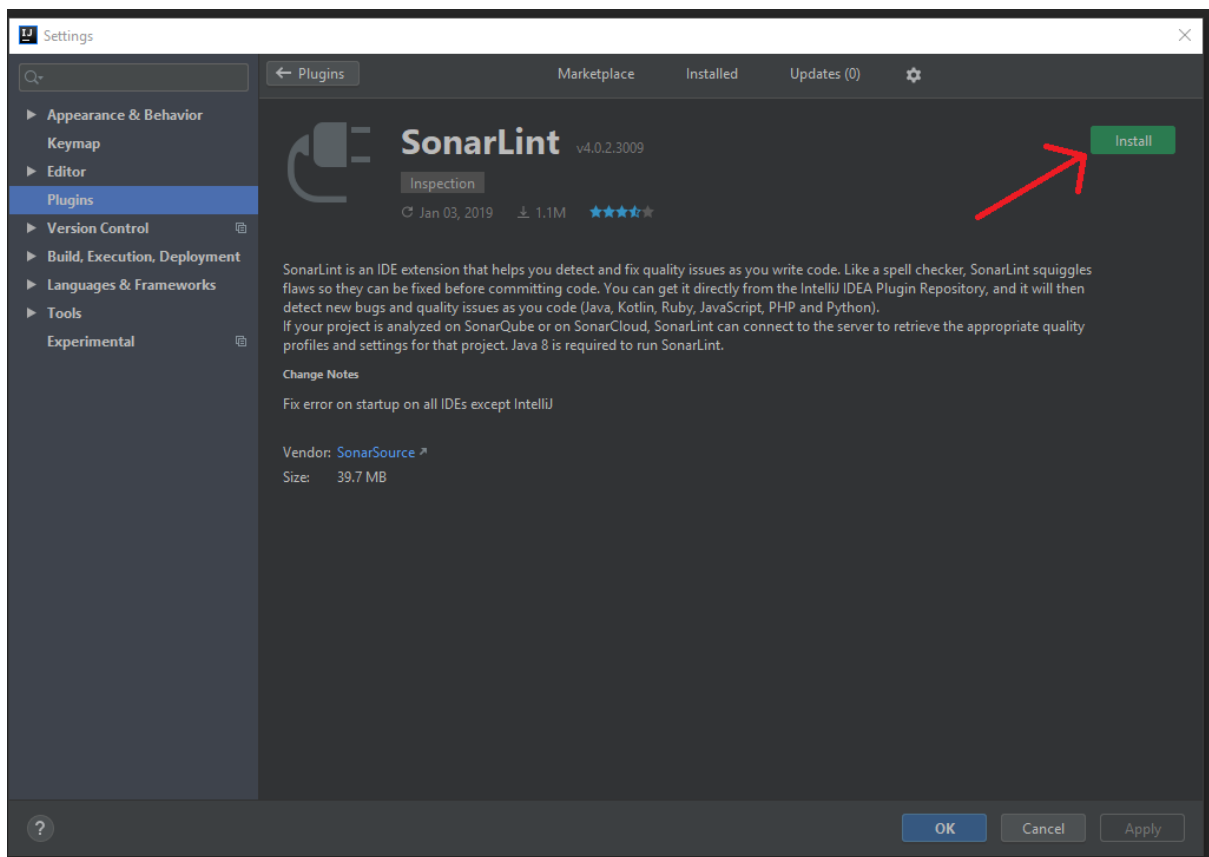
2.3.1 SonarLint Integration IntelliJ

Um SonarLint für IntelliJ zu verwenden, müssen folgende Schritte durchgeführt werden:

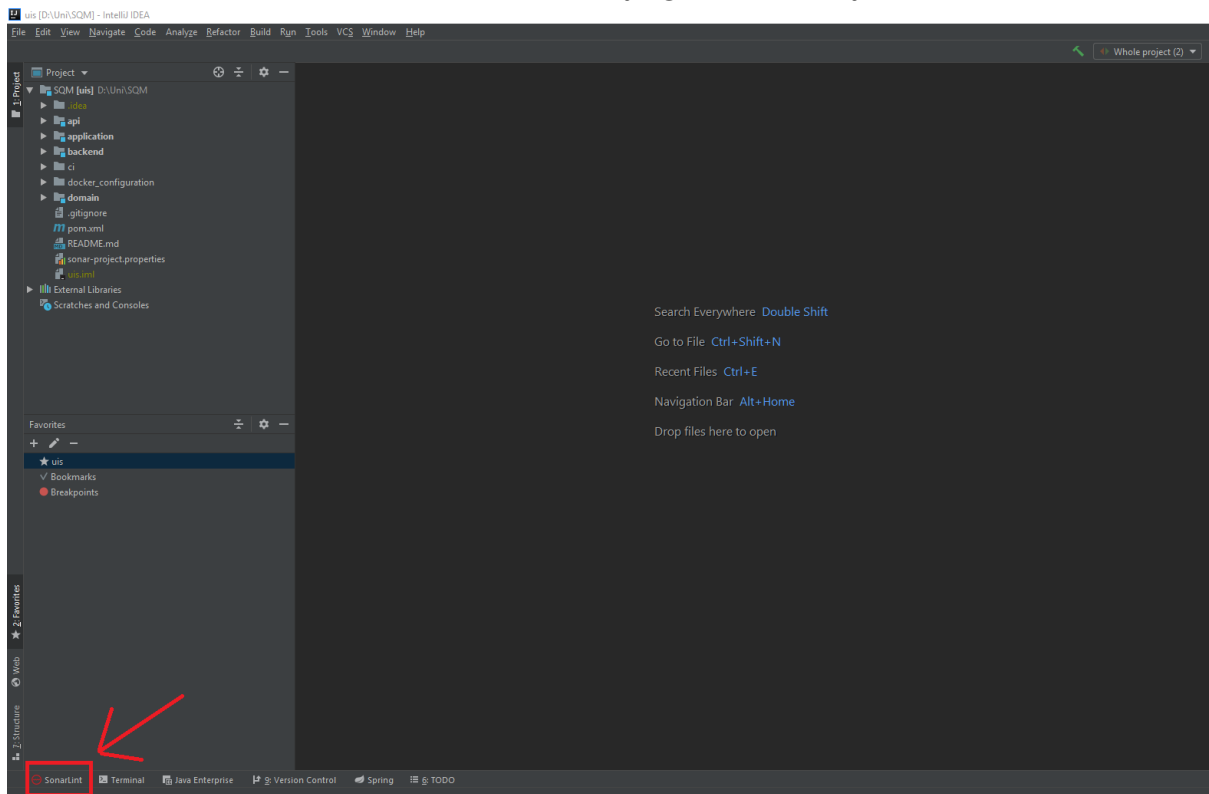
1) *Plugin-Einstellungen aufrufen (mittels Ctrl+Alt+S):*



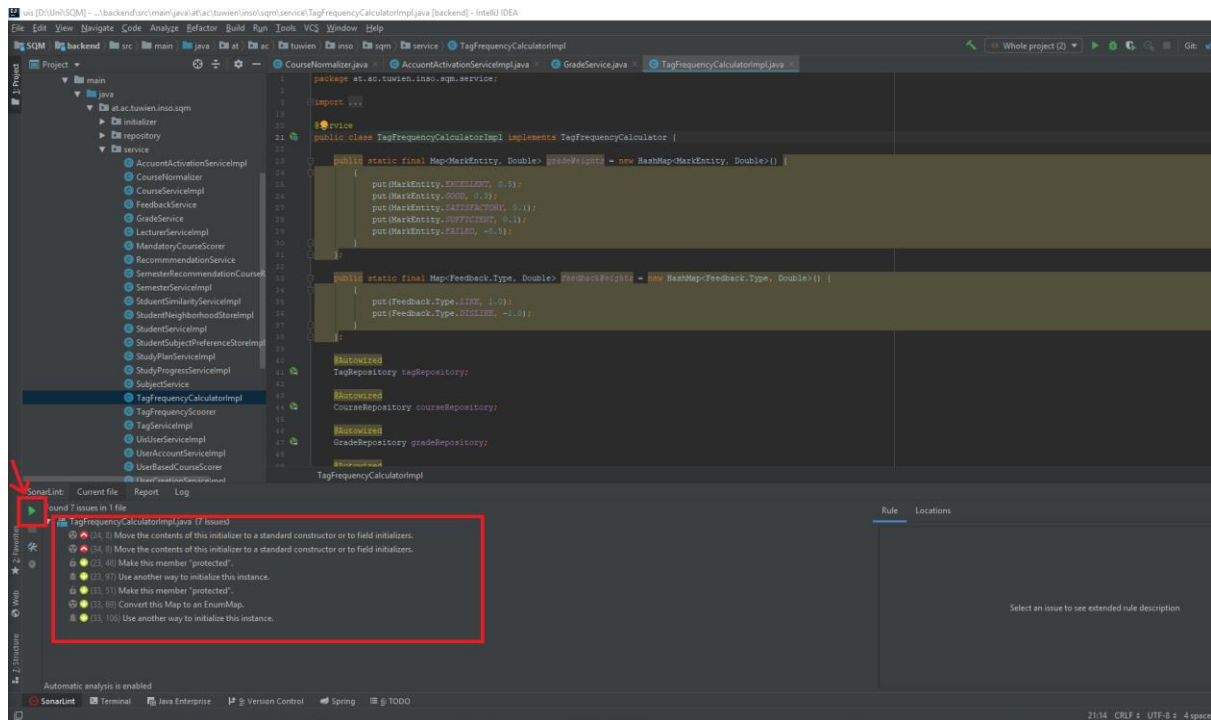
2) Suche nach „SonarLint“ und auf den Button „Install“ klicken:



3) Nach einem Neustart von IntelliJ sollte folgender Tab aufscheinen:



Die Analyse einer Datei kann anschließend folgendermaßen durchgeführt werden: SonarLint-Tab öffnen -> gewünschte Datei öffnen -> Analyse mittels Klick auf „Play“-Button starten:



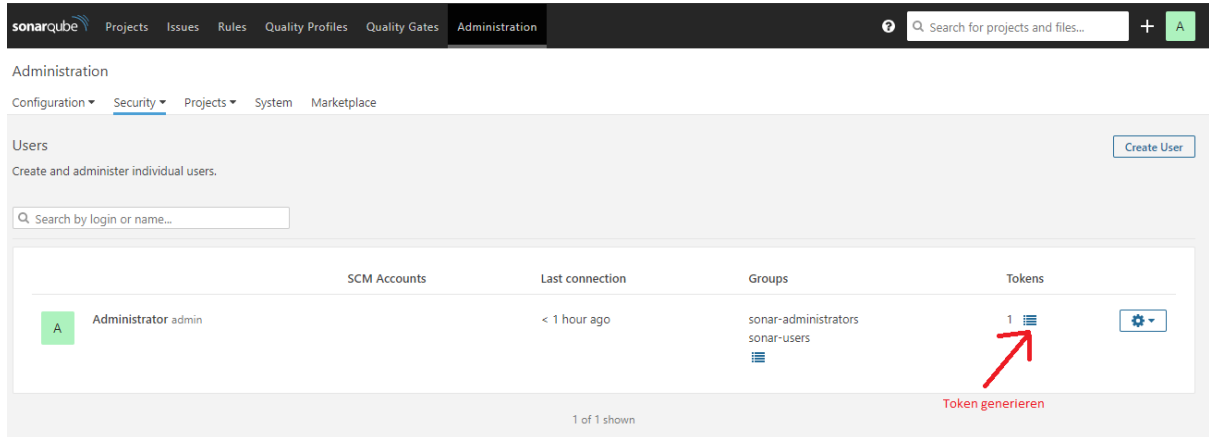
Die dabei gefundenen Issues werden dann wie in SonarQube aufgelistet. Bei Doppelklick auf einen Issue kann die verletzte Regel eingesehen werden.

2.3.2 SonarLint Integration SonarQube

Im ersten Schritt wurde die SonarLint-Erweiterung in IntelliJ hinzugefügt. Jedoch werden die Regeln nicht zentral gespeichert. Um eine konsistente Qualität zu gewährleisten, kann IntelliJ mit dem SonarQube-Server verbunden werden, um die Regeln für alle Entwickler zentral zur Verfügung zu stellen.

SonarQube-Einstellungen:

1) Anmelden in SonarQube -> Administration - Security - Users:



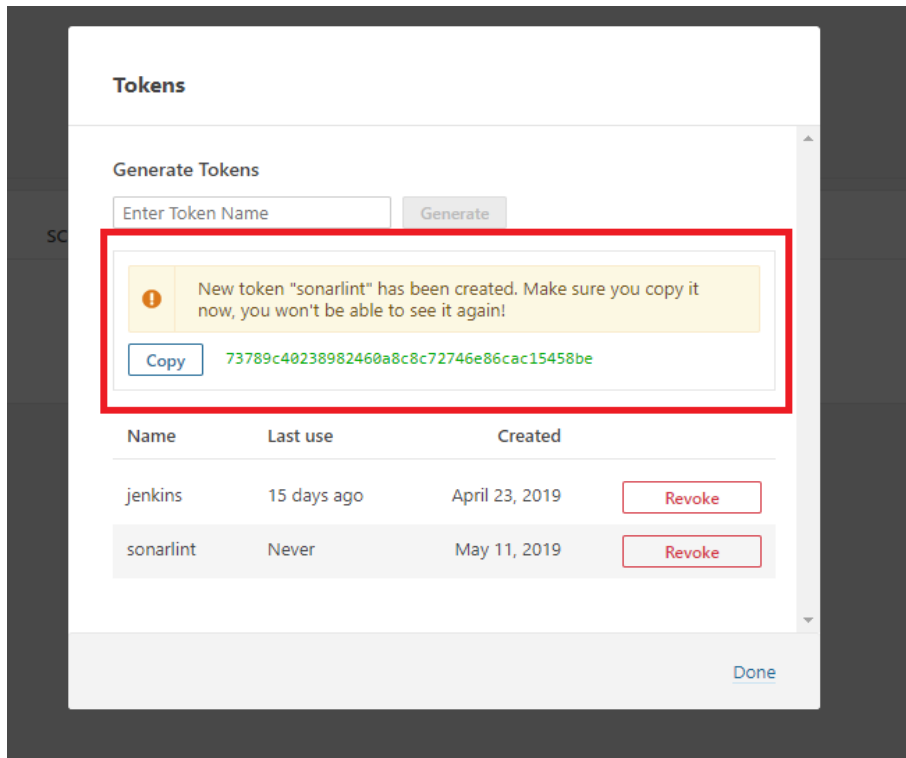
The screenshot shows the SonarQube Administration interface. The 'Users' section is active, displaying a table of users. The 'Administrator admin' user is listed. A red arrow points to the 'Tokens' column for this user, which contains a '1' and a menu icon. Below the table, the text 'Token generieren' is visible.

SCM Accounts	Last connection	Groups	Tokens
A Administrator admin	< 1 hour ago	sonar-administrators sonar-users	1 [Menu Icon]

1 of 1 shown

Token generieren

2) Token generieren - Tokentext kopieren, da dieser später benötigt wird:



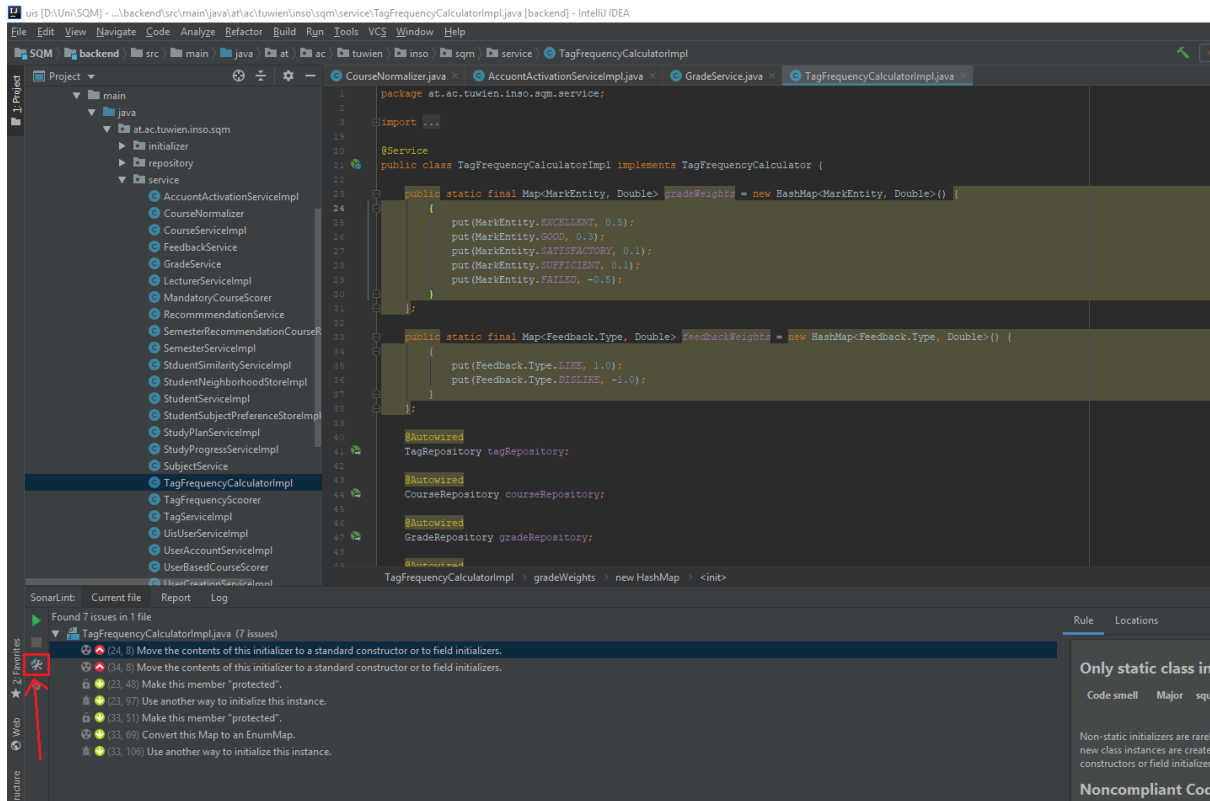
The screenshot shows the 'Tokens' page in SonarQube. A red box highlights a notification message: 'New token "sonarlint" has been created. Make sure you copy it now, you won't be able to see it again!'. Below the message is a 'Copy' button and the token value: 73789c40238982460a8c8c72746e86cac15458be. Below this is a table of existing tokens.

Name	Last use	Created	Revoke
jenkins	15 days ago	April 23, 2019	Revoke
sonarlint	Never	May 11, 2019	Revoke

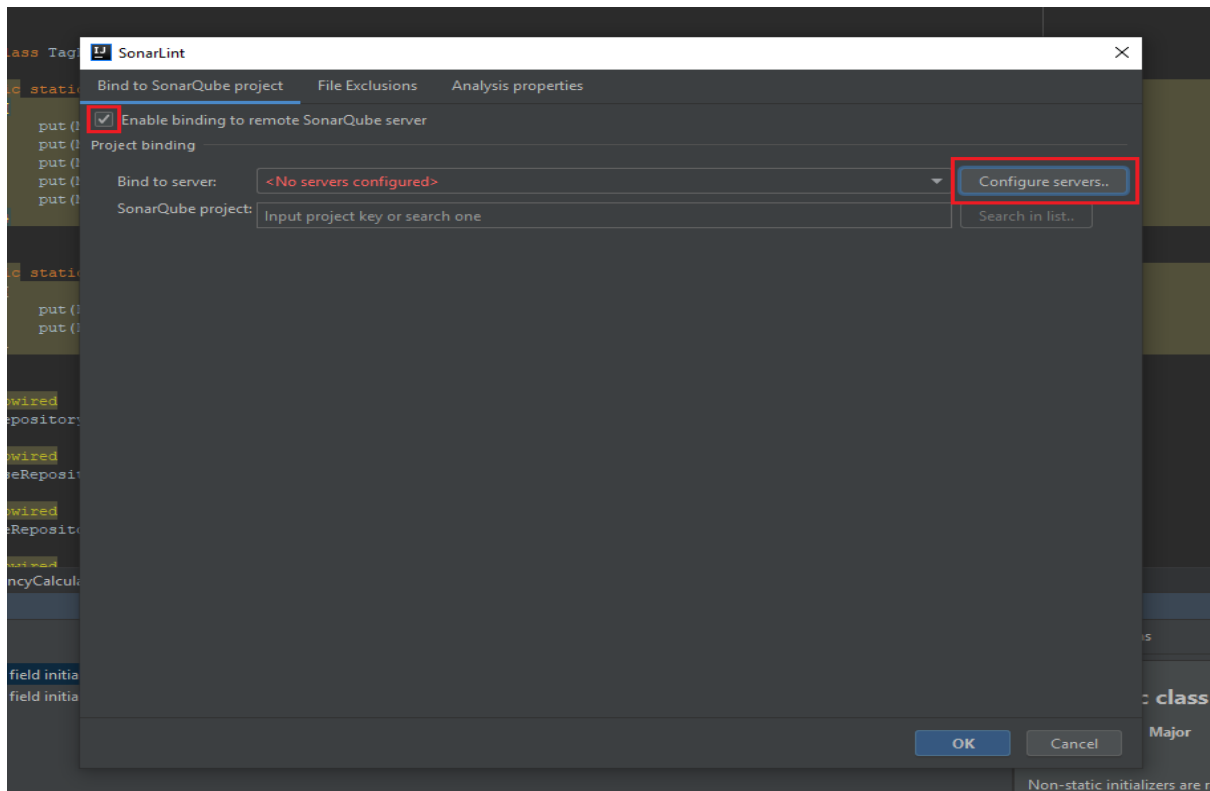
Done

IntelliJ-Einstellungen:

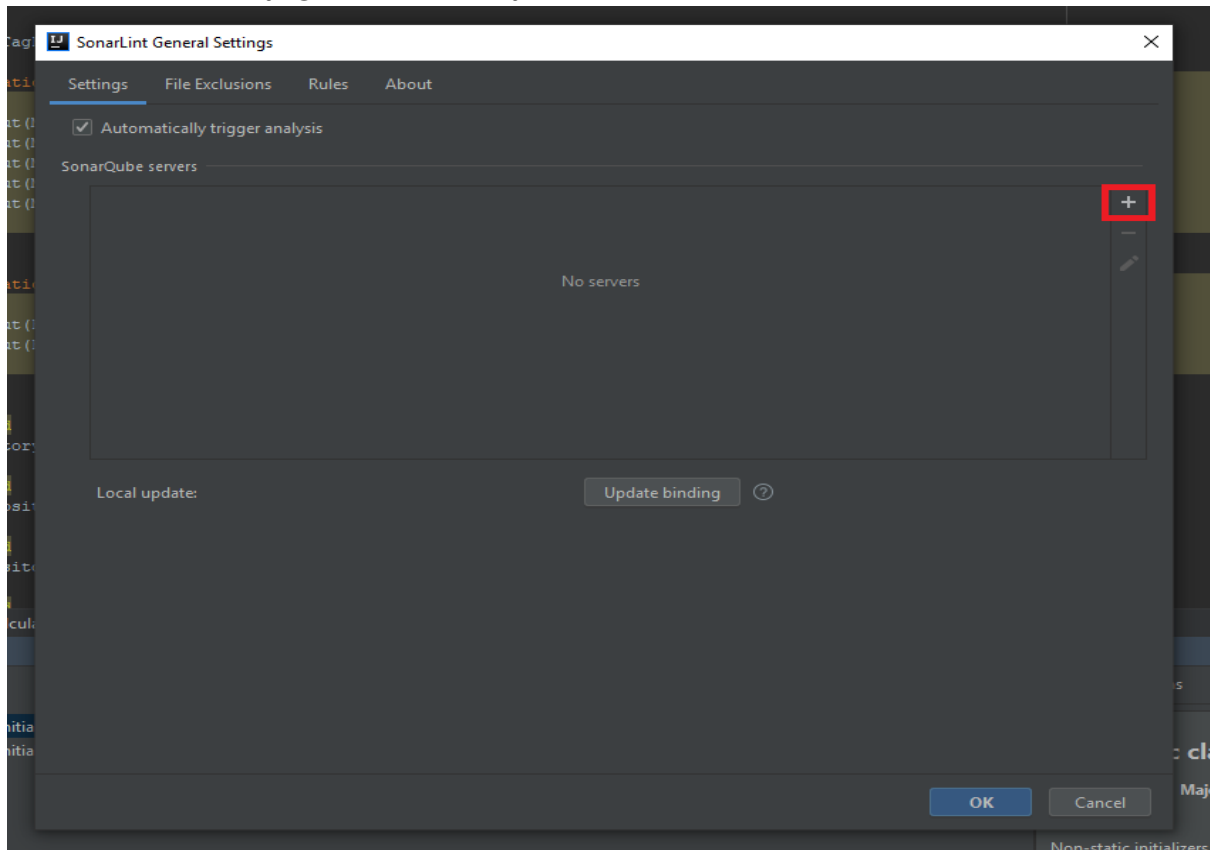
1) SonarLint konfigurieren:



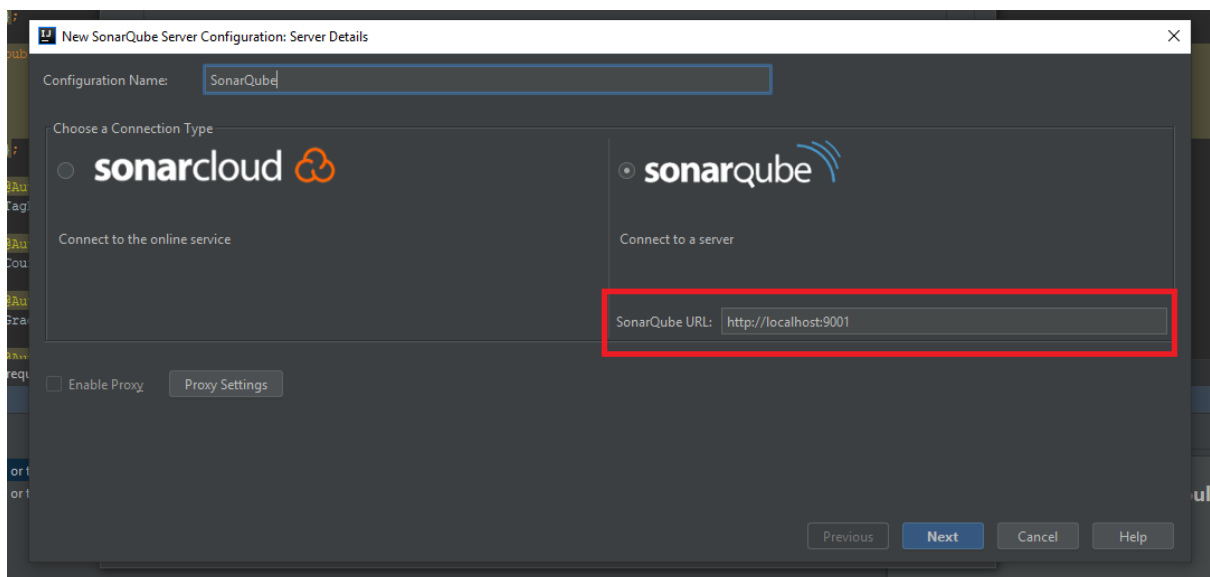
2) Server konfigurieren:



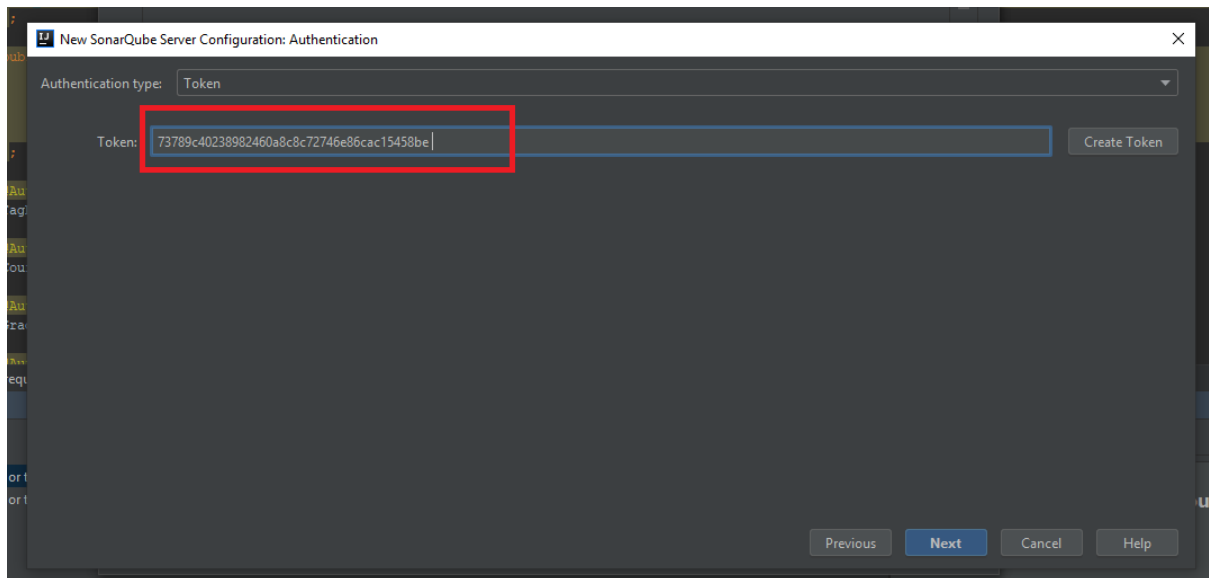
3) *Server hinzufügen mit Klick auf Plus-Icon:*



4) *SonarQube auswählen und eigene SonarQube-URL angeben:*



5) Angabe des Tokens, der in SonarQube erstellt wurde:



New SonarQube Server Configuration: Authentication

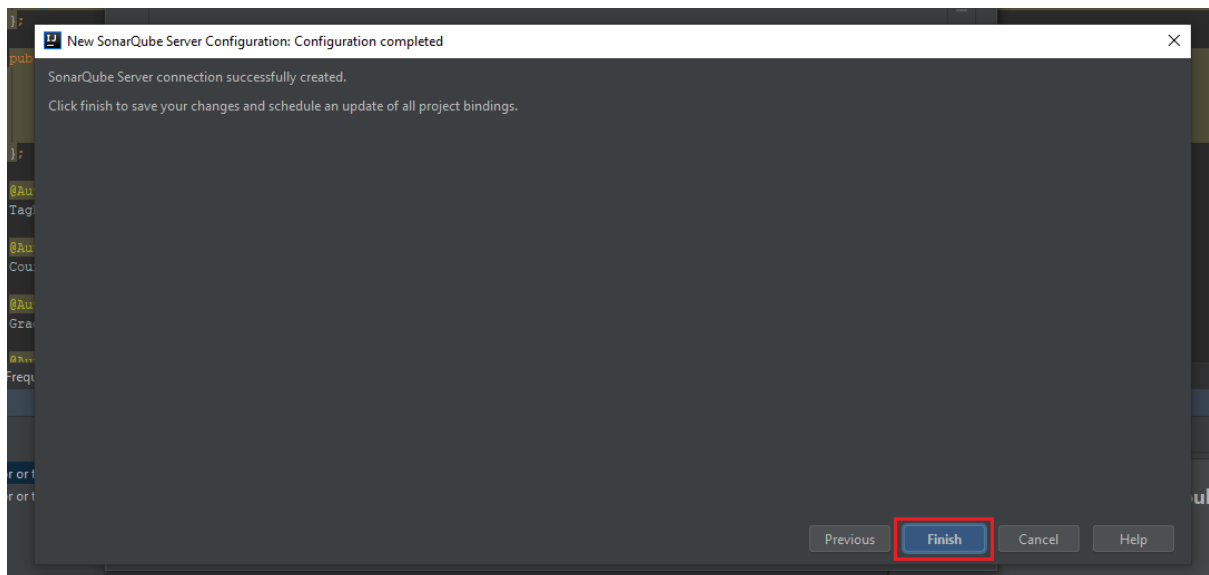
Authentication type: Token

Token: 73789c40238982460a8c8c72746e86cac15458be

Create Token

Previous Next Cancel Help

6) Einstellungen abschließen mit Klick auf den Button „Finish“:



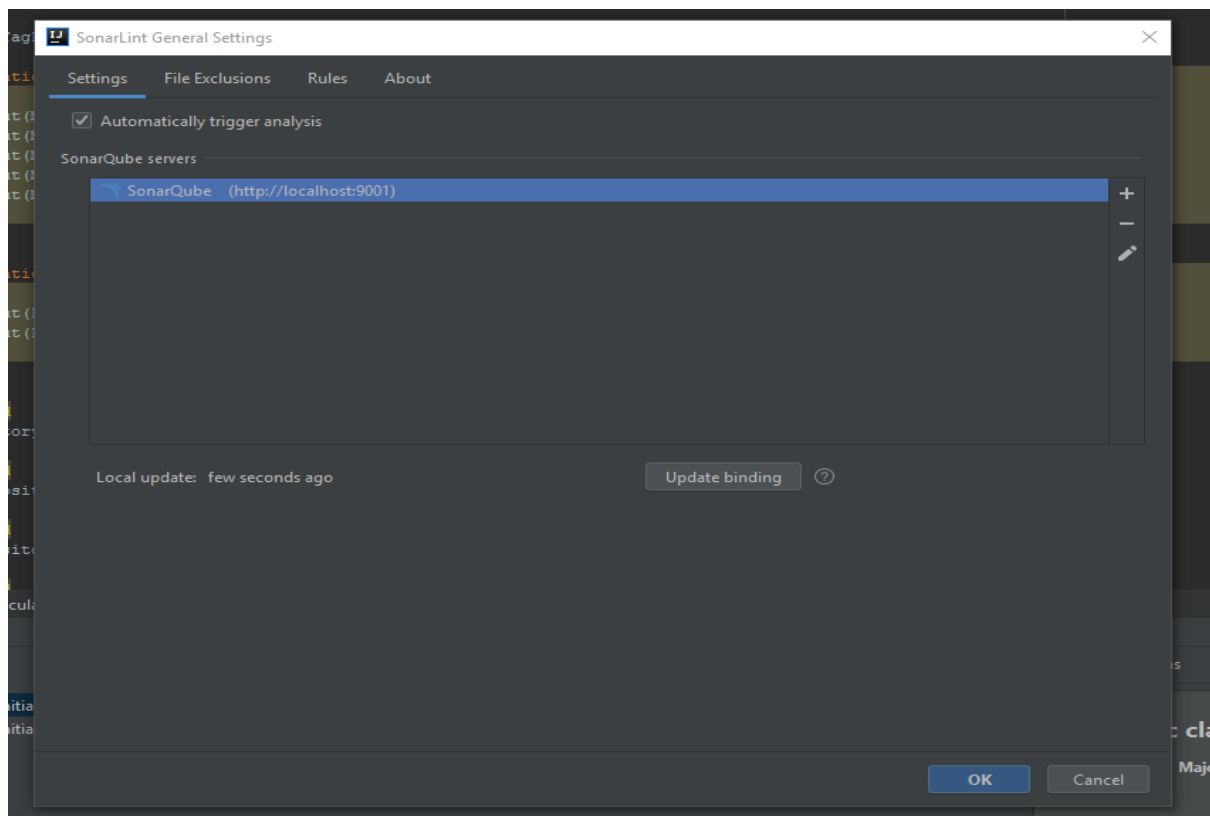
New SonarQube Server Configuration: Configuration completed

SonarQube Server connection successfully created.

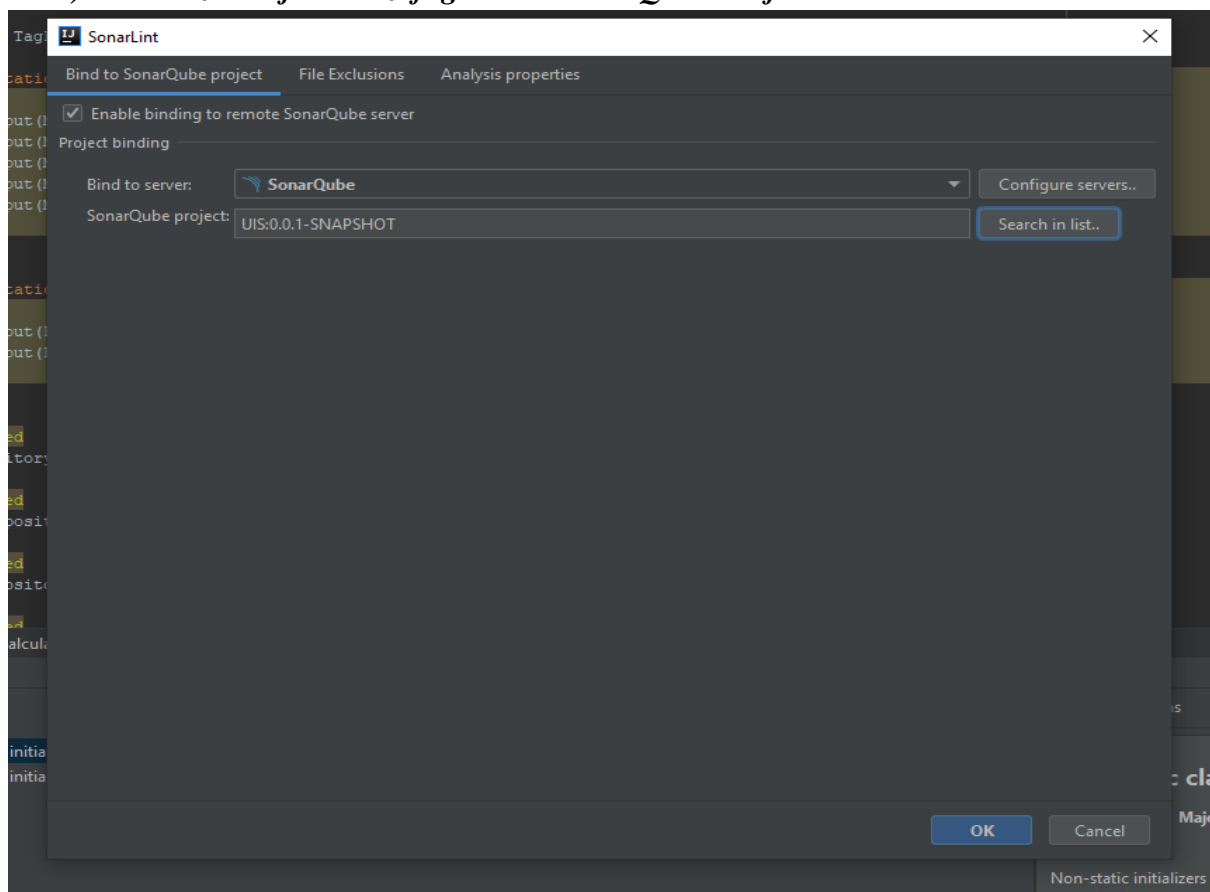
Click finish to save your changes and schedule an update of all project bindings.

Previous Finish Cancel Help

7) Fortfahren mit Klick auf OK:



8) Server zu Projekt hinzufügen und SonarQube-Projekt auswählen:



Nach dem Abschluss dieser Einstellungen werden für die statische Code-Analyse die SonarLint-Regeln aus SonarQube verwendet. Dadurch werden die Entwickler bereits beim Schreiben des Codes auf mögliche Fehler hingewiesen und müssen nicht auf den Abschluss des Builds am SonarServer warten.

Um einen reibungslosen Ablauf zu garantieren, sollten die entstandenen Issues bereits vor dem Pushen behoben werden, da ansonsten die Build-Prozesse unnötig blockiert werden. Eine weitere Möglichkeit, um schlechten Code zu verhindern, ist, dass Branches mit vorhandenen Issues nie auf den Master gemerged werden. Dabei könnten Merge/Pull Requests eingeführt werden, in denen ein anderer Entwickler den Code auf mögliche Fehler/Verbesserungen und Issues überprüft (also einen „Review“ macht, bevor der Code tatsächlich gepusht werden kann). Zahlreiche Tools wie Gitlab/Bitbucket unterstützen diese Merge Strategien und ermöglichen dadurch eine reibungslose Integration³⁵.

³⁵ <https://docs.gitlab.com/ee/gitlab-basics/add-merge-request.html>

3.1 Merge Requests and Protected Branches

Ein weiteres Konzept, um die Softwarequalität erheblich zu steigern, bieten Merge Requests. Es gibt zahlreiche verschiedene Merging-Konzepte, die von unterschiedlichen Anbietern empfohlen werden. Als ein erprobtes Konzept für eine Branching-Strategie und ein darin enthaltenes Code Review, empfehlen wir den GitLab Flow³⁶.

Branching-Strategie und Reviews in Bezug auf das Mergen:

Der GitLab Flow wird in zahlreichen Applikationen verwendet, er kombiniert „*feature-driven development*“ und „*feature branches*“³⁷. Der gewöhnliche Flow verwendet einen Development Branch. Um den Prozess zu vereinfachen, wurden jedoch folgende Entscheidungen getroffen:

- Feature Branches werden direkt vom Master Branch erstellt und dahin gemerged
- Feature Branches benötigen einen Merge Request, um in den Master gemerged zu werden
- Wenn der Feature Branch erfolgreich gemerged ist, muss dieser gelöscht werden
- Wenn benötigt können auch ein Produktions-Branch und Release-Branch direkt vom Master Branch angelegt werden
- Wenn benötigt können auch Hotfix Branches vom Master erstellt werden

Mit dieser beschriebenen Strategie benötigt jeder Merge einen Review, bevor dieser in den Master gelangen kann. Merge Requests können auch früher erstellt werden, um andere Projektmitglieder an der Entwicklung des Features teilhaben zu lassen. Diese Requests können dann mit dem Kürzel „WIP“ (*Work in Progress*) versehen werden. Dies ermöglicht dann direkt bei der Implementierung Feedback zu bekommen. Diese Vorgehensweise eignet sich speziell für komplexe Designentscheidungen bzw. Modularisierungen, die großen Einfluss an der Strukturierung des Projekts haben. Auch können sie einen guten Lerneffekt für neue Projektmitglieder bringen.

Überprüfung durch Reviewer und Definition einer Checkliste:

Damit der Review Prozess besser strukturiert wird, sollte eine Checkliste erstellt werden, die von den Entwicklern im Falle eines Reviews überprüft werden muss. Dies gewährleistet eine einheitliche Qualität des Reviewprozesses und bietet so auch neuen Projektmitgliedern einen besseren Leitfaden, worauf hingearbeitet werden sollte.

Die folgende Liste beschreibt die Punkte, die bei einem Review Prozess abgearbeitet werden müssen. Die Checkliste wurde dahingehend optimiert, welche Qualitätsmängel im Projekt bereits aufgetreten sind. Wenn ein Punkt der Liste nicht mit „*Erfüllt*“ beantwortet werden

³⁶ https://docs.gitlab.com/ee/workflow/gitlab_flow.html

³⁷ https://docs.gitlab.com/ee/workflow/gitlab_flow.html

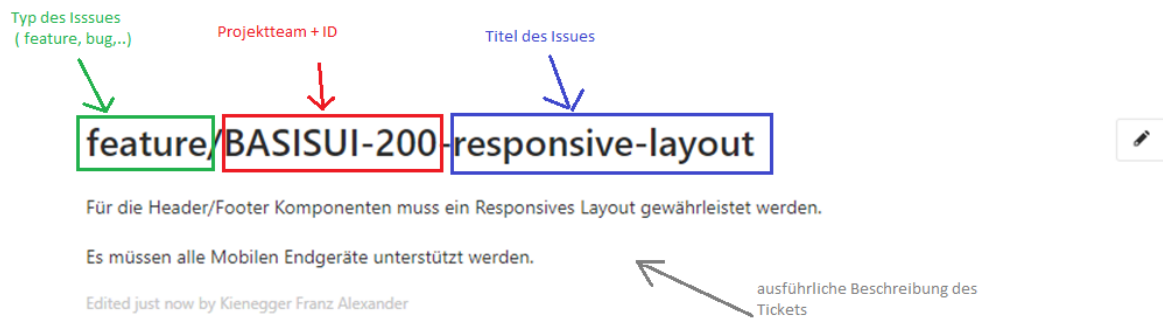
kann, muss der Merge Request verweigert werden. Diese Punkte müssen mit einer nützlichen Beschreibung des Mangels versehen werden, um den Entwicklern eine Hilfestellung zu bieten. Um die Merge Requests zugleich laufend zu verbessern, sollte die Checkliste auch während der Implementierungsphase angepasst werden, um auf Komplikationen zu reagieren.

Zu überprüfender Punkt	Erfüllt	Mangel (Beschreibung)
Ist der Code gut strukturiert?		
Ist der Code leicht verständlich?		
Ist der Code modular aufgebaut?		
Ist der Code frei von Duplikationen und Redundanzen?		
Ist der Code frei von Sicherheitsmängeln?		
Ist die Testabdeckung ausreichend?		
Wurden Unit- und Integration-Tests implementiert?		
Werden die definierten Namenskonventionen eingehalten?		
Wurde ein etwaiges Change Log angepasst?		
Wurde Logging mit den passenden Log-Leveln hinzugefügt?		
Funktioniert der Code? Können die Use Cases auch weiterhin ausgeführt werden?		
Ist der Code testbar?		
Werden gängige Coding-Konventionen eingehalten?		
Sind die öffentlichen Schnittstellen/APIs und sensible Code-Abschnitte mit Javadoc versehen?		
Ist der Code performant und frei von Speicherverschwendung?		

Namenkonventionen für Feature Branches:

Für jeden Feature Branch muss ein Issue existieren, worin der zu erledigende Task beschrieben wird. Zur Gewährleistung einer Verlinkung muss der Name des Branches mit dem Issue-Namen übereinstimmen. Der Name kann Aufschluss über den Typ, das Projektteam und den Inhalt geben.

In GitLab sollten Issues wie folgt systematisch benannt werden:



Der Branch innerhalb von Git muss dann mit denselben Namen des Issues versehen werden, z.B. „feature-BASISUI-200-responsive-layout“. Diese Vorgangsweise gewährleistet eine bessere Übersicht und eine eindeutige Zuordnung.

Wenn nach erfolgreichem Reviewprozess der Branch in den Master gemerged wurde, muss dieser gelöscht werden. Dies ermöglicht eine spätere Wiederverwendung des Namens, wenn die Qualität nicht ausreichend ist oder etwaige Features inkorrekt umgesetzt wurden, und eine Überarbeitung notwendig ist.

Berechtigungen für und Erfordernisse an Reviewer für das Akzeptieren von Merge Requests:

Generell sollten alle Entwickler, die an einem spezifischen Projekt arbeiten, in der Lage sein, Code-Reviews an einem Merge Request durchzuführen. Neue Projektmitglieder sollten in den Reviewprozess mittels Pair Programming eingebunden werden und erst nach einer Einarbeitungszeit die vollständigen Berechtigungen bekommen.

Eine weitere sinnvolle Einschränkung ist, dass nur Entwickler des jeweiligen Teams die Merge Requests akzeptieren, da diese die höchste Expertise aufweisen. Es besteht auch die Möglichkeit, die Reviews nach folgenden Kategorien zu kennzeichnen: Frontend, Backend oder Datenbank/Operations. Diese Requests werden dann Personen mit dem höchsten Know-How in der jeweiligen Kategorie zugewiesen.

Vorgangsweise, wenn ein Merge Request abgelehnt wird:

Falls ein Merge Request abgelehnt wird aufgrund von Verletzungen der Checkliste oder anderer organisatorischer Gründe, muss der Ersteller eine Begründung erhalten, warum dieser

nicht akzeptiert wurde. Daraufhin kann dieser seinen Code anpassen und den Merge Request erneut öffnen.

Falls ein Merge Request abgelehnt wird, bekommt der Ersteller eine Benachrichtigung, warum dieser in der angefragten Form nicht genehmigt werden konnte. Die Ursachen können sowohl fehlende Punkte in der Checkliste als auch organisatorische Gründe sein. Der Reviewer muss die Gründe für eine Ablehnung ausführlich darlegen, damit ein Lerneffekt eintritt. Daraufhin kann der Entwickler des Issues Anpassungen an seinem Branch vornehmen, um die beschriebenen Mängel zu beseitigen, und anschließend den Merge Request erneut öffnen.

Die organisatorische Aufgabe besteht darin, dass der Status des Issues und die verantwortlichen Personen dem Ticket korrekt zugeteilt werden, und bei Änderungen umgehend angepasst werden, und dass außerdem alle Betroffenen eine Benachrichtigung erhalten.

3.2 Knowledge Management

Knowledge Management in Bezug auf das Projekt ist von großer Bedeutung, da es den organisatorischen Rahmen für die Speicherung, Sammlung, Verbreitung und (Weiter)entwicklung von Wissen bietet. Mit Hilfe von Knowledge Management können nun das vorgeschlagene Tool Set-Up und die Richtlinien hinsichtlich Coding-Standards, Code-Qualität und Organisation von Merge Requests sowie Branches dokumentiert werden. Diese Dokumentation wird jedem Projektteammitglied zugänglich gemacht und dient als eine Art Referenz, die jedoch kontinuierlich erweitert und, falls notwendig, auch geändert werden kann. Außerdem bietet ein derartiges Wissensmanagementkonzept und -tool neuen Projektteammitgliedern eine gute Möglichkeit, sich einen Überblick über das Projekt und die wichtigsten Themen dazu zu verschaffen. Der Ansatz sollte auf einen längeren Zeithorizont ausgerichtet sein und sollte sich stetig weiterentwickeln.

Mögliche Tools für Knowledge Management im vorliegenden Projekt und Auswahl eines Tools davon:

Bevor die Entscheidung für ein Knowledge-Tool erfolgte, wurden verschiedene Möglichkeiten in Betracht gezogen. Als Wiki-Software und Content-Management-System wurde Drupal begutachtet, als Cloud-Wiki-Lösung Tiny MCE, als alleinstehende Wiki-Software MediaWiki und Atlassian Confluence. Als in GitLab integrierte und mit dem Projekt verbundene Lösung wurde GitLabWiki betrachtet.

Die Cloud-Lösung ***TinyMCE*** (i.e. Tiny Moxiecode Content Editor) bietet einen Open Source WYSIWYG-Editor für Wiki-Seiten basierend auf JavaScript an und kann kostenlos genützt werden. Die Benutzereingaben in einen Texteditor werden mittels JavaScript in HTML umgewandelt und angezeigt. Der Texteditor hat ein ähnliches Aussehen wie ein vereinfachtes Microsoft-Word-Programm.³⁸ TinyMCE könnte im Projekt UIS zwar verwendet werden, aber es ist nicht nahtlos an das Projekt angebunden und man müsste GitLab und die Wiki-Anwendung nebeneinander in zwei verschiedenen Programmen führen, was dazu führen könnte, dass die Developer das Wiki nicht so oft nutzen, da es ihnen zu mühsam erscheinen könnte.

Die weitere Möglichkeit ***Drupal*** ist ein freies Content-Management-System basierend auf PHP und bietet auch eine eigene Wiki-Lösung an.³⁹ Die Einträge des Benutzers müssen jeweils mit HTML-Markup erweitert werden, sodass sie passend angezeigt werden können.⁴⁰ Drupal ermöglicht auch Diskussionen und ein umfassendes Berechtigungs- und Rollenmanagement in Bezug auf Wiki-Seiten.⁴¹ Da Drupal diese HTML-Kenntnisse voraussetzt, ist die Verwendung etwas komplizierter. Außerdem würde die Drupal Application ebenfalls neben dem Projekt in GitLab stehen und nicht nahtlos angeschlossen sein.

³⁸ <https://www.tiny.cloud/docs/>: Anmerkung: Quelle bezieht sich auf den gesamten Absatz.

³⁹ <https://drupal-wiki.com/en/overview>

⁴⁰ <https://drupal-wiki.com/en/overview>

⁴¹ <https://www.computerwoche.de/g/die-besten-wiki-loesungen-im-ueberblick,102693,4>

Atlassian Confluence ist eine kommerzielle Wiki-Anwendung, die meist Hand in Hand mit der Verwendung von JIRA, einem ebenfalls von Atlassian kostenpflichtig angebotenen Issue- und Projektmanagementtool, geht. Confluence wird meist in Unternehmen für Wissensmanagement verwendet und legt den Fokus auf Java-Software-Development-Teams. Confluence bietet für die Erstellung von Wiki-Einträgen einen XML-basierten WYSIWYG-Texteditor, der auch Drag-&-Drop und Autovervollständigung ermöglicht.⁴² Für das Projekt UIS würde Atlassian Confluence eine gute Möglichkeit darstellen, wenn UIS tatsächlich ein Unternehmen mit stabiler Entwickler-Community wäre. Da aber in der momentanen Situation kein Unternehmen (auf längere Frist gesehen) besteht und kein Budget dafür zur Verfügung steht, könnte nur die kurzfristige kostenlose Testversion genutzt werden, die aber nach Ablauf der Testperiode eine Bezahlung und einen Lizenzerwerb notwendig machen würde.⁴³

MediaWiki ist jene Art von kostenloser und Open Source Wiki-Software (basierend auf PHP), mit der auch Wikipedia, das weltweit größte Wiki, erstellt wurde. MediaWiki bietet eine stabile langfristige Wartung und Unterstützung, vor allem für sehr große und aktive Wikiprojekte, da es die Skalierbarkeit gut unterstützt. Templates für Wikiseiten und Features können direkt in die eigene MediaWiki-Anwendung übernommen werden (unter Einhaltung der angegebenen Lizenz). Der Nachteil von MediaWiki ist allerdings, dass es komplex aufzusetzen und zu warten ist. Oftmals ist es für MediaWiki-Anwendungen notwendig, umfangreiche Designs und Inhalte von anderen Applikationen zu importieren. Die Dokumentation zu MediaWiki ist außerdem noch nicht vollkommen ausgereift und organisatorisch aufbereitet.⁴⁴

Schlussendlich haben wir uns nach der Durchsicht und dem Vergleich aller oben beschriebenen Möglichkeiten für **GitLabWiki** entschieden. GitLabWiki hat im Fall des Projekts UIS, das noch weiterentwickelt wird, den Vorteil, dass das Wiki direkt dem GitLab-Projekt zugeordnet werden kann, und dass man Teile davon leicht mit dem Wiki verlinken kann. Außerdem können die Mitglieder des Entwicklerteams dann schnell auf das Wiki zugreifen, wenn sie im Projekt und im GitLab-Repository arbeiten, d.h. es besteht keine Gefahr, dass das Wiki in Vergessenheit gerät, oder dass die Developer extra eine andere Anwendung öffnen müssen, und somit den Gebrauch des Wiki wegen dem hohen Aufwand reduzieren. Zusätzlich kann in den Wiki-Einträgen im GitLabWiki Java-Code angenehm angezeigt und eingebunden werden, und des Weiteren organisatorisch gut aufbereitet werden. Das GitLabWiki kann außerdem kostenlos verwendet werden und bietet auch die Möglichkeit, Änderungen nachzuverfolgen.

Für das ausgewählte Wiki, d.h. GitLabWiki, wurde nun ein **Prototyp** angelegt. Als Sprache für das Knowledge-Management-Tool wurde Englisch gewählt, da wir es zuvor auch als Projektsprache insgesamt vorgeschlagen haben. Die Inhalte des derzeitigen Wikis sind allen Projektmitgliedern zugänglich und sollen dazu dienen, dass neue Projektmitglieder einen guten Überblick über das Projekt-Set-Up, die Coding Standards, die Strategie für das Qualitätsmanagement und die Handhabung eines einheitlichen

⁴² <https://de.atlassian.com/software/confluence>: Anmerkung: Quelle bezieht sich auf den gesamten Absatz.

⁴³ <https://www.trustradius.com/products/atlassian-confluence/reviews/pros-and-cons?f=25>

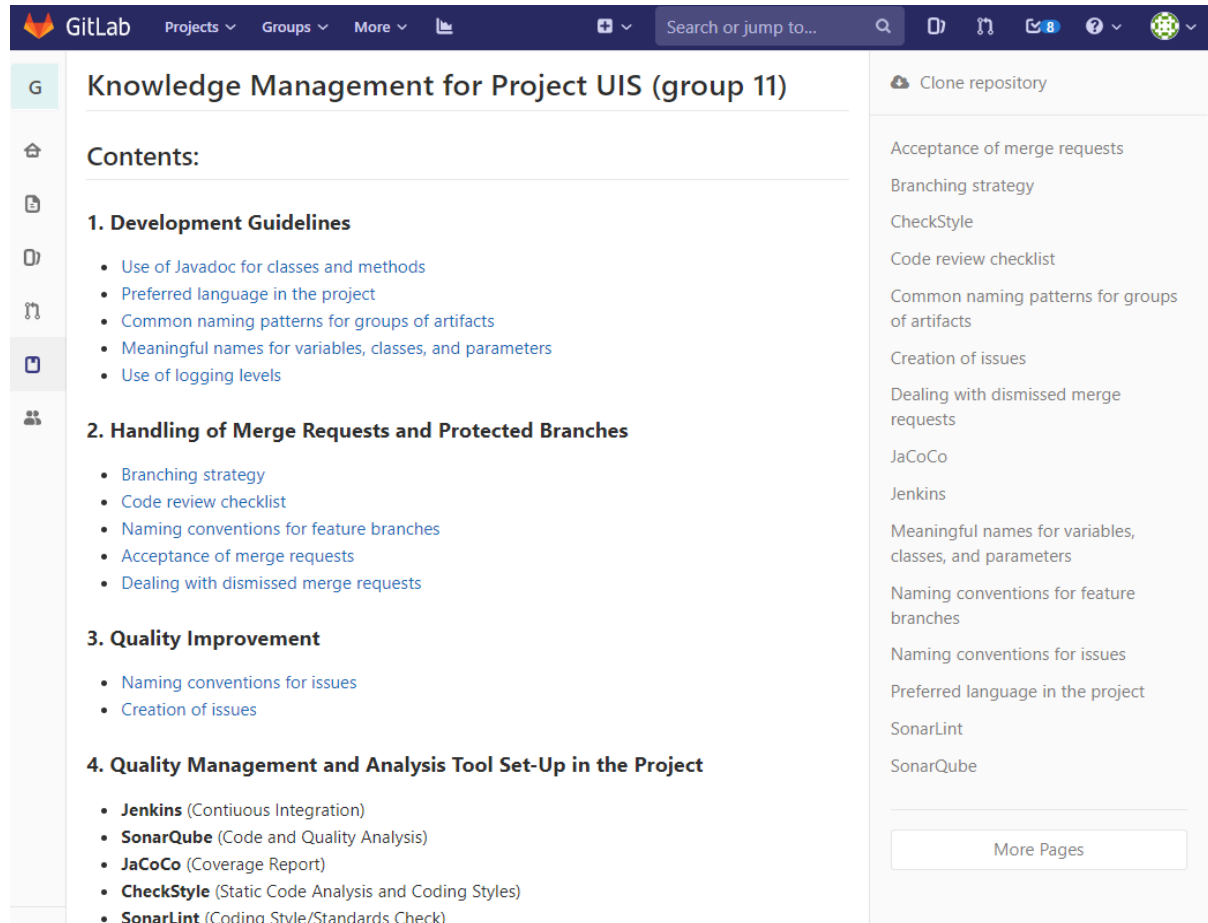
⁴⁴ https://www.mediawiki.org/wiki/Manual:Deciding_which_wiki_software_to_use: Anmerkung: Quelle bezieht sich auf den gesamten Absatz.

Softwareentwicklungsprozesses erhalten. Bereits am Projekt arbeitende Developer können das Wiki als Nachschlagwerk für gemeinsam bestimmte Richtlinien und Standards hinsichtlich Coding, Qualität und Abläufen (z.B. für Branching) konsultieren und gegebenenfalls erweitern, wenn neue Projektteile entwickelt werden und/oder sich die Standards des Projektteams ändern.

Vorgeschlagene Inhalte für das Knowledge-Management-Tool (Wiki) & Darlegung anhand des Prototyps:

Im Knowledge-Management-Tool (GitLabWiki) wurden anhand des Prototyps Informationen zu den Development Guidelines, die Handhabung von Merge Requests sowie Branches, Anleitungen zur Umsetzung von Qualitätsverbesserungen (mittels Issues und Feature Branches) und zum generellen Ansatz des Projektteams in Bezug auf Qualitätsmanagement und -analyse und einschlägigem Tool-Set-Up bereitgestellt und aufbereitet. Die Struktur des Wikis wurde in diese vier großen Themenblöcke aufgeteilt. Jeder Themenblock umfasst mehrere Wikiseiten, die genau angeben, worum es sich handelt, welche Richtung das Projektteam von UIS eingeschlagen hat, und welche „Main Take-Aways“ vom Leser nach Durchsehen der Wikiseite mitgenommen und im Projekt angewandt werden sollten. Die Inhalte können so je nach Thema leicht eingeordnet, wiedergefunden und verbessert werden. Das Wiki ist unter diesem Link zugänglich: <https://peso.inso.tuwien.ac.at/repo/sqm-ss19/group11/wikis/home>.

Hauptseite (Home) des GitLabWiki für das Projekt UIS (group 11):



The screenshot displays the GitLab Wiki interface for 'Project UIS (group 11)'. The top navigation bar includes the GitLab logo, 'Projects', 'Groups', and 'More' menus, along with a search bar and various utility icons. The main content area is titled 'Knowledge Management for Project UIS (group 11)' and features a 'Contents' section with four main categories:

- 1. Development Guidelines**
 - Use of Javadoc for classes and methods
 - Preferred language in the project
 - Common naming patterns for groups of artifacts
 - Meaningful names for variables, classes, and parameters
 - Use of logging levels
- 2. Handling of Merge Requests and Protected Branches**
 - Branching strategy
 - Code review checklist
 - Naming conventions for feature branches
 - Acceptance of merge requests
 - Dealing with dismissed merge requests
- 3. Quality Improvement**
 - Naming conventions for issues
 - Creation of issues
- 4. Quality Management and Analysis Tool Set-Up in the Project**
 - Jenkins (Continuous Integration)
 - SonarQube (Code and Quality Analysis)
 - JaCoCo (Coverage Report)
 - CheckStyle (Static Code Analysis and Coding Styles)
 - SonarLint (Coding Style/Standards Check)

The right sidebar contains a 'Clone repository' button and a list of links to specific wiki pages, including 'Acceptance of merge requests', 'Branching strategy', 'CheckStyle', 'Code review checklist', 'Common naming patterns for groups of artifacts', 'Creation of issues', 'Dealing with dismissed merge requests', 'JaCoCo', 'Jenkins', 'Meaningful names for variables, classes, and parameters', 'Naming conventions for feature branches', 'Naming conventions for issues', 'Preferred language in the project', 'SonarLint', and 'SonarQube'. A 'More Pages' button is located at the bottom of the sidebar.

Der erste Themenblock umfasst die *Development Guidelines*. Der untenstehende Screenshot zeigt die Wikiseite über die Verwendung von Javadoc für Klassen und Methoden. Auf der Seite wird angezeigt, welches Projektteammitglied die Seite zuletzt bearbeitet hat, und es ist auch möglich, den Verlauf der Bearbeitungen über *Page History* einzusehen. Die Seite kann, falls notwendig, auch direkt wieder bearbeitet werden, und es können auch weitere Unterseiten hinzugefügt werden, falls die Struktur noch umfangreicher werden sollte.

Der Inhalt ist einerseits beschreibend, was im oberen Bereich der Seite ersichtlich ist. Der Ansatz hinsichtlich des Themas wird dargelegt. Andererseits werden darunter die *Main Take-Aways* formuliert, die sich mit wichtigen Ratschlägen an die Developer richten, die diese nach der Lektüre der Wikiseite mitnehmen sollten. Weiters werden auch noch einige praxisnahe, am Projekt orientierte Beispiele angeführt, die das Thema greifbarer machen sollen.

Zum Anlegen der Wiki-Einträge wurde Markdown verwendet, sodass die Struktur und Formatierung mittels spezieller Syntax im Texteditor der Wikiseite bereits hinzugefügt werden kann. Auch Code kann somit mittels einer bestimmten Formatierung eingefärbt und abgehoben von den Textstellen eingebunden werden.

1. Development Guidelines > Use of Javadoc for classes and methods:

The screenshot shows a Wiki page interface. At the top, a breadcrumb trail reads 'sqm-ss19 > group11 > Wiki > Use of Javadoc for classes and methods'. The page title is 'Use of Javadoc for classes and methods', with a note 'Last edited by Eggerth Cordula Thekla 2 hours ago'. On the right, there are buttons for 'New page', 'Page history', and 'Edit'. The main content area contains the following text: 'The introduction of documentation for each and every class and method just for the sake of documentation should not be our main objective as it also uses valuable resources. We strive for creating classes and methods (notably the non-public code parts) in a way that their purpose is visible immediately and clearly without a huge amount of documentation. For the public REST API as well as other interfaces to external/public applications, we use Javadoc to state clearly what parameters, return and functionality are associated to a specific method, class or interface.' Below this, it says: 'We do not keep code which is set to commented state because this is a signal that is not an important code part anyway. Thus, such a code part should be either kept and re-set to production code (in uncommented state) or it should be deleted if it is not important any more.' A section titled 'Main take-aways:' follows, containing two bullet points: 'Public API towards external applications: use Javadoc for classes, methods, exceptions and parameters to state clearly what these parts of code are about.' and 'Non-public code parts: try to formulate and name the methods/classes/parameters/variables/exceptions in a way that no extensive documentation thereof is necessary and that they are easily understandable.'

G

🏠

📄

🔍

🔗

👤

- **Getter and setter methods:** no Javadoc or other comment necessary as these are commonly known functionalities
- **Commented code parts:** code should be production code, but not in commented form; if a code is written as a commented line, it should rather be deleted.

Good examples of the use of Javadoc:

```
/**
 * returns a user by name
 * can only be used if a user is authenticated
 *
 * @param name
 * @return {@link Tag} object
 */
@PreAuthorize("isAuthenticated()")
Tag findByName(String name);
```

```
/**
 *
 * @param id should not be null and not <1
 * @return UisUserEntity
 * @throws BusinessException
 *
 * returns an {@link UisUserEntity} with the provided id. if no user can be found
 * can only be used by ADMINS. if no user is found a {@link BusinessException}
 */
@PreAuthorize("hasRole('ADMIN')")
UisUserEntity findOne(long id) throws BusinessException;
```

1. Development Guidelines > Preferred language in the project:

G

🏠

📄

🔍

🔗

👤

sqm-ss19 > group11 > Wiki > Preferred language in the project

Preferred language in the project

Last edited by **Eggerth Cordula Thekla** 2 hours ago

New page

Page history

Edit

In this project, the most use language is currently English for classes, methods, exceptions and comments. In some cases, comments, exceptions variables or class names are still written in German. As the UIS project team, we strive to achieve 100% naming and commenting in English and to adopt English as the language of the project. For this reason, the parts that are still in German should be gradually phased out.

New parts of code, comments or Javadoc should always be subject to a spelling check by other project members than the one who wrote the concerned contribution. Additionally, the author should double check the spelling of his/her contribution before it is handed over to the spelling reviewers. Apart from this, grammar checks and typing auto-correction for English could be used.

Main take-aways:

- **English** should be the language of choice for naming parts of the code and commenting.
- Any piece of code or comment that is added to the project should undergo a **spelling check by project team members** before it is added to production code.

1. Development Guidelines > Common naming patterns for groups of artifacts:

sqm-ss19 > group11 > Wiki > Common naming patterns for groups of artifacts

Common naming patterns for groups of artifacts

Last edited by **Hasler Armin** just now

[New page](#)[Page history](#)[Edit](#)

When it comes to naming groups of artifacts such as classes in the package *entity*, either all classes should be extended with the suffix *Entity* or alternatively none of them could receive the suffix. But the most important lesson to consider at this point is that we are striving to use unified and consistent naming patterns.

In the *test* packages, we use the suffix *Test* to label classes, which are dedicated to testing purposes in order to easily recognize them.

For *interfaces*, we use the following naming pattern: *LecturerService*, *UserAccountService*. The implementing classes should end with the suffix *Impl*.

For *identifiers*, we use the common naming pattern *ID*.

For *"public constants"*, we use capital letters only, and if the constant consists of several words, the components of the constant should be connected using an underline symbol.

For *enums*, we aim to place them in the package next to the classes that are using them, and NOT in a general overall package or in separate packages within a specific package. Enums do not have to be specifically characterized in their naming, so no suffix like *Enum* is used.

Main take-aways:

- Append **classes in testing packages** by the word *Test*.
- **Interfaces** should follow the naming pattern like *LecturerService* or *UserAccountService*. Implementing classes should end with the suffix *Impl*.
- **Identifiers** use the common naming pattern *ID*.
- **Public constants** use capital letters only, and are connected using "_" in case of several components of the constant name.
- **Enums** should not use the suffix *Enum*, and should be placed in the package where they are used.

1. Development Guidelines > Meaningful names for variables, classes, and parameters:

sqm-ss19 > group11 > Wiki > Meaningful names for variables, classes, and parameters

Meaningful names for variables, classes, and parameters

Last edited by **Hasler Armin** just now

[New page](#)[Page history](#)[Edit](#)

With regard to semantics, the name of variables, classes and further parts of the code should reflect their actual meaning best possible. It should be clear from the name what the part is intended for.

If a term is not generally known, no abbreviations should be used.

The use of capital letters and small letters should be used in a common format. Variable names should start with a small letter, followed by only small letters. Class, exceptions and interface names should be started with a capital letter and each word in them should be started again with a capital letter. Within the words, the letters are small ones. Interfaces naming should correspond to the common naming pattern for interfaces (i.e. *LecturerService*).

Main take-aways:

- The **name** of variables, classes and further code parts should **reflect their actual meaning**.
- Use **no abbreviations** (which are not generally known).
- **Start classes, exceptions and interfaces** with **capital letters** in each word component.
- Use **uncapitalized letters** for **variable** naming.
- **Interface** names should correspond to the **common naming pattern** for interfaces.

Good examples for naming practices:

```
@Autowired
private UserAccountService userAccountService;

@Autowired
private LecturerRepository lecturerRepository;
```

```
SemesterTypeEnum(String name, int startMonth, int startDay)
```


1. Development Guidelines > Use of logging levels:

G

sqm-ss19 > group11 > Wiki > Use of logging levels

🏠

📄

🔍

🔗

📁

👤

Use of logging levels

New pagePage historyEdit

Last edited by **Eggerth Cordula Thekla** just now

In the UIS project, we use the application logging frameworks SLF4J and Log4J. These frameworks use certain log levels to categorize information that is generated by the application according to their urgency. The log events can be filtered by type and are ordered hierarchically.

In general, logging comprises various granularity steps:

- **ALL**: all log levels (incl. custom log levels)
- **DEBUG**: detailed events for diagnostic purposes (from developer perspective)
- **INFO**: info events at a less detailed level, which illustrate the normal behavior of the application
- **WARN**: potentially faulty situations (e.g. using deprecated syntax)
- **ERROR**: failure situations, in which the app can, however, possibly continue its service
- **FATAL**: severe failure situation, which (probably) leads to an abort of the app
- **OFF**: completely disables logging (i.e. nothing is logged at all)
- **TRACE**: more detailed events than DEBUG, but can lead to performance reduction due to its resource intensity

The log levels can be ordered hierarchically:

ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF

In this project, we use the levels ERROR, WARN, INFO and DEBUG, and we do not use TRACE or FATAL so far. The level OFF is not provided by Log4J, so it is not used either. The most used log level in the project is so far INFO, which illustrates the usual flow of events through the application.

G

Practice examples from the UIS project:

- **Log-level INFO:**

```
public String get(String path) {
    log.info("Nachrichten für den Pfad " + path + " erhalten.");
    return messageSource.getMessage(path, null, LOCALE);
}

public String msg(String path, Object... args) {
    log.info("Nachrichten für Objekt und Pfad " + path + " erhalten.");
    return messageSource.getMessage(path, args, LocaleContextHolder.getLocale());
}
```

72

- **Log-level ERROR:**

```
private static final Logger logger = LoggerFactory.getLogger(UisApplication.class);

public static void main(String[] args) {
    try {
        SpringApplication.run(UisApplication.class, args);
    }
    catch (Error e) {
        logger.error("An error occurred in the application", e);
        throw e;
    }
}
```

Weitere Codebeispiele sind für die Log-Levels WARN, ERROR und DEBUG verfügbar im Wiki, werden aber an dieser Stelle aus Platzgründen nicht nochmals aufgelistet, da sie genauso wie die vorherigen Log-Level-Beispiele aufgebaut sind.

Der zweite Themenblock umfasst die *Handhabung von Merge Requests und Protected Branches*. Die untenstehenden Screenshots zeigen die zugehörigen Wikiseiten.

2. Handling of merge requests and protected branches > Branching strategy:

G

sqm-ss19 > group11 > Wiki > Branching strategy

Branching strategy

Last edited by **Eggerth Cordula Thekla** 2 hours ago

New page

Page history

Edit

With our branching strategy, we aim to make each merge of a (feature) branch subject to a review prior to the actual merge of the code into the master branch. Merge requests can also be initiated before the planned merge, if a developer wants for instance to involve another developer in the coding or feedback process of the feature under concern. Such a branch can then be marked with the abbreviation **WIP** (i.e. *Work in Progress*), which means that the feature is not complete, but is under interim review. New project members should not receive immediate responsibility over whole feature branches, but should approach the project in the first phase via pair programming with experienced developers.

Main-takeaways on our branching strategy:

- **Feature branches** are directly **created from the master branch** and **merged into it**.
- A **feature branch needs a merge request**, if the feature developer wants **to merge it into the master**.
- Once the feature branch is successfully merged, it has to be deleted.
- If necessary, a production branch and release branch can be directly created from the master branch.
- If necessary, hotfix branches can be directly created from the master branch.
- A branch can be marked as **WIP**, if an **interim review** by a developer should take place.
- **New project members** should be involved via **pair programming**.

2. Handling of merge requests and protected branches > Code review checklist:

G

sqm-ss19 > group11 > Wiki > Code review checklist

🏠

📄

🔗

🔍

👤

👥

»

Code review checklist

Last edited by **Eggerth Cordula Thekla** 2 hours ago

New page

Page history

Edit


To better structure the review process, each one of our developer team members, who is chosen as a reviewer in line with a merge request, should go through the checklist one by one and answer the respective questions. The list of questions was optimized in a way that it takes common issues into account that have already occurred in the UIS project in the past.







If the reviewer judges the point of the checklist as **fulfilled*", s/he should just mark it as *fulfilled*. However, if the reviewers thinks that the **point of the checklist is not fulfilled**, s/he is required to **give a justification** and description of the situation and why it was not fulfilled.

Checklist for reviews of merge requests:

- Is the code well structured?
- Is the code easily understandable?
- Is the code free of duplications and redundancies?
- Is the code free of security breaches?
- Does the code comprise unit and integration tests?
- Does the code comply with the defined naming conventions?
- Was the change log adapted?
- Was logging (incl. suitable log levels) added?
- Does the build succeed?
- Can all use cases that existed so far still be run without problems?
- Is the code testable?
- Were the project coding conventions/standards respected?
- Do the public interfaces/APIs and sensitive code parts contain Javadoc?
- Is the code free of performance inhibitors?

2. Handling of merge requests and protected branches > Naming conventions for feature branches:

 sqm-ss19 > group11 > Wiki > Naming conventions for feature branches



Naming conventions for feature branches

Last edited by **Eggerth Cordula Thekla** 2 hours ago

[New page](#)[Page history](#)[Edit](#)

An issue has to be created for each feature branch. In this issue, the task to be completed has to be described. The name of the branch needs to correspond exactly to the name of the issue so that linking is possible. Also, the branch can easily be recognized and it should be uniquely identifiable.

The name of the branch can provide additional information on the type, the project team, and/or the content related to the feature.

Once the (feature) branch has been merged into the master, the (feature) branch should be deleted. In doing so, the name of the completed (feature) branch can be re-used afterwards.

Proposed naming pattern for issues on GitLab:

type_of_issue / project_team_-_ID / title_of_issue

Exemplary issue:

FEATURE / USERINTERACTION-100 / frontend_tracking

2. Handling of merge requests and protected branches > Acceptance of merge requests:

G

sqm-ss19 > group11 > Wiki > Acceptance of merge requests

🏠

📄

🔗

🔒

🔗

📄

👤

Acceptance of merge requests

Last edited by **Eggerth Cordula Thekla** 1 hour ago

New page

Page history

Edit

Overall, all qualified developers, who are part of the project UIS, should be able to do reviews of merge requests. If a developer still lacks experience in reviewing, s/he should be given training in the form of a pair reviewing phase and should be enabled to develop his/her reviewing skills under the auspices of an experienced reviewer. If a new developer joins the team, s/he should first undergo a phase of pair reviewing and pair programming before s/he takes responsibility for her/his own reviews of merge requests.

One further restriction might be that only developers from the respective teams accept merge requests because these have the highest expertise. Depending on the area to which the topic can be attributed, a merge requests reviewer might also be selected based on the aggregated area of the feature, i.e. frontend, backend, or database/operations. So, the merge requests can be attributed to the developers with the highest level of expertise in the specific domain.

Main take-aways:

- **All qualified developers** in the team should be able and prepared to **do reviews of merge requests**.
- **Not fully experienced developers and new project members** should be introduced to reviewing in the UIS project via a **pair programming** approach.

2. Handling of merge requests and protected branches > Dealing with dismissed merge requests:

G

sqm-ss19 > group11 > Wiki > Dealing with dismissed merge requests

🏠

📄

🔗

🔗

🔗

🔗

👤

Dealing with dismissed merge requests

Last edited by **Eggerth Cordula Thekla** 49 minutes ago

New page

Page history

Edit

If a merge request is dismissed due to lacking fulfilment of the checklist criteria or other organizational reasons, the developer who created the merge request must obtain a justification why the reviewer came to the conclusion that the request could not be accepted.

Subsequent to this, the creator of the merge request can open his/her merge request again and adapt the code accordingly.

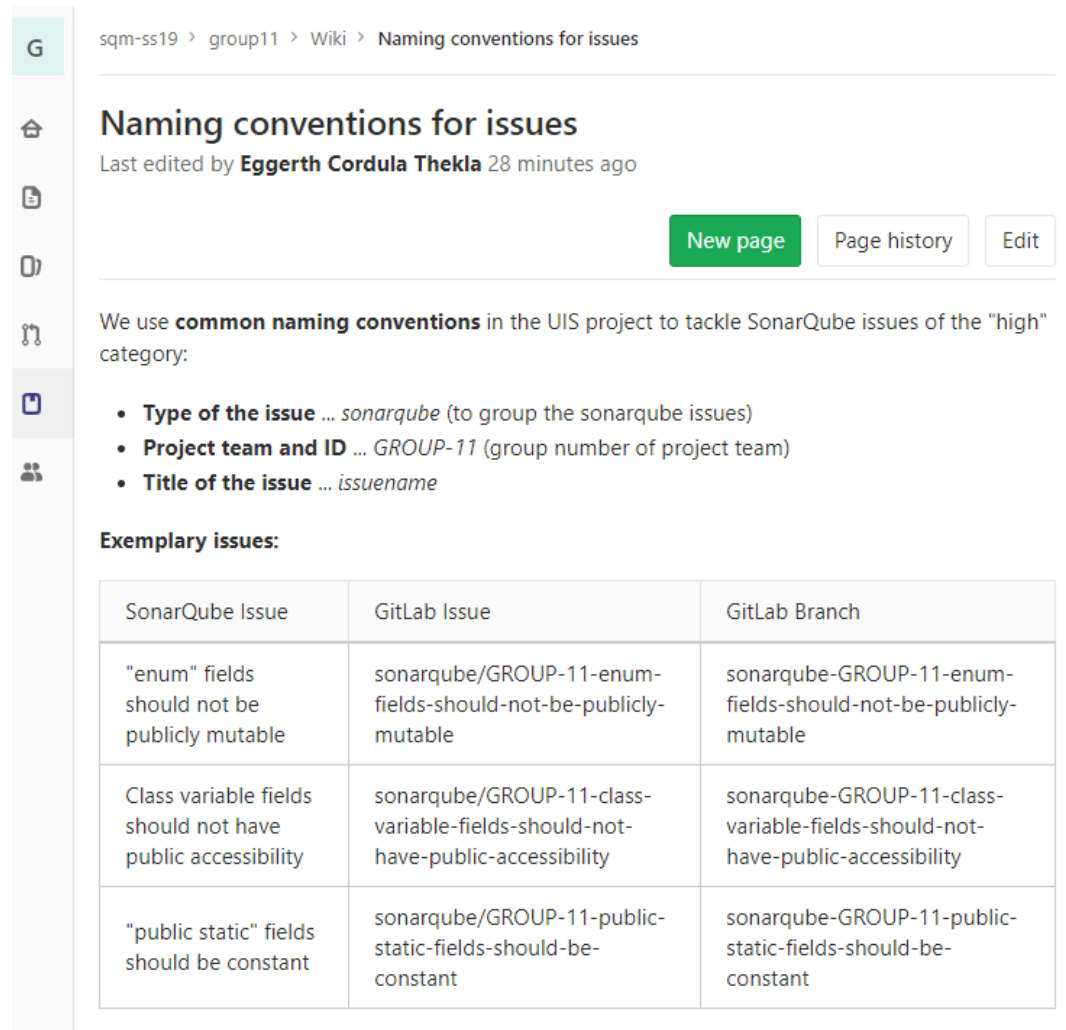
Overall, the organizational goal is to correctly assign the status of the issue to the responsible person, and that the feedback can be traced back to the reviewer, and can be integrated by the concerned project team member. Also, all involved project team members receive a notification in case of a change to the code under review.

Main take-aways:

- The developer who submitted the **dismissed merge request**, must receive a **justification why the reviewer did not accept** the conclusion.
- A **dismissed merge request should be re-opened** by the developer who created it to be **adapted according to the feedback** from the reviewer.

Der dritte Themenblock umfasst die *Verbesserung der Qualität*. Die untenstehenden Screenshots zeigen die dazugehörigen Wikiseiten:

3. Quality Improvement > Naming conventions for issues:



sqm-ss19 > group11 > Wiki > Naming conventions for issues

Naming conventions for issues

Last edited by **Eggerth Cordula Thekla** 28 minutes ago

[New page](#) [Page history](#) [Edit](#)

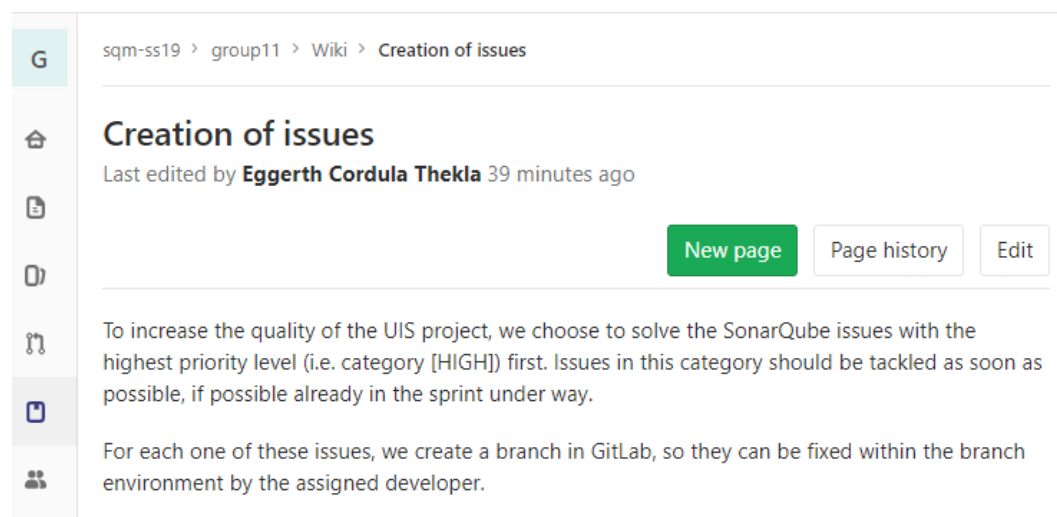
We use **common naming conventions** in the UIS project to tackle SonarQube issues of the "high" category:

- **Type of the issue** ... *sonarqube* (to group the sonarqube issues)
- **Project team and ID** ... *GROUP-11* (group number of project team)
- **Title of the issue** ... *issuename*

Exemplary issues:

SonarQube Issue	GitLab Issue	GitLab Branch
"enum" fields should not be publicly mutable	sonarqube/GROUP-11-enum-fields-should-not-be-publicly-mutable	sonarqube-GROUP-11-enum-fields-should-not-be-publicly-mutable
Class variable fields should not have public accessibility	sonarqube/GROUP-11-class-variable-fields-should-not-have-public-accessibility	sonarqube-GROUP-11-class-variable-fields-should-not-have-public-accessibility
"public static" fields should be constant	sonarqube/GROUP-11-public-static-fields-should-be-constant	sonarqube-GROUP-11-public-static-fields-should-be-constant

3. Quality Improvement > Creation of issues:



sqm-ss19 > group11 > Wiki > Creation of issues

Creation of issues

Last edited by **Eggerth Cordula Thekla** 39 minutes ago

[New page](#) [Page history](#) [Edit](#)

To increase the quality of the UIS project, we choose to solve the SonarQube issues with the highest priority level (i.e. category [HIGH]) first. Issues in this category should be tackled as soon as possible, if possible already in the sprint under way.

For each one of these issues, we create a branch in GitLab, so they can be fixed within the branch environment by the assigned developer.

Der vierte Themenblock umfasst generelle Informationen zu *Qualitätsmanagement und -analyse* im Projekt und zum zugehörigen Tool Set-Up:

4. Quality Management and Analysis Tool Set-Up in the Project:

- **Jenkins** (Continuous Integration)
- **SonarQube** (Code and Quality Analysis)
- **JaCoCo** (Coverage Report)
- **CheckStyle** (Static Code Analysis and Coding Styles)
- **SonarLint** (Coding Style/Standards Check)

3.3 Qualitätsverbesserung (*Quality Improvement*)

Um die Qualität des Projekts zu erhöhen, haben wir uns dazu entschlossen, zunächst die Sonarqube Issues mit der höchsten Kategorie [HIGH] zu beheben (siehe **1.3 SonarQube Integration**). Die Issues in dieser Kategorie sollten umgehend im aktuellen Sprint behoben werden.

Im Zuge dessen haben wir für jeden Sonarqube Issue der Kategorie [HIGH] einen Issue und einen dazugehörigen Branch in GitLab angelegt. Dabei haben wir uns an die von uns vorgeschlagenen Namenskonventionen für Issues und Branches gehalten:

- Typ des Issues: *sonarqube* (zur Gruppierung der Sonarqube Issues)
- Projektteam + ID: *GROUP-11* (der Einfachheit halber unsere Gruppennummer)
- Titel des Issues: Issue-Bezeichnung

Sonarqube Issue	GitLab Issue	GitLab Branch
"enum" fields should not be publicly mutable	sonarqube/GROUP-11-enum-fields-should-not-be-publicly-mutable	sonarqube-GROUP-11-enum-fields-should-not-be-publicly-mutable
Class variable fields should not have public accessibility	sonarqube/GROUP-11-class-variable-fields-should-not-have-public-accessibility	sonarqube-GROUP-11-class-variable-fields-should-not-have-public-accessibility
"public static" fields should be constant	sonarqube/GROUP-11-public-static-fields-should-be-constant	sonarqube-GROUP-11-public-static-fields-should-be-constant
"BigDecimal(double)" should not be used	sonarqube/GROUP-11-bigdecimal-constructor-should-not-be-used	sonarqube-GROUP-11-bigdecimal-constructor-should-not-be-used
Null pointers should not be dereferenced	sonarqube/GROUP-11-null-pointers-should-not-be-dereferenced	sonarqube-GROUP-11-null-pointers-should-not-be-dereferenced
Optional value should only be accessed after calling isPresent()	sonarqube/GROUP-11-optional-value-should-only-be-accessed-after-calling-isPresent	sonarqube-GROUP-11-optional-value-should-only-be-accessed-after-calling-isPresent
Methods should not be empty	sonarqube/GROUP-11-methods-should-not-be-empty	sonarqube-GROUP-11-methods-should-not-be-empty
Track uses of "FIXME" tags	sonarqube/GROUP-11-track-uses-of-fixme-tags	sonarqube-GROUP-11-track-uses-of-fixme-tags
Methods and field names should not be the same or differ only by capitalization	sonarqube/GROUP-11-methods-and-field-names-should-not-be-the-same-or-differ-only-by-capitalization	sonarqube-GROUP-11-methods-and-field-names-should-not-be-the-same-or-differ-only-by-capitalization
Track uses of "TODO" tags	sonarqube/GROUP-11-track-uses-of-todo-tags	sonarqube-GROUP-11-track-uses-of-todo-tags
Constant names should	sonarqube/GROUP-11-	sonarqube-GROUP-11-

comply with a naming convention	constant-names-should-comply-with-a-naming-convention	constant-names-should-comply-with-a-naming-convention
Boolean checks should not be inverted	sonarqube/GROUP-11-boolean-checks-should-not-be-inverted	sonarqube-GROUP-11-boolean-checks-should-not-be-inverted
Static non-final field names should comply with a naming convention	sonarqube/GROUP-11-static-non-final-field-names-should-comply-with-a-naming-convention	sonarqube-GROUP-11-static-non-final-field-names-should-comply-with-a-naming-convention
Cognitive Complexity of methods should not be too high	sonarqube/GROUP-11-cognitive-complexity-of-methods-should-not-be-too-high	sonarqube-GROUP-11-cognitive-complexity-of-methods-should-not-be-too-high
Generic exceptions should never be thrown	sonarqube/GROUP-11-generic-exceptions-should-never-be-thrown	sonarqube-GROUP-11-generic-exceptions-should-never-be-thrown
Local variables should not shadow class fields	sonarqube/GROUP-11-local-variables-should-not-shadow-class-fields	sonarqube-GROUP-11-local-variables-should-not-shadow-class-fields

An den Stellen, an denen wir Javadoc-Kommentare hinzufügen sollten, haben wir das nur dummymäßig gemacht, um die Sonarqube Issues zu beheben, da es unserer Meinung nach nicht Ziel dieses Kurses ist, fehlende Javadoc-Kommentare zu ergänzen.

Außerdem haben wir bei den als *TODO* markierten Bereichen nur jene Stellen behoben, wo die Lösung einfach und offensichtlich war, da wir über zu wenig Projektwissen verfügen, um alle *TODOs* beheben zu können.

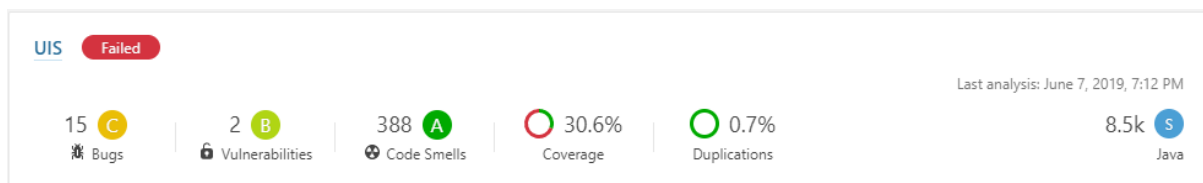
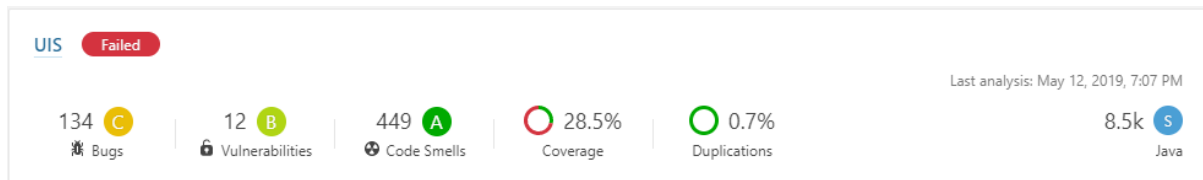
Bei der Cognitive Complexity haben wir nur eine von zwei betroffenen Klassen angepasst. Bei der Klasse `CoruseDetailsForStudent` resultiert die Komplexität daraus, dass die Klasse eine große Anzahl von Variablen enthält, die in der `equals`-Methode überprüft werden. Das ist unserer Meinung nach aber ein gültiger Ansatz, weshalb wir diese Klasse nicht geändert haben.

Zusätzlich zu den Sonarqube Issues haben wir noch folgende Issues hinsichtlich der Konventionen zur Namensgebung gemäß unseren Development Guidelines in GitLab angelegt und behoben:

Issue	GitLab Issue	GitLab Branch
Einheitliche Benennung der Test-Klassen	guidelines/GROUP-11-consistent-suffix-for-tests	guidelines-GROUP-11-consistent-suffix-for-tests
Einheitliche Benennung der Interfaces und implementierenden Klassen	guidelines/GROUP-11-consistent-naming-for-interfaces-and-implementing-classes	guidelines-GROUP-11-consistent-naming-for-interfaces-and-implementing-classes

Hier haben wir uns wieder an die von uns vorgeschlagenen Namenskonventionen für Issues und Branches gehalten, allerdings haben wir als Typ des Issues *guidelines* verwendet, weil es sich hier um Verbesserungen hinsichtlich der Development Guidelines handelt.

Die folgenden Screenshots zeigen den Sonarqube-Status vor und nach der Behebung der oben genannten Issues:



Wie man hier gut erkennen kann, haben die Behebungen der Issues zu einem nennenswerten Anstieg der Qualität geführt. So hat sich die Anzahl der Bugs von 134 auf 15, der Vulnerabilities von 12 auf 2 und der Code Smells von 449 auf 388 verringert. Außerdem ist sogar die Coverage noch leicht gestiegen.

3.4 Report Conclusio

Im ersten Abschnitt haben wir einen Vergleich verschiedener CI-Lösungen vorgenommen, wobei wir uns schlussendlich für Jenkins entschieden haben. Wir haben das System zur Qualitätsanalyse und -verbesserung mit Docker aufgesetzt und eine Pipeline erstellt, die unter anderem nach jedem Projekt-Build Schritte zur Überprüfung der Testabdeckung mit JaCoCo und eine SonarQube-Analyse enthält. Die Testabdeckung ist derzeit nicht ausreichend und sollte unserer Meinung nach bei mindestens 80% liegen. Durch die Integration von SonarQube kann die Qualität des Projekts verbessert werden, indem vorhandene Bugs, Vulnerabilities und Code Smells aufgezeigt werden. Die Build-Pipeline sollte so konfiguriert werden, dass der Build bei einer Unterschreitung der definierten Testabdeckung oder Codequalität fehlschlägt.

Im zweiten Abschnitt wurde Checkstyle in das Projekt integriert, um eine nachhaltig hohe und einheitliche Codequalität zu gewährleisten. Allerdings führt das nachträgliche Hinzufügen von Checkstyle-Regeln in ein bereits weit fortgeschrittenes Projekt zu einigen Problemen und sollte deshalb im Idealfall schon von Projektbeginn an integriert werden. Zur Gewährleistung eines möglichst einheitlichen und lesbaren Codes wurden außerdem Development Guidelines definiert, die jeder Entwickler berücksichtigen sollte. Die Integration von SonarLint in die vorhandene Entwicklungsumgebung unterstützt die Entwickler dabei, bereits bei der lokalen Entwicklung eine statische Code-Analyse durchzuführen. So können Fehler schon vor dem Pushen in das Repository behoben werden. Die verwendeten Qualitätsregeln können dabei direkt aus dem SonarQube-Server gelesen werden.

Im letzten Abschnitt werden Merge Requests beschrieben, mit denen eine erhebliche Steigerung der Softwarequalität erreicht werden kann. Da vor jedem Merge in den Master eine zweite Person den Code begutachten und bestätigen muss, sinkt die Wahrscheinlichkeit, grobe Fehler einzubauen. Außerdem führt dieser Reviewprozess zu einem Lerneffekt für neue Projektmitglieder. Wir haben für den Reviewprozess eine Checkliste definiert, die den Reviewern als Grundlage für die Entscheidung zur Bestätigung bzw. Ablehnung der Merge Requests dienen soll. Um das gesammelte Wissen sinnvoll aufarbeiten zu können und allen Projektmitgliedern zur Verfügung zu stellen, wurde ein Projektwiki eingeführt, in dem alle bis hierhin und in Zukunft gesammelten Informationen strukturiert aufgearbeitet werden. Zum Abschluss haben wir außerdem die SonarQube Issues mit der höchsten Priorität und einige Issues bezüglich unserer Development Guidelines behoben und gemäß dem von uns definierten Reviewprozess bearbeitet.

<i>Gewonnene Erfahrungen und Einsichten:</i>

- Verwendung von Code Reviews, um Qualität und Wissen über Code-Abschnitte zu teilen
- Verwendung der richtigen Branching-Strategie mit den GitLab Issues
- Anwendung der Checkstyle- und SonarQube-Analyse während der Entwicklung
- Vermeidung von Sonar Issues oder Verringerung auf ein akzeptables Maß
- Checkstyle-Überprüfung bereits bei Projektbeginn integrieren
- Development Guidelines sollten schon vor Projektbeginn erstellt werden
- Beachtung der Richtlinien bezüglich der Development Guidelines

<i>Verbesserungsvorschläge für die Zukunft:</i>
--

- Einführung von TDD (Test Driven Development), um eine bessere Qualität und gute Testabdeckung zu erreichen
- Einführung eines SCRUM-Entwicklungsprozesses, um stets lauffähige Artefakte zu generieren, und einen strukturierten sowie agilen Rahmen für Softwareprojekte zu schaffen
- Automatisches Deployment der Applikation auf einem CI-Server für Produktions- und Testumgebung
- Stetige Anpassung und Erweiterung der Regeln, um zeitnah auf mögliche Schwierigkeiten reagieren zu können