Bachelorarbeit

Sönke Peters

Automatische Generierung von Issues aus Testreports

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Sönke Peters

Automatische Generierung von Issues aus Testreports

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. rer. nat. Thomas Lehmann

Eingereicht am: 25. Februar 2019

Sönke Peters

Thema der Arbeit

Automatische Generierung von Issues aus Testreports

Stichworte

Testen, Automatisierung, GitLab, Issue, Issue-Tracker

Kurzzusammenfassung

Am Ende eines automatisierten Testdurchlaufs steht ein Report. Dieser enthält Informationen über erfolgreiche und fehlgeschlagene Tests. Letztere sind für den Entwickler relevant, weil sie Hinweise auf mangelhaften oder nicht funktionierenden Code liefern. Im Entwicklungsprozess wird für die Bearbeitung eines fehlgeschlagenen Tests ein sogenanntes Issue in einem Issue-Tracker, wie zum Beispiel dem Tracker der zu GitLab gehört, erstellt. Das Erstellen der Issues bedeutet Aufwand für die Person, die den Issue formulieren muss. In dieser Arbeit wird gezeigt, wie aus einem Testreport automatisch Issues in einem Issue-Tracker erstellt werden können. Es wird dabei auf den Umgang mit Duplikaten, irrelevant gewordenen Issues und das Zusammenfassen von ähnlichen Fehlermeldungen eingegangen. Das können Fehlermeldungen sein, deren beschriebene Fehler nahe (im Code) bei einander liegen oder den selben Fehler bezogen auf andere Abschnitte melden. Das Programm, welches im Rahmen dieser Arbeit entwickelt wird, kann verschiedene Reports in Issues verwandeln und wird in eine Pipeline integriert. Die Funktionalität wird zum einen mit einem Testreport einer statischen Codeanalyse, zum anderen mit einem Testreport eines Regressionstest geprüft.

Sönke Peters

Title of Thesis

Automatic Generation of Issues from test reports

Keywords

Testing, Automatation, GitLab, Issue, Issue-Tracker

Abstract

At the end of an automated test run there is a report. This report contains information about successful and failed tests. The latter are relevant for the developer because they provide information about defective or non-functionioning In the development process, an issue is created in an issue tracker, such as the tracker belonging to GitLab, to process a failed test. Creating the issues means effort for the person who has to formulate the issue. This paper shows how to automatically create issues from a test report in an issue tracker. It deals with the handling of duplicates, issues that have become irrelevant and the summary of similar error messages. These can be error messages whose described errors are close (in code) to each other or report the same error related to other sections. The program developed as part of this work can convert various reports into issues and is integrated into a pipeline. The functionality is tested on the one hand with a test report of a static code analysis, on the other hand with a test report of a regression test.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **x**

1 Einleitung **1**

2 Verwandte Arbeiten **3**

2.1 Wissenschaftliche Arbeiten zum Thema Issues 3
2.2 Betrachtung verwandter Programme im Bereich des automatisierten Testens 5
2.3 Fazit der Betrachtung und Einordnung der Arbeit. 8

3 Anforderungsanalyse **10**

3.1 Wirtschaftlicher Hintergrund der Softwarequalität und Automatisierung . 10
3.2 Hintergrund der Automatisierung des Übertragungsprozesses von Testreport zu Issue 11
3.3 Übersicht und Empfehlungen zum Aufbau und Inhalt eines Bug-Reports . 12
3.3.1 Aufbau und Merkmale des Testreports auf Basis des Incident Reports von IEEE-29119 17
3.4 Erwartete Herausforderungen 17
3.4.1 Duplikate 18
3.4.2 Wanderung und Verlust der Relevanz von Fehlern 20
3.4.3 Interaktion mit dem Nutzer 21
3.5 Idee zur Verbesserung der Übersicht und Optimierung: Zusammenfassen von Fehlermeldungen in Issues 22
3.6 Kennzeichnung von ähnlichen Fehlermeldungen 23
3.7 Paralleles Arbeiten mit verschiedenen Testreportquellen 23
3.8 Abgrenzung der Anforderungen 23
3.9 Zusammenfassung der Anforderungen an ein Programm zur automatischen Generierung von Issues aus Testreports 24

4 Konzept **25**
4.1 Automatisierung mit der Hilfe einer Pipeline 25
4.2 Hintergrund zur Nutzung von GitLab als Issue-Tracker 26
4.2.1 Übersicht über den Funktionsumfang von GitLab 26
4.2.2 Darstellung von Issues in GitLab 27
4.2.3 CI/CD: Automatisierung in GitLab 28
4.2.4 Einsatz von Docker-Container in der GitLab Pipeline 31
4.2.5 GitLab-Issue-Tracker als Datenbank 31
4.2.6 GitLab API: Zugriff auf alle Funktionen? 32
4.3 Cppcheck. Statische Codeanalyse als Testdatengenerator. 32
4.4 Wahl der Programmiersprache für die Implementierung des IssueMakers . 34
4.5 Übersichtliche Darstellung der Issues mit Hilfe von Markdown 35
4.6 Einsatz von Filtern zur Verhinderung von Duplikaten und irrelevanten Issues 36
4.7 Arbeitsablauf des Programms 36
4.8 Zusammenarbeit der Komponenten in der Pipeline 39

5 Implementierung **40**
5.1 Implementierung des Issue-Tracker-Interfaces für die Kommunikation mit der Gitlab-API 40
5.2 Der ReportManager als zentrale Komponente 42
5.3 Die Datenklasse Report: Issues in einem Array 43
5.4 Verschiedene Filter für die unterschiedlichen Herausforderungen 44
5.5 Parser: Das Lesen und Verstehen verschiedener Testreports 47
5.5.1 Parsen des Cppcheck Reports 48
5.5.2 Parsen des Reports des Regressionstest 49
5.6 Erfassen von Metainformationen für spätere Auswertungen 50
5.7 Der Issue, wie ihn GitLab sieht 51
5.8 Einbindung des Programms in die GitLab Pipeline 51

6 Ergebnisse **52**
6.1 Ausführung des IssueMakers. 52
6.2 Konfiguration und Allgemeiner Test mit Cppcheck 54
6.3 Definition der Testfälle auf Basis der Anforderungsanalyse 58
6.4 Ausführung der definierten Testfälle mit Hilfe von GitLab und Cppcheck . 62
6.4.1 Testfall 1 64
6.4.2 Testfall 2 65

6.4.3 Testfall 3 67
6.4.4 Testfall 4 69
6.4.5 Testfall 5 70
6.4.6 Testfall 6 71
6.4.7 Testfall 7 72
6.4.8 Testfall 8 75
6.4.9 Testfall 9 78
6.4.10 Testfall 10 80
6.4.11 Testfall 11 81
6.5 Ausführung mit Regressionstest 83
6.5.1 Testfall 1R 87
6.5.2 Testfall 2R 89
6.5.3 Testfall 3R 90
6.5.4 Testfall 4R 91
6.6 Parallele Ausführung des IssueMakers mit verschiedenen Testreport-Quellen in einer Pipeline 92

7 Fazit und Ausblick **95**
7.1 Ausbau der Filtermöglichkeiten des IssueMakers 95
7.2 Erweiterung des Funktionsumfangs durch weitere Parser 96
7.3 Alternativen zu GitLab als Issue-Tracker 96
7.3.1 YouTrack als Beispiel für einen GitLab Konkurrenten 96
7.4 Automatische Zuweisung der erstellten Issues 98
7.5 Automatische Aufwandsschätzung 99
7.6 Schlusswort 99

A Anhang **106**
A.1 Anleitung 106
A.1.1 Access Token 106
A.1.2 Access Token als CI Variable anlegen 107
A.1.3 Bereitstellung eines Runners für die Docker-Container 107
A.1.4 Pipeline erzeugen 108
A.1.5 Pipeline starten 111

Selbstständigkeitserklärung **112**

Abbildungsverzeichnis

4.1 Allgemeiner Aufbau einer Pipeline. 26
4.2 Arbeitsablauf des Programms. 37
4.3 Interface eines Reportparsers. 37
4.4 Interface eines Issue-Trackers. 37
4.5 Reportmanager. 38
4.6 Interface eines Filters. 38
4.7 Pipelineaufbau in GitLab. 39

5.1 Implementierung der Report-Wrapperklasse. 43
5.2 Implementierung der Parameter einer Cppcheck Fehlermeldung. 49
5.3 Implementierung aller Parameter eines Issues aus GitLab, die für die Erstellung zu beachten sind. 50

6.1 Darstellung einer erstellten Pipeline in GitLab. 56
6.2 Ausgabe des Terminals während der Ausführung des Issuemakers. 56
6.3 Darstellung eines erstellten Issues in GitLab. 57
6.4 Testfall 1: Ansicht der Pipeline in GitLab mit fehlgeschlagener Cppcheck-Stufe. 64
6.5 Testfall 1: Ausgabe der Konsole der Cppcheck-Pipeline-Stufe, wenn bei dem Aufruf in der .gitlab-ci.yml ein Pfad angegeben wird, an dem sich kein Programmcode befindet. 64
6.6 Testfall 2: Ansicht der Pipeline-Übersicht in GitLab nach dem erfolgreichen Durchlauf und mit fehlerfreiem Programmcode. 65
6.7 Testfall 2: Konsolenausgabe des IssueMakers im Falle eines leeren Issue-Trackers und eines leeren Testreports. 66
6.8 Testfall 3: Konsolenausgabe des IssueMakers nach erfolgreichem Lauf mit Fehlerfund. 68
6.9 Testfall 3: Ansicht des Issue-Trackers samt neu generiertem Issue. 68

6.10 Testfall 4: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der gefundene Fehler ist derselbe wie der, der bereits im Issue-Tracker vorhanden ist. Es wird kein neuer Issue im Issue-Tracker erstellt. 69
6.11 Testfall 5: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Issue-Tracker enthält einen Issue. Der neu erstellte Report findet keinen Fehler. Der irrelevant gewordene Issue wird geschlossen. 70
6.12 Testfall 6: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Report von Cppcheck enthält keinen Fehler. Im Issue-Tracker ist ebenfalls keiner vorhanden. 71
6.13 Testfall 7: Ein vom IssueMaker erstellter Issue in Gitlab. 73
6.14 Testfall 7: Der Issue aus Abbildung 6.13 ohne Beschreibung und geädertem Titel. 73
6.15 Testfall 7: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der gefundene Issue hat den gleichen Hash wie der geänderte Issue aus Abbildung 6.14 und wird als Duplikat verworfen. 74
6.16 Testfall 8: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Issue aus Abbildung 6.18 wurde als irrelevant erkannt, weil er keinen passenden Hash mehr besitzt und wurde geschlossen. 75
6.17 Testfall 8: Ein durch den IssueMaker erstellter Issue. Am Ende der Beschreibung ist der Hash markiert. 76
6.18 Testfall 8: Der Hash des Issues in Abbildung 6.17 wurde entfernt. 77
6.19 Testfall 9: Das Label „IssueMaker“ wurde dem Issue entfernt. 78
6.20 Testfall 9. Es existieren zwei Issues mit dem gleichen Inhalt aber verschiedenen Labeln. 78
6.21 Testfall 9. Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Issue mit dem fehlenden aus Abbildung 6.19 wurde nicht heruntergeladen. 79
6.22 Testfall 10: Es wurde ein zusätzlicher Testissue im Issue-Tracker erstellt. . 80
6.23 Testfall10: Ausgabe des Terminals nach der Ausführung des IssueMakers. Es wurde nur ein Issue heruntergeladen. 80
6.24 Testfall 11. Es wurde ein Issue mit den Labeln des IssueMakers erstellt. . 81
6.25 Testfall 11. Ausgabe des Terminals nach der Ausführung des IssueMakers. Beide Issues wurden aus dem IssueMaker geladen und der Testissue wurde als irrelevanter Issue geschlossen. 82
6.26 Beispiel einer Ausgabe des Regressionstest der Hochschule für Angewandte Wissenschaften (HAW) für das Software-Engineering Praktikum. 83

6.27 Testfall 1R: Konsolenausgabe des IssueMakers nach der Erstellung eines Issues basierend auf dem Report des Regressionstests 87
6.28 Testfall 1R: Darstellung eines mithilfe des IssueMakers erstellten Issues, der eine Meldung des Regressionstests enthält. 88
6.29 Testfall 2R: Konsolenausgabe des IssueMakers nach der Erkennung eines Duplikats basierend auf dem Report des Regressionstests. 89
6.30 Testfall 3R: Konsolenausgabe nach dem Test der Identifikation eines irrelevant gewordenen Issues. 90
6.31 Testfall 4R: Konsolenausgabe des IssueMakers nach einem erfolgreichen Regressionstest. 91
6.32 Parallele Ausführung von zwei Testtools und zwei IssueMaker Instanzen in einer GitLab Pipeline. 94

A.1 Profil-Einstellungen. 107
A.2 Menü der Profil-Einstellungen. 108
A.3 Erstellen eines Access Tokens. 109
A.4 Button zum Bestätigen der Eingaben. 109
A.5 Der erstellte Access Token erscheint über dem Formular. 110
A.6 Menü im Projekt. 110

Tabellenverzeichnis

3.1 Aufbau einer Fehlermeldung aus Basiswissen Softwaretest [37, S.199] . . . 14
3.2 Aufbau eines Issues nach IEEE-829. [17, S.60] 15
3.3 Aufbau eines Issues nach IEE29119. [29, S.45 ff.] 16

4.1 Aufbau eines Issues in GitLab. [14] 27
4.2 Vergleich der Attribute eines Fehler Reports nach IEEE-29119 [29] und eines GitLab-Issues [14] 29
4.3 Aufbau eines Fehler-Elements des XML-Reports von Cppcheck. 34
4.4 Lokalisierungsattribute einer Cppcheck-Fehlermeldung. 34

6.1 Testfälle 60
6.2 Fortsetzung der Testfälle 61
6.3 Testfälle des Regressionstests 86

Listings

4.1 Beispiel einer .gitlab-ci.yml, die eine Pipeline erzeugt. 28
4.2 Aufruf von Cppcheck mithilfe eines Terminals. 32
4.3 Auzug eines Reports, der von Cppcheck erstellt wurde. 33

6.1 Start des IssueMakers mit Hilfestellung. 52
6.2 Konsolenausgabe des IssueMakers mit Hilfestellung. 52
6.3 Start des IssueMakers mit Parametern. 53
6.4 Main.cpp. 54
6.5 SomeClass.h. 55
6.6 SomeClass.cpp. 55
6.7 .gitlab-ci.yml des GitLab-Projects mit Cppcheck und IssueMaker. 62
6.8 Programmcode ohne für Cppcheck identifizierbare Mängel. 65
6.9 Main.cpp. 67
6.10 Konfigurationsdatei .gitlab-ci.yml für den Regressionstest 84
6.11 Konfigurationsdatei .gitlab-ci.yml für den Regressionstest 92

A.1 Beispiel einer .gitlab-ci.yml, die eine Pipeline erzeugt. 108

1 Einleitung

Im Softwareentwicklungsprozess wird viel Energie und Zeit in das Testen von Programmcode investiert. Der Testprozess beginnt möglichst früh und der Abnahmetest bildet den letzte Schritt vor der Auslieferung eines Programms oder Moduls an einen Kunden. Um den Aufwand der eigentlichen Testausführung zu reduzieren, wird viel Zeit in die Automatisierung von Tests investiert.

Am Ende eines automatisierten Testdurchlaufs steht ein Report. Dieser enthält neben Informationen über erfolgreiche und fehlgeschlagene Tests auch Metainformationen wie den Zeitpunkt der Prüfung. Die Ergebnisse sind für den Entwickler relevant, weil sie Hinweise auf mangelhaften oder nicht funktionierenden Code liefern. Im Entwicklungsprozess wird für die Bearbeitung eines fehlgeschlagenen Tests ein sogenanntes Issue (vgl. Abschnitt zu Issues auf Seite 3.3) in einem Issue-Tracker (vgl. Abschnitt zu Issue-Trackern auf Seite 4.2.5), wie zum Beispiel dem Issue-Tracker von GitLab, erstellt. Das Erstellen der Issues bedeutet Aufwand für die Person, die den Issue formulieren muss.

Gegenstand der Arbeit soll daher die Frage sein, inwiefern die Automatisierung nach der Erstellung des Testreports weitergehen kann. Die meist strukturierten Reports sollen von einem Programm eingelesen werden. Dieses übernimmt im Anschluss die Aufgabe aus den Informationen Issues zu erstellen. Diese werden den Entwicklern über den Issue-Tracker zur Verfügung gestellt.

Des Weiteren soll erörtert werden, welche Hindernisse diese Automatisierung überwinden muss und welche Probleme dabei auftreten können. Als Basis für die Testreports werden die Ergebnisse der statischen Analyse durch Cppcheck (S. 32) und eines Regressionstests genutzt. Diese werden in einer Continious-Integration-Pipeline durchgeführt. Im nächsten Pipeline-Schritt werden die Testreports in Issues des jeweiligen Issue-Trackers überführt.

Damit das Programm den Issue-Tracker nicht mit redundanten Issues befüllt, muss es die Fähigkeit besitzen, bereits erstellte Issues mit den Funden der Testreports zu vergleichen.

Außerdem kann es passieren, dass die Fehlerwirkungen durch Seiteneffekte hervorgerufen worden sind, die im Rahmen eines anderen Issues behoben wurden. Dies führt in bestimmten Situationen zu irrelevanten Issues, welche im Issue-Tracker entfernt werden sollten. Das Programm muss also Duplikate vermeiden (S. 18) und sollte irrelevante Issues entfernen (S. 20). Das Erkennen von Duplikaten und Säubern des Issue-Trackers sind bisher Aufgaben, die von Menschen ausgeführt werden müssen.

Zu Beginn dieser Arbeit wird ein Überblick über den Stand der Forschung und ähnlichen Ansätze zu diesem Themengebiet gegeben (S. 3). Im Anschluss werden auf Basis dieser Recherche Anforderungen für ein Programm ermittelt (S. 10). Anschließend ist das Konzept für deren Umsetzung dokumentiert (S. 25) und darauf aufbauend wird die Implementierung erläutert (S. 40). Die Implementierung wird am Ende mit verschiedenen Testreports geprüft (S. 52).

Im letzten Teil wird sich mit den Ergebnissen zusammengefasst und kritisch hinterfragt. Es wird betrachtet, ob sich das entwickelte Programm auch auf andere Issue-Tracker als GitLab anwenden lässt. Außerdem werden Erweiterungsmöglichkeiten diskutiert (S. 95).

2 Verwandte Arbeiten

Da im Rahmen dieser Arbeit ein Programm entwickelt wird, wird im Folgenden nicht nur Bezug auf verwandte wissenschaftliche Arbeiten genommen, die sich mit Aspekten dieser Arbeit beschäftigt haben, sondern es wird im zweiten Abschnitt auch eine Diskussion über bereits bekannte Programme geben, die sich in diesem Bereich finden.

2.1 Wissenschaftliche Arbeiten zum Thema Issues

In der Wissenschaft wird sich heute umfangreich mit dem Thema Issues auseinandergesetzt. Dabei wird nicht nur auf den Aufbau eingegangen, sondern vor allem auf den Umgang mit Duplikaten in den Issue-Trackern oder auch Bug-Repositories.

John Anvik, Lyndon Hiew und Gail C. Murphy haben die offenen Bug Repositories von Eclipse und Firefox untersucht [38]. Dabei kam heraus, dass Duplikate und der Umgang mit ihnen ein großes Problem in öffentlichen Bug Repositories sind, weil es sehr viele unterschiedliche Nutzer gibt, die ähnliche Probleme melden. Die Erkennung der Duplikate wird von sogenannten „Triagers“ gemacht. Das sind Personen, die sich sehr gut im Repository auskennen und mit diesem Wissen Duplikate schnell identifizieren können. Es wird weiter erörtert, ob die Duplikatenerkennung automatisiert werden kann. Mithilfe von Machine Learning sollen Nutzer darauf hingewiesen werden, dass es bereits ähnliche Bug-Reports gibt.

Xiaoyin Wang, Lu Zhang, Tao Xie haben zusammen mit John Anvik und Jiasu Sun die Arbeit an der automatischen Erkennung von Duplikaten fortgesetzt [49]. Es wird die Beschreibung der Bug-Reports mit Hilfe der kombinierten Anwendung aus Natural-Language-Processing und Execution-Information verglichen, um doppelte Bug-Reports zu identifizieren. Dieser Ansatz wird anhand der Repositories von Firefox und Eclipse getestet. Tatsächlich sind sie erfolgreich gewesen und konnten eine signifikante Anzahl an

Duplikaten identifizieren. Sie haben sich allerdings stets auf von echten Nutzern erstellte Bug-Reports bezogen. Im Gegensatz dazu wird in dieser Arbeit die Untermenge der Duplikate betrachtet, die automatisch erstellt wurde. Den Formulierungen der Fehlermeldungen fehlt es an Individualität, weshalb komplexe Verfahren wie Natural-Language-Processing zu aufwendig wären.

Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann und Sunghun Kim betrachten die Frage, ob Duplikate nicht sogar Mehrwert bieten [41]. Sie schreiben, dass Duplikate oft zusätzliche Informationen enthalten können, weil Bug-Reports, die denselben Fehler beschreiben, oft nicht identisch formuliert sind. Sie wurden in der Regel von unterschiedlichen Nutzern geschrieben, die einen anderen Blickwinkel auf das Problem besitzen. Weiter wird darauf eingegangen, warum unterschiedliche Nutzer überhaupt ein Duplikat erstellen und wie dies verhindert werden kann. Bei automatisch erstellten Issues ist das Format allerdings immer gleich und der Auslöser für Duplikate ist nicht der Nutzer, sodass die Behandlung und Beurteilung anderen Kriterien unterliegen muss.

Sarah Rastkar, Gail C. Murphy und Gabriel Murray betrachten die Möglichkeit, Bug-Reports zusammenzufassen [45]. Dies soll helfen, wenn es zu einem aktuellen Issue bereits ähnliche Bug-Reports in der Vergangenheit gab. Damit der Entwickler nicht die gesamte Historie des vergangenen Bug-Reports erarbeiten muss, soll eine Zusammenfassung erstellt werden, die den Aufwand dafür stark verringert. Zusammenfassungen können also beim Erkennen von Duplikaten helfen. Es wurde sich hier auf die Vorteile beschränkt, Bug-Reports zusammenzufassen, die von Menschen erstellt wurden. Die bekannte Struktur und das Hinzufügen von eindeutigen Identifikationsmöglichkeiten bei automatisch erstellten Bug-Reports, macht das Zusammenfassen von ähnlichen Fehlermeldungen in einem Issue unter Umständen leichter.

Dass Zusammenfassungen unter bestimmten Voraussetzungen helfen können, Duplikate zu verhindern, zeigen auch Hongying Gu, Long Zhao und Chang Shu [48]. Durch das Erzeugen von Zusammenfassungen von bereits erstellten Bug-Reports können Nutzer Vorschläge für ähnliche Bug-Reports bekommen, während sie einen eigenen erstellen. Dies soll die Nutzer dazu bewegen, selbst zu prüfen, ob es bereits identische oder sehr ähnliche Bug-Reports gibt und ob sie diese mit ihren Informationen erweitern können.

Ruchika Malhotra und Laavanye Bahl haben ein Programm entwickelt, welches alle Commits eines Git-Repositories kategorisiert [44]. Interessant sind die Dateien aller Commits, die eine Lösung eines zuvor erkannten Defekts beinhalten. Die Namen der Dateien werden in einer Excel-Tabelle samt der Commit-Message abgelegt. Zusätzlich wird eine statische

Code-Analyse, deren Ergebnisse auch in der Tabelle eingetragen werden, auf diese Dateien angewendet. Die Daten können dann für weitere Analysen verwendet werden, wie zum Beispiel die Vorhersage von fehlerhaften Klassen oder Benchmarks der Softwarequalität. Aus diesen Informationen können Issues erstellt werden, die darauf hinweisen, dass bestimmte Programmcodeabschnitte überarbeitet werden müssen.

2.2 Betrachtung verwandter Programme im Bereich des automatisierten Testens

Neben den wissenschaftlich dokumentierten Arbeiten gibt es bereits einige Programme für die Verbindung von Analyse der Testreports und Erstellung von Issues in einem Issue-Tracker. Sie verfolgen unterschiedliche Ansätze und Automatisierungsgrade. Manche dieser Programme führen selbst Prüfungen durch, andere verwalten eigene Fehlerdatenbanken. Einige bieten eine Integration in bestehende Issue-Tracker, aber die Spezialisierung auf bestimmte Prüfmechanismen schränkt die Nutzungsmöglichkeiten ein.

Für Webanwendungen gibt es Programme, die es erlauben, Issues in einem Issue-Tracker zu dokumentieren, wenn Fehler manuell gefunden wurden. Ein Beispiel für ein solches Programm ist Ybug [33]. Fehler, die bei einer Inspektion oder Review gefunden werden, können mithilfe dieser Browser-Erweiterung direkt an Asana, Slack, Trello, GitHub, GitLab und weitere geleitet werden. Es lassen sich nur Fehler in Webanwendungen verwalten. Außerdem testet es nicht automatisch. Es eignet sich also nicht für eine komplette Automatisierung des Testprozesses.

Ein ähnliches Konzept verfolgt Marker.io [32]. Es ist ebenfalls eine Browser-Erweiterung und somit nur auf Webanwendungen anwendbar. Mit Marker.io kann man per Knopfdruck ein Issue auf GitHub, GitLab, Jira und Bitbucket erstellen. Es hilft Entwicklern, die Zugriff auf das Repository von Webanwendungen haben, indem Screenshots und weitere Informationen direkt in einem Issue gepflegt werden können, wenn ein Fehler oder Mangel entdeckt wird. Somit wird zumindest ein Teil der Arbeit automatisiert.

Eine andere Art dieser Issue-Generatoren sind Reporting-Tools, wie sie heute in fast jeder Software zu finden sind. Sie ermöglichen das optionale Versenden von Fehler-Logs, nachdem ein Fehler aufgetreten ist. Manuell oder auch automatisch erstellte Fehler-Logs

können mithilfe dieser Systeme an ein Bug-Repository gesendet werden. Bei vielen Programmen und sogar bei Betriebssystemen wird der Nutzer bei einem Absturz gefragt, ob der Fehler-Log an den Hersteller der Software gesendet werden darf.

Für IDEs gibt es Erweiterungen, um direkt aus der Entwicklungsumgebung heraus Issues zu erstellen, wie zum Beispiel der Atlassian Connector [3]. Er wird von Atlassian für verschiedene IDEs bereitgestellt und erlaubt es über die Entwicklungsumgebung in Jira Tickets zu erstellen.

Diese Erweiterungen oder Features haben den Nachteil, dass die eigentliche Erstellung der Issues per Hand geschehen oder zumindest zugestimmt werden muss.

Daneben gibt es eine Reihe von Programmen, die es erlauben, Fehlermeldungen per E-Mail zu versenden.

Bugzilla [5] bietet neben diversen Features auch die automatische Erkennung von doppelten Bugs und es kann diese entsprechend filtern. Es kann zwar nicht selbst Testreports von anderen Tools einlesen, aber es kann in diverse IDEs, wie zum Beispiel Eclipse, als Plugin hinzugefügt werden. In der IDE erkannte Bugs können dann an das Bugzilla Bug-Repository gesendet werden und können von dort weiterverarbeitet werden. Für dort erkannte Fehler können dann Benachrichtigungen per E-Mail versendet werden. Ein Nachteil von Bugzilla ist das Alter. Es wird zwar ständig weiterentwickelt, aber die Bedienung und das Design sind trotzdem nicht mehr zeitgemäß und es bietet von Haus aus keine Integration in modernere Systeme wie GitLab und Jira.

Ein weit verbreitetes Programm, dass ebenfalls E-Mails versenden kann, ist SonarQube [30]. Es ist zum Testen der Codequalität von verschiedenen Programmiersprachen entwickelt worden. Es lässt sich gut in den Entwicklungsprozess einbinden, indem man sogenannte Quality-Gates setzt. Nur wenn der Code die Bedingungen, die in diesen Gates festgeschrieben sind, erfüllt, darf der Programmcode in den produktiven Teil übernommen werden. Zusätzlich ist es mit SonarQube möglich das Hinzufügen von Programmcode zu verbieten, wenn die Qualitätsstandards nicht eingehalten wurden. Ein Nachteil ist, dass es sich nicht mit jedem Issue-Tracker verbinden lässt, sodass Fehler nicht im Issue-Tracker, sondern in SonarQube angezeigt werden. Außerdem ist es auf die Codequalität beschränkt.

Die Fähigkeit, E-Mails zu versenden, ist eine Funktion, die sich zur Weiterverarbeitung eignet. Wenn ein Testtool, für jeden Fund eine E-Mail versenden kann und ein Issue-Tracker aus E-Mails eigene Issues generieren kann, könnte man aus einer Kombination

von beidem eine Automatisierung des Prozesses erreichen. Neben GitLab [13] kann auch Jira [18] aus E-Mails Issues generieren. Der Issue-Tracker von Atlassian erlaubt dies allerdings nur mit einigem Aufwand. Es muss nicht nur der E-Mail-Server, sondern auch ein E-Mail-Handler eingestellt werden. Zusätzlich müssen die E-Mails ein bestimmtes Format besitzen. Das Testtool muss also erlauben, dass man die Funde entsprechend formatiert. Das Versenden von E-Mails könnte als Brücke fungieren zwischen Testtool und Issue-Tracker.

Neben der Kombination aus Tests und Issue-Tracker gibt es auch Software, die Code prüft und eigene Fehlerdatenbanken führt. Sie ist also entkoppelt von spezialisierten Issue-Trackern.

FindBugs [9] ist ein Projekt der Universität Maryland. Es kann ausschließlich auf Java-Code angewendet werden. FindBugs gehört zu den statischen Analysatoren. Gefundene Fehler werden im eigenen Cloud-Storage gehalten. Dies ist ähnlich zu einem Issue-Tracker. Es ist aber ansonsten nicht mit anderen Trackern ohne weiteres kompatibel.

Etwas entfernt vom eigentlichen Erstellen von Issues in einem Issue-Tracker sind Programme, deren Funktionalität sich auf das Testen beschränkt, dabei aber Attribute besitzen, welche es ermöglichen mit wenig Aufwand aus den gefundenen Mängeln neue Issues zu generieren.

Pmdtester [26] ist ein Tool, welches zum PMD Source Code Analyzer Projekt gehört. Zu diesem Projekt gehört auch PMD selbst, welches ein Tool zur statischen Code-Analyse ist. Der Pmdtester ist ausschließlich für Regressionstests geeignet. Das Tool lässt sich nur in eine Travis-CI-Pipeline integrieren und speichert fertige Reports in einem lokalen Verzeichnis. Es ist also in der Automatisierung beschränkt und kann nur über Travis-CI mit einem Issue-Tracker verbunden werden.

Zu den umfangreichsten Testmanagement-Programmen gehören ReportPortal, TestRail und Azure DevOps. ReportPortal ist ein Programm zur Unterstützung von Continuous-Integration und Continuous-Testing. Es bietet eine Integration zu Jira und Jenkins und diversen Unit-Test-Tools. Außerdem lässt es sich beliebig erweitern, da das Projekt der GNU General Public License v3.0 unterliegt [22]. Mithilfe des Zugriffs auf die verschiedenen Tools, wird eine Übersicht über die Testprozesse geboten. Die wichtigste Eigenschaft von ReportPortal ist die automatische Analyse aktueller und historischer Daten. Damit ist es dem Tool möglich dem Anwender eine Empfehlung zu machen, ob ein Issue zu erstellen ist und welcher Entwickler sich für die Bearbeitung eignen könnte. Wird ein

Fehler gefundene, erkennt das ReportPortal anhand der Zugriffe auf die unterschiedlichen Plattformen, ob dieser Fehler bereits in Bearbeitung ist. Außerdem können abstraktere Fehler zu bestimmten Commits zugeordnet werden und so die Suche nach der Ursache des Problems einschränken. Ermöglicht wird diese Analyse durch Machine-Learning [19].

TestRail bietet eine Übersicht und Auswertungsmöglichkeiten sowie Integration mit verschiedenen Testtools und Jira. Ähnlich wie bei ReportPortal können Übersichten und Diagramme erstellt werden, die die Arbeit der Tester übersichtlicher gestalten. TestRail bietet allerdings keine Empfehlungen auf Basis von Machine-Learning-Algorithmen und es muss von einem Tester manuell bestätigt werden, dass für einen fehlgeschlagenen Test ein Issue erstellt wird [20].

ReportPortal und TestRail haben zwar Zugriff auf die Issue-Tracker und können dort Issues erstellen, aber sie machen das nicht automatisch, sondern erst nach Bestätigung durch einen Nutzer.

Keines der betrachteten Programme so umfangreich wie Azure DevOps [4]. Es bietet ein ähnliches Konzept und Umfang wie GitLab. Außerdem bietet es mit Test Plans ein Testmanagement-Tool, wie TestRail und ReportPortal. Mit dessen Hilfe ist es möglich, gefundene Fehler mit den Anforderungen abzugleichen und daraus Issues zu generieren. Man ist allerdings auf diese Funktion beschränkt und kann zur Zeit dieser Arbeit nicht beliebige Testtools mit Azure DevOps verwalten.

2.3 Fazit der Betrachtung und Einordnung der Arbeit.

Fast alle genannten Tools beschränken sich auf einen kleinen Bereich der Testautomatisierung. So schränken sie Plattformen oder Programmiersprachen auf die sie angewendet werden können ein. Auch die Integration in den Arbeitsprozess ist oft eingeschränkt. Entweder ist eine Verbindung nur zu bestimmten Issue-Trackern möglich oder die Verbindung benötigt zusätzlichen Aufwand durch den Umweg über das Versenden von E-Mails. Einige müssen per Hand gestartet und automatisieren nur kleine Teile der Erstellung der Issues in dem sie zum Beispiel Pflichtfelder automatisch befüllen oder Anhänge wie Screenshots verwalten.

Es ist außerdem möglich, dass eine Software nur integriert getestet werden kann. Besonders in der Welt der Echtzeitsysteme ist Software oft auf Sensoren und Aktoren, die entweder simuliert oder bereitgestellt werden müssen, angewiesen. Eine automatisierte

Test-Pipeline und automatisch erstellte Issues für fehlgeschlagene Tests sind hier hilfreich. Es gibt kein Programm, welches flexibel genug ist verschiedene Testreports zu lesen und daraus automatisch in verschiedenen Issue-Trackern neue Issues zu erstellen.

3 Anforderungsanalyse

Das Programm, welches im Rahmen dieser Arbeit entwickelt werden soll, soll flexibel genug sein, um auch individuell entwickelte Testtools und deren Testreports lesen zu können und daraus in verschiedenen Issue-Trackern Issues zu erstellen. Im folgenden wird es als IssueMaker bezeichnet.

Nachdem ein Testreport erstellt wurde, wird er vom IssueMaker eingelesen. Die gefundenen Fehler werden mit denen, die bereits im Issue-Tracker vorhanden sind abgeglichen, sodass keine Duplikate erzeugt werden. Neue Fehler aus dem Testreport werden automatisch als Issues im Issue-Tracker erstellt.

Im folgenden werden die Anforderungen an das Programm herausgestellt. Auf deren Basis kann das System und die einzelnen Komponenten aufgebaut werden. Der Fokus der Anforderungen wird auf den Einsatzbereich und die Einsatzumgebung gelegt. Außerdem werden für bekannte Herausforderungen Konzepte diskutiert.

3.1 Wirtschaftlicher Hintergrund der Softwarequalität und Automatisierung

Softwarequalität ist ein entscheidener Faktor für den Erfolg von Programmen [37]. Deshalb ist Testen ein wichtiger Bestandteil der Softwareentwicklung und Softwarewartung. Laut Martin Pohl liegt der Anteil des Testens am Budget eines Softwareprojekts bei 30 bis 40 Prozent [37]. Er hebt besonders das strukturierte Vorgehen, eine angemessene Organisationsstruktur sowie entsprechende Infrastruktur hervor.

> Da Time-to-Market, Wettbewerb, Globalisierung und Quality of Service, einschließlich der Qualität von Softwaresystemen, für viele Firmen zu einem wichtigen Überlebensfaktor geworden sind, steigt der Bedarf nach angemessenen Testprozessen immer weiter an. Sowohl die immer größerer Bedeutung

> von Software in unserer Gesellschaft als auch das Budget, das für das Testen ausgegeben wird, bestätigen den Bedarf an einem gut strukturierten und verlässlichen Testprozess innerhalb der Softwareentwicklung. Hierbei sind ein strukturiertes Vorgehen, eine angemessene Organisationsstruktur und die entsprechende Infrastruktur notwendig. [37, xxvii]

In einem Artikel zu Techniken des Softwaretestens von Lu Luo geht der Autor sogar noch etwas weiter.

> The testing of software is an important means of assessing the software to determine its quality. Since testing typically consumes 40 - 50 % of development efforts, and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering. [43]

Eine Automatisierung der Tests kann laut Luo den gesamten Testaufwand verringern und somit Geld einsparen [43]. Außerdem werden Fehler früher gefunden, was im Verlauf des Entwicklungsprozesses Kosten einsparen kann.

3.2 Hintergrund der Automatisierung des Übertragungsprozesses von Testreport zu Issue

Verschiedene Arten von Tests können bereits automatisiert ausgeführt werden, wie beispielsweise Komponententests oder die statische Code-Analyse, die bereits in Teilen von manchen Entwicklungsumgebungen während der Programmierung ausgeführt wird. Mit der Hilfe von automatischen Tests kann die Continuous Integration eines Projekts voran getrieben werden. Eine Regelung könnte beispielsweise sein, dass wenn nach einer Codeänderung alle Tests des automatischen Testprozesses erfolgreich gewesen sind, wird ein Update des bestehenden Systems gemacht. Fehler werden so früher gefunden und können leichter den entsprechenden Codeänderungen zugewiesen werden [42].

Am Ende eines einzelnen Tests steht das Testergebnis. Wenn aber mehrere Tests einer Testsuite ausgeführt wurden, steht am Ende ein Testreport, der aus mehreren Testergebnissen besteht. Entwickler benötigen die Testergebnisse, um Fehler identifizieren und beheben zu können. Da Programmierer ihre Aufgaben heute häufig Software-gestützt erhalten, müssen die Ergebnisse des Reports in einen Issue-Tracker geschrieben werden. Von dort holen sich die Entwickler einzelne Issues, um diese nacheinander zu bearbeiten.

Das Übertragen der Testergebnisse in Issues muss per Hand gemacht werden. Dies kostet Zeit und Geld.

Das Übertragen der negativen Testergebnisse in einen Issue-Tracker ist aus verschiedenen Gründen wichtig. Vor allem bietet der Issue-Tracker eine zentrale Datenbank, die ausgewertet werden kann und für zukünftige Aufwandsplanungen und Recherchen etc. verwendet werden kann.

Für den späteren Prozessablauf gilt, dass nachdem ein Testlauf beendet und ein Testreport erstellt wurde, der Testreport gelesen wird und auf dessen Basis Issues im entsprechende Issue-Tracker erstellt werden.

3.3 Übersicht und Empfehlungen zum Aufbau und Inhalt eines Bug-Reports

Für die Übertragung der Testergebnisse in ein Issue sollte der Inhalt in einem Format aufgebaut sein, dass es dem Entwickler leicht macht, die nötigen Informationen für die Bearbeitung schnell zu erfassen. In einer Umfrage von Bettenburg, Just, Schröter, Weiss, Premaj und Zimmermann [40] aus dem Jahr 2008 wurden Entwickler und Reporter gefragt, was einen guten Bug-Report ausmacht. Entwickler nutzen beobachtetes und erwartetes Verhalten, Testfall, Stacktraces und die Schritte zur Reproduktion am meisten. Am wichtigsten sind ihnen dabei die Schritte zur Reproduktion von Fehlern. Diese können genutzt werden, um mithilfe des Debuggers die Ursache für das Fehlverhalten schnell identifizieren können. Sie gaben aber an, dass unvollständige Informationen und Duplikate sowie Fehler in den Schritten zum Reproduzieren die größten Probleme machen und viel Zeit kosten.

Fehlerreporter geben am häufigsten das beobachtete und erwartete Verhalten an. Nur wenige haben den Stacktrace oder Testfall angegeben, was daran liegt, dass diese für Reporter in der Regel schwer zu beschaffen sind. Die Schritte zur Reproduktion des Fehlerfalls und die Testfälle bewerten sie zwar am nützlichsten für Entwickler, aber auch am schwersten zu liefern. Diese Informationen beziehen sich nicht auf automatisch generierte Bug-Reports, aber sie zeigen, was den Entwicklern am besten hilft, um Fehler zu beheben.

Der IssueMaker sollte entsprechende Informationen aus Testreports übernehmen und nach Nutzen für die Entwickler anordnen. Die Anforderung an die automatische Generierung von Issues ist also, dass die Informationen der Testreports sinnvoll und übersichtlich dargestellt werden. Auf den Inhalt und Umfang der Informationen kann kaum Einfluss genommen werden, weil sie vom Testtool abhängig sind.

Im Buch Basiswissen Softwaretest [37] wird für eine Fehlermeldung ein Aufbau wie in Tabelle 3.1 vorgeschlagen. Anforderung, Fehlerquelle, Testfall und Problem würden Informationen zur Reproduzierbarkeit, dem beobachteten und erwarteten Verhalten sowie den Stacktrace enthalten, falls diese Informationen geliefert werden können. Zur Identifikation von möglichen Duplikaten, falls diese bekannt sind, können die Verweise herangezogen werden. Die restlichen Informationen bieten Hilfen in der Verwaltung. Die laufende Meldungsnummer kann als Datenbankindex genutzt werden und die Priorität kann Auswirkungen auf den Zeitpunkt der Bearbeitung haben. Das Buch orientiert sich an dem IEEE Standard Nummer 829 für Software und Systemtest Dokumentation[17]. Es gibt aber einige Anpassungen.

Der Aufbau eines Issues nach IEEE-829 ist in 3.2 dargestellt. Dort wird für einen „Anomaly-Report“ oder Issue folgende Definition geliefert.

> The purpose of the AR is to document any event that occurs during the testing process that requires investigation. This may be called a problem, test incident, defect, trouble, issue, anomaly, or error report. [17, S.60]

Zum ersten Einordnen des Fehlerfundes ist hier das Feld Zusammenfassung (Summary) vorgesehen. Informationen zur Reproduzierbarkeit werden hier in der Beschreibung (Description of the anomaly) benannt. So werden auch die Informationen zum erwarteten und beobachteten Verhalten aufgeführt. In den Issues nach der IEEE-829 wird vorgesehen, dass auch Informationen zur Fehlerbehebung vom Entwickler angegeben werden. Dafür ist das Feld „Description of the corrective action“ vorgesehen. Inzwischen wurde dieser Standard durch den IEEE-29119 abgelöst[29].

Attribut	Beschreibung
Nummer	laufende, eindeutige Meldungsnummer
Testobjekt	Bezeichnung des Testobjekts
Version	Identifikation der genauen Version des Testobjekts
Plattform	Identifikation der HW-/SW-Plattform bzw. der Testumgebung, in der das Problem auftritt
Entdecker	Identifikation des Testers, der das Problem meldet
Entwickler	Name des für das Testobjekt verantwortlichen Entwicklers oder des Teams
Erfassung	Datum, ggf. Uhrzeit, an dem das Problem betrachtet wurde
Status	Bearbeitungsfortschritt der Meldung; möglichst mit Kommentar und Datum des Eintrags
Klasse	Klassifizierung der Schwere des Problems
Priorität	Klassifizierung der Dringlichkeit der Korrektur
Anforderung	Verweis auf die (Kunden) Anforderung, die wegen der Fehlerwirkung nicht erfüllt oder verletzt ist
Fehlerquelle	Soweit feststellbar, die Projektphase, in der die Fehlerhandlung begangen wurde
Testfall	Beschreibung des Testfalls bzw, der Schritte, die zur Reproduktion der Fehlerwirkung führen
Problem	Beschreibung der Fehlerwirkung; vorausgesagte vs. tatsächliche Ergebnisse bzw. Verhalten
Kommentar	Stellungnahme der Betroffenen zum Meldungsinhalt
Korrektur	Korrekturmaßnahmen des zuständigen Entwicklers
Verweis	Querverweise auf andere zugehörige Meldungen

Tabelle 3.1: Aufbau einer Fehlermeldung aus Basiswissen Softwaretest [37, S.199]

Attribut	**Beschreibung**
Document Identifier	Uniquely identify a version of the document
Scope	Briefly describe any contextual information not covered elsewhere in the AR that is needed to make this AR understandable.
References	List all of the applicable reference documents.
Summary	Summarize the anomaly.
Date abomaly discovered	Record the date (and possibly also the time) that the anomaly was first identified.
Context	Identify the software or system item (including any version numbers), or software or system configuration item, and/or the software or system life cycle process in which the anomaly was observed.
description of anomaly	Provide a description of the anomaly. Indicate whether the anomaly is reproducible, and provide enough information to make it reproducible if it is.
Impact	Indicate (if known) the depth and breath of the impact this anomaly will have on technical and business issues
Originiator's assessment of urgency	Provide an evaluation of the need for an immediate repair.
Description of the corrective action	Summarize the activities during the corrective action taken to resolve the reported anomaly.
Status of the anomaly	Identify the current status of the anomaly.
Conclusion and recommendations	Specify any recommendations for changes to the development and/or testing processes and documentation that would help to prevent this kind of anomaly in the future.
Document change procedures and history	Specify the means for identifying, approving, implementing, and recording changes to the master test process.

Tabelle 3.2: Aufbau eines Issues nach IEEE-829. [17, S.60]

Attribut	Beschreibung
Timing information	Records the date (and possibly also the time) when the incident was first observed.
Originator	Specifies the name(s) and title(s) of the individual(s) who identified the incident.
Context	Identifies the context in which the incident was observed.
Description of the incident	Provides a detailed description of the incident.
Originator's assessment of severity	Indicates, from the originator's point of view, the depth and breadth of the impact this incident will have on technical and business issues.
Originator's assessment of priority	Provides an evaluation of the urgency for the repair.
Risk	Provides information on the introduction of new risks or changes to the status of existing risks, where applicable.
Status of the incident	Identifies the current status of the incident, which will be "Open" or similar in this context.

Tabelle 3.3: Aufbau eines Issues nach IEE29119. [29, S.45 ff.]

Hier wird für einen „Incident Report“ oder Issue folgende Definition geliefert.

> A test incident is any issue that is noticed during testing that requires action(s) to be documented. Test incidents are recorded in incident reports. There will be one incident report for each unique incident (incident reports may also be known as defect reports, Bug-Reports, fault reports, etc.).[29, S.45]

Der Aufbau eines Incident Reports nach IEEE-29119 ist in Tabelle 3.3 dargestellt. Er hat weniger Attribute als der Issue nach IEEE-829 und die Informationen zur Reproduzierbarkeit und zum Verhalten sind wieder in der Beschreibung des „Incidents“ unterzubringen. Da Issues in der Regel bereits in projektspezifischen Datenbanken gehalten werden, wurde hier auf entsprechende Metainformationen verzichtet. Dieser Aufbau ist der modernste der drei vorgestellten, deshalb sollte sich der IssueMaker an ihm orientieren.

3.3.1 Aufbau und Merkmale des Testreports auf Basis des Incident Reports von IEEE-29119

Der IssueMaker muss den Report lesen können, um auf dessen Inhalt basierend Issues zu erstellen. Im Idealfall sind alle Reports in einem identischen Format, das leicht durch Software zu lesen ist. Das Format sollte sich deshalb an dem des IEEE-29119 Standards in der Tabelle 3.3 orientieren.

Es sollte gelten, dass einzelne Fehler samt ihrer Informationen eindeutig zu erkennen sind und es nicht passieren kann, dass sich Informationen von unterschiedlichen Tests miteinander vermischen. XML [23] und JSON [21] bieten hier die nötigen Fähigkeiten, um einen Testreport so zu formatieren, dass die Fehlerfälle einzeln identifiziert werden können und der IssueMaker diese einlesen kann. Die Formate sind zudem auch für Menschen lesbar. Der IssueMaker muss den Ort und Pfad zu dem Report kennen. Außerdem müssen entsprechende Lesezugriffsrechte eingeräumt werden.

Es ist nicht immer möglich das Format des Reports vorzugeben. Deshalb ist eine Anforderung an das Programm, dass es beliebig erweitert werden können muss, um beliebige Testreports lesen zu können.

Ein Testreport kann auch erfolgreich abgeschlossene Testfälle beinhalten. Wenn ein solcher Report vorliegt, muss der IssueMaker dies erkennen können. Ansonsten würde er Issues für Testfälle erstellen, die keinen Bedarf für eine genaue Inspektion durch den Entwickler haben. Für das Programm ist es also wichtig, dass es fehlgeschlagene Testfälle in einem Testreport erkennen kann, oder dass der Testreport, auf den der IssueMaker Zugriff erhält, keine oder nur negative Testfälle beschreibt. Dies muss bei der Implementierung des Moduls zum Lesen des Reports beachtet werden.

3.4 Erwartete Herausforderungen

Bei der automatischen Generierung kommt es zu mehreren Herausforderungen. Als Erstes ist die Erzeugung von Duplikaten zu verhindern. Außerdem kann es dazu kommen, dass Meldungen nicht mehr an der angegebenen Stelle zu finden sind, weil beispielsweise Programmcode eingefügt wurde. Ein weiterer wesentlicher Punkt ist die Interaktion mit Nutzern des Issue-Trackers.

3.4.1 Duplikate

Die Möglichkeit, dass der Prozess unter verschiedenen Umständen erneut gestartet werden kann, ist wichtig, damit er den Softwareentwicklungsprozess nicht einschränkt. Allerdings erzeugt es auch eine große Herausforderung. Wenn nicht alle Issues behoben wurden, bevor die Pipeline erneut gestartet wird, kann es zu Duplikaten kommen. Der IssueMaker muss erkennen können, welche Issues er selbst erstellt hat und diese dann mit den neu erstellten abgleichen können.

Bei der Identifizierung von Duplikaten gibt es diverse Herausforderungen. Es beginnt bei der Frage, wie es eigentlich zu Duplikaten kommen kann und endet bei der Frage, wie damit umgegangen werden soll. Die Identifizierung ist deshalb schwierig, weil der Issue-Tracker und die Issues, die er auflistet, jederzeit von Anwendern manipuliert werden können. Es kann dazu kommen, dass ein erstellter Issue in der für den IssueMaker bekannten Form nicht mehr existiert, weil ein Nutzer die Inhalte angepasst hat.

In der Regel geht man davon aus, dass keine neuen Informationen in redundanten Einträgen vorhanden sind. Dies gilt aber oft nicht für Fehler in Software. Eine Fehlermeldung, die häufiger als einmal auftritt, ist ohne Kontext zwar erst einmal redundant. Sind aber weitere Informationen im Kontext des Fehlers versteckt, kann eine Dopplung sinnvoll sein [41]. Anstatt doppelte Issues zu löschen, könnte man sie beispielsweise als Duplikate markieren. Dies könnte allerdings zu einer Flutung mit als Duplikat markierten Issues führen. Es gäbe also Duplikate, nur sie wären als solche zu erkennen.

Weiter muss der IssueMaker erkennen können in welchem Status sich ein Issue befindet. Ein Issue kann in der Regel mindestens zwei Status haben. Entweder ist er `offen`. Dann wurde er erstellt, aber noch nicht gelöst. Oder ein Issue ist `geschlossen`, was impliziert, dass der Fehler, der im Issue beschrieben wurde, gelöst wurde und nicht mehr auftreten sollte. Verschiedene Issue-Tracker lassen außerdem beliebig viele selbst definierte Status zu. Alle abzudecken ist nicht möglich. Wenn ein Issue-Tracker angesprochen werden soll, der weitere Status zulässt, muss das bei den Anpassungen beachtet werden. Für die Erkennung von Duplikaten ist im Fall von `offen` und `geschlossen` wichtig, dass Issues kein Duplikat sind, wenn deren Pendant im Issue-Tracker geschlossen ist.

Wenn ein Fehler, der eigentlich bereits als gelöst im Issue-Tracker gelistet ist, wieder auftaucht, gibt es zwei Möglichkeiten. Zum einen kann der geschlossene Issue wieder geöffnet werden. Allerdings kann dies zu Verwirrungen führen, wenn bereits Commits oder Pull-Request an den Issue gebunden wurden. Geschlossene Issues sollten nicht wieder

geöffnet werden. Denn wenn ein Issue geschlossen ist, sollte die Arbeit, die in dem Issue beschrieben ist, abgeschlossen sein. Wenn der Fehler des Issues allerdings wieder im Test gefunden wird, hat dieser eine Existenzberechtigung und sollte durch einen neuen Issue repräsentiert werden. Der bereits geschlossene Issue kann im neuen verlinkt werden, sodass dessen Lösung bei der Bearbeitung des neuen Auftretens helfen kann.

Neben den Status kann ein Issue zusätzlich einem Entwickler zugewiesen werden. Diese Zuweisung beschreibt einen Substatus, der interessant sein kann, weil eine Zuweisung bedeuten kann, dass ein Issue bereits in Bearbeitung ist. Dann wäre es wünschenswert, dass er bei der Duplikaterkennung betrachtet wird. Wenn ein Issue als `offen` geführt werden kann, solange er nicht geschlossen ist, reicht es zu prüfen, dass alle Issues, die nicht geschlossen sind, bei der Prüfung nach Duplikaten betrachtet werden sollten.

Anstatt einen Issue, der als Duplikat erkannt wurde, zu filtern oder ihn als Duplikat zu erstellen, kann man eine weitere Eigenschaft von Issues in Betracht ziehen. Issues können priorisiert werden. Jeder der in den Tabellen 3.1, 3.2 und 3.3 hat ein Feld dafür vorgesehen. Auch Testfälle werden priorisiert, denn vollständiges Testen ist kaum möglich. So liegt es nahe, dass in Projekten eine Priorisierung der Tests gemacht wird und diese in den Issues übernommen wird. Im Buch „Basiswissen Softwaretest“[37] werden zum Beispiel folgende Kriterien für die Priorisierung von Testfällen vorgeschlagen:

- Die Nutzungshäufigkeit einer Funktion bzw. die Eintrittswahrscheinlichkeit einer Fehlerwirkung beim Betrieb der Software. Funktionen die öfter genutzt werden und Fehler enthalten, werden diese öfter auslösen. Deshalb sollten Funktionen, die oft genutzt werden, im Testplan höher priorisiert werden als Funktionen, die selten genutzt werden.
- Aus der Kombination von Fehlerschwere (=erwarteter Schaden) und Eintrittwahrscheinlichkeit ergibt sich das Fehlerrisiko. (Erwarteter Schaden × Eintrittswahrscheinlichkeit = Risiko) Testfälle, die risikoreiche Fehler aufdecken können, sollten dementsprechend auch eine höhere Priorität haben als Tests, die risikoarme Ausfälle aufdecken sollen.
- Fehler, die der Kunde bzw. Nutzer wahrnimmt, sind höher priorisiert, weil dies den Nutzer verunsichert.
- Fehler, die Funktionen nicht benutzbar machen.

- Tests, die nachweisen, ob die Qualitätsanforderungen des Kunden eingehalten wurden.
- Komponenten, die beim Ausfall schwere Auswirkungen haben
- Komplexe Programmabschnitte sollten getestet werden, weil hier die Wahrscheinlichkeit, dass ein Entwickler einen Fehler gemacht hat, größer ist.
- Fehler mit hohem Projektrisiko

Die Priorisierung ist aber vom Projekt selbst abhängig. Ein denkbares Szenario wäre also, dass die Priorität eines Issues erhöht wird, wenn ein Issue erneut auftaucht. Das hätte den Vorteil, dass auch Issues, die auf den ersten Blick unwichtig erscheinen, nicht unbearbeitet „verhungern". Aber dies kann auch durch die Betrachtung des Erstellungsdatums des Issues kompensiert werden. Eine automatische Erhöhung der Priorität würde außerdem dazu führen, dass die Issues bei jedem Durchlauf der Pipeline an Priorität gewinnen würden. Wenn nach der Bearbeitung eines Issues jedes Mal die Pipeline angestoßen werden würde, würde es entweder sehr viele Stufen voraussetzen oder die Issues würden sehr schnell die maximale Priorität erreichen. Dann würden unwichtige Issues wichtige verdrängen oder es würde nach einer Weile nur noch maximal priorisierte Issues geben und die Priorisierung wäre überflüssig. Eine Erhöhung der Priorität aufgrund wiederholtem Auffinden eines Mangels ist also nicht zu empfehlen.

Der IssueMaker sollte möglichst keine Duplikate erzeugen. Dafür wird die Liste der bereits im Issue-Tracker vorhandenen Issues mit denen des neu erstellten Testreports bei jedem Durchlauf verglichen. Tritt der Fall ein, dass ein Issue des Reports bereits im Issue-Tracker enthalten ist, wird der Issue im Report entfernt und nicht im Issue-Tracker. So kommt es zu keinem Duplikat.

3.4.2 Wanderung und Verlust der Relevanz von Fehlern

Eine andere Herausforderung für das Verhindern von Duplikaten sind Refactoring und Änderungen im Code. Diese können Zeilenverschiebungen oder Verschiebungen von Teilen des Programmcodes von einer Datei in eine andere beinhalten. Wenn ein bekannter Fehler durch solche Arbeiten seine Position ändert, darf kein neues Issue für den Fehler erstellt werden, wenn der Prozess erneut angestoßen wird. Stattdessen sollte erkannt werden können, dass sich die Position des Fehlers im Code lediglich verschoben hat und

der bereits bekannte Issue sollte ein Update erfahren. Ansonsten müssen alle Informationen des alten Issues in den neuen übernommen werden. Am wichtigsten ist, dass die Änderungen der Position erkannt werden kann und kein Duplikat erzeugt wird.

Ein weiteres Problem der automatischen Erstellung von Issues ist, dass sie an Relevanz verlieren können, ohne dass sie bearbeitet wurden. Ein Issue ist relevant, solange der Inhalt noch aktuell ist. Also der Fehler, der im Issue beschrieben wurde, noch im Code existiert.

Beispielsweise kann ein Test einer Funktionalität fehlschlagen und dies kann dazu führen das andere Funktionen, die auch auf der fehlerhaften Funktion basieren, ebenfalls fehlschlagen. Für jeden fehlgeschlagenen Test wird ein Issue erstellt. Wenn nun die Funktionalität der ersten Fehlerquelle aufgelöst wird und somit auch alle anderen Fehler nicht weiter auftreten, müssen deren Issues entfernt werden. Der IssueMaker muss also bei einer erneuten Ausführung die bereits erstellten Issues auf Relevanz prüfen und irrelevant gewordene Issues eigenständig löschen können. Dieser Vorgang wird im folgenden als Housekeeping bezeichnet. Das Housekeeping kann auch bei Positionsänderungen von Fehlern helfen, indem die irrelevanten Issues entfernt werden würden.

3.4.3 Interaktion mit dem Nutzer

Die Arbeit mit einem Issue-Tracker bedeutet, dass auch Entwickler/Nutzer die Issues manipulieren können. Änderungen können sein, dass Kommentare an den Issue angehängt werden oder Parameter wie Zeitschätzungen oder Zuweisungen angepasst werden. Dies sind keine kritischen Fälle, weil die Issues aufgrund ihrer Beschreibung identifiziert werden.

Hier ist dann der folgende kritische Fall zu erkennen: Wenn ein Entwickler oder Nutzer die Beschreibung eines Issues, welcher vom IssueMaker erstellt wurde, manipuliert, ist es bereits bei kleine Änderungen nicht mehr möglich, sie als gleiche Issues zu erkennen.

Das Housekeeping würde sie entfernen, weil der Issue in der aktualisierten Form nicht erneut entdeckt werden würde. Damit würden die zusätzlichen Informationen verschwinden, die der Nutzer hinzugefügt hat. Deshalb muss klar sein, dass ein Entwicklungsprozess, der den IssueMaker nutzt, so definiert ist, dass Issues, die von ihm erstellt wurden, entweder in der Beschreibung nicht verändert werden dürfen oder es muss einen Identifikationsmechanismus geben mit dessen Hilfe sich der Issue auch nach Änderungen noch

identifizieren lässt. Für einen solchen Mechanismus muss dann gelten, dass er nicht vom Nutzer manipuliert werden kann. Hier sollte erwähnt werden, dass Issues geschlossen werden sollten und nicht gelöscht werden, wenn sie als Irrelevant erkannt werden sollten. Denn ansonsten gehen möglicherweise zusätzliche Informationen verloren.

3.5 Idee zur Verbesserung der Übersicht und Optimierung: Zusammenfassen von Fehlermeldungen in Issues

Einer der Grundsätze des Softwaretestens ist die Häufung von Fehlern [37]. In der Nähe von Softwarefehlern finden sich meistens weitere. Ein Entwickler könnte demnach in bestimmten Situationen mehrere Funde innerhalb eines Issues auflösen. Hier wäre es also praktisch, wenn das Programm erkennen würde, wenn sich viele Fehler zum Beispiel vom selben Typ in derselben Datei befinden und der IssueMaker diese in einem Issue zusammenfassen würde. Dies ist typisch bei der statischen Codeanalyse, wenn beispielsweise neue Entwickler sich nicht an die Coding-Konventionen des Teams gehalten haben.

Allerdings können Fehler aus unterschiedlichen Gründen ähnlich sein. Zum Beispiel:

- Es handelt sich um den gleichen Fehler aber in einer anderen Datei.
- Mehrere Fehler sind in derselben Datei.
- Der Fehlertyp und damit die Lösung ist bei mehreren Fehlern gleich.

Durch das Zusammenfassen von Fehlern innerhalb eines Issues ergeben sich allerdings Probleme. Nicht nur die Beschreibung, sondern auch andere Informationen wie beispielsweise Kommentare sind in einem Issue vorhanden. Die Felder in 3.3 erlauben es durchaus ähnliche Fehler in einem Issue zusammenzufassen. Dies ist aber nicht ohne weiteres möglich. Als Beispiel ist hier das Datum genannt. Dieses kann nicht immer sinnvoll zusammengefasst werden.

Nur wenn ähnliche Fehler bei demselben Durchlauf des Prozesses gefunden werden, hätten sie denselben Zeitstempel. Wenn sie aber bei unterschiedlichen Durchläufen gefunden werden würden, wäre der Zeitstempel eine Liste von Zeiten, die den eigentlichen Fehlern nicht mehr zuzuordnen sind. Das würde bedeuten, dass Fehler nur in Issues zusammengefasst werden dürfen, wenn sie ähnlich sind und im selben Testlauf gefunden wurden.

Ein weiteres Problem beim Zusammenfassen von Fehlern tritt auf, wenn Teile eines Issues, der mehrere Fehlerinstanzen enthält, im Zuge eines anderen Issues gelöst wurden andere aber nicht. Der Issue müsste aktualisiert werden, was wiederum eine Interpretation des Inhalts voraussetzen würde, was aufwendig ist.

3.6 Kennzeichnung von ähnlichen Fehlermeldungen

Während das Zusammenfassen von Fehlern es erschwert Duplikate zu erkennen, ist es unter Umständen sinnvoll, Fehler zu kennzeichnen, die von einem bestimmten Typ sind. Fehler, die denselben Typ haben, bilden eine Gruppe. Wenn nun das Programm prüfen muss, ob es sich bei dem neuen Issue um ein Duplikat handelt, kann am Typ bereits eine Vorauswahl getroffen werden. Dies verringert die Menge der zu vergleichenden Issues.

3.7 Paralleles Arbeiten mit verschiedenen Testreportquellen

Der IssueMaker soll Testreports unterschiedlicher Quellen einlesen können. Dies ermöglicht eine parallele Ausführung mit unterschiedlichen Testverfahren und mehreren Instanzen des Programms zum Erstellen der Issues. Dafür muss das Programm entweder entsprechend konfigurierbar sein oder erkennen, welche Art und vor allem welches Format der Testreport hat.

3.8 Abgrenzung der Anforderungen

Um den Funktionsumfang auch nach oben abzugrenzen, sei erwähnt, dass der IssueMaker kein Testtool ist. Er wird selbst keine Tests durchführen können und der Testreport ist ihm in einem ihm bekannten Format bereitzustellen. Der IssueMaker wird lediglich in der Lage sein aus strukturierten Testreports Issues zu erstellen. Es wird ein Key-Value-Format wie XML oder JSON bevorzugt. Die Struktur ist wichtig, da die Informationen des Testreports nur auf Gleichheit überprüft werden können, wenn sie gleich formuliert und aufgebaut sind. Außerdem sollte der Testreport bereits für den Menschen lesbar sein, da er die Daten des Testreports nicht interpretieren kann. Änderungen, der vom IssueMaker erstellten Issues, sind nur im begrenzten Maße vom IssueMaker tolerierbar.

3.9 Zusammenfassung der Anforderungen an ein Programm zur automatischen Generierung von Issues aus Testreports

Schließlich ergeben sich folgende Anforderungen.

- Funktionale Anforderungen
  1. Der IssueMaker soll automatisch im Anschluss eines Testlaufs auf Basis des erstellten Testreports Issues im entsprechenden Issue-Tracker erstellen.
  2. Anpassbarkeit und Erweiterbarkeit muss gegeben sein. Es sollen Testreports aus verschiedenen Quellen und in beliebigen Formaten eingelesen werden können. Außerdem sollen verschiedene Issue-Tracker angesprochen werden können.
  3. Duplikate müssen erkannt und verhindert werden können.
  4. Irrelevant gewordene Issues müssen geschlossen werden.
  5. Es dürfen nur Issues manipuliert werden, die vom IssueMaker erstellt wurden.
  6. Durch den Nutzer vorgenommene Änderungen an erzeugten Issues dürfen nicht verloren gehen.
- Nicht-funktionale Anforderungen
  1. Die Informationen aus den Testreports sollen sinnvoll und übersichtlich dargestellt werden.
  2. Die erzeugten Issues halten sich nah am Aufbau des Incident Reports aus IEEE-29119.

4 Konzept

Das Konzept beschreibt auf Basis der Anforderungsanalyse 3 die geplante Umsetzung des IssueMakers. Es werden konkrete Randbedingungen für die Implementierung vorgestellt und auf deren Basis wird eine Architektur für das spätere Programm erarbeitet.

4.1 Automatisierung mit der Hilfe einer Pipeline

Damit automatisch Issues auf Basis der Reports erstellt werden können, muss das Programm automatisch nach der Erstellung eines Reports ohne Nutzeraktion starten. Dies kann ermöglicht werden, indem es nahtlos in den Prozess eingebunden wird. Hierfür bietet sich eine Pipeline an, die die Aufgaben nacheinander ausführt. GitLab bietet im eigenen CI/CD Bereich die Möglichkeit eine Pipeline zu erstellen.

Im Idealfall wird die Pipeline regelmäßig zu fest definierten Ereignissen ausgeführt. Dies kann bedeuten, dass sie bei jedem Update des Programmcodes ausgeführt wird oder, dass sie regelmäßig zu einer bestimmten Zeit ausgeführt wird. Denkbar wäre es, zu einem Zeitpunkt an dem die meisten Entwickler nicht arbeiten, damit die Ausführung der Pipeline keine Ressourcen belegt, die während der Entwicklung gebraucht werden oder weil die Tests, auf denen der Report basiert sehr lange dauern. Es sollte auch möglich sein, dass das Tool bei Bedarf ausgeführt wird.

Eine solche Pipeline könnte wie in Abbildung 4.1 aufgebaut sein.

1. Der Test wird angestoßen. Dies kann durch unterschiedliche Mechanismen geschehen. Möglich wäre eine manuelle, periodische oder durch ein Event ausgelöste Ausführung.

2. Wenn der Test abgeschlossen wurde, gibt es einen Report, der aus einer Menge erfolgreicher und nicht erfolgreicher Testergebnisse zusammen gesetzt ist.

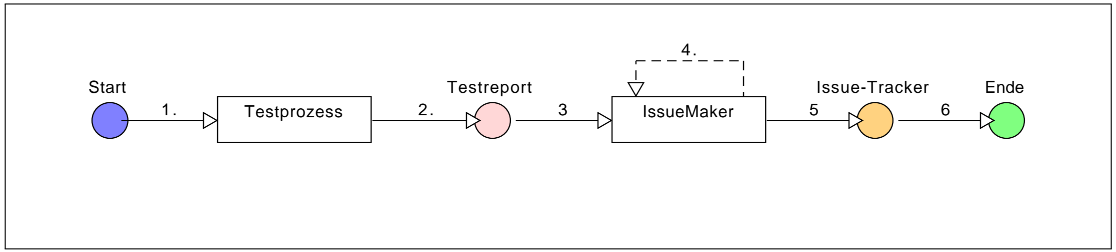

Abbildung 4.1: Allgemeiner Aufbau einer Pipeline.

3. Der IssueMaker liest den Report.
4. Er kann die erfolgreiche und fehlgeschlagene Testergebnisse einordnen und entscheiden, welche in den Issue-Tracker geladen werden sollen.
5. Alle Testergebnisse der fehlgeschlagenen Tests werden vom IssueMaker jeweils in Issues im spezifizierten Issue-Tracker erstellt.
6. Die Pipeline endet.

Die erstellten Issues können anschließend von den Entwicklern bearbeitet werden.

4.2 Hintergrund zur Nutzung von GitLab als Issue-Tracker

An der Hochschule für angewandte Wissenschaften (HAW) in Hamburg gibt es für die Studierenden des Fachs technische Informatik im 4. Semester im Rahmen der Software-Engineering Ausbildung ein Praktikum, indem die Studierenden eine Fließbandanlage betreiben müssen. Damit die Teilnehmer ihre Programme regelmäßig testen können, wurde eine Simulation der Anlage und ein Regressionstest erstellt. Dieser kann über eine Pipeline in der GitLab-Instanz der HAW gestartet werden. Die Testergebnisse des Regressionstests sollen über einen weiteren Pipeline-Schritt nun als Issues in den GitLab-Projekten der Studierenden erstellt werden. Deshalb wird für dieses Projekt als Issue-Tracker GitLab genutzt.

4.2.1 Übersicht über den Funktionsumfang von GitLab

Die Webanwendung GitLab bietet neben einer auf Git basierten Versionskontrolle einen eigenen Issue-Tracker. Außerdem gibt es die Möglichkeit eine eigene CI/CD Pipeline

Attribut	Beschreibung
created_at	Timestamp of creation.
author	Person that created the issue.
state	Identifies the current state.
title	Provides a title.
project_id	Identifier for the project this issue belongs to.
description	Description of the issue.
updated_at	Timestamp of the last change in any of the fields.
closed_at	Timestamp when the issue was closed.
closed_by	Person that closed the issue.
iid	The internal ID of a project's issue.
labels	Comma-separated label names for an issue.
id	The ID or URL-encoded path of the project owned by the authenticated user.
assignees	Comma-separated IDs of users assigned to this issue.
assignee	Deprecated. Id of the user assigned to this issue.
milestone	The global ID of a milestone to assign issue
subscribed	Indicates a subscription.
user_notes_count	Number of user notes.
due_date	Due date of the issue.
web_url	Web url of the issue.
time_stats	Time stats like estimated time.
confidential	Indicates if an issue is confidential.
weight	The weight of the issue.
discussion_locked	Indicates if an issue allows discussions.
links	List of links mentioned in this issue.

Tabelle 4.1: Aufbau eines Issues in GitLab. [14]

für ein Projekt anzulegen. Für die Verwendung von GitLab gibt es zwei Möglichkeiten. Entweder man nutzt die von GitLab selbst gehostete Instanz oder man erstellt eigenverantwortliche eine eigene. Zweiteres hat die HAW für ihre Studierenden und Forschungsprojekte bereits vorbereitet.

4.2.2 Darstellung von Issues in GitLab

Der Aufbau eines Issues ist in der Tabelle 4.1 dargestellt. In Tabelle 4.2 sind die Attribute des GitLab-Issues und die Empfehlung der IEEE-29119 Dokumentation zur Testdokumentation gegenübergestellt und eingeordnet. Informationen zur Schwere und zum Risiko

sind in einem GitLab-Issue nicht grundsätzlich vorgesehen und müssen in der Beschreibung einen Platz finden. Außerdem haben GitLab-Issues einige Felder, die nicht dem Fehler Report nach IEEE-29119 zugeordnet werden können.

Diese Felder sind zum Teil nicht fehlerspezifisch, sondern entstammen anderen Funktionen der GitLab-Software. So ermöglichen die Flags `confidential` und `discussion_locked` die Kommunikation über und Sicht auf die Issues für bestimmte Benutzer einzuschränken.

Andere Felder sind für den Arbeitsprozess in GitLab wichtig. Wie zum Beispiel der oder die Assignee(s). Ein Issue kann einer oder mehreren Person(en) zur Bearbeitung zugewiesen werden. Diese Person(en) heißen Assignee(s). Nach der Bearbeitung eines Issues kann er geschlossen werden. Für Rückfragen, kann es wichtig sein, zu wissen, wer den Issue geschlossen hat. Dies wird unter `closed_by` gespeichert.

Der Issue-Tracker von GitLab kann die Issues auf verschiedene Weisen anzeigen. Es ist einerseits möglich alle Issues in einer einzigen Liste anzeigen zu lassen. Es ist aber auch möglich sie in einem frei gestaltbaren Board bestehend aus mehreren Listen für unterschiedliche Stadien der Issues vgl. Kanban und Scrum aufzulisten. Damit in diesen Listen der Inhalt der einzelnen Issues bereits überblickt werden kann, hat jeder Issue einen Titel, der eine sehr kurze Beschreibung des Inhalts beinhalten sollte.

Eine Anforderung an die automatisch erstellten Issues ist, dass sie alle Informationen, die im Testreport enthalten sind, im Issue aufführt. Zusätzlich werden Informationen wie das Erstellungsdatum des Issues gepflegt. Der Aufbau des Issues soll den Entwicklern möglichst gut helfen. In der Studie aus „What makes a good bug report?“ [40] wurden diese identifiziert.

4.2.3 CI/CD: Automatisierung in GitLab

Die Konfiguration der Pipeline in GitLab erfolgt über eine Datei, die im Repository Root-Verzeichnis liegen muss. In der Regel heißt sie `.gitlab-ci.yml` und ist im YAML-Format [25] geschrieben. Die `.gitlab-ci.yml` ist in 4.1 zu sehen.

```
1 variables:
2  DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"
3  SERVICE_NAME: bachelorthesis
4  CI_PROJECT_NAMESPACE: ach339
```

IEEE-29119	**GitLab**
Timing Information	created_at, updated_at, closed_at, due_d ate, time_stats
Organiator	author
Context	project_id, milestone, links, labels, id, iid
Description of the incient	description
Originator's assessment of severity	—
Originator's assessment of priority	weight
Risk	—
Status of the incident	state
—	title
—	closed_by
—	assignees, (assignee)
—	subscribed
—	user_notes_count
—	web_url
—	confidential
—	discussion_locked

Tabelle 4.2: Vergleich der Attribute eines Fehler Reports nach IEEE-29119 [29] und eines GitLab-Issues [14]

```
5
6  stages:
7   - test
8   - generate
9
10 cppcheck:
11  stage: test
12  image: soenke2/cppcheckforissuemaker
13  script:
14  - "cppcheck --enable=all --std=posix \
15  --inconclusive --force TestProject/ --xml 2> report.xml"
16  artifacts:
17  paths:
18  - report.xml
19
20
21 issueMaker:
22  stage: generate
23  allow_failure: true
24  image: $DOCKER_REGISTRY/
25     $CI_PROJECT_NAMESPACE/
26     $SERVICE_NAME:latest
27  only:
28  - develop
29  script:
30  - Main.py
31  dependencies:
32  - cppcheck
```

Listing 4.1: Beispiel einer .gitlab-ci.yml, die eine Pipeline erzeugt.

Als Erstes können in der Datei Variablen definiert werden, die im späteren Verlauf genutzt werden können. Darauf folgen die Stationen der Pipeline, die durch `stages` gekennzeichnet sind. Stationen werden in der Reihenfolge ausgeführt, in der sie hier definiert sind. In diesem Fall gibt es zwei. Als Erstes muss der Testreport erstellt werden. Dafür wird in diesem Beispiel Cppcheck genutzt [6]. Cppcheck selbst wird in einem Docker-Container ausgeführt. Im zweiten Schritt wird der IssueMaker ebenfalls in einem Docker-Container gestartet. Mit diesen Informationen kann GitLab eine Pipeline erstellen und die Programme werden automatisch ausgeführt, wenn die Pipeline gestartet wird.

4.2.4 Einsatz von Docker-Container in der GitLab Pipeline

Der IssueMaker soll in einer beliebigen GitLab-Pipeline laufen können und muss deshalb in einem Docker-Container liegen oder im Repository vorhanden sein. Damit nicht jeder Nutzer die Dateien des Programms in seinem Repository halten muss, wurde sich für die Lösung mit Docker entschieden.

Docker vereinfacht die Bereitstellung von Anwendungen, weil sie sich in Containern, die alle nötigen Pakete enthalten, befinden und sich leicht als Dateien transportieren und installieren lassen [7]. Ein Container, der innerhalb einer GitLab-Pipeline gestartet wird, hat Zugriff auf das gesamte Repository und auf Artefakte, die in vorausgegangenen Schritten erstellt wurden.

Diese Artefakte ermöglichen eine einfache Kommunikation zwischen den Pipeline-Schritten und in diesem Fall die Übergabe des Testreports an die Issue-Generierung. Ein weiterer Vorteil des Docker-Containers ist, dass über eine Docker-Registry der Container on-demand geladen werden kann. GitLab sucht in der Docker-Hub-Registry [8] und in jeder weiteren, die in einer Konfigurationsdatei angegeben werden kann, nach dem in der `.gitlab-ci.yml` angegebenen Container. Dies ermöglicht auch, dass Container in der eigenen privaten Registry gehalten werden können. Das bedeutet, dass sie nicht frei im Internet verfügbar sind, sondern zum Beispiel nur im Intranet einer Hochschule oder eines Unternehmens genutzt werden können.

Die GitLab-Pipeline erlaubt auch das parallele Ausführen von Skripten und Containern, sodass ein Programm, das in einem Container liegt, mehrfach nebeneinander laufen kann, indem der Container mehrfach gestartet wird. Da der IssueMaker verschiedene Reports lesen können soll, könnten so problemlos zwei Instanzen des Tools für unterschiedliche Reports in jeweils einem eigenen Container in der Pipeline parallel laufen.

4.2.5 GitLab-Issue-Tracker als Datenbank

Da der IssueMaker in der Pipeline ausgeführt wird und diese nach der Ausführung wieder bereinigt wird, können die Issues nicht in einer Datenbank innerhalb der Pipeline persistent gespeichert werden. Das Programm muss also bei jedem Lauf, alle Issues aus dem Issue-Tracker laden, um diese mit den neuen vergleichen zu können. Alternativ könnte eine Datenbank unabhängig der Pipeline und des Issue-Trackers gehalten werden. Diese würde aber redundant zum Issue-Tracker selbst sein.

4.2.6 GitLab API: Zugriff auf alle Funktionen?

Der zweite Container enthält den IssueMaker, welcher aus dem Report, den der erste Container erstellt hat, die Issues im GitLab-Issue-Tracker erstellt. Damit das Programm Issues in GitLab erstellen kann, muss die GitLab-Api angesprochen werden, diese ist eine RESTful-API und bietet diverse Möglichkeiten zum Manipulieren des eigenen GitLab-Repositories und der eigenen GitLab-Instanz [11].

4.3 Cppcheck. Statische Codeanalyse als Testdatengenerator.

Der erste Container beinhaltet Software um einen Testreport, die als Testdaten für das Projekt genutzt werden können, zu erstellen. Um einfach Testdaten zu produzieren wurde sich für Cppcheck entschieden. Cppcheck ist ein Programm zur statischen Codeanalyse. Es ist spezialisiert auf Fehler in C/C++ Code, die dem Compiler selbst entgehen [6]. Der Report einer Cppcheck-Analyse kann im XML-Format ausgegeben werden, welches man leicht lesen kann aber auch leicht durch ein Programm einlesen lassen kann. Als Beispiele für Alternativen von Cppcheck können Clang, Cpplint von Google, flint++ etc. gewählt werden. Hier wird sich allerdings auf Cppcheck beschränkt.

Es liegt nahe, ein Programm zur statischen Codeanalyse zu wählen, weil dabei der Code nicht ausgeführt werden muss. Außerdem ist es leicht Fehler einzubauen, die in der statischen Codeanalyse gefunden werden, um so Testreports zu generieren, die von dem Programm als Issues angelegt werden können. Später wird auf Basis der Erfahrungen mit Cppcheck das Programm am Regressionstest des Projekts der HAW überprüft.

Wenn Cppcheck mit dem Aufruf 4.2 ausgeführt wird, schreibt es den Report in die `report.xml` Datei. In 4.3 ist zur Verdeutlichung ein Ausschnitt des Reports dargestellt.

```
1 cppcheck --enable=all --std=posix --force TestProject/ --xml 2>
    report.xml
```

Listing 4.2: Aufruf von Cppcheck mithilfe eines Terminals.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
 <cppcheck version="1.84"/>
 <errors>
 <error id="functionConst"
  severity="style"
  msg="Technically the member function
  &apos;SomeClass::toString&apos;
  can be const."
  verbose="The member function
  &apos;SomeClass::toString&apos;
  can be made a const function. Making this function
  &apos;const&apos; should not cause compiler
  errors. Even though the function can be made
  const function technically it may not make
  sense conceptually.
  Think about your design and the
  task of the function first -
  is it a function that must not change object
  internal state?"
  cwe="398"
  inconclusive="true">
  <location file0="TestProject/SomeClass.cpp"
  file="TestProject/SomeClass.h" line="17"/>
  <location file="TestProject/SomeClass.cpp" line="8"/>
  <symbol>SomeClass::toString</symbol>
 </error>
[...]
 </errors>
</results>
```

Listing 4.3: Auzug eines Reports, der von Cppcheck erstellt wurde.

Im XML-Report von Cppcheck in Listing 4.3 ist zu sehen, dass es sich hier um eine Liste von Fehlern mit dem XML-Tag `error` handelt. Eine Liste alle XML-Attribute des `error`-Elements ist in der Tabelle 4.3 nachzulesen. Das Ziel ist, dass später für jeden `error` ein Issue mit den Attributen erstellt wird. Außerdem hat ein `error` XML-Element noch `location` Elemente die in der Tabelle 4.4 erläutert werden und auch im erstellten Issue einen Platz finden sollen.

Es wird eine Klasse benötigt, die diese XML-Datei lesen kann und in ein Issue-Objekt

Attribut	Beschreibung
id	Id des Fehlers z.B. „unreadVariable“
severity	Art des Fundes. Kann entweder error, warning, style, performance, portability oder information sein.
msg	Kurzform der Fehlerbeschreibung.
verbose	Ausführliche Fehlerbeschreibung.
inconclusive	Kann entweder wahr oder falsch sein. Es kann vorkommen, dass Cppcheck sich nicht sicher ist, dass es sich tatsächlich um einen Fehler handelt. Dann wird die entsprechende Meldung als „inconclusive “ bezeichnet.
cwe	Der CWE Identifier des Fundes [31] wird nur übermittelt, wenn diese bekannt ist.

Tabelle 4.3: Aufbau eines Fehler-Elements des XML-Reports von Cppcheck.

Attribut	Beschreibung
file	Dateipfad. Kann relativ oder absolut angegeben werden.
file0	Name der Quelldatei.
line	Zeile in der Fehler gefunden wurde.
info	Zusätzlich Informationen.

Tabelle 4.4: Lokalisierungsattribute einer Cppcheck-Fehlermeldung.

überführt mit dem weiter gearbeitet werden kann.

4.4 Wahl der Programmiersprache für die Implementierung des IssueMakers

Der IssueMaker wird in der Programmiersprache Python geschrieben.

Python hat diverse Vorteile für die Entwicklung eines Prototyps. So gibt es bereits eine umfangreiche Bibliothek, die die Anfragen an die GitLab-API abstrahiert [28]. Dies ermöglicht, dass Anfragen an die Schnittstelle implementiert werden können.

Außerdem sind die Rahmenbedingungen für die Ausführung eines Python-Skripts sehr gering, sodass das Docker-Image nur wenige zusätzliche Pakete benötigt [27].

Es kann zum Beispiel auf dem Docker-Image Alpine basieren. Alpine ist ein sehr kleines Docker Image, das auf Alpine Linux mit einem kompletten Paketindex basiert und

5 MB groß ist [1]. Zusätzlich gibt es Abhängigkeiten von `python3` und `pip` sowie `gitlab-python`. Diese müssen nach geladen werden.

Nicht zuletzt wird für ein lauffähiges Python-Skript wenig zusätzlicher Code benötigt, um das Programm starten zu können, sodass man sich auf die funktionalen Anforderungen konzentrieren kann.

Wenn der IssueMaker in einer anderen Programmiersprache wie beispielsweise Java geschrieben werden würde, bräuchte man wesentlich mehr Ressourcen um unter anderem die Java Virtual Machine ausführen zu können.

4.5 Übersichtliche Darstellung der Issues mit Hilfe von Markdown

Die Issues sollen ein für den Entwickler leicht zu lesendes und verständliches Format besitzen und gegebenenfalls Zusatzinformationen enthalten.

Issues in GitLab bieten Felder, um Informationen zu halten, wie in Tabelle 4.1 zu sehen ist. Die meisten sind bereits von der API mit entsprechenden Funktionen vorbelegt.

Für einen Fehler eines Reports bleiben lediglich zwei Felder. Zum einen der Titel des Issues und zum anderen die Beschreibung.

Wie bereits im vorherigen Teil beschrieben bestehen Fehler aber oft aus verschiedenen Schlüsselwertpaaren. Beispielsweise sei auf die `Severity` verwiesen. `Severity` ist der Schlüssel und `Performance` ist ein möglicher entsprechender Wert.

Die Informationen des Reports, die nicht im GitLab-Issue vorgesehen werden, müssen in der Beschreibung des Issues ihren Platz finden. Um Issue-Beschreibungen strukturiert in GitLab darstellen zu können, wird Markdown unterstützt [15].

Markdown ist eine Textformatierungssprache, die es ermöglicht mit wenigen Zeichen einen Text zu formatieren. Überschriften einfache Tabellen und einfaches hervorheben von bestimmten Textbausteinen ist einfach zu bewerkstelligen. Vor allem kann man dies einfach im Text mitsenden. GitLab kümmert sich schließlich um die Darstellung.

4.6 Einsatz von Filtern zur Verhinderung von Duplikaten und irrelevanten Issues

In der Anforderungsanalyse wurde erörtert, dass es in bestimmten Situationen zu Duplikaten kommen kann. Für das Beachten von Duplikaten und das Housekeeping muss das Programm Zugriff auf den Issue-Tracker bekommen. Es müssen bereits im Issue-Tracker erstellte Issues mit den aus dem neuen Bug-Report entstandenen Issues abgeglichen werden. Duplikate und irrelevante Issues müssen erkannt werden und dann entsprechend gefiltert oder geschlossen werden.

GitLab-Issues können ein Label erhalten, dieses kann von den Nutzern frei erstellt werden. Der IssueMaker kann also diese Label nutzen, um von ihm erstellte Issues zu kennzeichnen und zu erkennen. Ein Label hat einen Namen und eine Farbe. Dies ermöglicht, dass nur Issues eines bestimmten Labels für die Duplikatenerkennung und das Housekeeping aus dem Issue-Tracker geladen werden können. Es wird eine Vorauswahl getroffen.

Für die anschließende Behandlung von Duplikaten und irrelevanten Issues werden verschiedene Filter implementiert, die nacheinander auf die Listen angewendet werden. Am Ende soll eine Liste von Issues vorliegen, die über ein Interface wieder an den Issue-Tracker von GitLab geladen werden kann.

4.7 Arbeitsablauf des Programms

Aus den Anforderungen und dem Konzept bildet sich für den IssueMaker der Arbeitsablauf, wie er in der Abbildung 4.2 zu sehen ist. Dabei ist zu beachten, dass `Filter`, `Report Parser` und `Issue-Tracker` Interfaces sind. Mit ihrer Hilfe ist es möglich das Programm jederzeit zu erweitern.

Der ReportParser ist ein Interface, welches von dem Parser, der den eigentlichen Bug-Report einliest, implementiert werden muss (Siehe: 4.3).

Dies ermöglicht eine kontinuierliche Weiterentwicklung der Software durch die Anpassung an verschiedene Testtools, die eigene Reports produzieren. Für das Lesen des Reports von Cppcheck wird ein XML-Parser benötigt.

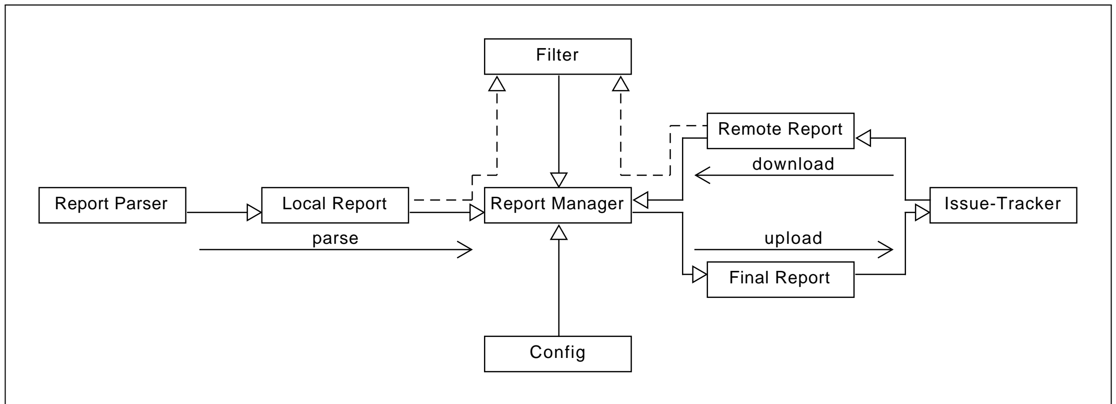

Abbildung 4.2: Arbeitsablauf des Programms.

«Interface»
ReportParser

+parseReport(filename):Report

Abbildung 4.3: Interface eines Reportparsers.

Der `ReportParser` muss einen Report erstellen. Dieser ist nichts weiter als eine Liste von Issues in einem internen Format, welches sich am Aufbau des Incident-Reports aus IEEE-29119 wie er in der Tabelle 3.3 zu sehen ist, orientiert.

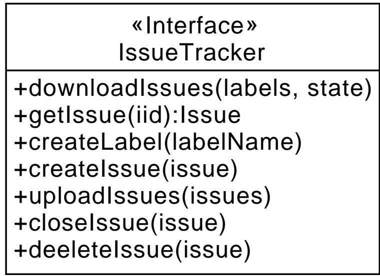

Abbildung 4.4: Interface eines Issue-Trackers.

Am anderen Ende der Architektur befindet sich ein Issue-Tracker Interface (Siehe: 4.4). Dieses stellt eine Schnittstelle zum Issue-Tracker wie GitLab zur Verfügung. Über das Interface muss es möglich sein Issues aus den internen Reports in einem Issue-Tracker wie GitLab zu erstellen. Außerdem muss es zur Behandlung von irrelevanten Issues mög-

lich sein, Issues im Issue-Tracker zu schließen oder zu löschen. Damit Duplikate erkannt werden können, muss es über das Interface möglich sein, Issues aus dem Issue-Tracker abhängig vom Status (`offen`, `geschlossen`) und anhand von Labeln herunterzuladen.

ReportManager
-Config
+init(config) -getRemoteReport():Report -getLocalReport():Report -uploadFinalReport(finalReport) +start() -applyFilter(localReport,remoteReport)

Abbildung 4.5: Reportmanager.

Im Kern des IssueMakers zwischen Parser und dem Tracker-Interface befindet sich der `ReportManager` (Siehe Abbildung 4.5). Er benötigt einen Zugriff auf die Config-Instanz, auf die alle anderen Komponenten zugreifen können, um für den Programmablauf spezifische Parameter zu laden. Diese nutzt er, um eine Verbindung zum passenden Issue-Tracker zu erstellen und den richtigen Parser für das konfigurierte Testtool zu wählen. Dann vergleicht er den Report des Parsers mit dem Report des Issue-Trackers und filtert, entsprechend der Richtlinien zur Duplikatvermeidung und des Houskeepings, Issues aus dem Report heraus.

«Interface» Filter
+filterLocalReport(localReport, remoteReport) +filterRemoteReport(localReport, remoteReport)

Abbildung 4.6: Interface eines Filters.

Neue Filter können erstellt werden, indem sie das Filterinterface implementieren (Siehe Abbildung 4.6). Am Ende existiert ein bereinigter Report in dem nur die Issues sind, die im Tracker neu erstellt werden sollen. Das Hochladen wird ebenfalls über das Issue-Tracker Interface ermöglicht.

4.8 Zusammenarbeit der Komponenten in der Pipeline

Der IssueMaker wird in eine Pipeline-Struktur wie sie in 4.7 zu sehen ist, eingebunden. Im ersten Schritt wird ein Test angestoßen. Der Test erzeugt einen Testreport. Im nächsten wird der Testreport gelesen und gefiltert. Im Issue-Tracker werden irrelevante Issues gelöscht und neue Issues erstellt. Die Pipeline endet und sämtliche Artefakte werden bereinigt und Ressourcen wieder freigegeben.

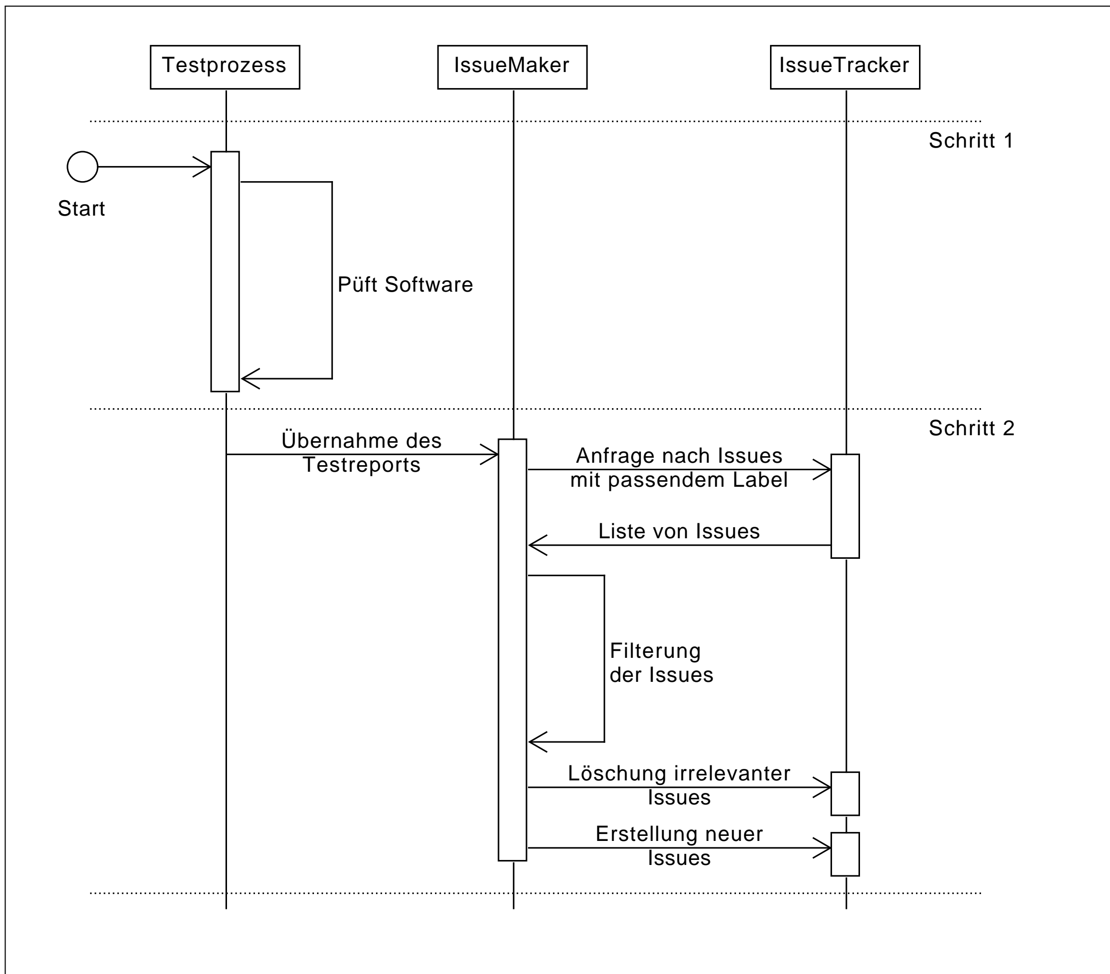

Abbildung 4.7: Pipelineaufbau in GitLab.

5 Implementierung

In diesem Kapitel wird eine Implementierung des Konzepts vorgestellt. Es wird die Programmierung der einzelnen Klassen und Interfaces in Bezug auf GitLab als Issue-Tracker und Cppcheck zur Erstellung der Testreports erläutert.

Außerdem wird beschrieben, welche Unterschiede es gibt, wenn ein Parser für einen selbst konzipierten Test geschrieben werden muss. Dafür wird der Testreport des bereits erwähnten Regressionstests, der für das Projekt der HAW entwickelt wurde, gelesen und ebenfalls in Issues umgewandelt.

5.1 Implementierung des Issue-Tracker-Interfaces für die Kommunikation mit der Gitlab-API

Als Issue-Tracker wird GitLab verwendet. Um den Zugriff auf den Issue-Tracker von GitLab zu bekommen, muss dessen API angesprochen werden. Außerdem muss das Interface aus Abbildung 4.4 für den Issue-Tracker implementiert und erweitert werden.

- `downloadIssues(State, LabelList):Report` Zum Prüfen der Issues aus dem Issue-Tracker, wird eine Methode benötigt, die, abhängig von der übergebenen `LabelList` und dem gewünschten `State`, alle Issues aus dem Issue-Tracker herunterlädt.

  Für GitLab wird nur der `State offen` betrachtet. Für andere Issue-Tracker kann dies variieren. Die GitLab-API unterstützt Paging. Dies ist bei dem Zugriff auf den Issue-Tracker zu beachten. Paging bedeutet, dass nicht der gesamte Issue-Tracker betrachtet wird, sondern immer nur ein Teil aller Issues. Der IssueMaker benötigt aber für die zuverlässige Erkennung von Duplikaten alle Issues im Issue-Tracker.

- `uploadIssues(Report)` Am Ende der Filterung müssen alle neu zu erstellenden Issues in den Issue-Tracker geladen werden. Dies wird durch die Implementierung dieser Methode erreicht. Die GitLab-API erlaubt es nur einen Issue zurzeit zu erstellen, weshalb bei der Implementierung dieser Methode alle Issues des Reports iteriert werden müssen.

- `createLabel(Labelname)` `Label` werden als erste Möglichkeit der Filterung genutzt. Anstatt alle Issues aus dem Issue-Tracker zu laden, werden nur die geladen, die das `Label` „IssueMaker" besitzen. Da nur diese von diesem Programm erstellt wurden.

  Außerdem kann jede Instanz des IssueMakers nur auf einen Testreport zurzeit angewendet werden. Dieser muss konfiguriert sein. Deshalb wird ein weiteres `Label` erstellt, welches den `Identifier` des Programms enthält, das den Testreport erstellt hat. In diesem Beispiel wäre es „Cppcheck". Dies ermöglicht, dass bei wiederholter Ausführung des Programms nur Issues aus dem Tracker mit den Labeln „IssueMaker" und „Cppcheck" geladen werden.

  Wenn diese `Label` aber noch nicht im Issue-Tracker existieren, müssen sie mit dieser Methode erst über die API des Issue-Trackers erstellt werden. Falls ein Issue-Tracker keine Unterstützung für `Label` besitzt, kann diese Methode auch leer bleiben.

- `getLabels(): LabelList` Da der IssueMaker keinen persistenten Speicher hat, sondern im Container on-demand gestartet und beendet wird, muss bei jedem Start geprüft werden, ob die nötigen `Label` bereits im Issue-Tracker existieren. Ansonsten kann es zu Fehlern kommen, wenn der Issue-Tracker versucht `Label` zu erstellen, die bereits existieren.

- `deleteIssues(iid)` Da das Programm in der Lage sein soll, dass Issues aus dem Issue-Tracker entfernt werden, wenn sie irrelevant geworden sind, braucht jeder Issue-Tracker eine Methode mit der er Issues im Issue-Tracker löschen kann. Beispielsweise könnten irrelevante Issues, die keine Änderungen durch Nutzer erfahren haben, mit dieser Methode gelöscht werden.

- `closeIssue(iid)` Issues, die an Relevanz verloren haben aber bereits von Nutzern manipuliert wurden, werden nur geschlossen, da ansonsten die Änderungen der Nutzer verloren gehen. Geschlossene Issues können im Issue-Tracker wiedergefunden werden, gelöschte können dies nicht.

Dieses Interface muss von allen Issue-Trackern implementiert werden, da es die Schnittstelle zum restlichen Programm bildet.

Die konkrete Implementierung des Issue-Trackers für GitLab in Abbildung 4.4 hat außerdem einen Konstruktor. Innerhalb des Konstruktors, bzw. der `init()`-Methode in Python, wird mithilfe der Parameter, die beim Programmstart übergeben werden müssen, der Zugriff auf den GitLab-Issue-Tracker gewährt. Mit den Werten kann anschließend das Projekt identifiziert werden, indem die Issues erstellt werden sollen.

5.2 Der ReportManager als zentrale Komponente

Der `ReportManager` delegiert die Aufgaben an die anderen Komponenten und wendet die Filter auf den Testreport und die heruntergeladenen Issues an. Er funktioniert für alle Issue-Tracker und Testreports gleich (siehe Abbildung 4.5).

- `start()` Diese Methode ist vergleichbar mit einem Konstruktor. Allerdings ist sie angelehnt an eine `main()`-Methode in der Programmiersprache Java. Sie ruft alle Methoden auf, die für das weitere Vorgehen nötig sind.

  Als Erstes werden mit der Methode `getLocalReport():Report` der Parser für den neu erstellten Testreport aufgerufen. Anschließend werden mit der Methode `getRemoteReport:Remote` über den Issue-Tracker die für den Durchlauf interessanten Issues aus dem Issue-Tracker geladen.

  Im Anschluss werden alle bekannten Filter auf die beiden Reports angewendet, indem die Methode `applyFilter(localReport, remoteReport):void` aufgerufen wird. Schließlich wird der finale Report, der alle noch relevanten Issues enthält, mit der Methode `uploadReport(finalReport):void` in den Issue-Tracker hochgeladen.

- `applyFilter(localReport, remoteReport) : Report` Mit dieser Methode werden alle bekannten Filter nacheinander auf die Reports angewendet. Für die Implementierung des IssueMakers wurde ein Filter für Duplikate und einer für irrelevante Issues angelegt.

  Da die GitLab-API es ermöglicht bereits beim Herunterladen der Issues aus dem Issue-Tracker eine Vorauswahl zu treffen, wird kein Filter benötigt, um die Issues zu entfernen, die nicht vom IssueMaker und aus dem Report von Cppcheck erstellt

wurden, zu filtern. Dies wird durch die Übergabe der Label bei der download-Methode der Issue-Tracker Implementierung erreicht. Am Ende der Filterung steht ein Report, der alle Issues enthält, die im Issue-Tracker angelegt werden sollen.

- `getRemoteReport()` Diese Methode spricht das Issue-Tracker-Interface an und holt sich alle Issues aus dem GitLab-Issue-Tracker, die die `Label` „IssueMaker“ und „cppcheck“ besitzen.
- `getLocalReport()` Im Gegensatz zu der Methode `getRemoteIssues()` wird mit dieser Methode der Parser angesprochen, der den Testreport liest.
- `uploadFinalReport(finalReport)` Hier wird nun wieder das Issue-Tracker-Interface angesprochen, um alle neuen Issues in den Issue-Tracker hochzuladen.

Report
-issueList
+init(issues) +addIssue(issue) +removeIssue(position) +getIssue(position):Issue +getIssues():issueList

Abbildung 5.1: Implementierung der Report-Wrapperklasse.

5.3 Die Datenklasse Report: Issues in einem Array

Der `Report` ist eine Wrapper-Klasse, die eine Liste mit Issues hält und über entsprechende Methoden den Zugriff auf einzelne Issues des Reports gewährt. Entsprechend hat die Klasse die üblichen Methoden für den Zugriff eines solchen Datentyps.

- `removeIssue(Position)` Mit dieser Methode kann ein Issue an einer beliebigen Position der internen Liste entfernt werden.
- `addIssue(Issue)` Diese Methode erlaubt es, Issues an die Liste anzuhängen.
- `getIssueList()` Damit innerhalb anderer Methoden leicht über alle Issues der Liste iteriert werden kann, kann auch die gesamte Liste zurückgegeben werden.

- `getIssue(Position)` Auch einzelne Issues müssen aus dem Report gelesen werden können.

5.4 Verschiedene Filter für die unterschiedlichen Herausforderungen

Das Entfernen von unerwünschten Issues aus dem Issue-Tracker oder aus dem Testreport wird über Filter ermöglicht. Das Filter-Interface ist in der Abbildung 4.6 zu sehen. Filter können sich entweder auf die Filterung des lokalen Reports, der aus dem aktuellen Test entstanden ist, oder auf den Report, der aus dem Issue-Tracker geladen wurde, beziehen. In Abhängigkeit davon, was sie filtern, gibt es die Methoden `filterLocalReport(LocalRepot, RemoteReport):LocalReport` und `filterRemoteReport(LocalReport, RemoteReport):RemoteReport`.

Diese geben jeweils einen gefilterten `LocalReport` oder `RemoteReport` zurück. Dadurch können alle Filter hintereinander ausgeführt werden, ohne zu wissen, welchen Report sie manipulieren. Damit in der Filterung auch Vergleiche beider Reports möglich sind, müssen beide als Parameter übergeben werden.

- `filterLocalReport(LocalReport, RemoteReport):LocalReport` Diese Methode filtert den Report, der am Ende des Testprozesses erzeugt wurde.
- `filterRemoteReport(LocalReport, RemoteReport):RemoteReport` Diese Methode filtert den Report, der aus den Issues des Issue-Trackers mit den entsprechenden `Labeln` erstellt wird.

Es wurden zwei Filter implementiert. Einer manipuliert den `LocalReport` und versucht Duplikate zu entfernen und der andere manipuliert den `RemoteReport` des Issue-Trackers und versucht irrelevante Issues zu entfernen.

Duplikate filtern

Der `DuplicateFilter` entfernt aus dem lokalen Report alle Issues, die bereits im Issue-Tracker also im `RemoteReport` enthalten sind. Laut des Konzepts kann dies erreicht

werden, indem ein neuer Issue mit den bereits im Issue-Tracker vorhandenen Issues verglichen wird. Wenn sie den gleichen Inhalt haben, wurde ein Duplikat entdeckt und es muss entfernt werden.

Im Kapitel 4 Konzept auf der Seite 27 wurden bereits der Aufbau eines GitLab-Issues gezeigt. Nicht alle Felder können über die GitLab-API beschrieben werden. Außerdem ist für den eigentlichen Inhalt der Fehlermeldung nur der Titel und die Beschreibung vorgesehen. Der erste Schritt würde also bedeuten, dass man die Beschreibung und den Titel vergleicht. Wenn diese identisch sind, wird davon ausgegangen, dass ein Duplikat identifiziert wurde.

Es ist nicht nötig ein komplexes Verfahren mittels Natuaral Language Processing wie in [47] von Runeson et al. oder in [49] von Wang et al. zu konstruieren um den Inhalt des Titels und der Beschreibung zu vergleichen. Da die Texte maschinell erstellt wurden, sind sie identisch, wenn es sich um denselben Fehler handelt. Es sollte aber auch beachtet werden, dass nicht der reine XML-Text für jeden Fund in die Beschreibung geschrieben wird. Stattdessen wird mittels Markdown Bausteinen aus dem XML-Text einen benutzerfreundlichen und übersichtlichen Issue erstellt.

Es wird also der Text des Bug-Reports durch Sonderzeichen und Leerzeichen sowie Zeilenumbrüche erweitert, damit er in GitLab nach der Markdown-Interpretation eine optimale Gestalt hat.

Dies funktioniert sehr gut, wenn ein Issue aus dem Report in GitLab erstellt wird. Wenn aber Issues aus dem Issue-Tracker geladen werden, bekommt man wieder nur den durch Markdown versetzten Text zurück. Die bedeutet, dass der Aufwand beim Vergleichen von zwei Issues größer ist, weil mehr Zeichen verglichen werden müssen. Zudem ist es so nicht mehr möglich zusätzliche Informationen in den Issue zu schreiben, die vom IssueMaker erzeugt wurden und nicht vom Testreport. Siehe dazu den Abschnitt zu 5.6 auf Seite 50.

Ein weiteres Problem sind Änderungen, die im Issue-Tracker am Issue selbst gemacht wurden. Ein Nutzer könnte zum Beispiel Zusatzinformationen in die Beschreibung des Fehlers schreiben. Es wäre nicht mehr möglich ein Duplikat zu dem Issue zu erkennen.

Um den Vergleich von zwei Issues zu reduzieren und Metainformationen sowie Änderungen der Beschreibung zuzulassen, wird einen Hashwert über die Daten des Bug-Reports gebildet und im Issue gespeichert. Leider bietet GitLab dafür kein eigenes Feld, weshalb der Hashwert in die Beschreibung des Issues geschrieben werden muss. Dadurch reicht es

aber aus, den Hash zu bilden und zu vergleichen, um ein Duplikat zu identifizieren. Es kommt nun nur zu unerwünschten Duplikaten, wenn der Hash im Issue-Tracker manipuliert wird.

Über welche Parameter des Reports der Hash gebildet wird, hängt vom Test-Tool ab und muss beim Schreiben des Parsers beachtet werden. Siehe hierfür den Abschnitt zum Parser auf Seite 47. Für die Bildung des Hashwertes sollte eine Hashfunktion aus dem Bereich der Kryptographie gewählt werden. Diese Funktionen haben in der Regel eine hohe Kollisionsresistenz [46].

Da bei einer Hashfunktion aus ein Wertebereich auf einen kleineren abgebildet wird, kann es passieren, dass unterschiedliche Eingangswerte denselben Hashwert bilden. Das nennt man Kollision. Mit der Entdeckung einer Kollision kann dann die Hashfunktion herausgefunden werden, was in der Kryptografie unerwünscht ist.

Eine Kollision zweier Hashwerte würde in diesem Anwendungsfall zu einem Duplikat oder irrelevanten Issue führen. Da dies zu vermeiden gilt, sollte also eine Hashfunktion mit geringer Kollisionswahrscheinlichkeit bzw. hoher Kollisionsresistenz gewählt werden. Da die Mengen an Issues und somit Menge an Hashwerten aber nicht außerordentlich groß sein wird, reicht eine beliebige bekannte Hashfunktion aus dem Bereich der Kryptografie aus.

Es kann eine Hashfunktion wie `md5` genommen werden. Diese hat auch den Vorteil, dass sie in der Python-Bibliothek `hashlib` vorhanden ist. Sie ist ausreichend resistent gegenüber Kollisionen und der Hashwert ist relativ kurz, sodass keine Übersichtlichkeit in der Issue-Beschreibung verloren geht, wenn der Hashwert dort aufgeführt wird. Er muss allerdings in der Beschreibung zu finden sein. Für den Cppcheck-Parser wird er deshalb in `HASH:` und `:HASH` eingeklammert.

Zwei Issues haben also den gleichen Inhalt, wenn die Hashwerte gleich sind und nur dann kann von einem Duplikat gesprochen werden. Der Filter vergleicht also die Hashwerte in dem lokalen Report mit den Hashwerte der Issues des Issue-Trackers und entfernt gefundene Duplikate aus dem lokalen Report, da im Issue-Tracker bereits zusätzliche Informationen vorhanden sein können, wie Zeitschätzungen, Kommentare und Zuweisungen.

Irrelevante Issues schließen

Irrelevante Issues werden genau wie Duplikate über den Hashwert identifiziert. Nur Issues, die aus dem Issue-Tracker geladen wurden, können irrelevant sein. Ein Issue ist irrelevant, wenn es bei einem erneuten Durchlauf des IssueMakers nicht mehr vorkommt. Das bedeutet, dass der Fehler der ursprünglich bestand, nicht mehr gefunden wurde. Dies kann auftreten wenn zum Beispiel ein Fehler eines Issues innerhalb der Lösung eines Fehlers eines anderen Issues gelöst wird. Der erste Fehler taucht dann nicht mehr auf, aber der Issue existiert noch im Issue-Tracker. Der Issue ist somit irrelevant.

Da Issues aber trotzdem bereits im Issue-Tracker von Nutzern verändert werden konnten, werden Issues, die als irrelevant erkannt wurden nicht gelöscht, sondern sie werden geschlossen. Dies erlaubt, dass sie nicht verschwinden, sondern in der Datenbank erhalten bleiben und für Recherchen oder Auswertungen in Betracht gezogen werden können.

Der Filter funktioniert, indem er den `LocaleReport` und `RemoteReport` vergleicht. Jeder Issue, der im `RemoteReport` aber nicht im `LocalReport` vorhanden ist, ist irrelevant geworden. Der Filter entfernt ihn aus dem `RemoteReport` und schließt ihn im Issue-Tracker, sodass spätere Filter eine kürzere Liste durcharbeiten müssen.

5.5 Parser: Das Lesen und Verstehen verschiedener Testreports

Die Aufgabe des Parsers ist es, die Reports des vorangegangenen Testprozesses in eine Form zu bringen mit der der IssueMaker arbeiten kann. Das bedeutet in erster Linie die Informationen in den Titel und Beschreibung des Issues zu schreiben. Deshalb hat das `ReportParser`-Interface auch nur eine Methode (siehe Abbildung 4.3). Es werden hier zwei unterschiedliche Parser gezeigt, um zu zeigen, dass es möglich ist aus verschiedenen Testreports Issues zu erstellen. Zum einen wird ein Testreport eines bekannten Analyse-Tools (Cppcheck) im XML-Format eingelesen, zum anderen wird die Ausgabe eines spezialisierten Regressionstests in Issues umgewandelt.

- `parseReport(Filename):Report` Diese Methode muss auf den entsprechenden Testreport angepasst werden. Das heißt, dass sie den richtigen Dateinamen

übergeben bekommt, sodass sie die Report-Datei finden kann. Zusätzlich muss innerhalb dieser Methode aus dem Report des Testprozesses ein Report des IssueMakers gemacht werden, also eine Liste aus Issues.

5.5.1 Parsen des Cppcheck Reports

Für Cppcheck muss ein Parser implementiert werden, der den XML-Report in Issues umwandelt. Im Abschnitt 4.3 auf der Seite 32 wurde bereits der Aufbau eines Fehlers in Cppcheck erläutert. Die einzelnen Felder des Fehlers werden in der Klasse `CppcheckError` 5.2 geladen und dort mithilfe der `toString()`-Methode in Titel und Beschreibung eines GitLab-Issues geformt. Außerdem wird in der `toString()`-Methode der Inhalte des Fehlers mit Markdown-Bausteinen so angepasst, dass ihn GitLab später strukturiert darstellen kann.

Für die Identifizierung eines Duplikats wird im Zuge der Erzeugung der Issue-Beschreibung auch der Hash generiert und angehängt. Für den Hash ist es wichtig, dass er genügend Informationen enthält, die spezifisch sind für den Fehler, den er markiert. Es würde nicht reichen nur die `Message` und das Gewicht in den Hash aufzunehmen, da diese bei der statischen Analyse für verschiedene Fehler identisch sein können. Deshalb wird der Hash über die Felder `Message` und `Locations` gebildet. Diese reichen aus, da die `Message` Informationen über den Fehlertyp und die `Locations` die Dateien und Zeilen in denen der Fehler gefunden wurde, enthält. Die anderen Felder können zwar auch in den Hashwert aufgenommen werden aber sie bringen keinen Mehrwert. Da sie nur allgemeine Fehlerinformationen beinhalten.

Da ein Cppcheck-Fehler eine Liste von `Locations` beinhalten kann, gibt es eine Klasse, die `CppErrorLocation` heißt. Sie enthält die Informationen des `Location`-Elements des XML-Baums und mithilfe einer eigenen `toString()`-Methode bereitet die Klasse die Informationen so auf, dass sie in der `toString()`-Methode der `CppcheckError`-Klasse einfach aufgerufen werden kann.

Nach Abschluss des Parsens des Cppcheck Reports, werden alle identifizierten `CppcheckErrors` in Issues umgewandelt indem die `Message` des Cppcheck-Fehlers als Titel und die restlichen Informationen als Beschreibung des Issues verwendet werden. Anschließend werden die Issues in der Report-Klasse gekapselt und an den `ReportManager` zurückgegeben.

CppcheckError
-id
-severity
-message
-verbose
-cwe
-date
-inconclusive
-location
-meta
-hash
+toString():String
-createCompareHash:Hash

Abbildung 5.2: Implementierung der Parameter einer Cppcheck Fehlermeldung.

5.5.2 Parsen des Reports des Regressionstest

Im Gegensatz zu Cppcheck wird hier die Meldung nicht im XML-Format verpackt. Allerdings sind die Informationen auch nicht so umfangreich. Es gibt nur zwei Felder, die aus dem Report des Regressionstests gezogen werden. Zum einen das Feld `Executing Test`, welches zum Beispiel den Wert `regression_test` beinhaltet. Zum anderen das Feld `Verdict`, welches das Ergebnis des Tests enthält.

Um diese Informationen aus dem Report ziehen zu können wurde ein Text-Parser geschrieben. Dieser liest die Datei als Ganzes ein und durchsucht mittels Pattern-Matching den String nach den Schlüsselwörtern `Executing Test` und `Verdict`. Dies ist keinesfalls eine sichere Herangehensweise, aber da der Report in keinem strukturierten Format vorliegt, ist dies die einzige Möglichkeit. Zusätzlich zu diesen Informationen wird das Datum des Testlaufs und wieder ein Hashwert berechnet. Der Hashwert des Parsers für den Regressionstest beinhaltet den Testplan, das Ergebnis (`Verdict`) und den Namen des Tools.

Das Datum wird nicht in die Berechnung des Hashwerts aufgenommen. Wenn der Regressionstest jede Nacht angestoßen wird und fehlschlägt, würden jedes Mal neue Issues erzeugt werden, weil das Datum den Hashwert abändert. Die alten Issues würden zwar im Zuge der Filterung von irrelevanten Issues geschlossen werden, aber das ist nicht erwünscht. Denn die Issues werden nicht irrelevant, wenn sie an verschiedenen Tagen gefunden werden.

Anschließend werden die Informationen des Reports wieder in einen Issue und anschließend einen Report gekapselt und im `ReportManager` weiter verarbeitet.

5.6 Erfassen von Metainformationen für spätere Auswertungen

Metainformationen sind für die Auswertung der Issues im späteren Projektverlauf interessant und können außerdem Informationen für den `Context`-Bereich des IEEE-29119 Incident Reports aus [29] liefern. Für die Identifikation von Duplikaten dürfen sie aber nicht in den Hash mit aufgenommen werden. Außerdem sollten sie am Ende der Beschreibung des Issues angehängt werden, da sie in erster Linie kaum Hilfe bieten, den Fehler zu beheben.

Zu den Metainformationen gehört das Datum, an dem der Issue erstellt wurde. Dies kann bei der Priorisierung helfen. Issues, die sehr alt sind, können entweder gelöscht werden, oder sollten höher Priorisiert werden, damit sie nicht vergessen werden. Außerdem ist der Commit und der Branch auf dem der Fehler gefunden wurde, der in dem Issue benannt wird, interessante und möglicherweise hilfreiche Kontextinformation. Dies sind lediglich Beispiele für Informationen, die hilfreich sein können und die gegebenenfalls nicht in einem Report mitgeliefert werden und deshalb durch den IssueMaker automatisch an jedem Issue einen Platz finden könnten.

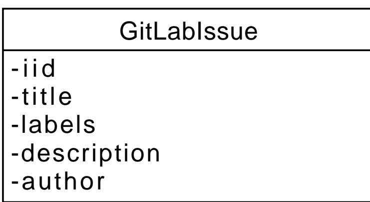

Abbildung 5.3: Implementierung aller Parameter eines Issues aus GitLab, die für die Erstellung zu beachten sind.

5.7 Der Issue, wie ihn GitLab sieht

Grundsätzlich sollte ein Issue wie in IEEE-29119 [29] bereitgestellt werden. Nun wurde bereits im Konzept die Differenz zwischen dem Incident Report und dem GitLab-Issue vorgestellt. Neben diesen Unterschieden sind Informationen wie die `Timing Information` und GitLab eigene Felder für den IssueMaker nicht interessant, da diese nicht vom Report mitgeliefert werden, automatisch von GitLab bei der Erstellung und Bearbeitung gesetzt werden oder durch die API nicht änderbar sind. Für den Issue im IssueMaker sind lediglich die Parameter `title`, `description`, `labels` und `iid` interessant. Das hat den Hintergrund, dass nur der Titel, die Beschreibung und die Label vom IssueMaker manipuliert werden und der Issue-Identifier (`iid`) zum Schließen und Löschen von Issues benutzt werden kann. Die anderen Werte werden intern nicht benötigt. Daraus folgt der Aufbau wie er in dem Klassendiagramm 5.3 zu sehen ist.

5.8 Einbindung des Programms in die GitLab Pipeline

Trotz der Anpassungen der Komponenten gegenüber dem vorgestellten Konzept, ist die Architektur unverändert.

Damit das Programm in einer GitLab-Pipeline laufen kann, muss es noch in einen Docker-Container eingebettet werden. Dies wird erreicht, indem ein `Dockerfile` angelegt wird, welches die Dateien des IssueMakers sowie Python beinhaltet.

Anschließend muss das Image des Containers angelegt und in eine GitLab bekannte Docker-Registry geladen werden. Hierfür kann zum Beispiel Docker-Hub [8] genommen werden. Da die HAW eine eigene Registry für Docker-Container führt, habe ich mich entschieden diese für das Image zu nutzen.

Danach kann das Image über eine `.gitlab-ci.yml` wie im zum Beispiel im Listing 6.7 in eine GitLab Pipeline eingebunden werden.

6 Ergebnisse

Im Folgenden werden die Ergebnisse der Arbeit beschrieben. Zu Beginn wird eine Beispielausführung des IssueMakers zusammen mit Cppcheck in GitLab gezeigt. Dafür wird ein einfaches fehlerbehaftetes C++ Programm geprüft. Anschließend werden verschiedene Testfälle betrachtet und die Ergebnisse der Anforderungsanalyse gegenübergestellt. Am Ende werden ausgewählte Testfälle mit dem Regressionstest durchgeführt.

6.1 Ausführung des IssueMakers.

Um den IssueMaker zu starten, müssen verschiedene Parameter mitgegeben werden. Um sich ein Überblick zu verschaffen, kann mit dem Aufruf in Listing 6.1 die Hilfe des Issuemakers aufgerufen werden. Die Ausgabe dieses Aufrufs ist in Listing 6.2 zu sehen. Elemente die in eckigen Klammern eingerahmt sind, können weggelassen werden, wenn der IssueMaker mit der GitLab-Instanz der HAW arbeitet.

```
1 IssueMaker.py -h
```

Listing 6.1: Start des IssueMakers mit Hilfestellung.

```
1 usage: IssueMaker.py [-h] -t <ACCESS TOKEN> -p <PROJECT ID> -f <PATH
     TO
2                 REPORT> -tt <TEST TOOL> [-it <ISSUE-TRACKER>]
3                 [-u <API-URL>]
4
5 IssueMaker. Perfect to generate issues in your favourite issue-
     tracker. At
6 least four arguments are needed for Gitlab of the HAW-Hamburg; six
     for
7 every other issue-tracker.
8
```

```
9   optional arguments:
10    -h, --help show this help message and exit
11    -t <ACCESS TOKEN> Access token for the api of the choosen issue-
         tracker.
12    -p <PROJECT ID> Project identifier for the issue-tracker project
13                     destination.
14    -f <PATH TO REPORT> Path to the generated report file.
15    -tt <TEST TOOL> Identifier for the test tool, that generated the
16                     report.
17    -it <ISSUE-TRACKER> Identifier or name of the chosen issue-tracker.
18    -u <API-URL> Api url of the issue-tracker.
19
20  Process finished with exit code 0
```

Listing 6.2: Konsolenausgabe des IssueMakers mit Hilfestellung.

Für den Aufruf innerhalb der `.gitlab-ci.yml` kann der Aufruf in Listing 6.3 übernommen werden. Die Variablen müssen dafür ebenfalls in der yml-Datei oder im GitLab-Projekt hinterlegt sein. Variablen mit dem Präfix `CI` werden von GitLab bereitgestellt. Eine komplette Liste aller vordefinierten Variablen, die GitLab von bereitstellt werden, ist in [24] zu finden.

```
1  IssueMaker.py - t $TEST_ACCESS_TOKEN -p \
2  $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
```

Listing 6.3: Start des IssueMakers mit Parametern.

6.2 Konfiguration und Allgemeiner Test mit Cppcheck

Für den Testdurchlauf ist in Listing 4.1 die für den Testlauf vollständige `.gitlab-ci.yml` abgebildet. Der Test wird mit einem Testreport von Cppcheck durchgeführt.

Man könnte für den Test auch einen fertig erstellten Testreport nehmen und im Repository ablegen, aber um auch das Zusammenspiel zwischen Erstellung des Testreports und Generierung der Issues zu demonstrieren wird nicht nur Cppcheck in der Pipeline ausgeführt, sondern auch Programmcode in das Repository geladen.

Der Code ist in den Listings 6.4, 6.5 und 6.6 zu sehen. Er ist mit diversen Fehlern bestückt, die Cppcheck erkennen kann. Das Hochladen des Codes startet die Pipeline automatisch. Dies kann in Gitlab über das Menü „CI/CD“ und „Pipelines“ verfolgt werden. Ein erfolgreicher Durchlauf ist in der Abbildung 6.1 zu sehen.

Die einzelnen Stationen der Pipeline können durch einen Klick ebenfalls betrachtet werden. Dies ist beispielhaft in der Abbildung 6.2 zu sehen.

Wenn der IssueMaker erfolgreich war und wie in diesem Fall durch Cppcheck Fehler im Code gefunden wurden, werden Issues erstellt. Ein Issue, der aus einem Cppcheck-Fund erstellt worden ist, ist in der Abbildung 6.3 zu sehen.

Die für die Entwickler wichtigen Informationen sind im oberen Bereich der Beschreibung des Issues dargestellt und die Metainformationen wie im Konzept erklärt darunter. Am Ende steht der Hashwert über den der IssueMaker Duplikate und irrelevante Issues identifizieren kann.

```
1  #include <iostream>
2  #include "SomeClass.h"
3  
4  int main() {
5   std::cout << "Hello, World!" << std::endl;
6   int k = 5000;
7   int p = 9;
8   int *ptr = NULL;
9   std::cout << p<< std::endl;
10  std::cout << *ptr <<std::endl;
11  SomeClass someClass = SomeClass();
12  SomeClass someClass1 = SomeClass(345, "Some Class");
```

```
13  std::cout << someClass1 << std::endl;
14  std::cout << someClass1.toString() << std::endl;
15  someClass1 = nullptr;
16  std::cout << someClass1.toString() << std::endl;
17  someClass = someClass1;
18
19  return 0;
20 }
```

Listing 6.4: Main.cpp.

```
1 #ifndef TESTPROJECT_SOMECLASS_H
2 #define TESTPROJECT_SOMECLASS_H
3
4 #include <string>
5
6 class SomeClass {
7 private:
8  SomeClass()=default;
9  int index;
10  std::string text;
11 public:
12  SomeClass(int index, std::string text):
13             index(index), text(text){}
14  std::string toString();
15  int getIndex();
16  int getText();
17  std::string getText();
18 };
19
20 #endif //TESTPROJECT_SOMECLASS_H
```

Listing 6.5: SomeClass.h.

```
1 #include "SomeClass.h"
2
3 std::string SomeClass::toString() {
4  return index +": " + text;
5 }
```

Listing 6.6: SomeClass.cpp.

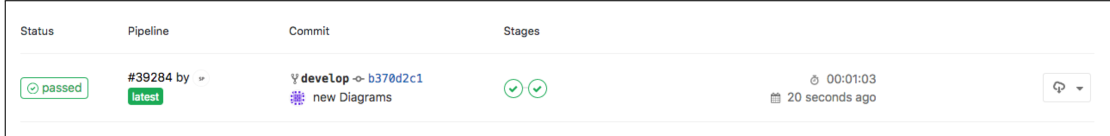

Abbildung 6.1: Darstellung einer erstellten Pipeline in GitLab.

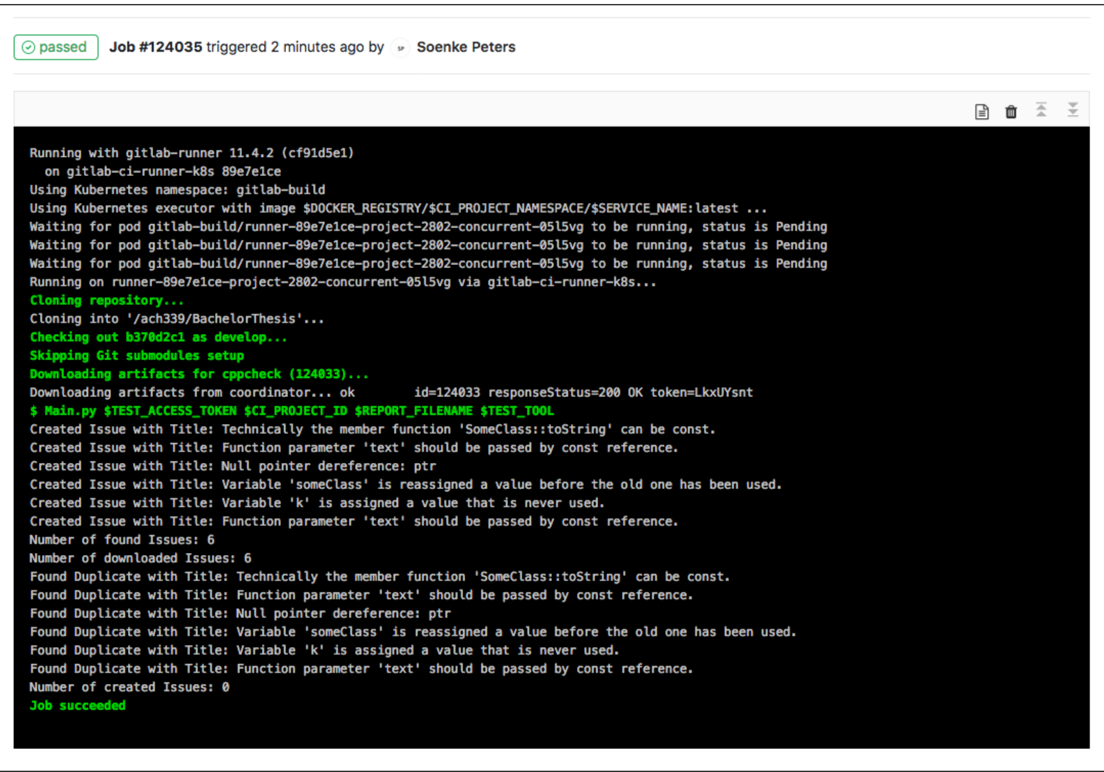

Abbildung 6.2: Ausgabe des Terminals während der Ausführung des Issuemakers.

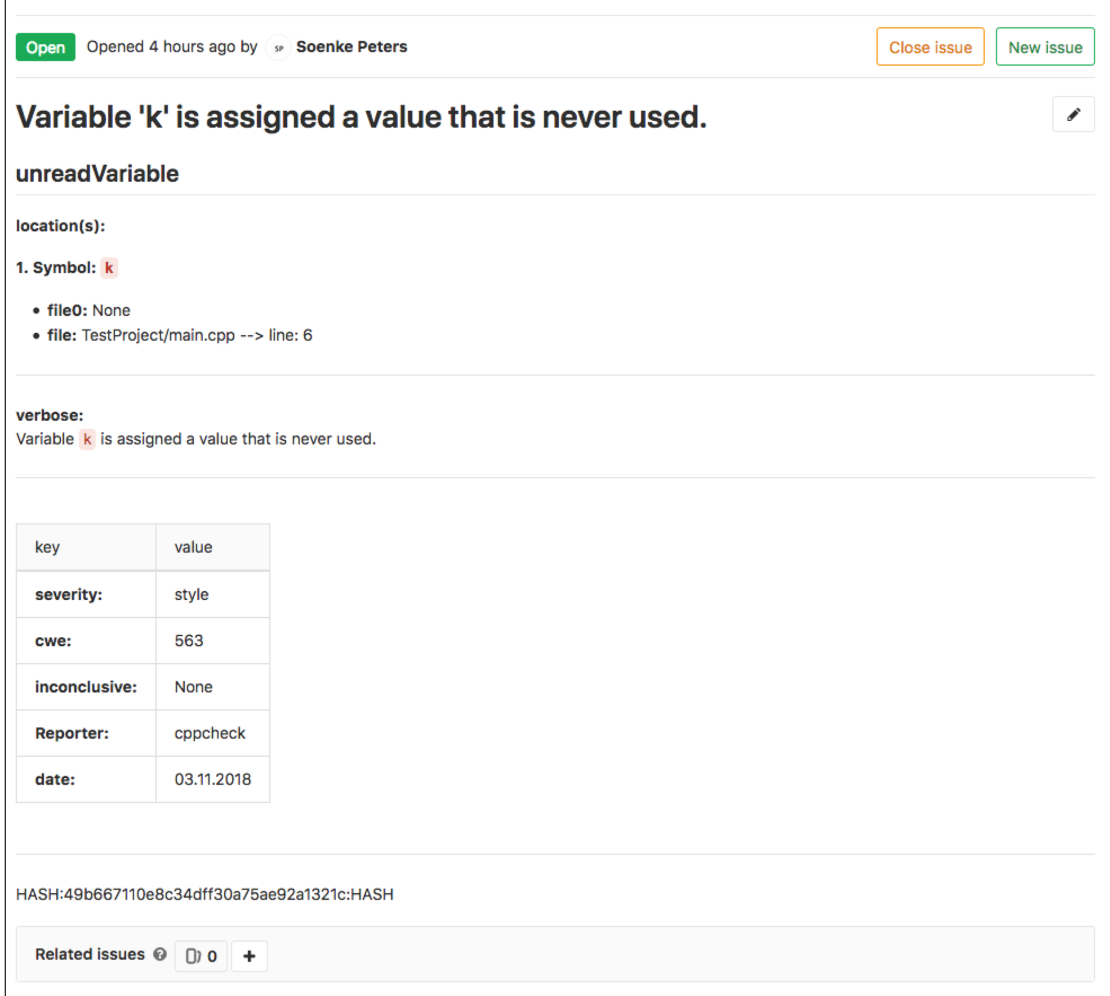

Abbildung 6.3: Darstellung eines erstellten Issues in GitLab.

6.3 Definition der Testfälle auf Basis der Anforderungsanalyse

In den Tabellen 6.1 und 6.2 sind Testfälle aufgelistet, die sich aus den zu Beginn erläuterten Anforderungen und der abschließenden Implementierung ergeben.

Der Systemtest soll zeigen, dass alle zu Beginn erläuterten Anforderungen, die implementiert wurden, erfüllt wurden. Die erste Anforderung an das Programm ist, dass es automatisch im Anschluss eines Testlaufs auf Basis dessen Reports Issues erstellen kann. Der IssueMaker muss dafür in einer Pipeline hinter einen solchen Testlauf geschaltet sein. Außerdem muss ein Parser existieren, den der IssueMaker nutzen kann, um den Testreport des Testlaufs einzulesen. Die folgenden Testfälle setzen voraus, dass der IssueMaker einen solchen Parser besitzt und dass er in der GitLab-Pipeline ausgeführt wird. Die Grundfunktionalität wird in den Testfällen 1, 2, 3 und 4 ab Seite 64 geprüft.

Für die zweite Anforderung gilt, dass verschiedene Testreports aus verschiedenen Quellen vom IssueMaker eingelesen werden können. Es wird einerseits der im XML-Format vorliegende Testreport von Cppcheck (Testfall 1 bis 11 ab Seite 62), andererseits der Testreport eines Regressionstests (Testfall 1R bis 4R ab Seite 83), der in keinem gängigen Format vorliegt, eingelesen. Das XML-Format steht hier als Repräsentant für formatierte Reports. Der IssueMaker ist immer dann dazu imstande einen Report einzulesen, wenn ein entsprechender Parser implementiert wurde. Eine klare Grenze ist vorhanden, wenn das Format des Testreports keine wiederkehrenden Strukturen aufweist, wie zum Beispiel Schlüsselwörter.

Die Anpassbarkeit und Erweiterbarkeit des IssueMakers wird zusätzlich durch die Bereitstellung von Interfaces für Filter, Parser und Issue-Tracker gegeben. Möchte ein Anwender den IssueMaker für eine bestimmte Testreport-Quelle nutzen, muss ein entsprechender Parser geschrieben werden. Soll die Filterung von Issues ausgebaut werden, können weitere Filter hinzugefügt werden. Wenn ein anderer Issue-Tracker angebunden werden soll, kann dies durch eine weitere Issue-Tracker Klasse geschehen.

Die Erkennung von Duplikaten sowie das Schließen von irrelevanten Issues wird durch die Evaluierung des Hashwertes in der Beschreibung eines Issues ermöglicht. Dies funktioniert mit Einschränkungen, denn zumindest für GitLab gilt, dass Nutzer alle Felder eines Issues ändern können und somit die Gefahr besteht, dass ein Anwender den Hash ändert. Die Testfälle 6.4.5, 6.4.6 und 6.4.7 zeigen das Erkennen der Duplikate und der irrelevanten

Issues mit der Hilfe des Hashwerts innerhalb der Beschreibung auf. Der Testfall 6.4.8 zeigt die Probleme dieses Lösungsansatzes auf. Eine Alternative zu GitLab wird im Abschnitt 7.3 vorgestellt.

Damit der IssueMaker keine Issues entfernt, die von einem Nutzer erstellt wurden, versieht er jeden von sich erstellten Issue mit einem Label. Der IssueMaker bezieht lediglich Issues aus dem Issue-Tracker mit dem passenden Label in die Filterung mit ein. Die Testfälle 6.4.9, 6.4.10 und 6.4.11 zeigen, dass der Nutzer diese Label nicht an eigene Issues binden darf, da sie ansonsten vom IssueMaker mit beachtet werden. Dies verhindert, dass der IssueMaker vom Nutzer erstellte Issues beachtet. Allerdings soll der Nutzer Informationen an vom IssueMaker erstellte Issues anhängen können. Da Issues nur durch die Kombination aus Label und Hash vom IssueMaker erkannt und evaluiert werden. Kann der Nutzer auch Issues ändern, die vom IssueMaker erstellt wurden, solange er den Hash und die Label unberührt lässt.

Die Issues des IssueMakers sollten sich an IEEE-29119 orientieren. Da GitLab dies aber nicht vorsieht und die Testreports nicht unbedingt alle Daten dafür vorsehen, konnte dies nicht vollständig erfüllt werden. Stattdessen wurden die Informationen so aufbereitet, dass ein Entwickler die wichtigsten Daten schnell erfassen kann. Der Aufbau der Issues im entsprechenden Tracker kann durch Änderungen des Parsers einfach angepasst werden, sodass auch individuelle Anforderungen umgesetzt werden können.

Nr.	Beschreibung	
1	Vorbedingungen	Es gibt keinen Programmcode, der getestet werden kann.
	Ausführung	Die Pipeline wird gestartet.
	Ergebnis	Cppcheck erzeugt keinen Report. Der IssueMaker soll keine Issues erzeugen.
2	Vorbedingungen	Es gibt fehlerfreien Programmcode.
	Ausführung	Die Pipeline wird gestartet.
	Ergebnis	Der Report von Cppcheck findet keine Fehler und es wird kein Report erzeugt. Der IssueMaker sollte keine Issues erzeugen.
3	Vorbedingungen	Es gibt mangelhaften Programmcode.
	Ausführung	Die Pipeline wird gestartet.
	Ergebnis	Der von Cppcheck erstellte Report hat Elemente, die vom IssueMaker als Issues im Projekt angelegt werden.
4	Vorbedingungen	Nach Test 3 wird die Pipeline ohne Codeänderungen erneut ausgeführt.
	Ausführung	Die Pipeline wird gestartet.
	Ergebnis	Der IssueMaker macht keine Änderungen an den Issues im Issue-Tracker.
5	Vorbedingungen	Im Issue-Tracker existieren Fehler, die vom IssueMaker erstellt wurden.
	Ausführung	Es wird ein Fehler eines Issues, das vom IssueMaker erstellt wurde, behoben, aber nicht der Issue geschlossen. Die Pipeline wird gestartet.
	Ergebnis	Der IssueMaker schließt das erstellt Issue.
6	Vorbedingungen	Im Issue-Tracker existieren Fehler, die vom IssueMaker erstellt wurden.
	Ausführung	Es wird ein vom IssueMaker erstelltes Issue gelöst. Die Pipeline wird gestartet.
	Ergebnis	Die noch bestehenden Issues werden nicht manipuliert und der IssueMaker erstellt das gelöste Issue nicht erneut.
7	Vorbedingungen	Im Issue-Tracker existieren Fehler, die vom IssueMaker erstellt wurden.
	Ausführung	Es wird ein Issue, welches vom IssueMaker erzeugt wurde, bis auf den Hash komplett geändert. Die Pipeline wird gestartet.
	Ergebnis	Die Issues werden nicht manipuliert.

Tabelle 6.1: Testfälle

Nr.	Beschreibung	
8	Vorbedingungen	Im Issue-Tracker existieren Fehler, die vom IssueMaker erstellt wurden.
	Ausführung	Der Hash eines Issues, das vom IssueMaker erstellt wurde, wird entfernt. Die Pipeline wird gestartet.
	Ergebnis	Der manipulierte Issue wird erneut vom IssueMaker erzeugt. Es gibt ihn zweimal. Einmal mit und einmal ohne Hash.
9	Vorbedingungen	Im Issue-Tracker existieren Fehler, die vom IssueMaker erstellt wurden.
	Ausführung	Das Label mit dem Namen `IssueMaker` eines Issues, das vom IssueMaker erstellt wurde, wird entfernt. Die Pipeline wird gestartet.
	Ergebnis	Der Issue wird erneut vom IssueMaker erzeugt. Es gibt ihn zweimal. Einmal mit und einmal ohne dem entfernten Label.
10	Vorbedingungen	Es existieren Issues im Issue-Tracker, die nicht vom IssueMaker erstellt wurden. Der zu testende Programmcode enthält Fehler.
	Ausführung	Die Pipeline wird gestartet.
	Ergebnis	Es werden Issues auf Basis des erzeugten Testreports erstellt. Die bereits erstellten Issues werden nicht manipuliert.
11	Vorbedingungen	Im Issue-Tracker existieren Fehler, die vom IssueMaker erstellt wurden. Deren Label sind bekannt.
	Ausführung	Es wird manuell ein Issue erstellt, welches die selben Label hat, wie die Issues, die vom IssueMaker erstellt wurden. Die Pipeline wird gestartet.
	Ergebnis	Es werden Issues auf Basis des erzeugten Testreports erstellt. Die bereits erstellten Issues werden nicht manipuliert.

Tabelle 6.2: Fortsetzung der Testfälle

6.4 Ausführung der definierten Testfälle mit Hilfe von GitLab und Cppcheck

Um den Test in eine kontrollierte Umgebung zu bringen, wird ein neues Projekt in GitLab erzeugt. Das Projekt wird so konfiguriert, dass eine Pipeline vorhanden ist. Dafür wird die `.gitlab-ci.yml` aus dem Listing 6.7 in dem Repository abgelegt. Außerdem werden alle nötigen Parameter in den Projekteinstellungen erstellt (siehe Anhang 106).

```
1  variables:
2     DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"
3     REPORT_FILENAME: "report.xml"
4     TEST_TOOL: "cppcheck"
5     SERVICE_NAME: bachelorthesis
6     CI_PROJECT_NAMESPACE: ach339
7     CODE_PATH: "."
8
9  stages:
10   - testanddeploy
11   - generate
12
13 cppcheck:
14   stage: testanddeploy
15   image: soenke2/cppcheckforissuemaker
16   script:
17     - cppcheck --enable=all --std=posix --inconclusive \
18     --force $CODE_PATH --xml 2> $REPORT_FILENAME
19   artifacts:
20     paths:
21       - report.xml
22
23 issueMaker:
24   stage: generate
25   allow_failure: true
26   image: $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
27   script:
28     - IssueMaker.py -t $TEST_ACCESS_TOKEN -p\
29      $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
30   dependencies:
31     - cppcheck
```

```
32 }
```

Listing 6.7: .gitlab-ci.yml des GitLab-Projects mit Cppcheck und IssueMaker.

6.4.1 Testfall 1

Der erste Testfall kann ausgeführt werden. Abbildung 6.4 zeigt, dass, da es keinen Programmcode gibt, bereits der erste Pipeline-Schritt mit Cppcheck fehl. Der Grund ist in der Konsolenausgabe in Abbildung 6.5 zu sehen. „No C or C++ source files found.“ erklärt eindeutig, dass die Funktionsweise von Cppcheck eingeschränkt ist, wenn es keinen Programmcode zur Überprüfung vorfindet. Da die erste Stufe der Pipeline fehlgeschlagen ist, startet die darauf folgende nicht und deshalb werden auch keine Issues erzeugt.

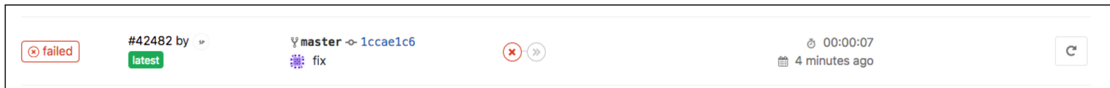

Abbildung 6.4: Testfall 1: Ansicht der Pipeline in GitLab mit fehlgeschlagener Cppcheck-Stufe.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image soenke2/cppcheckforissuemaker ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0q2zsd to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0q2zsd to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0q2zsd via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 1ccae1c6 as master...
Skipping Git submodules setup
$ cppcheck --enable=all --std=posix --inconclusive --force $CODE_PATH --xml 2> $REPORT_FILENAME
cppcheck: No C or C++ source files found.
ERROR: Job failed: command terminated with exit code 1
```

Abbildung 6.5: Testfall 1: Ausgabe der Konsole der Cppcheck-Pipeline-Stufe, wenn bei dem Aufruf in der .gitlab-ci.yml ein Pfad angegeben wird, an dem sich kein Programmcode befindet.

6.4.2 Testfall 2

Für den zweiten Testfall wird fehlerfreier Programmcode benötigt. Fehlerfrei bedeutet hier, dass Cppcheck keine Fehler entdeckt. Der Programmcode im Listing 6.8 wird in das Repository hochgeladen.

Die Pipeline startet nach dem Hochladen automatisch. Der erfolgreiche Lauf wird in der Übersicht der GitLab-Pipeline durch einen grünen Haken für jede erfolgreiche Station dargestellt, siehe Abbildung 6.6.

In der Konsolenausgabe 6.7 ist die Ausgabe des IssueMakers gezeigt. In den letzten vier Zeilen ist zusammengefasst, dass kein Issue gefunden wurde, keiner aus dem Issue-Tracker geladen wurde, weil dieser noch leer ist und somit kein Issue erstellt wurde.

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello, World!" << std::endl;
5     return 0;
6 }
```

Listing 6.8: Programmcode ohne für Cppcheck identifizierbare Mängel.

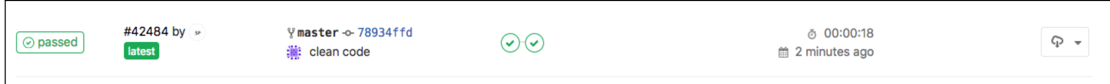

Abbildung 6.6: Testfall 2: Ansicht der Pipeline-Übersicht in GitLab nach dem erfolgreichen Durchlauf und mit fehlerfreiem Programmcode.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0hksvg to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0hksvg to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0hksvg via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 78934ffd as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (133993)...
Downloading artifacts from coordinator... ok        id=133993 responseStatus=200 OK token=-g3kPbs9
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Number of found Issues: 0
Number of downloaded Issues: 0
Number of created Issues: 0
Job succeeded
```

Abbildung 6.7: Testfall 2: Konsolenausgabe des IssueMakers im Falle eines leeren Issue-Trackers und eines leeren Testreports.

6.4.3 Testfall 3

Als Nächstes wird ein Fehler in den Programmcode implementiert. Dafür wird eine Integer-Variable definiert und initialisiert aber nicht genutzt. Der Programmcode ist in Listing 6.9 zu sehen. Dies ist ein Verstoß der Regeln, die in Cppcheck definiert sind.

Das Hochladen dieses mangelhaften Codes lässt wieder die Pipeline starten. Nachdem erfolgreichen Durchlauf ist in der Zusammenfassung des IssueMakers in Abbildung 6.8 zu lesen, dass ein Issue mit dem Titel „Variable 'a' is assigned a value that is never used." erstellt wurde.

Die erfolgreiche Generierung des Issues ist im Issue-Tracker GitLabs in Abbildung 6.9 zu sehen.

Der IssueMaker hat erfolgreich den Report von Cppcheck eingelesen, die Fehlermeldung aufbereitet und ein Issue erstellt.

```
1 #include <iostream>
2 
3 int main() {
4     int a = 4;
5     std::cout << "Hello, World!" << std::endl;
6     return 0;
7 }
```

Listing 6.9: Main.cpp.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0lqbbv to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0lqbbv to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0lqbbv via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 49641cc7 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (133996)...
Downloading artifacts from coordinator... ok        id=133996 responseStatus=200 OK token=no54svRQ
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Created Issue with Title: Variable 'a' is assigned a value that is never used.
Number of found Issues: 1
Number of downloaded Issues: 0
Number of created Issues: 1
Created Issue: Variable 'a' is assigned a value that is never used.
Job succeeded
```

Abbildung 6.8: Testfall 3: Konsolenausgabe des IssueMakers nach erfolgreichem Lauf mit Fehlerfund.

Abbildung 6.9: Testfall 3: Ansicht des Issue-Trackers samt neu generiertem Issue.

6.4.4 Testfall 4

Für den vierten Fall müssen keine Änderungen am Code gemacht werden. Es geht darum, dass Issues, die vom IssueMaker erzeugt wurden, nicht erneut erstellt werden. Deshalb muss lediglich die Pipeline manuell neugestartet werden. Nachdem sie fertig ist, kann in der Konsolenausgabe 6.10 gelesen werden, dass der Report einen Fehler beinhaltet hat („Number of found Issues: 1"). In der darauf folgenden Zeile steht, dass auch im Issue-Tracker ein Issue gefunden wurde und dieses geladen wird. Anschließend ist zu sehen, dass der gefundene Fehler bereits im heruntergeladenen Issue Erwähnung findet. Das Duplikat wurde verhindert und es wurde kein neuer Issue generiert.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-05ggzk to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-05ggzk to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-05ggzk via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 49641cc7 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (134004)...
Downloading artifacts from coordinator... ok        id=134004 responseStatus=200 OK token=2rry5Pbb
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Created Issue with Title: Variable 'a' is assigned a value that is never used.
Number of found Issues: 1
Number of downloaded Issues: 1
Found Duplicate with Title: Variable 'a' is assigned a value that is never used.
Number of created Issues: 0
Job succeeded
```

Abbildung 6.10: Testfall 4: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der gefundene Fehler ist derselbe wie der, der bereits im Issue-Tracker vorhanden ist. Es wird kein neuer Issue im Issue-Tracker erstellt.

6.4.5 Testfall 5

Der fünfte Testfall prüft die alltägliche Arbeit mit dem Programm. Ein Entwickler wählt ein Issue und behebt den Fehler, der dort beschrieben ist. Als Simulation wird der ursprünglich fehlerfreie Programmcode wiederhergestellt, indem die Zeile mit der unnützen Integer Initialisierung entfernt wird. Wird die Pipeline erneut gestartet, wird der Fehler nicht erneut durch Cppcheck gefunden und er taucht nicht mehr in dem neuen Report auf. In der Konsolenausgabe des IssueMakers in der Abbildung 6.11 ist zu sehen, dass kein Issue gefunden wurde, aber einer heruntergeladen wurde. Der heruntergeladene Issue wird als irrelevant erkannt, weil der Fehler behoben wurde.

Der IssueMaker schließt den irrelevanten Issue und erzeugt keinen weiteren, da der Report keine Fehler enthalten hat. Dieses Verhalten kann ungewohnt sein, da ein Issue geschlossen wird, nachdem die Arbeit am Programmcode beendet wurde. Es kann aber noch Arbeiten am Issue selbst geben. Abhängig vom Prozess des Entwicklerteams müssen noch Metainformationen, wie zum Beispiel die Arbeitszeit, oder Kommentare an den Issue angehängt werden, bevor es geschlossen wird.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0hzb6m to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0hzb6m to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0hzb6m via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 416c0f30 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (134011)...
Downloading artifacts from coordinator... ok        id=134011 responseStatus=200 OK token=ZFbZwL4o
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Number of found Issues: 0
Number of downloaded Issues: 1
Closed Issue with title: Variable 'a' is assigned a value that is never used.
Number of created Issues: 0
Job succeeded
```

Abbildung 6.11: Testfall 5: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Issue-Tracker enthält einen Issue. Der neu erstellte Report findet keinen Fehler. Der irrelevant gewordene Issue wird geschlossen.

6.4.6 Testfall 6

Testfall sechs ist vom Verhalten weitestgehend äquivalent zum fünften Testfall. Allerdings wird der Issue, der den Fehler beschreibt, manuell geschlossen bevor der neue Programmcode hochgeladen wird.

Wie in der Abbildung 6.12 zu lesen ist, wird weder ein Issue gefunden noch einer erneut heruntergeladen. Im Gegensatz zum Testfall Nummer 5 wird also auch kein irrelevanter Issue identifiziert und kein neuer Issue erzeugt.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-04pgmw to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-04pgmw to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-04pgmw to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-04pgmw via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 7ac985a9 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (136975)...
Downloading artifacts from coordinator... ok        id=136975 responseStatus=200 OK token=ye6uLQbd
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Number of found Issues: 0
Number of downloaded Issues: 0
Number of created Issues: 0
Job succeeded
```

Abbildung 6.12: Testfall 6: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Report von Cppcheck enthält keinen Fehler. Im Issue-Tracker ist ebenfalls keiner vorhanden.

6.4.7 Testfall 7

Im siebten Testfall wird die Situation nachgestellt, dass ein Issue von einem Entwickler manipuliert wird. Dafür muss zuallererst wieder ein Issue erzeugt werden. Deshalb wird der Programmcode wie in Listing 6.9 geändert und erneut hochgeladen. Es wird wieder ein Issue vom IssueMaker erzeugt.

Im Kapitel Implementierung auf Seite 40 wurde entschieden, dass es sicherer ist, einen Hash zur Identifikation der Issues zu wählen. Dadurch können Anwender des Issue-Trackers den Inhalt der Issues manipulieren, indem sie zum Beispiel weitere Informationen, die zur Lösung beitragen können, anhängen ohne die Identifikation von Duplikaten und irrelevanten Issues zu erschweren.

Um zu zeigen, dass die Identifikation eines Issues über den Hashwert funktioniert, wird dem Issue in Abbildung 6.13 die Beschreibung bis auf den Hash entfernt. Außerdem wird zur Verdeutlichung der Titel des Issues geändert. Die geänderte Version des Issues ist in der Abbildung 6.14 zu sehen.

Wird anschließend die Pipeline erneut gestartet, erzeugt der IssueMaker die Konsolenausgabe in Abbildung 6.15. Der Fehler wurde erneut von Cppcheck im Report erwähnt und somit als Issue im IssueMaker erstellt. Anschließend lädt der IssueMaker den abgewandelten Issue aus dem Issue-Tracker und vergleicht die Hashwerte. Er erkennt, dass es sich um ein Duplikat handelt und erzeugt keinen neuen Issue.

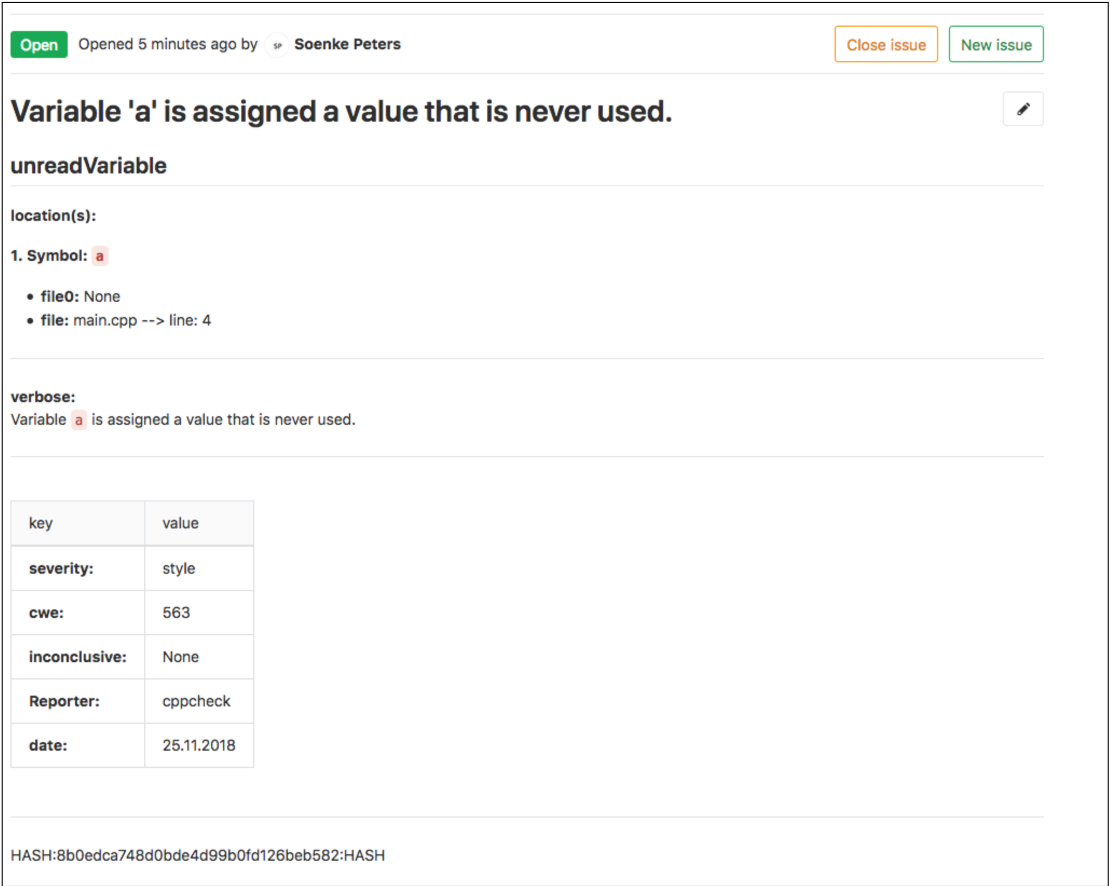

Abbildung 6.13: Testfall 7: Ein vom IssueMaker erstellter Issue in Gitlab.

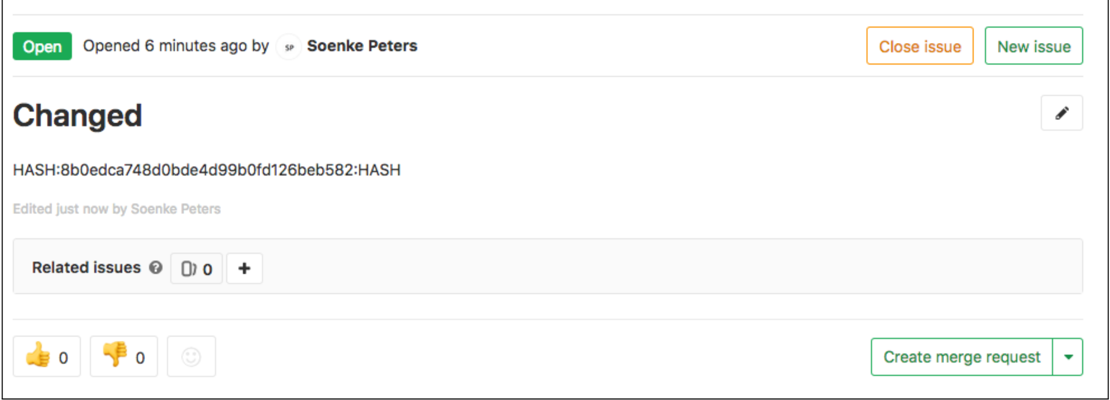

Abbildung 6.14: Testfall 7: Der Issue aus Abbildung 6.13 ohne Beschreibung und geänderten Titel.

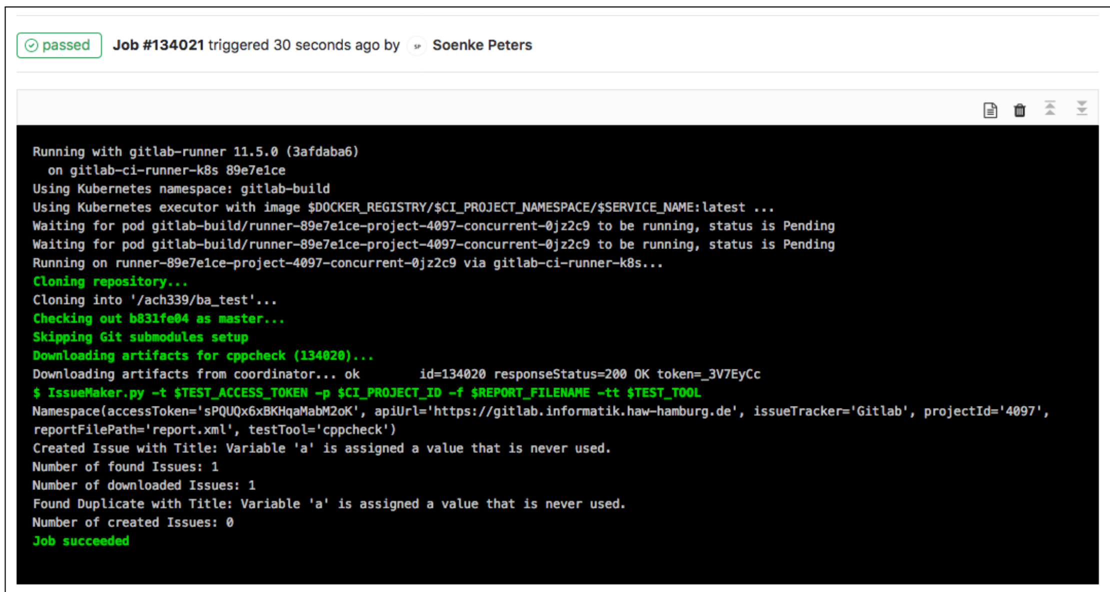

Abbildung 6.15: Testfall 7: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der gefundene Issue hat den gleichen Hash wie der geänderte Issue aus Abbildung 6.14 und wird als Duplikat verworfen.

6.4.8 Testfall 8

Komplementär zum siebten Test wird im achten Testfall nun nicht der Inhalt des Issues, sondern der Hash entfernt. Dafür muss zuallererst der Status Quo erzeugt werden. Das bedeutet, im Issue-Tracker befindet sich ein Issue, welcher vom IssueMaker generiert wurde, siehe Abbildung 6.17.

Anschließend wird der Hash des Issues, in Abbildung 6.18 zu sehen, entfernt und die Pipeline erneut gestartet. Das Ergebnis ist in der Abbildung 6.16 zu lesen.

Der Report von Cppcheck enthält einen Fehler und deshalb wird ein Issue gefunden. Außerdem wird aus dem Issue-Tracker ein Issue heruntergeladen.

Der Vergleich der Hashwerte schlägt fehl und der Issue im Issue-Tracker wird als irrelevant geschlossen. Anschließend wird ein neuer Issue mit denselben Informationen aber mit Hash erzeugt. Da der IssueMaker die Issues nicht löscht, sondern nur schließt gehen hier keine Informationen verloren. Aber GitLabs Schwachstelle ist zu erkennen. Es ist nicht möglich den Nutzer davor zu schützen Issues so zu manipulieren, dass sie vom IssueMaker falsch interpretiert werden, weil keine Felder im GitLab Issue dafür zur Verfügung stehen und keine eigenen Felder definiert werden können.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-04b7hr to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-04b7hr to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-04b7hr via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out b831fe04 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (134026)...
Downloading artifacts from coordinator... ok        id=134026 responseStatus=200 OK token=sobKNSsr
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Created Issue with Title: Variable 'a' is assigned a value that is never used.
Number of found Issues: 1
Number of downloaded Issues: 1
Closed Issue with title: Variable 'a' is assigned a value that is never used.
Number of created Issues: 1
Created Issue: Variable 'a' is assigned a value that is never used.
Job succeeded
```

Abbildung 6.16: Testfall 8: Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Issue aus Abbildung 6.18 wurde als irrelevant erkannt, weil er keinen passenden Hash mehr besitzt und wurde geschlossen.

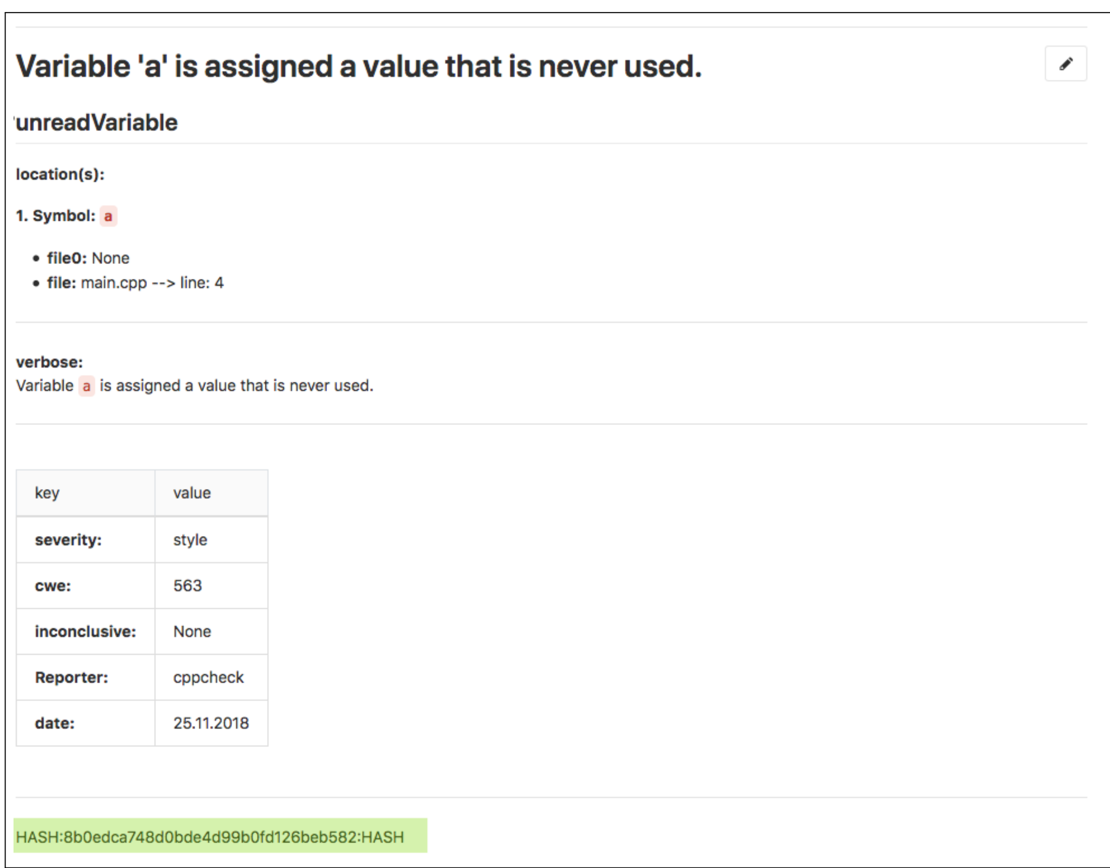

Abbildung 6.17: Testfall 8: Ein durch den IssueMaker erstellter Issue. Am Ende der Beschreibung ist der Hash markiert.

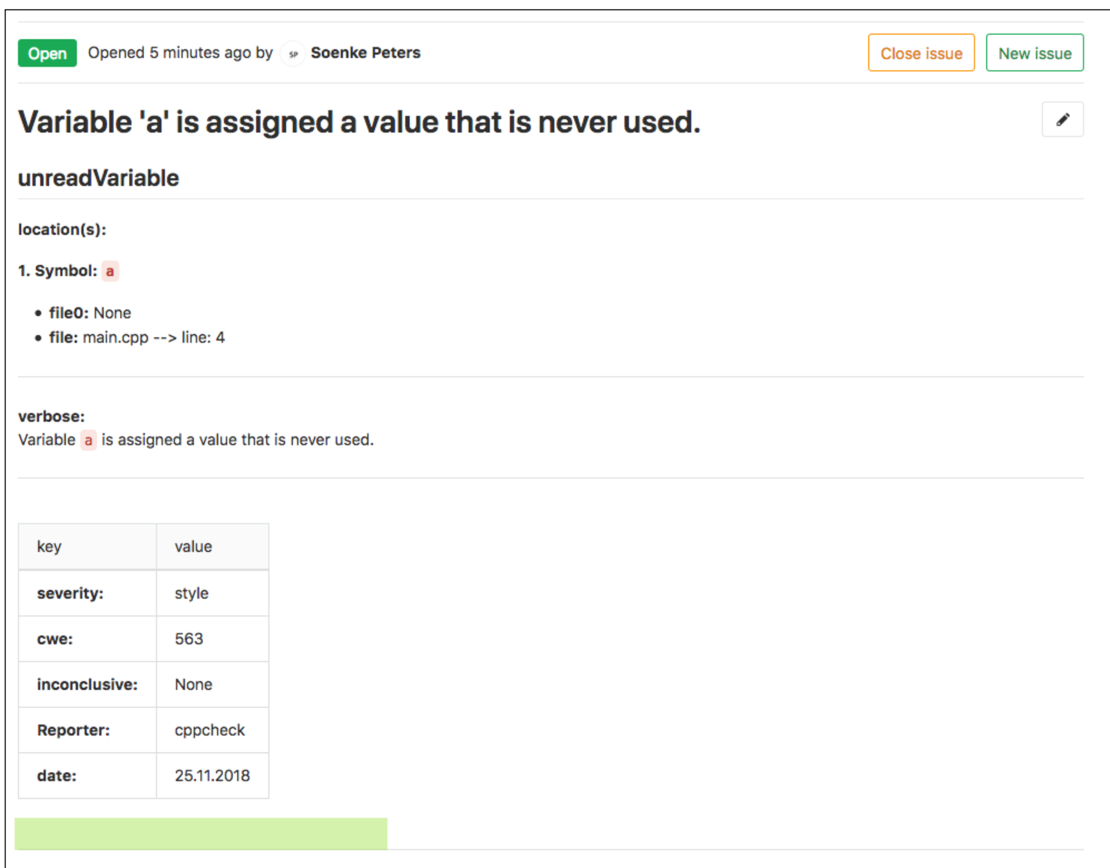

Abbildung 6.18: Testfall 8: Der Hash des Issues in Abbildung 6.17 wurde entfernt.

6.4.9 Testfall 9

Damit der IssueMaker nur Issues betrachtet, die er selbst erstellt hat und im entsprechenden Durchlauf sich auch nur an dem Programm, mit dem der Test ausgeführt wurde, orientiert, nutzt der IssueMaker die Möglichkeit Issues mit verschiedenen Labeln zu versehen.

Es wird immer ein Label für den IssueMaker und eines für das Programm mit dem der Testreport erstellt wurde genutzt. Es soll im neunten Testfall gezeigt werden, dass die Label eine valide Möglichkeit zum Identifizieren von Issues sind, aber gleichzeitig auch problematisch sein können. Dafür wird ein Label eines Issues, der vom IssueMaker erstellt wurde, entfernt, siehe Abbildung 6.19. Die Pipeline wird erneut gestartet und es wird ein Duplikat erzeugt. Das Ergebnis ist in Abbildung 6.20 zu sehen.

In der Konsolenausgabe in der Abbildung 6.21 findet sich der Grund. Es wurde kein Issue aus dem Issue-Tracker geladen. Zumindest für GitLab gilt, dass die Label von den Nutzern beliebig geändert werden können. Gleichzeitig sind sie die beste Möglichkeit den Aufwand zu reduzieren, weil die API es ermöglicht die Issues, die heruntergeladen werden, nach Labeln zu filtern.

Dadurch müssen nicht immer alle Issues eines Issue-Trackers durchsucht werden, sondern nur die Untermenge, die vom IssueMaker erstellt wurde. Trotzdem bleibt die Möglichkeit, dass durch Änderung der Label Duplikate entstehen.

Abbildung 6.19: Testfall 9: Das Label „IssueMaker“ wurde dem Issue entfernt.

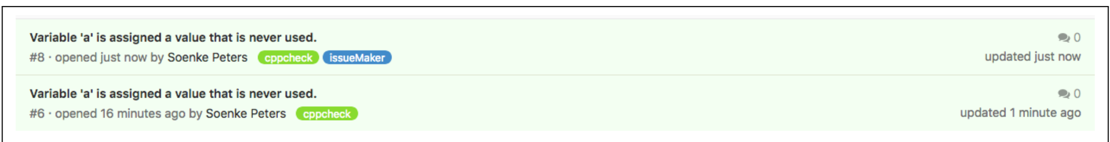

Abbildung 6.20: Testfall 9. Es existieren zwei Issues mit dem gleichen Inhalt aber verschiedenen Labeln.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kknx5 to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0kknx5 via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out b831fe04 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (134267)...
Downloading artifacts from coordinator... ok        id=134267 responseStatus=200 OK token=CSK3yCPD
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Created Issue with Title: Variable 'a' is assigned a value that is never used.
Number of found Issues: 1
Number of downloaded Issues: 0
Number of created Issues: 1
Created Issue: Variable 'a' is assigned a value that is never used.
Job succeeded
```

Abbildung 6.21: Testfall 9. Ausgabe des Terminals nach der Ausführung des IssueMakers. Der Issue mit dem fehlenden aus Abbildung 6.19 wurde nicht heruntergeladen.

6.4.10 Testfall 10

Bisher wurden nur Issues betrachtet, die vom IssueMaker erstellt wurden. Der Testfall 10 beleuchtet das Zusammenspiel des IssueMakers mit einem Issue-Tracker, der bereits Issues aus anderen Quellen enthält. Dafür wird ein Issue mit beliebigem Inhalt erstellt. Der Issue ist in Abbildung 6.22 zu sehen. Wenn die Pipeline erneut gestartet wird, wird erwartet, dass keines der, nicht vom IssueMaker erstellten, Issues vom IssueMaker betrachtet oder geändert wird.

Die Ausgabe der Konsole in Abbildung 6.23 zeigt, dass nur ein Issue aus dem Issue-Tracker geladen wurde, obwohl zwei im Issue-Tracker vorhanden sind. Da einem die Label fehlen, die ihn für den IssueMaker interessant machen, wird dieser vom IssueMaker ignoriert.

Abbildung 6.22: Testfall 10: Es wurde ein zusätzlicher Testissue im Issue-Tracker erstellt.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kf5qq to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0kf5qq to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0kf5qq via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out b831fe04 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (134259)...
Downloading artifacts from coordinator... ok        id=134259 responseStatus=200 OK token=5_61Us4Q
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Created Issue with Title: Variable 'a' is assigned a value that is never used.
Number of found Issues: 1
Number of downloaded Issues: 1
Found Duplicate with Title: Variable 'a' is assigned a value that is never used.
Number of created Issues: 0
Job succeeded
```

Abbildung 6.23: Testfall10: Ausgabe des Terminals nach der Ausführung des IssueMakers. Es wurde nur ein Issue heruntergeladen.

6.4.11 Testfall 11

Nun wird geprüft, was mit Issues passiert, die vom Nutzer erstellt wurden und die Label des IssueMakers und Cppcheck besitzen. Der Issue mit dem manipulierten Label ist in Abbildung 6.24 dargestellt. Es ist egal, ob der Nutzer diesem Issue einen Hash gibt, solange dieser nicht bereits in einem anderen Issue vorhanden ist. In diesem Fall würde er als Duplikat entfernt werden.

Es kann der Issue aus Testfall 10 genommen werden und um die Label `Cppcheck` und `IssueMaker` erweitert werden. Der Arbeitsablauf des IssueMakers ist in der Abbildung 6.25 zu sehen.

Der IssueMaker lädt den Issue herunter und identifiziert ihn indirekt als nicht relevant, weil er keinen bekannten Hashwert besitzt und schließt den Issue im Anschluss. Die Nutzer eines Issue-Trackers, der auch durch den IssueMaker manipuliert werden kann, müssen wissen, dass die Label eine Schwachstelle sind. Es sollte kein Anwender Issues erstellen und mit Labeln versehen, die vom IssueMaker verwendet werden. Da diese vom IssueMaker heruntergeladen werden würden und mit hoher Wahrscheinlichkeit gefiltert werden würden.

Abbildung 6.24: Testfall 11. Es wurde ein Issue mit den Labeln des IssueMakers erstellt.

Diese Testfälle orientieren sich an den Anforderungen und sollen diese abdecken. Es sind also keine exotischen Anwendungsfälle abgedeckt.

```
Running with gitlab-runner 11.5.0 (3afdaba6)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0sk4nv to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0sk4nv to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0sk4nv to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0sk4nv to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0sk4nv via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out b831fe04 as master...
Skipping Git submodules setup
Downloading artifacts for cppcheck (134261)...
Downloading artifacts from coordinator... ok        id=134261 responseStatus=200 OK token=ghyZPnLk
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.xml', testTool='cppcheck')
Created Issue with Title: Variable 'a' is assigned a value that is never used.
Number of found Issues: 1
Number of downloaded Issues: 2
Closed Issue with title: Test
Found Duplicate with Title: Variable 'a' is assigned a value that is never used.
Number of created Issues: 0
Job succeeded
```

Abbildung 6.25: Testfall 11. Ausgabe des Terminals nach der Ausführung des IssueMakers. Beide Issues wurden aus dem IssueMaker geladen und der Testissue wurde als irrelevanter Issue geschlossen.

6.5 Ausführung mit Regressionstest

Für den Testdurchlauf auf Basis des Regressionstests müssen die in den Tabellen6.1 und 6.2 nicht erneut herangezogen werden. Diese beziehen sich auf die Funktionsweise des IssueMakers im Zusammenspiel mit Cppcheck und sollen die gesamte Funktionsweise abbilden. Der Unterschied zwischen dem Regressionstests und der statischen Codeanalyse mit Cppcheck ist hauptsächlich beim Parser zu finden, da auch hier als Issue-Tracker weiterhin GitLab genutzt wird. Für einen alternativen Issue-Tracker sei auf den Ausblick 7.3 auf Seite 96 verwiesen. Es muss ein eigener Parser für die Ausgabe des Regressionstests geschrieben werden. Dafür muss das `Reportparser`-Interface eingebunden werden.

```
Running with gitlab-runner 11.3.1 (0aa5179e)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image docker-hub.informatik.haw-hamburg.de/cies/tessa-esep:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-2807-concurrent-0sc8wm to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-2807-concurrent-0sc8wm to be running, status is Pending
Running on runner-89e7e1ce-project-2807-concurrent-0sc8wm via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/CIES/TeSSA-ESEP-Regression'...
Checking out 9cc29d0f as master...
Skipping Git submodules setup
$ cd $CI_PROJECT_DIR
$ testgateway_client.py -b $BINARY -tp $TESTPLAN
info     | bin: binaries/regression_test testplan: testplans/testplan_regression.json
info     | connecting to testgateway
info     | connection established
info     | transfering testplan:
info     | ['regression_test']
info     | tranfering binary file to host
info     | transfer successfull
Host     | Initializing target...
Host     | Target Aquired target-1[inuse:True, Qnx:192.168.101.101, Simulation:192.168.101.33]!
Host     | Executing Test - regression_test
Host     | Testexecution finished! Verdict: False
Job succeeded
```

Abbildung 6.26: Beispiel einer Ausgabe des Regressionstest der HAW für das Software-Engineering Praktikum.

Bei dem Regressionstest des Projekts der HAW besteht die Ausgabe des Regressionstests aus weniger Komponenten als die des statischen Codeanalyse-Reports von Cppcheck. Es gibt einen Testtitel (z.B. `regression_test` und ein Ergebnis, welches in der Ausgabe durch `Verdict` angekündigt wird. Die gesamte Ausgabe des Regressionstests ist in der Abbildung 6.26 zu sehen. Im Gegensatz zu Cppcheck kann also in dem Report auch ein positives Ergebnis erscheinen und der Parser muss das beachten, da ansonsten Issues erzeugt werden würden, in denen steht, dass der Regressionstest erfolgreich gewesen ist.

Für eine Pipeline, die Issues auf Basis des Regressionstests generiert, muss die `.gitlab-ci.yml` angepasst werden. Der Teil, der bisher für den Aufruf von Cppcheck

genutzt wurde, muss mit dem Aufruf für den Regressionstest ausgetauscht werden. Die fertige `.gitlab-ci.yml` ist in Listing 6.10 zu sehen.

Neben den Unterschieden, die speziell für den Regressionstest sind und nicht Teil dieser Arbeit, ist auf den Aufruf des Skripts in der Pipeline-Station `verify` zu achten. Da bei der Implementierung des Regressionstests nicht vorgesehen gewesen ist, dass die Ausgabe in eine Datei geschrieben wird, aber der IssueMaker dies benötigt, wird mit dem Befehl cmtee $REPORT_FILENAME die Ausgabe in eine Datei gelenkt. So kann der IssueMaker dann genau wie zuvor bei Cppcheck aufgerufen werden.

Wenn ein Parser geschrieben wurde und eine neue Version des IssueMakers, die den neuen Parser beinhaltet, gebaut ist, können die Testfälle bearbeitet werden. Eine Untermenge der Testfälle von Cppcheck, mit auf den Regressionstest angepassten Bedingungen, ist in der Tabelle 6.3 zu sehen.

Es müssen nicht alle Tests erneut ausgeführt werden, weil hier nur die Zusammenarbeit mit einem weiteren Parser getestet werden soll. Es geht also darum, dass der IssueMaker den richtigen Parser wählt, und dem Testtool entsprechende Issues erzeugt.

```
1  variables:
2    DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"
3    REPORT_FILENAME: "report.txt"
4    TEST_TOOL: "esep_test"
5    SERVICE_NAME: bachelorthesis
6    CI_PROJECT_NAMESPACE: ach339
7    BINARY: "binaries/regression_test"
8    TESTPLAN: "testplans/testplan_regression.json"
9
10 stages:
11   - verify
12   - generate
13
14 verify:
15     stage: verify
16     image: docker-hub.informatik.haw-hamburg.de/cies/tessa-esep:latest
17     script:
18         - cd $CI_PROJECT_DIR
19         - testgateway_client.py -b $BINARY -tp $TESTPLAN | tee $
             REPORT_FILENAME
20   artifacts:
```

```
21      paths:
22        - $REPORT_FILENAME
23
24  issueMaker:
25    stage: generate
26    allow_failure: true
27    image: $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
28    only:
29     - develop
30    script:
31      - IssueMaker.py -t $ACCESS_TOKEN -p $CI_PROJECT_ID -f $
          REPORT_FILENAME -tt $TEST_TOOL #execute IssueMaker.py
32    dependencies:
33      - verify
```

Listing 6.10: Konfigurationsdatei .gitlab-ci.yml für den Regressionstest

<table>
<tr><th>Nr.</th><th colspan="2">Beschreibung</th></tr>
<tr><td rowspan="3">1R</td><td>Vorbedingungen</td><td>Im Issue-Tracker existiert kein Issue,
der vom IssueMaker basierend
auf dem Regressionstest, erstellt
wuden. Der Regressionstest muss fehlschlagen.</td></tr>
<tr><td>Ausführung</td><td>Die Pipeline wird gestartet.</td></tr>
<tr><td>Ergebnis</td><td>Es wird ein Issue im Issue-Tracker erzeugt.</td></tr>
<tr><td rowspan="3">2R</td><td>Vorbedingungen</td><td>Im Issue-Tracker existiert ein Issue,
der vom IssueMaker basierend
auf dem Regressionstest, erstellt
wude. Der Regressionstest muss mit dem Fehler
aus 1 fehlschlagen.</td></tr>
<tr><td>Ausführung</td><td>Die Pipeline wird gestartet.</td></tr>
<tr><td>Ergebnis</td><td>Es wird kein neuer Issue erzeugt.</td></tr>
<tr><td rowspan="3">3R</td><td>Vorbedingungen</td><td>Im Issue-Tracker existiert ein Issue,
der vom IssueMaker basierend
auf dem Regressionstest, erstellt
wuden. Der Regressionstest muss mit einen anderen
Fehler als in 1 fehlschlagen.</td></tr>
<tr><td>Ausführung</td><td>Die Pipeline wird gestartet.</td></tr>
<tr><td>Ergebnis</td><td>Es wird ein neuer Issue erzeugt.</td></tr>
<tr><td rowspan="3">4R</td><td>Vorbedingungen</td><td>Im Issue-Tracker existiert ein Issue,
der vom IssueMaker basierend
auf dem Regressionstest, erstellt
wuden. Der Regressionstest muss erfolgreich enden.</td></tr>
<tr><td>Ausführung</td><td>Die Pipeline wird gestartet.</td></tr>
<tr><td>Ergebnis</td><td>Es wird kein neuer Issue erzeugt.
Alle alten Issues basierend auf dem Regressionstest
werden geschlossen</td></tr>
</table>

Tabelle 6.3: Testfälle des Regressionstests

6.5.1 Testfall 1R

Der erste Testfall beschreibt die Situation, dass der Regressionstest einen Fehler findet und der Issue-Tracker zu der Zeit keine Issues des Regressionstests beinhaltet. Es wird hier gezeigt, dass der Parser funktioniert, die richtigen Label erzeugt werden und ein Issue in der definierten Form im Issue-Tracker generiert wird. In der Abbildung 6.28 ist der erstellte Issue mit den Labeln `esep_test` zu sehen und in der Abbildung 6.27 die Konsolenausgabe des IssueMakers zum passenden Report.

```
Running with gitlab-runner 11.5.1 (7f00c780)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-059q2p to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-059q2p to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-059q2p via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out a89b23bb as master...
Skipping Git submodules setup
Downloading artifacts for regressionTest (139672)...
Downloading artifacts from coordinator... ok        id=139672 responseStatus=200 OK token=9C5A-vrN
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.txt', testTool='esep_test')
Number of found Issues: 1
Number of downloaded Issues: 0
Number of created Issues: 1
Created Issue: regression_test
Job succeeded
```

Abbildung 6.27: Testfall 1R: Konsolenausgabe des IssueMakers nach der Erstellung eines Issues basierend auf dem Report des Regressionstests

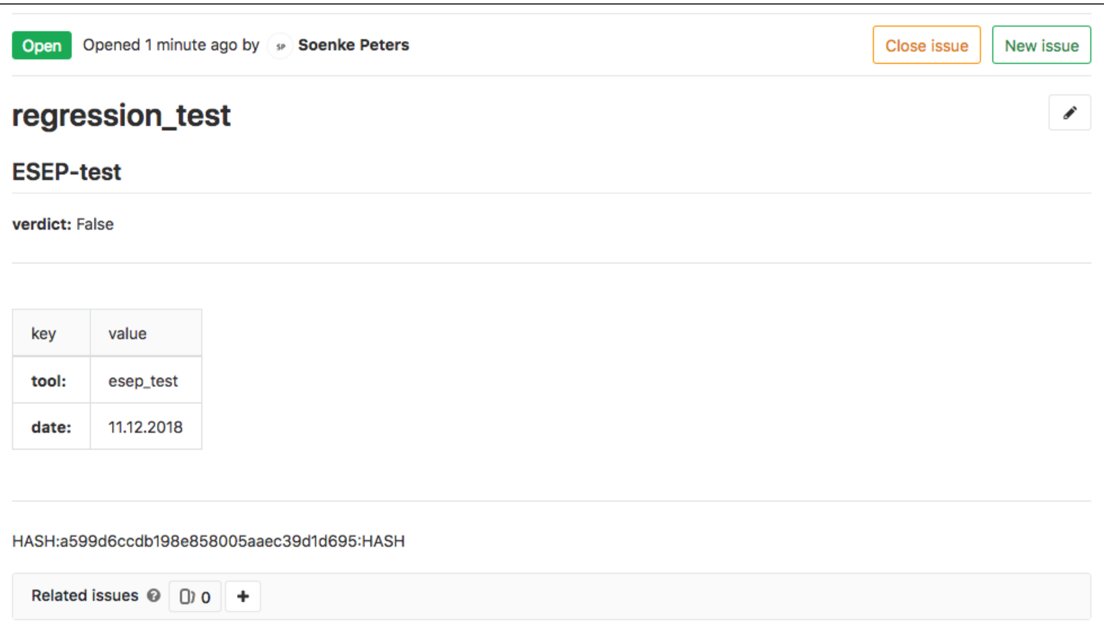

Abbildung 6.28: Testfall 1R: Darstellung eines mithilfe des IssueMakers erstellten Issues, der eine Meldung des Regressionstests enthält.

6.5.2 Testfall 2R

Da auch hier Duplikate vermieden werden sollen, muss geprüft werden, ob der Parser den Hashwert erkennt und auswertet. Dafür muss der Regressionstest mit demselben Programmcode erneut ausgeführt werden und denselben Fehler finden. Der IssueMaker erkennt, dass der Issue aus dem ersten Testfall noch im Issue-Tracker existiert und der Hashwert der gleiche ist. Es wird also kein neuer Issue generiert. Die passende Konsolenausgabe ist in der Abbildung 6.29 zu sehen.

```
Running with gitlab-runner 11.5.1 (7f00c780)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0bwwm8 to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0bwwm8 to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0bwwm8 via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out a89b23bb as master...
Skipping Git submodules setup
Downloading artifacts for regressionTest (139684)...
Downloading artifacts from coordinator... ok        id=139684 responseStatus=200 OK token=vkA_C_7G
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.txt', testTool='esep_test')
Number of found Issues: 1
Number of downloaded Issues: 1
Found Duplicate with Title: regression_test
Number of created Issues: 0
Job succeeded
```

Abbildung 6.29: Testfall 2R: Konsolenausgabe des IssueMakers nach der Erkennung eines Duplikats basierend auf dem Report des Regressionstests.

6.5.3 Testfall 3R

Der dritte Testfall soll die Duplikatenerkennung aus der anderen Richtung verifizieren. Dafür muss der Regressionstest basierend auf einem anderen Problem erfolglos enden. Der IssueMaker erkennt, durch den Unterschied der Hashwerte, dass es sich um ein anderes Problem handelt als in Testfall 1R und generiert einen neuen Issue. Die entsprechende Ausgabe des IssueMakers ist in der Abbildung 6.30. Es ist außerdem eine Besonderheit aufgetreten. Der Issue des zuvor gefunden Fehlers wurde entfernt. Das liegt daran, dass dieser Regressionstest fehlschlägt, sobald er einen Fehler gefunden hat. Danach wird er nicht weiter ausgeführt. Es kann also nicht passieren, dass im Report des Tests mehr als ein Fehler vorhanden ist. Wenn ein neuer Fehler gefunden wird wird der IssueMaker in diesem Fall immer alle alten Issues als irrelevant schließen.

```
Running with gitlab-runner 11.5.1 (7f00c780)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0xb86m to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-0xb86m to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-0xb86m via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out 7ac7051d as master...
Skipping Git submodules setup
Downloading artifacts for regressionTest (139686)...
Downloading artifacts from coordinator... ok        id=139686 responseStatus=200 OK token=PzHNSzm8
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.txt', testTool='esep_test')
Number of found Issues: 1
Number of downloaded Issues: 1
Closed Issue with title: regression_test
Number of created Issues: 1
Created Issue: regression_test
Job succeeded
```

Abbildung 6.30: Testfall 3R: Konsolenausgabe nach dem Test der Identifikation eines irrelevant gewordenen Issues.

6.5.4 Testfall 4R

Der letzte Testfall ist spezieller als die vorhergegangenen. Wie bereits in der Einführung zu diesen Testfällen geschrieben wurde, kann der Regressionstest auch erfolgreich enden. In diesem Fall wird aber auch ein Report erstellt. Das liegt daran, dass die gesamte Ausgabe des Regressionstests in eine Datei für den IssueMaker umgeleitet wird. Es wird also der Regressionstest ausgeführt, der erfolgreich endet. Durch die Prüfung der Konsolenausgabe in Abbildung 6.31 ist zu sehen, dass der Parser keinen Grund zur Generierung eines Issues in dem Report des Regressionstests gefunden hat. Mit den Test-

```
Running with gitlab-runner 11.5.1 (7f00c780)
  on gitlab-ci-runner-k8s 89e7e1ce
Using Kubernetes namespace: gitlab-build
Using Kubernetes executor with image $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest ...
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-067m4j to be running, status is Pending
Waiting for pod gitlab-build/runner-89e7e1ce-project-4097-concurrent-067m4j to be running, status is Pending
Running on runner-89e7e1ce-project-4097-concurrent-067m4j via gitlab-ci-runner-k8s...
Cloning repository...
Cloning into '/ach339/ba_test'...
Checking out d5ed3c0b as master...
Skipping Git submodules setup
Downloading artifacts for regressionTest (139688)...
Downloading artifacts from coordinator... ok        id=139688 responseStatus=200 OK token=niYc_DDM
$ IssueMaker.py -t $TEST_ACCESS_TOKEN -p $CI_PROJECT_ID -f $REPORT_FILENAME -tt $TEST_TOOL
Namespace(accessToken='sPQUQx6xBKHqaMabM2oK', apiUrl='https://gitlab.informatik.haw-hamburg.de', issueTracker='Gitlab', projectId='4097',
reportFilePath='report.txt', testTool='esep_test')
Number of found Issues: 0
Number of downloaded Issues: 1
Closed Issue with title: regression_test
Number of created Issues: 0
Job succeeded
```

Abbildung 6.31: Testfall 4R: Konsolenausgabe des IssueMakers nach einem erfolgreichen Regressionstest.

fällen zum Regressionstest sei gezeigt, dass der IssueMaker flexibel genug ist, auch mit unterschiedlichen Testreport-Quellen umzugehen. Solange für den Testreport ein entsprechender Parser (vgl. Seite 36) geschrieben wurde, kann der IssueMaker die Erstellung von Issues in einem Issue-Tracker übernehmen.

6.6 Parallele Ausführung des IssueMakers mit verschiedenen Testreport-Quellen in einer Pipeline

Abschließend ist in Listing 6.11 eine `.gitlab-ci.yml` aufgeführt, die eine Pipeline erzeugt in der Parallel der Regressionstest und Cppcheck ausgeführt werden und anschließend der IssueMaker die erzeugten Testreports einliest und entsprechende Issues erstellt. Pipeline-Schritte können parallel ausgeführt werden, wenn es für die `Stages` in der `.gitlab-ci.yml` mehrere auszuführende Komponenten gibt, sodass es für diese Aufgabe zwei `Stages` mit jeweils zwei Komponente geben muss.

In der ersten `Stage verify` wird der Regressionstest und Cppcheck ausgeführt. Beide schreiben in eine eigene Testreport-Datei. Im zweiten Schritt wird für jeden Testreport eine eigene IssueMaker-Instanz konfiguriert.

Wenn die Pipeline nun startet werden erst die Prüfungen durchgeführt und sobald beide fertig sind, wird die nächste `Stage` mit den IssueMakern parallel ausgeführt.

Wichtig ist hier, dass die `Stages` jeweils das Schlüsselwertpaar `allow_failure: true` haben. Ansonsten kann ein Problem in einer der Komponenten die gesamte Pipeline fehlschlagen lassen.

Die auf der Basis der `.gitlab-ci.yml` in Listing 6.11 ist in der Abbildung 6.32 zu sehen.

Das Problem bei diesem Aufbau ist, dass der Regressionstest in der Regel wesentlich länger dauert als der Test von Cppcheck. Hier muss aber die IssueMaker Instanz, die den Testreport von Cppcheck behandelt, warten bis auch der Regressionstest fertig ist. Leider ist es mit GitLab noch nicht anders möglich, aber es gibt bereits ein Issue dafür im GitLab Projekt [16], welches für 2019 geplant ist.

Damit ist auch die Anforderung der parallelen Ausführung des IssueMakers gezeigt.

```
1 variables:
2   DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"
3   REPORT_FILENAME: "report.txt"
4   TEST_TOOL: "esep_test"
5   REPORT_FILENAME2: "report.xml"
6   TEST_TOOL2: "cppcheck"
7   SERVICE_NAME: bachelorthesis
8   CI_PROJECT_NAMESPACE: ach339
```

```
9     BINARY: "binaries/regression_test"
10    TESTPLAN: "testplans/testplan_regression.json"
11
12 stages:
13   - verify
14   - generate
15
16 regressionTest:
17     stage: verify
18     image: docker-hub.informatik.haw-hamburg.de/cies/tessa-esep:latest
19     script:
20         - cd $CI_PROJECT_DIR
21         - testgateway_client.py -b $BINARY -tp $TESTPLAN | tee $
              REPORT_FILENAME
22   artifacts:
23     paths:
24       - $REPORT_FILENAME
25
26 cppcheck:
27   stage: verify
28   image: soenke2/cppcheckforissuemaker
29   script:
30     - cppcheck --enable=all --std=posix --force $CODE_PATH --xml 2> $
          REPORT_FILENAME2
31   artifacts:
32     paths:
33       - $REPORT_FILENAME2
34
35 issueMaker1:
36   stage: generate
37   allow_failure: true
38   image: $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
39   only:
40    - develop
41   script:
42     - IssueMaker.py -t $ACCESS_TOKEN -p $CI_PROJECT_ID -f $
          REPORT_FILENAME -tt $TEST_TOOL #execute IssueMaker.py
43   dependencies:
44     - regressionTest
45
```

```
46 issueMaker2:
47   stage: generate
48   allow_failure: true
49   image: $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
50   only:
51    - develop
52   script:
53     - IssueMaker.py -t $ACCESS_TOKEN -p $CI_PROJECT_ID -f $
         REPORT_FILENAME2 -tt $TEST_TOOL2 #execute IssueMaker.py
54   dependencies:
55     - cppcheck
```

Listing 6.11: Konfigurationsdatei .gitlab-ci.yml für den Regressionstest

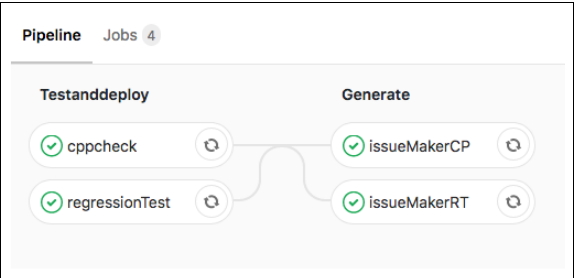

Abbildung 6.32: Parallele Ausführung von zwei Testtools und zwei IssueMaker Instanzen in einer GitLab Pipeline.

7 Fazit und Ausblick

Es wurde gezeigt, dass der IssueMaker den Anforderungen der Analyse auf Seite 24 weitestgehend genügt. Die Erweiterbarkeit durch weitere Filter, Parser und Issue-Tracker wird durch die Vorgabe eines Interfaces den Anwendern ermöglicht. Der Aufbau eines Issues nach IEEE-29119 konnte, wegen der Einschränkungen durch den Issue-Tracker in GitLab nicht vollständig umgesetzt werden und die Änderbarkeit der vom IssueMaker erstellten Hashwerte zur Identifikation der Issues ist eine zu beachtende Schwachstelle.

7.1 Ausbau der Filtermöglichkeiten des IssueMakers

Im Rahmen dieser Arbeit wurden zwei Filter implementiert. Einer ist dafür verantwortlich, dass keine Duplikate entstehen und der andere entfernt irrelevante Issues aus dem Issue-Tracker. Abhängig vom Testprozess und den Ansprüchen an ein Projekt sind weitere Filter denkbar. Der IssueMaker könnte über die Filter so konfiguriert werden, dass er Management-Aufgaben im Issue-Tracker übernimmt. Er kann alte Issues mit einem zusätzlichen Label versehen, sodass sie erneut von den Entwicklern geprüft werden und nicht vergessen werden, oder sie direkt schließen, weil sie abhängig von ihrem Alter wahrscheinlich nicht mehr bearbeitet werden.

Wenn man den IssueMaker auf ein Issue-Tracker anwendet, der benutzerdefinierte Felder zulässt, könnte ein Filter prüfen, ob in erstellten Issues Informationen in diesen Feldern hinterlegt wurden. Wenn die Felder wichtig sind, könnten in bestimmten Fällen die fehlenden Informationen nachgetragen werden oder der Ersteller könnte informiert werden. Diese Art von Manipulation des Issue-Trackers kann durch einen Filter, der nur auf den Remote-Report angewendet wird erreicht werden.

7.2 Erweiterung des Funktionsumfangs durch weitere Parser

Neben den Filtern wurden auch zwei Parser entwickelt. Einer für die Reports von Cppcheck und einer für die Ausgabe eines benutzerdefinierten Regressionstests. Denn unterschiedliche Testverfahren und Programme liefern unterschiedliche Reports und es sollte für jeden Testreport ein Parser geschrieben werden. Nur so kann der Mehrwert von gut formatierten Issues im Issue-Tracker erreicht werden. Wenn ein Team einen eigenen Parser für ein bekanntes Testtool entwickelt hat, sollte dieser über ein Pull-Request in das Repository des IssueMaker übernommen werden, sodass andere Teams davon profitieren können.

7.3 Alternativen zu GitLab als Issue-Tracker

GitLab ist eine beliebte Anwendung. Aber der Issue-Tracker hat auch Nachteile. Zum Beispiel ist die fehlende Möglichkeit eigene Felder an einem Issue zu definieren ein Hindernis. Denn um einzelne Informationen aus der Issue-Beschreibung zu gewinnen muss die Beschreibung geparsed werden. Das ist aufwendig und wenn Nutzer die Beschreibung geändert haben fehleranfällig. Außerdem ist es nicht möglich Issues eindeutig zu markieren, weshalb es zu der beschriebenen Kombination aus Hash und Label kommen muss.

In Konkurrenz zu GitLab stehen beispielsweise Jira [2], GitHub [10] und andere, welche auch eine API bieten, die es ermöglicht Issues zu erstellen. Sodass untersucht werden kann, inwiefern es Unterschiede bei automatischen Erstellung von Issues in den anderen Programmen gibt.

7.3.1 YouTrack als Beispiel für einen GitLab Konkurrenten

Im Kapitel Implementierung wurde bereits gezeigt, wie mehrere unterschiedliche Parser angebunden werden können. Dabei hat der Nutzer viel Freiraum.

Etwas schwieriger verhält es sich bei einem Issue-Tracker. Hier muss der Zugriff eingerichtet werden und die API entsprechende Funktionen zur Erzeugung von Issues bereitstellen.

In der Implementierung wurde sich an die GitLab-API gehalten. Diese wird nun dem Issue-Tracker YouTrack gegenübergestellt. YouTrack ist eine Web-basierte Issue-Tracker- und Projektmanagement-Plattform von JetBrains [35]. Es bietet ähnliche Möglichkeiten wie das Issue-Tracker Modul von GitLab und eignet sich deshalb für eine genauere Betrachtung und Vergleich.

Für die automatisierte Nutzung des IssueTrackers sollte er in eine Pipeline integriert werden. YouTrack besitzt aber kein eigenes Pipeline-Modul. Es muss sich also um eine Continuous-Integration-Pipeline gekümmert werden.

Um die GitLab-API nutzen zu können, wird ein Token benötigt. Dasselbe Prinzip wird auch bei YouTrack verwendet. Die Authentifizierung stellt deshalb kein Problem dar. Auch YouTrack liefert eine RESTful-API und eine Base-URL mit der die API erreicht werden kann. Außerdem werden die Issues einem Projekt zugeordnet. Der IssueMaker kann also ähnlich gestartet werden wie bei der GitLab-API, allerdings mit an die YouTrack-API angepassten Werten.

Im Kapitel Konzept wurden die Methoden, die ein Issue-Tracker implementieren muss aufgelistet und im Interface 4.4 dargestellt.

Damit der IssueMaker Duplikate und irrelevante Issues erkennen kann und da er keine eigene Datenbank besitzt, benötigt er die Möglichkeit Issues, die von ihm erstellt wurden, aus dem Issue-Tracker herunterzuladen. Dafür muss die Methode `downloadIssues(State, Labels):Report` implementiert werden. Dies kann fast identisch zu GitLab auch für YouTrack geschehen. Allerdings kann bei der Anfrage an YouTrack nicht `State` und `Label` übergeben werden. Stattdessen muss ein Query in der YouTrack eigenen „Search-Query-Grammar“ erstellt werden [36]. Dieser kann dann in der Anfrage übermittelt werden, sodass nur die Issues geladen werden, die die entsprechenden `States` und `Label` haben.

Nachdem der IssueMaker entschieden hat, welche Issues neu erstellt werden sollen. Muss er die API über die Methode `uploadIssues(Report):void` aufrufen. Die Issues haben ähnliche Felder wie die in GitLab. Allerdings heißt das Feld für den Titel hier `Summary`. YouTrack unterstützt auch Markdown, sodass ein Parser, der Issues mit Markdown erzeugt hat, auch in YouTrack genutzt werden kann.

Zusätzlich erlaubt YouTrack eigens definierte Felder. In einem solchen Feld kann zum Beispiel der Hash gespeichert werden, sodass die Identifikation von Issues erleichtert und

verbessert wird. Außerdem könnten die Informationen besser in eigene Felder geschrieben werden, als sie mit Markdown in ein angemessenes Format gebracht werden können.

Ebenso wie GItLab unterstützt YouTrack das Konzept der Label. Allerdings heißen sie hier Tags. Ansonsten können sie äquivalent genutzt und erstellt werden, sodass bei der Implementierung der Methode `createLabel(Labelname): void` lediglich beachtet werden muss, dass der Begriff ein anderer ist.

Die Methoden `deleteIssue(Issue):void`, `closeIssue(Issue):void` und `getIssue(iid):Issue` können ebenfalls implementiert werden, denn auch YouTrack nutzt einen eindeutigen Issue-Identifier, um einzelne Issues über die API zu manipulieren.

Es können also alle Methoden des Interfaces mit überschaubarem Aufwand implemeniert werden. Dieser kann weiter reduziert werden, indem die Python Bibliothek der YouTrack API genutzt wird [34]. Allerdings ist diese nicht dokumentiert.

Der IssueMaker beschränkt sich also nicht nur auf GitLab. Solange eine API existiert mit dessen Hilfe der Issue-Tracker manipuliert werden kann, könnte es nur noch bei der Authentifizierung des IssueMakers bei dem Issue-Tracker Probleme geben. Denn nicht jeder Issue-Tracker ermöglicht es einen Schlüssel zur Authentifizierung zu hinterlegen. Eine Kombination aus Benutzername und Passwort ist im IssueMaker für den Login über die API nicht vorgesehen. Außerdem kann es zu Namensvariationen kommen. Die „Labels“ in GitLab heißen beispielsweise bei YouTrack „Tags“ , obwohl sie dieselbe Funktion haben und der Titel eines Issues heißt bei GitLab „Titel“ , während YouTrack den Begriff „Summary“ nutzt. Diese Unterschiede zwischen den Issue-Trackern können die Implementierung mindestens erschweren.

7.4 Automatische Zuweisung der erstellten Issues

Neben den Erweiterungen durch neue Parser, Filter oder andere Issue-Tracker kann auch die Funktionalität der Anwendung erweitert werden.

Im Abschnitt zu den verwandten Arbeiten, wurde darauf eingegangen, wie mittels Natural-Language-Processing Duplikate in öffentlichen Bug Trackern gefunden werden können. Diese Projekte nutzen die hohe Anzahl an Bug Reports und Issues als Testdaten um ihr System zu trainieren. Aber nicht nur Duplikate können erkannt werden. Mit der hohen

Anzahl an Daten können auch Systeme trainiert werden, die den Fehlerreport und den Entwickler, der den Fehler behoben hat, analysieren und dann bei neuen Bug-Reports Vorschläge machen, wer den Fehler beheben sollte [39]. Ähnlich wie bei den Duplikaten ist das bei automatisch erstellten Reports leichter, weil das Natural-Language-Processing wegfällt. Indem geschlossene Issues des Issue-Tracker analysiert werden und geschaut wird, welcher Entwickler, welche Fehler behoben hat, könnte das Programm nicht nur automatisch Issues aus Reports erstellen, sondern sie direkt einem Entwickler zuweisen.

Eine andere Möglichkeit wäre es den Prozess dafür zu nutzen. Der IssueMaker startet, wenn er in einer GitLab-Pipeline ausgeführt wird, nach einem Testdurchlauf. Die Pipeline wird bei jedem Push in das Repository automatisch gestartet. Nun speichert GitLab bei jedem Push den Benutzernamen, der Person, die ihn ausgeführt hat. Wenn also Fehler gefunden werden, können die erzeugten Issues, direkt der Person zu gewiesen werden, die die Fehler produziert hat.

7.5 Automatische Aufwandsschätzung

Ein andere Frage, die sich auch mit Data-Mining beschäftigt lautet: „How long will it take to fix this bug?“ und wurde 2007 von Cathrin Weiss et al. gestellt [50].

In der Veröffentlichung wurden manuell erstellte Issues betrachtet und damit wurde ein System trainiert die voraussichtliche Zeit zum Bearbeiten zu schätzen. Der IssueMaker könnte auch um eine solche Funktion erweitert werden und wenn die API des Issue-Trackers es erlaubt, bei erstellten Issues direkt eine Aufwandsschätzung liefern. Die sonst manuell von den Entwicklern gemacht werden müsste. Dafür muss er allerdings alle bereits erstellten Issues des Issue-Trackers analysieren und es wäre ein schnellerer Zugriff auf die Datenbank nötig als eine Web-Schnittstelle.

7.6 Schlusswort

Bei der automatischen Erstellung von Issues aus Bug-Reports gibt es viele Herausforderungen. Der unterschiedliche Report-Aufbau der verschiedenen Testtools verlangt, dass für jede neue Art ein neuer Parser geschrieben werden muss. Auf der anderen Seite bieten Issue-Tracker wie GitLab nur einen Titel und eine Beschreibung um Informationen in Issues unterzubringen, die Fehler-spezifisch sind. Es ist optimiert auf manuell erstellte

Issues und Web-Interfaces und nicht auf alle Anforderungen, die Entwickler in den Umfragen [40] sich wünschen. Vor allem sind sie nicht angepasst an automatisch erstellte Reports und deren Struktur und somit ist es aufwendig Duplikate zu identifizieren, bei denen die bekannte Struktur ansonsten helfen würde.

Da die Arbeit aus dem Bedarf des automatischen Testens der Projekte der Studenten der HAW entstanden ist, sollte der Nutzen dieses Programms in diesem Rahmen auch betrachtet werden. Das Problem bei der Projektarbeit war, dass den Studenten oft Rückmeldung fehlte und sie nur mit den Anlagen vor Ort testen konnten. Durch die automatischen Tests, die bereits vor dieser Arbeit ermöglicht wurden, können die Studenten nun auch von Zuhause aus ihren Programmcode testen. Der IssueMaker erweitert die Funktionalität, so dass die Testergebnisse direkt in dem Issue-Tracker der Studenten als Issues erzeugt werden und sie dadurch sofort eine Rückmeldung bekommen, ob ihr Programmcode funktioniert und wenn nicht welche Module noch Arbeit benötigen.

Generell sehe ich den Einsatz des Programms in Szenarien, in denen Tests sehr aufwendig sind, wie zum Beispiel Regressionstest oder Performanz- und Last-Analysen. Das kann bedeuten, dass sie langwierig sind, extra Hardware benötigen oder beides. Die Reports, die dann erstellt werden, müssen aufwendig gesichtet und per Hand in die Issue-Tracker übertragen werden. Hier setzt der IssueMaker an und ermöglicht es, dass bereits nach den Tests automatisch die Ergebnisse als Issue im Issue-Tracker der Entwickler erstellt werden, ohne dass sie die Reports manuell inspizieren müssen. Am Ende ist der IssueMaker aber davon abhängig wie gut die Informationen des Reports sind um aussagekräftige Issues zu erstellen.

Literaturverzeichnis

[1] *Alpine Dokumentation.* https://wiki.alpinelinux.org/wiki/Alpine_Linux:Documentation. – Accessed: 2018-10-31

[2] *Atlassian Jira API Documentation.* https://developer.atlassian.com/cloud/jira/platform/rest/v3/. – Accessed: 2018-11-01

[3] *Atlasssian Conector Plugin Dokumentation.* https://confluence.atlassian.com/ideplugin/overview-of-the-atlassian-ide-connectors. – Accessed: 2018-10-26

[4] *Azure DevOps Dokumentation.* https://docs.microsoft.com/en-us/azure/devops/?view=vsts. – Accessed: 2018-10-26

[5] *Bugzilla Dokumentation.* https://www.bugzilla.org/docs/. – Accessed: 2018-10-26

[6] *Cppcheck Dokumentation.* http://cppcheck.sourceforge.net/manual.pdf. – Accessed: 2018-10-31

[7] *Docker Dokumentation.* https://docs.docker.com/. – Accessed: 2018-10-31

[8] *Docker-Hub Dokumentation.* https://docs.docker.com/docker-hub/. – Accessed: 2018-10-31

[9] *Findbugs Dokumentation.* http://findbugs.sourceforge.net/findbugs2.html. – Accessed: 2018-10-26

[10] *Github API Documentation.* https://developer.github.com/v3/. – Accessed: 2018-11-01

[11] *Gitlab API Dokumentation.* https://docs.gitlab.com/ee/api/. – Accessed: 2018-10-31

[12] *Gitlab Continious Integration Dokumentation.* https://docs.gitlab.com/ee/ci/yaml/. – Accessed: 2018-10-31

[13] *Gitlab Dokumentation. Thema: Issues aus EMails erstellen.* https://docs.gitlab.com/ee/user/project/issues/create_new_issue.html#new-issue-via-email. – Accessed: 2018-10-26

[14] *Gitlab Issue Dokumentation.* https://docs.gitlab.com/ee/user/project/issues/. – Accessed: 2018-10-28

[15] *Gitlab Markdown Dokumentation.* https://docs.gitlab.com/ee/user/markdown.html. – Accessed: 2018-10-31

[16] *GitLabs Issue-Tracker mit Issue des Feature Requests für echt parallele Pipelines.* https://gitlab.com/gitlab-org/gitlab-ce/issues/47063. – Accessed: 2018-12-06

[17] : *IEEE Standard for Software and System Test Documentation*

[18] *Jira Dokumentation. Thema: Issues aus EMails erstellen.* https://confluence.atlassian.com/adminjiraserver073/creating-issues-and-comments-from-email-861253784.html. – Accessed: 2018-10-26

[19] *Link zur Homepage von ReportPortal.* http://www.reportportal.io. – Accessed: 2018-12-8

[20] *Link zur Homepage von Testrail.* https://www.gurock.com/testrail. – Accessed: 2018-12-8

[21] *Link zur Webseite der JSON Dokumentation.* https://www.json.org/. – Accessed: 2018-12-8

[22] *Link zur Webseite der Open Source Lizenz GNU Gerneral Public Licanse v.3.0.* http://www.gnu.org/licenses/gpl-3.0.html. – Accessed: 2018-12-8

[23] *Link zur Webseite der W3C zum Bereich XML.* https://www.w3.org/XML/. – Accessed: 2018-12-8

[24] *Liste aller vordefinierten Variaben in GitLab.* https://docs.gitlab.com/ee/ci/variables/. – Accessed: 2018-10-31

[25] *Offizielle Homepage der YAML-Spezifikation.* https://yaml.org/. – Accessed: 2018-12-8

[26] *PMD Dokumentation.* https://pmd.github.io/pmd/pmd_devdocs_pmdtester.html. – Accessed: 2018-10-26

[27] *Python Dokumentation.* https://docs.python.org/3/. – Accessed: 2018-10-31

[28] *Python Gitlab API Dokumentation.* https://python-gitlab.readthedocs.io/en/stable/. – Accessed: 2018-10-31

[29] : *Software and systems engineering Software testing Part 3:Test documentation*

[30] *SonarQube Dokumentation.* https://docs.sonarqube.org/display/SONAR/Documentation. – Accessed: 2018-10-26

[31] *Webseite der Common Weakness Enumeration.* https://cwe.mitre.org/. – Accessed: 2018-11-01

[32] *Website Marker.io.* https://marker.io/. – Accessed: 2018-10-26

[33] *YBug Dokumentation.* https://ybug.io/docs/installation. – Accessed: 2018-10-26

[34] *YouTrack Python Bibliotek.* https://github.com/JetBrains/youtrack-rest-python-library. – Accessed: 2018-11-13

[35] *YoutTrack Homepage.* https://www.jetbrains.com/help/youtrack/incloud/YouTrack-InCloud.html. – Accessed: 2018-11-13

[36] *YoutTrack Search Query Grammar Dokumentation.* https://www.jetbrains.com/help/youtrack/incloud/Search-Query-Grammar.html. – Accessed: 2018-11-13

[37] Andreas Spillner, Tilo L.: *Basiswissen Softwaretest.* Dpunkt.Verlag GmbH, 2012. – URL https://www.ebook.de/de/product/19361935/andreas_spillner_tilo_linz_basiswissen_softwaretest.html. – ISBN 3864900247

[38] Anvik, John ; Hiew, Lyndon ; Murphy, Gail C.: Coping with an open bug repository. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse '05*, ACM Press, 2005, S. 35–39

[39] Anvik, John ; Hiew, Lyndon ; Murphy, Gail C.: Who should fix this bug? In: *Proceeding of the 28th international conference on Software engineering - ICSE '06*, ACM Press, 2006, S. 361–370

[40] Bettenburg, Nicolas ; Just, Sascha ; Schröter, Adrian ; Weiss, Cathrin ; Premraj, Rahul ; Zimmermann, Thomas: What makes a good bug report? In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, ACM Press, 2008, S. 308–318

[41] Bettenburg, Nicolas ; Premraj, Rahul ; Zimmermann, Thomas ; Kim, Sunghun: Duplicate bug reports considered harmful … really? In: *2008 IEEE International Conference on Software Maintenance*, IEEE, 2008, S. 337–345

[42] Fowler, Martin ; Foemmel, Matthew: Continuous integration. In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006), S. 14

[43] Luo, Lu: Software Testing Techniques / Carnegie Mellon University. 2001. – Forschungsbericht. H

[44] Malhotra, Ruchika ; Bahl, Laavanye: A defect tracking tool for open source software. In: *2017 2nd International Conference for Convergence in Technology (I2CT)*, IEEE, apr 2017, S. 901–905

[45] Rastkar, Sarah ; Murphy, Gail C. ; Murray, Gabriel: Summarizing software artifacts. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, ACM Press, 2010, S. 505–514

[46] Rogaway, Phillip ; Shrimpton, Thomas: Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In: *Fast Software Encryption*. Springer, Januar 2004. – URL http://dx.doi.org/10.1007/978-3-540-25937-4_24. – ISBN 978-3-540-22171-5

[47] Runeson, Per ; Alexandersson, Magnus ; Nyholm, Oskar: Detection of Duplicate Defect Reports Using Natural Language Processing. In: *29th International Conference on Software Engineering (ICSE'07)*, IEEE, may 2007, S. 1–10

[48] Shu, Hongying Gu ; Long Zhao ; C.: Analysis of duplicate issue reports for issue tracking system. In: *The 3rd International Conference on Data Mining and Intelligent Information Technology Applications*, 2011, S. 86–91

[49] WANG, Xiaoyin ; ZHANG, Lu ; XIE, Tao ; ANVIK, John ; SUN, Jiasu: An approach to detecting duplicate bug reports using natural language and execution information. In: *Proceedings of the 13th international conference on Software engineering - ICSE '08*, ACM Press, 2008, S. 461–470

[50] WEISS, Cathrin ; PREMRAJ, Rahul ; ZIMMERMANN, Thomas ; ZELLER, Andreas: How Long Will It Take to Fix This Bug? In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, IEEE, may 2007, S. 1–8

A Anhang

A.1 Anleitung

Die folgende Anleitung beschränkt sich auf GitLab.

A.1.1 Access Token

Für die in Inbetriebnahme des Issue-Makers im CI von GitLab muss als Erstes einer der Nutzer des Repositories einen sogenannten `Acces-Token` erstellen. Dafür muss auf der Nutzer sich auf GitLab anmelden und oben rechts auf sein Icon klicken. Es erscheint ein Dropdown-Menü. Dort muss
cmSettings ausgewählt werden A.1

Anschließend muss im Menü auf der linken Bildschirmseite `Access-Token` ausgewählt werden A.2.

In der darauf folgenden Ansicht muss das Feld Name ausgefüllt werden. Außerdem muss ein Datum gewählt werden, bis zu dem der Token gültig sein soll. Bei den Check-Boxen muss schließlich `api` angekreuzt werden A.3. Um den Token zu erzeugen muss auf den Button `Create personal access token` A.4 am Ende des Formulars geklickt werden.

GitLab erstellt einen Token und zeigt diesen nun oberhalb des Formulars an. Dieser ist unbedingt zu sichern, da er nicht erneut angezeigt werden kann A.5.

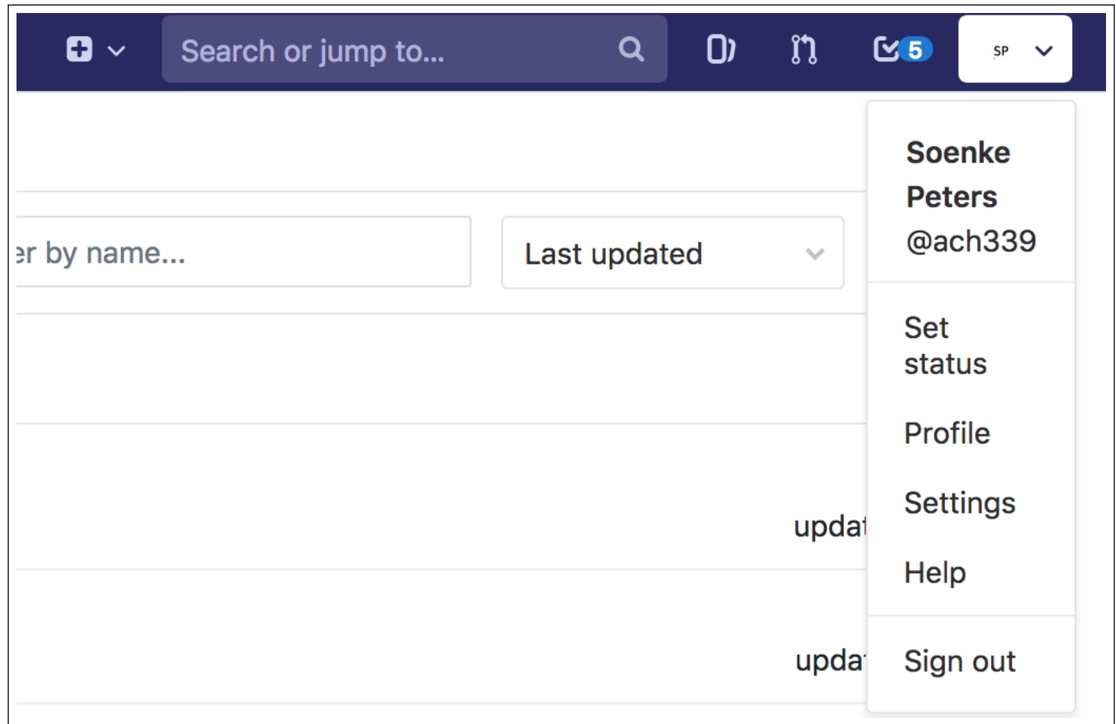

Abbildung A.1: Profil-Einstellungen.

A.1.2 Access Token als CI Variable anlegen

Als nächstes muss der erzeugte Token im GitLab-Projekt, welches ihn nutzen können soll, hinterlegt werden. Dafür öffnet man das Projekt und dort die Einstellungen A.6 und wählt `CI/CD` aus. Dort muss der Bereich `Variables` expandiert werden. Als `Input variable key` sollte hier der Name `ACCESS_TOKEN` und als `Input variable value` der erstellte Access Token hinterlegt werden.

A.1.3 Bereitstellung eines Runners für die Docker-Container

Eine Pipeline in GitLab benötigt ein sogenannten `Runner` auf dem die Container ausgeführt werden können. Dafür muss in der CI/CD-Einstellungen ein `Runner` erstellt werden. Wenn die bereitgestellte GitLab-Instanz einen `Shared Runner` besitzt kann dieser ausgewählt werden. Allerdings muss man sich dann den `Runner` mit anderen Nutzern teilen, was zu Wartezeiten führen kann. Alternativ kann ein `Specific Runner` erzeugt werden. Dafür muss der entsprechenden Anleitung auf GitLab gefolgt werden.

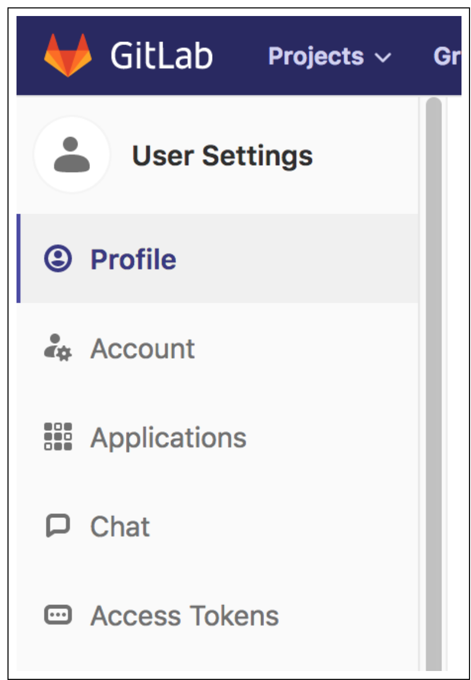

Abbildung A.2: Menü der Profil-Einstellungen.

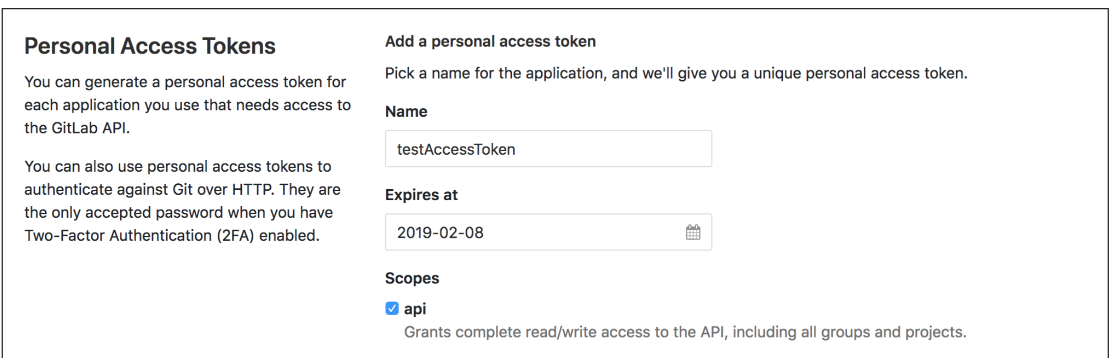

Abbildung A.3: Erstellen eines Access Tokens.

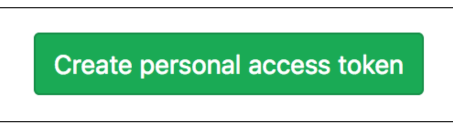

Abbildung A.4: Button zum Bestätigen der Eingaben.

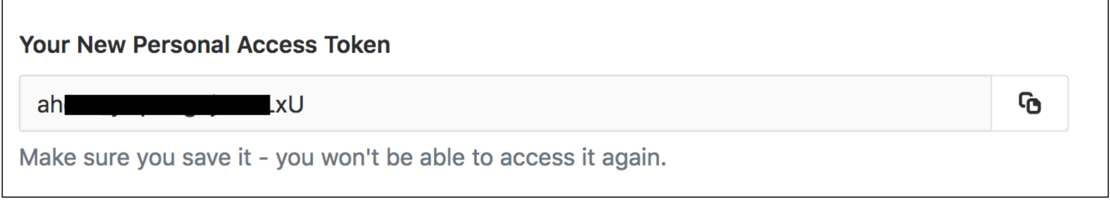

Abbildung A.5: Der erstellte Access Token erscheint über dem Formular.

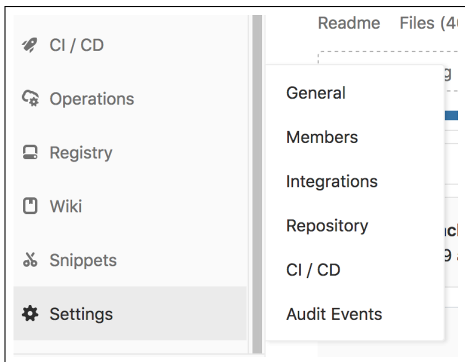

Abbildung A.6: Menü im Projekt.

A.1.4 Pipeline erzeugen

In GitLab kann eine Pipeline erzeugt werden, indem dem Repository im Root-Verzeichnis eine `.gitlab-ci.yml` hinzugefügt wird. Der IssueMaker benötigt zwei `Stages`. In der ersten wird ein Testreport erzeugt und in der zweiten wird der IssueMaker ausgeführt.

Eine Beispiel ist in dem Listing A.1 zu sehen. Dort wir in der `Stage test` mittels Cppcheck ein Testreport in der Datei `report.xml` erzeugt. Innerhalb der `Stage generate` wird anschließend der IssueMaker ausgeführt. Für nähere Informationen zur `.gitlab-ci.yml` wird auf die Dokumentation [12] verwiesen.

```
1  variables:
2    DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"
3    REPORT_FILENAME: "report.xml"
4    TEST_TOOL: "cppcheck"
5    SERVICE_NAME: bachelorthesis
6    CI_PROJECT_NAMESPACE: ach339
7    UUT: TestProject/
8  stages:
9   - test
10  - generate
11
12 cppcheck:
13  stage: test
14  image: soenke2/cppcheckforissuemaker
15  script:
16   - cppcheck --enable=all --std=posix --force $UUT --xml 2> $
        REPORT_FILENAME
17  artifacts:
18   paths:
19     - report.xml
20
21 issueMaker:
22  stage: generate
23  allow_failure: true
24  image: $DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
25  only:
26   - develop
27  script:
```

```
28      - Main.py $TEST_ACCESS_TOKEN $CI_PROJECT_ID $REPORT_FILENAME $
            TEST_TOOL
29    dependencies:
30      - cppcheck
```

Listing A.1: Beispiel einer .gitlab-ci.yml, die eine Pipeline erzeugt.

A.1.5 Pipeline starten

Wenn das Repository nun durch einen Push geändert wird, oder über den Menüpunkt `CI/CD` eine Pipeline manuell gestartet wird, wird das Skript der `.gitlab-ci.yml` ausgeführt.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg	25.02.2019	
Ort	Datum	Unterschrift im Original