Hochschule München
University of Applied Sciences
Fakultät für
Informatik und Mathematik

Bachelor - Thesis

Ein konfigurierbares und skalierbares Monitoringsystem für Delta-Metriken innerhalb des kontinuierlichen Integrationsservers Jenkins

Gutachter: Prof. Dr. Ullrich Hafner
Vorgelegt von: Simon Symhoven
Adresse: Boschetsriederstraße 59a
D-81379 München
E-Mail: simon.symhoven@hm.edu
Matr.-Nr.: 49651418

München, den 12. Juli 2021

Eidesstattliche Erklärung

Erklärung gemäß §16 Abs. 10 APO i.V.m. §35 Abs. 7 RaPO

Hiermit erkläre ich, Simon Symhoven, die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der von mir angegebenen Literatur verfasst zu haben.

Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Diese Arbeit hat in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

München, den 12. Juli 2021

Simon Symhoven

Gender Erklärung

In dieser Arbeit wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten sind dabei ausdrücklich mitgemeint, soweit es für die Aussage erforderlich ist.

Abstract

Das Ziel in der vorliegenden Arbeit ist es, ein neues *Dashboard* in Form eines *Plugins* für den *CI*-Server *Jenkins* zu entwickeln. Das *Plugin* soll ein für den Nutzer konfigurierbares und beliebig erweiterbares *Dashboard* zur Verfügung stellen, welches verschiedene Delta-Softwaremetriken von *Pull Requests* visualisiert. Anders als bisherige Darstellungen innerhalb des *Jenkins UI* sollen die Resultate eines *Pull Requests* auf eine gefilterte Darstellung der tatsächlich vorgenommenen Änderungen an der Codebasis reduziert werden.

Das *Pull Request Monitoring Plugin* wurde auf Basis der *JavaScript* Bibliothek *Muuri* entwickelt und bietet eine Schnittstelle für andere *Plugins*, ihre Metriken und Qualitätsmerkmale dem *Dashboard* in Form eines *Portlets* zur Verfügung zu stellen.

Es werden die technische Umsetzung des *Plugins* erläutert, Entscheidungen diskutiert und resultierende Ergebnisse vorgestellt. Dabei wird auf die zwei umgesetzten *Portlets* des *Warnings Next Generation* und *Code Coverage API Plugins* eingegangen und die für den Anwender relevante grafische Benutzeroberfläche des *Plugins* innerhalb des *Jenkins UI* präsentiert.

Abkürzungsverzeichnis

API *Application Programming Interface*

CI *Continuous Integration*

SCM *Source Code Management*

UI *User Interface*

URL *Uniform Resource Locator*

VCS *Version Control System*

Inhaltsverzeichnis

1 Einleitung

Die *Continuous Integration* (*CI*) hat in den letzten Jahren enorm an Bedeutung gewonnen. Viele Software Teams haben ihre Entwicklungsprozesse auf *Pull Requests* umgestellt. Änderungen an der Software werden auf *Feature Branches* unter Anwendung eines *Git-Workflows* entwickelt und in diesem mithilfe eines *Source Code Management* (*SCM*) Systems in dem zugehörigen *Repository* eingecheckt, der dann durch einen *Pull Request* in die bestehende Codebasis integriert wird [**?**]. Dieser wird anschließend manuell einem *Code Review* unterzogen und zusätzlich automatisiert im *CI*-Server, z. B. dem *Jenkins*, gebaut [**?**]. Dieser liefert eine Qualitätsaussage zu der jeweiligen Softwareversion.

Seit der Entwicklung des *Jenkins* im Jahre 2004 ist die Visualisierung der Ergebnisse des automatisierten *Builds* oder *Runs* eher im Kontext des Gesamtprojekts zu betrachten. Für Entwickler ist diese Darstellung unzureichend. Statt die Resultate als Übersicht für das Gesamtprojekt zu visualisieren, ist eine gefilterte Darstellung der Ergebnisse auf die tatsächlich vorgenommenen Änderungen durch den *Pull Request* an der Codebasis hilfreicher [**?**].

Ziel ist es, den *CI*-Server *Jenkins* um ein neues *Dashboard* zu erweitern, welches die Ergebnisse eines *Pull Requests* visualisiert. Wichtige Ergebnisse können sein: Testergebnisse (Fehler, hinzugefügte Tests), Veränderungen der *Code Coverage*, Warnungen einer statischen Code Analyse, *API Contract* Verstöße oder Veränderungen der Metriken zur Softwarequalität. Das *Dashboard* soll als neues *Plugin*, dem *Pull Request Monitoring Plugin* [**?**] für den *Jenkins* bereit gestellt werden und eine Möglichkeit bieten, dieses *Dashboard* als Nutzer zu konfigurieren und die Ergebnisse diverser Metriken zu visualisieren. Diese Metriken sollen im Allgemeinen von anderen Plugins bereitgestellt werden. Das *Dashboard* soll erweiterbar sein, sodass andere Plugins eine Möglichkeit haben, ihre

Ergebnisse dem *Dashboard* beisteuern zu können. Auch sollen erste *Plugins*, wie das *Warnings Next Generation Plugin* [**?**] das *Application Programming Interface* (*API*) des erstellten *Pull Request Monitoring Plugins* nutzen können, um die Metriken in Form von *Portlets* dem *Dashboard* und somit dem Nutzer bereitzustellen. Auf Basis der *JavaScript* Bibliothek *Muuri* soll das *Plugin* entwickelt werden. *Muuri* liefert ein sortierbares, filterbares und verschiebbares Layout *Muuri* [**?**]. Dieses Layout soll mit den bereitgestellten *Portlets* anderer *Plugins* ausgestattet werden, wobei ein *Portlet* eine Komponente in Form einer Kachel mit der darzustellenden Metrik oder Qualitätsaussage innerhalb der Benutzeroberfläche definiert.

Kapitel 2 stellt bereits bestehende Arbeiten im wissenschaftlichen Kontext vor. In Kapitel 3 werden dann zunächst die nötigen Grundlagen geschaffen. Dabei werden vor allem auf die im weiteren Verlauf benötigten Begrifflichkeiten eingegangen und diese im Kontext der Arbeit erläutert. Auf die praktische Umsetzung und die damit verbundenen Entscheidungsfindungen wird in Kapitel 4 eingegangen. In Kapitel 5 werden schließlich die Ergebnisse vorgestellt. Kapitel 6 greift einige Diskussionspunkte auf, welche während der Entwicklung der Arbeit entstanden sind und begründet die getroffenen Entscheidungen. Neben der Zusammenfassung wird in Kapitel 7 auch ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben sowie auf die Zusammenarbeit mit der Comquent GmbH eingegangen.

2 Verwandte Arbeiten

Es gibt bereits zahlreiche Plugins für den *CI*-Server *Jenkins*, die Softwaremetriken berechnen. Darunter das *Code Coverage API Plugin*, welches die *Code Coverage* einer Codebasis ermittelt [**?**], das *Warnings Next Generation Plugin*, welches Compiler-Warnungen sammelt, die von statischen Analysetools gemeldet werden [**?**], oder das *Git Forensics API Plugin*, welches Daten aus einem Versionskontroll-*Repository* ausliest und analysiert [**?**]. Jedes *Plugin* bietet eine eigene Darstellung ihrer Ergebnisse. Meist werden die Metriken jedoch nur bezogen auf das Gesamtprojekt berechnet und nicht speziell für *Pull Requests*.

Erste Ansätze eines dedizierten *Dashboards*, welches Ergebnisse anderer *Plugins* aggregiert, sind in dem *Dashboard View Plugin* zu erkennen. Es können *Views* angelegt werden und diese mit Ergebnissen anderer *Plugins* befüllt werden, sofern diese ihre Ergebnisse in dem entsprechenden Format bereitstellen. Die *Portlets*, die von anderen *Plugins* bereitgestellt werden, beziehen sich aber nicht zwangsläufig auf reine Softwaremetriken oder gar die Darstellung einzelner Resultate für einen *Pull Request* [**?**]. Vielmehr ist dieses *Plugin* im Kontext des Gesamtprojekts zu betrachten und dient der generellen Überwachung einzelner Projekte. Damit wurde das Problem adressiert, das historisch gewachsene *User Interface* (*UI*) des *Jenkins* zu modernisieren und diverse Resultate in einem *Dashboard* zu aggregieren. Es fehlt der Bezug zum *Pull Request* und die Reduzierung der Resultate auf die durch den *Pull Request* veränderten Codebestandteile.

3 Grundlagen

3.1 *Source Code Management* (*SCM*)

Source Code Management, auch *SCM* genannt, dient dazu Versionen von Softwarekomponenten zu verwalten und wird verwendet, um Änderungen an einem Quellcode *Repository* zu verfolgen und die Historie aller Änderungen nachzuvollziehen. Das *Repository* bezeichnet dabei den Speicherort von Softwarekomponenten. Grundsätzlich soll es allen Mitwirkenden eines Projekts helfen, Aktualisierungen an einer Codebasis zusammenzuführen. Oft wird *SCM* als Synonym für *Version Control System* (*VCS*) verwendet [**?**].

3.2 *Git-Workflow* und *Pull Requests*

Ein solches *SCM* ist *Git*, welches Änderungen an einer Codebasis verwaltet und dokumentiert [**?**]. Üblicherweise wird die Codebasis eines Projektes im *SCM* System auf dem *master-Branch* verwaltet. Vom *master-Branch* wird auf den *developer-Branch* abgezweigt, von dem aus die Entwicklung startet. Eine Änderung, z. B. in Form einer neuen Funktionalität wird dann beispielsweise auf einem *feature-Branch* entwickelt. Dieser entspringt dem *developer-Branch* und wird am Ende der Entwicklung wieder in den *developer-Branch* integriert. Basierend auf dem *developer-Branch* werden *release-Branches* erzeugt, auf denen *Release*-Versionen auf Produktivumgebungen getestet und gegebenenfalls *Bugs* behoben werden. Ist der *Release* vollständig, so wird der *release-Branch* sowohl in den *developer-* als auch den *master-Branch* integriert. Dieser Prozess dient der besseren Zusammenarbeit und Skalierung des Entwicklungsteams und wird als *Git-Workflow* bezeichnet [**?**].

Das Zusammenführen der *Branches* und der damit verbundene Arbeitsablauf wird allgemein als *Pull Request* bezeichnet und wird immer von einem anderen Mitglied aus dem Team geprüft (*Review*) und anschließend akzeptiert (*Merge*) oder abgelehnt (*Close*). Durch den *Pull Request* signalisiert der Entwickler also die Fertigstellung seines Beitrags und bittet somit um die Prüfung der Änderung und das Zusammenführen in die Codebasis, fasst **?**] zusammen. Je nach verwendetem *SCM* System, wird der *Pull Request* teilweise auch als *Merge Request* bezeichnet [**?**].

Das Konzept der *Branches*, der *Git-Workflow* und der darin enthaltene Prozess des Pull Requests wird in Abbildung 3.1 beschrieben.

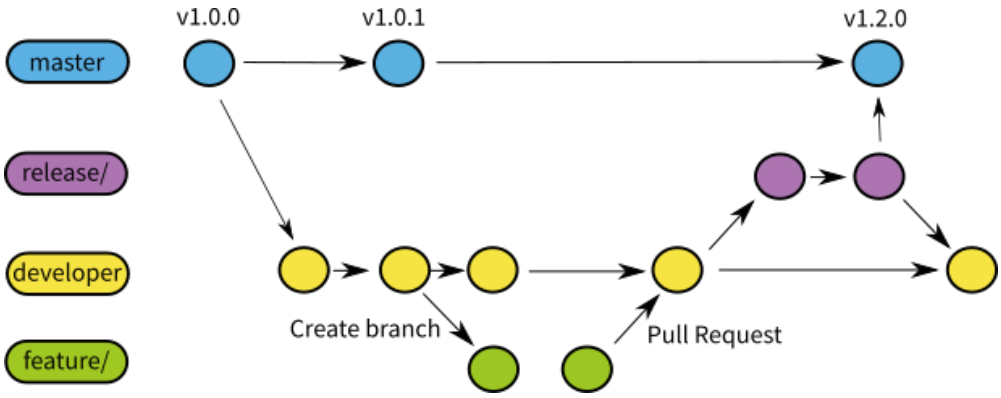

Abbildung 3.1: Visualisierung des *Git-Workflows*, Quelle: Eigene Darstellung nach **?** .

3.3 *Continuous Integration* (*CI*)

Der Begriff *Continuous Integration*, auch *CI* genannt, entstammt der Software Entwicklung und beschreibt dabei einen Prozess der permanenten Integration. Wie **?**] beschreibt, war es in den frühen Anfängen der Softwareentwicklung üblich, dass alle Änderungen der beteiligten Team Mitglieder zu einem Zeitpunkt, der sogenannten Integrationsphase, in der Codebasis vereint wurden und daraus ein *Release* erzeugt wurde. Diese Phase ging in der Regel mit harter Arbeit und monatelangem Lösen von Konflikten einher, da diese nur schwer vorher-

zusehen sind. Noch schwieriger ist es dann, diese zu lösen, da der Code unter Umständen mehrere Monate alt ist.

Der Prozess der *CI* wurde entwickelt, um diese Probleme zu adressieren. **?**] merkt an, dass eine *CI* zwar keine *Bugs* beseitigt, es aber deutlich einfacher macht, diese frühzeitig zu finden und zu eliminieren. Das Risiko und die Gefahr von unvorhersehbaren Kosten, verspäteten Software-*Releases* und in Folge unglücklicher Kunden wird minimiert [**?**].

Abbildung 3.2 visualisiert einen repräsentativen *CI* unterstützten technischen Prozess.

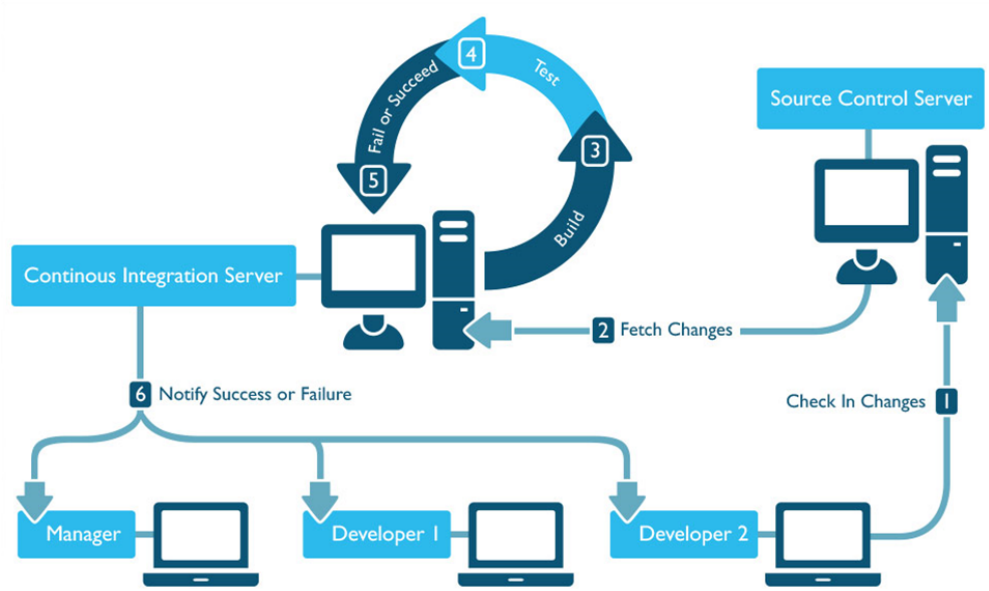

Abbildung 3.2: Technische Umsetzung von *CI*, Quelle: **?** .

Im Wesentlichen beobachtet der *CI*-Server ein *SCM* System (2) und führt eine definierte *Build Pipeline* bei jeder Änderung der dazugehörige Codebasis aus (3). So ist es denkbar, dass eine *Build Pipeline* die Codebasis kompiliert, automatisierte Tests ausführt und der Code einer statischen Code Analyse unterzieht (4). Dieser Prozess kann beliebig erweitert und an die Bedürfnisse der einzelnen Teams und deren Projekte angepasst werden. Sollte irgendetwas

nicht wie erwartet funktionieren, ein Test fehlschlagen oder gar das Kompilieren der Codebasis misslingen (5), so wird das Team unverzüglich von dem System darüber informiert (6) [**?**]. All das setzt voraus, dass jeder im Team seine *Features* häufig, normalerweise täglich, in das *SCM* System integriert (1) [**?**].

Nach **?**] ist somit der ganze Sinn einer *CI* schnelles Feedback zu erhalten, um auftretende *Bugs* dadurch frühzeitig zu finden und zu eliminieren, statt diese über lange Zeit in einem *Release* zu kumulieren. Durch die permanente Integration der einzelnen Codebestandteile (*Features*, *Bugfixes*, etc.) wird die Qualität der Software verbessert und Risiken minimiert.

Der *Jenkins* ist ein in *Java* geschriebener *Open Source CI*-Server. Die Entwicklung begann im Jahr 2004 durch Kohsuke Kawaguchi, damals unter dem Namen *Hudson* und zählt heute zu den populärsten *CI*-Servern. Dieser wird als Webapplikation bereitgestellt und bietet durch eine große, aktive Community und mehreren hunderten von *Plugins* flexible und diverse Einsatzmöglichkeiten in der permanenten Integration von Softwarekomponenten [**?**]. *Plugins* bezeichnen Zusatzmodule, die die Basisfunktionalität der eigentlichen Software erweitern. Das *Git Plugin* ermöglicht z. B. die Integration eines *Source Code Repositorys* in den *Jenkins*, sodass dieses als Projekt hinzugefügt werden kann und durch den *Jenkins* beobachtet und bei Änderungen analysiert werden kann [**?**].

3.4 Delta-Softwaremetriken

Nach **?**] bilden Softwaremetriken bestimmte Eigenschaften von Software als Zahlenwerte ab. Dadurch werden diese vergleichbar und dienen als Maßstab für die Qualität von Software. Beispielsweise bestimmt die *Code Coverage* den prozentualen Anteil des durch automatisierte Tests abgedeckten Quellcodes. Ein Delta bezeichnet im Allgemeinen eine Differenz.

Eine Delta-Softwaremetrik meint nun die Maßzahl einer bestimmen Softwaremetrik bezogen auf eine Referenz, für die es zwei verschiedene Betrachtungsweisen gibt - die absolute und die relative Referenz. Bei der absoluten Betrachtung

wird als Referenz die gesamte Codebasis des Ziel-*Branches* betrachtet. Das Delta der Metrik gibt also Aufschluss darüber, wie sich die Metrik bezogen auf das Gesamtprojekt durch einen *Pull Request* verändert.

Ein Beispiel: Auf dem *developer-Branch* eines *Repositorys* beträgt die *Code Coverage* 80 %. Ein Teammitglied entwickelt ein neues *Feature* basierend auf dem *developer-Branch* und stellt einen *Pull Request*, damit seine Änderungen in den *developer-Branch* integriert werden. Die *Code Coverage* auf dem *feature-Branch* beträgt 85 %. Durch einen erfolgreichen *Merge* wird die *Code Coverage* demnach um 5 % gesteigert.

Die relative Betrachtung einer Delta-Metrik bezieht sich dagegen auf den *Pull Request* selbst und gibt an, welche Zeilen des im *Pull Requests* veränderten, gelöschten oder hinzugefügten Quellcodes in der Metrik berücksichtigt sind.

Ein Beispiel: In einem *Pull Request* wurde der Codebasis eine neue Klasse mit 100 Zeilen hinzugefügt. Für diese Klasse existieren Tests, die den Code automatisiert testen. Diese Tests decken 67 der 100 Zeilen ab. Das bedeutet, dass die Zeilen, die sich in diesem *Pull Request* geändert haben, zu 67 % durch Tests abgedeckt sind. In der relativen Delta-Betrachtung, besitzt dieser *Pull Request* demnach eine *Code Coverage* von 67 %. Diese Delta-Betrachtung reduziert die Kennzahl der jeweiligen Softwaremetrik also auf die im *Pull Request* veränderten Codezeilen, anstatt das Gesamtprojekt zu beurteilen.

Häufig sind es diese Delta-Betrachtungen, die bei einem *Pull Request* relevant sind und für die *Reviewer* von Interesse sind, um die Qualität eines *Pull Requests* und der darin enthaltenen Weiterentwicklung beurteilen zu können.

4 Methodik und Umsetzung des Plugins

4.1 Konzeption

Zu Beginn der Arbeit existierten nur wenige Anforderungen für das in Kapitel 1 definierte Ziel, welche der Ausschreibung des Themas dieser Arbeit entstammen und in einem *Abstract* [**?**] verfasst wurden. Nach gemeinsamer Diskussion und Einarbeitung ergaben sich dadurch die im Folgenden beschriebenen Anforderungen.

Anforderung	*Dashboard*
Beschreibung	Das zu entwickelnde *Plugin* soll ein *Dashboard* zur Verfügung stellen, über das die wichtigsten Eigenschaften eines *Pull Requests* visualisiert werden.

Tabelle 4.1: 1. Anforderung: *Dashboard*

Anforderung	*Portlets* für das *Dashboard*
Beschreibung	Die in dem *Dashboard* angezeigten Eigenschaften eines *Pull Requests* sollen durch andere *Plugins* bereitgestellt werden.

Tabelle 4.2: 2. Anforderung: *Portlets*

Anforderung	Konfiguration
Beschreibung	Das *Dashboard* soll konfigurierbar sein, d. h. der Nutzer kann bestimmen, wie das *Dashboard* aufgebaut ist und welche *Portlets* in welcher Form angezeigt werden.

Tabelle 4.3: 3. Anforderung: Konfiguration

Anforderung	*API* für andere *Plugins*
Beschreibung	Damit andere *Plugins* solche *Portlets* aus Tabelle 4.2 bereitstellen können, muss eine *API* konzeptioniert und entwickelt werden.

Tabelle 4.4: 4. Anforderung: *API*

Anforderung	Distribution
Beschreibung	Das zu entwickelnde *Plugin* soll unter der *MIT*-Lizenz als *Jenkins Plugin* veröffentlicht werden, sodass dieses über das *Jenkins Update Center* zu finden ist [**?**].

Tabelle 4.5: 5. Anforderung: Distribution

Die Art und Weise der Umsetzung der einzelnen Anforderungen entschied sich größtenteils erst während der Einarbeitung bzw. der Arbeit selbst, beispielsweise durch das Feedback anderer Entwickler. Da ein neues *Plugin* erschaffen werden sollte, bestanden auch keine Abhängigkeiten zu anderen *Plugins*. Optional wurde als Ziel festgehalten, die ersten *Portlets* in anderen *Plugins* zu implementieren, sodass das *Plugin* im besten Fall im Rahmen dieser Arbeit bereits durch andere Nutzer verwendet werden kann. Zunächst soll in Abschnitt 4.2 die technische Realisierung dokumentiert werden, bevor in Kapitel 5 die Ergebnisse präsentiert werden.

4.2 Realisierung

Im Rahmen der vorliegenden Arbeit wurde ein *Jenkins Plugin* entwickelt. Es wird dessen Entwicklung und die verschiedenen Komponenten erläutert, die zur Umsetzung der in Kapitel 4 definierten Anforderungen maßgebend sind. Zur Veranschaulichung des Zusammenspiels der einzelnen Klassen soll Abbildung 4.1 dienen. Es zeigt ein vereinfachtes UML-Klassendiagramm der für die zu implementierenden *Plugins* bedeutsamen Klassen. Im Folgenden werden diese erläutert und ihre Funktion beschrieben werden. Aus Gründen der Übersichtlichkeit wird die Code-Dokumentation in den ausgewiesenen Code-Beispielen entfernt. Die jeweilige Originalquelle wird referenziert.

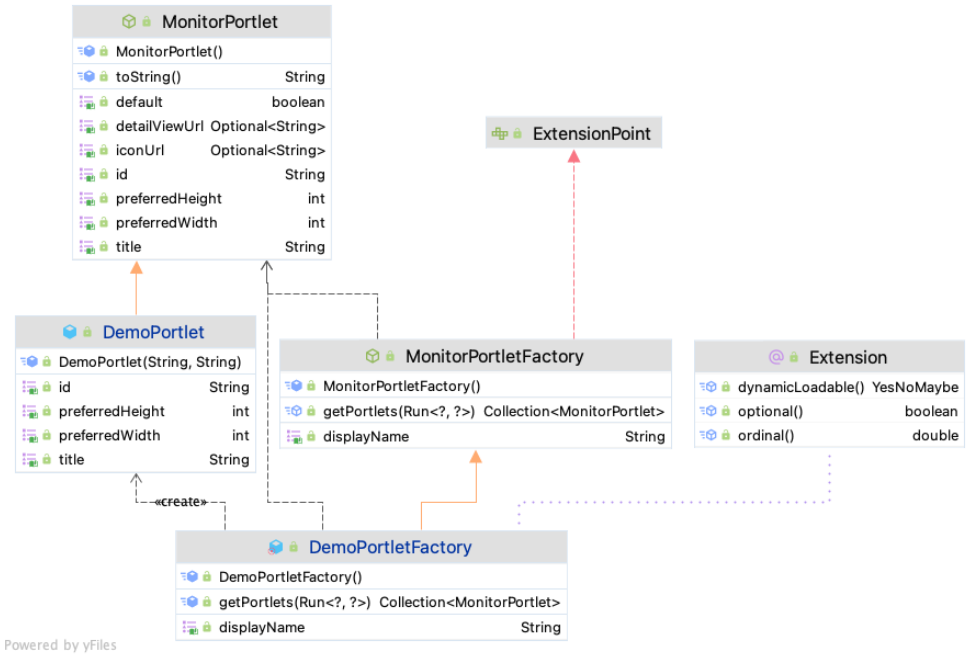

Abbildung 4.1: Vereinfachtes UML-Klassendiagramm der für die zu implementierenden *Plugins* relevanten Klassen, Quelle: Eigene Darstellung.

4.2.1 Abstrakte Klassen

Die in dem *Plugin* vorhandenen abstrakten Klassen fungieren als Schnittstelle für *Plugins*, die ein *Portlet* bereitstellen möchten. Die beiden abstrakten Klassen *MonitorPortletFactory* und *MonitorPortlet* werden dabei stets zusammen verwendet und stehen in einer *1*:*n*-Beziehung. Jedes *Plugin* besitzt eine *MonitorPortletFactory*, die die *MonitorPortlets* erzeugt und ausliefert. Diese *«create»* Beziehung ist in Abbildung 4.1 veranschaulicht. Dadurch wird sichergestellt, dass ein *Plugin* gegebenenfalls mehrere Instanzen der Klasse *MonitorPortlet* bereitstellen kann. Dieser Aspekt ist für die Entwicklung des *Portlets* für das *Warnings Next Generation Plugin* von Bedeutung, welches in Unterabschnitt 5.2.1 vorgestellt wird. Damit das implementierende *Plugin* die beiden Klassen verwenden kann, bedarf es einer Referenz des *Pull Request Monitoring Plugins* in der *pom.xml*, dem *Project Object Model*. Darin werden alle wichtigen Aspekte des Projektes, wie Projektinformationen oder Projektbeziehungen, also Abhängigkeiten zu anderen Projekten beschrieben. Diese zentrale Steuerungsdatei wird verwendet, um mithilfe des auf *Java* basierenden Build-Managment-Tools *Maven* das Projekt zu verwalten und zu steuern [**?**]. Alle *Jenkins Plugins* sind in *Java* geschrieben und nutzen dieses Build-Managment-Tool [**?**].

Listing 4.1: Auszug der Datei *plugin/pom.xml* aus dem *Warnings Next Generation Plugin* [?].

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
      www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
      maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.
      xsd">
2   <modelVersion>4.0.0</modelVersion>
3
4   <groupId>io.jenkins.plugins</groupId>
5   <artifactId>warnings-ng</artifactId>
6   <packaging>hpi</packaging>
7   <name>Warnings Next Generation Plugin</name>
8   <version>${revision}${changelist}</version>
9
```

```
10    <properties>
11      <revision>9.2.0</revision>
12      <changelist>-SNAPSHOT</changelist>
13
14      <pull-request-monitoring.version>1.7.1</pull-request-monitoring.
            version>
15    </properties>
16
17    <dependencies>
18      <dependency>
19        <groupId>io.jenkins.plugins</groupId>
20        <artifactId>pull-request-monitoring</artifactId>
21        <version>${pull-request-monitoring.version}</version>
22        <optional>true</optional>
23      </dependency>
24    </dependencies>
25  </project>
```

Listing 4.1 demonstriert die Referenzierung anhand des *Warnings Next Generation Plugins*. Zeile 22 deklariert die Abhängigkeit als optional. Dadurch wird diese Abhängigkeit nur dann verwendet, wenn die zugehörige Funktionalität tatsächlich benutzt wird. Wird diese Funktionalität nicht benutzt, so wird auch die Abhängigkeit nicht verwendet.

MonitorPortlet

Die *MonitorPortlet* Klasse definiert die für ein *Portlet* zu implementierenden Methoden. Ein *Plugin*, welches ein oder mehrere *Portlets* bereitstellen möchte, muss von dieser abstrakten Klasse erben und die Methoden implementieren und gegebenenfalls überschreiben, falls die vordefinierte Implementierung nicht dem jeweiligen Zweck entspricht. Listing 4.2 zeigt die abstrakte Klasse mit ihren Methoden. Ein Titel (vgl. Zeile 3), eine eindeutige ID (vgl. Zeile 5), die Breite (vgl. Zeile 11) und die Höhe (vgl. Zeile 13) sind in jedem Fall zu implementieren. Zu den optionalen Methoden zählen die Definition eines *Icons* (vgl. Zeile 15) oder einer Unterseite in Form eines *Uniform Resource Locator*s (*URL*s) (vgl.

Zeile 19), die geöffnet wird, wenn der Nutzer auf den Titel des im *Dashboard* hinzugefügten *Portlets* klickt. Seit Version 1.7.0 kann ein *Portlet* entscheiden, ob dieses standardmäßig (vgl. Zeile 7) in jedem *Dashboard* angezeigt werden soll [**?**]. Näheres dazu wird in Abschnitt 6.5 diskutiert.

Listing 4.2: Auszug der Klasse *io.jenkins.plugins.monitoring.MonitorPortlet.java* [**?**].

```
1  public abstract class MonitorPortlet {
2
3      public abstract String getTitle();
4
5      public abstract String getId();
6
7      public boolean isDefault() {
8          return false;
9      }
10
11     public abstract int getPreferredWidth();
12
13     public abstract int getPreferredHeight();
14
15     public Optional<String> getIconUrl() {
16         return Optional.empty();
17     }
18
19     public Optional<String> getDetailViewUrl() {
20         return Optional.empty();
21     }
22
23     @Override
24     public String toString() {
25         return "MonitorPortlet{'" + getId() + "'}";
26     }
27 }
```

Eine minimalistische Implementierung der *MonitorPortlet* Klasse wird im *Pull Request Monitoring Plugin* mit ausgeliefert und stellt dem Anwender ein erstes beispielhaftes *Portlet* für das *Dashboard* zur Verfügung, welches Listing 4.3 zeigt.

Listing 4.3: Auszug der Klasse *io.jenkins.plugins.monitoring.DemoPortlet.java* [**?**].

```
1  public class DemoPortlet extends MonitorPortlet {
2      private final String id;
3      private final String title;
4
5      public DemoPortlet(final String title, final String id) {
6          super();
7          this.id = id;
8          this.title = title;
9      }
10
11     @Override
12     public String getTitle() {
13         return title;
14     }
15
16     @Override
17     public String getId() {
18         return id;
19     }
20
21     @Override
22     public int getPreferredWidth() {
23         return 300;
24     }
25
26     @Override
27     public int getPreferredHeight() {
28         return 200;
29     }
30 }
```

Jenkins realisiert seine *UI* mittels *Jelly*, einer *Java*- und *XML*-basierten Skripting- und Verarbeitungsengine zur Umwandlung von *XML* in ausführbaren Code [**?**]. Die zugehörige *UI*-Komponente eines *MonitorPortlets* wird als *Jelly*-Datei ausgeliefert und trägt den Namen *monitor.jelly*. *Jelly*-Dateien sind direkt an Klassen gebunden. Das bedeutet, dass sie Methoden dieser Klassen aufrufen können. Um die Datei zu referenzieren, an die sie gebunden sind, verwenden *Jelly*-Dateien das Schlüsselwort `it`. Dabei bedarf es einer bestimmten Verzeichnisstruktur der *Java* Klassen und der *Jelly*-Dateien [**?**]. Die *DemoPortlet* Klasse liegt unter *./src/main/java/io/jenkins/plugins/-monitoring/DemoPortlet*. Die *Jelly*-Datei, also die *UI*-Komponente zu dieser Klasse liegt dann unter *./src/main/resources/io/jenkins/plugins/monitoring/-DemoPortlet/monitor.jelly*. Listing 4.4 zeigt die *UI*-Komponente der Klasse *DemoPortlet*.

Listing 4.4: *io/jenkins/plugins/monitoring/DemoPortlet/monitor.jelly* der Klasse *io.jenkins.plugins.monitoring.DemoPortlet.java* [**?**].

```
1 <?jelly escape-by-default='true'?>
2
3 <j:jelly xmlns:j="jelly:core">
4
5     <p>${it.title}</p>
6
7 </j:jelly>
```

`it` ist somit die Referenz auf die konkrete Implementierung *DemoPortlet* der abstrakten Klasse *MonitorPortlet*. Zeile 5 kann dadurch auf der zugehörigen *Java*-Klasse die Methode aus Listing 4.3 Zeile 14 ausführen und erhält als Ergebnis den Titel des *Portlets*.

MonitorPortletFactory

Die Klasse *MonitorPortletFactory* erzeugt die Instanzen der Klasse *MonitorPortlet*. Die abgewandelte Form des *Factory Patterns*, wie es erstmals **?**] beschrieben, ermöglicht es den Anwendern, mehrere Instanzen derselben Klasse zu erzeugen, sodass mehrere baugleiche *Portlets* erstellt werden können. Ein

Plugin, welches ein oder mehrere *Portlets* bereitstellen möchte, muss von dieser abstrakten Klasse erben und die abstrakten Methoden implementieren. Listing 4.5 zeigt die Klasse mit ihren abstrakten Methoden.

Listing 4.5: Auszug der Klasse *io.jenkins.plugins.monitoring.MonitorPortletFactory.java* [**?**].

```
1 public abstract class MonitorPortletFactory implements ExtensionPoint {
2
3     public abstract Collection<MonitorPortlet> getPortlets(Run<?, ?>
          build);
4
5     public abstract String getDisplayName();
6
7 }
```

Der Name (vgl. Zeile 5) der *MonitorPortletFactory* dient lediglich der Anzeige im *Dashboard*, unter der alle verfügbaren *Portlets* gelistet sind. Siehe dazu Abbildung 5.4. Eine *MonitorPortletFactory* liefert eine Menge an *MonitorPortlets* aus (vgl. Zeile 3). Diese Methode wiederum wird später verwendet, um die verfügbaren *Portlets* abzurufen. Damit der Inhalt der zu erzeugenden *Portlets* an den jeweiligen *Run* angepasst werden kann, wird der jeweils aktuelle *Run* der Schnittstellenmethode hinzugefügt.

Listing 4.6 zeigt die Implementierung der *DemoPortletFactory* der in dem *Pull Request Monitoring Plugin* mit ausgelieferten Demo-*Portlets*. Da diese *Factory* lediglich zwei statische *Portlets* ausliefert und nur der Demonstration dient, wird der übergebene *Run* ignoriert. In einer realen Implementierung, wie der aus Unterabschnitt 5.2.1 und Unterabschnitt 5.2.2, werden über diesen *Run* Informationen abgegriffen, um den Inhalt der *Portlets* entsprechend anzupassen.

Listing 4.6: Auszug der statischen Klasse *io.jenkins.plugins.monitoring.DemoPortletFactory.java* der Klasse *io.jenkins.plugins.monitoring.DemoPortlet.java* [**?**].

```
1 @Extension(optional = true)
2 public static class DemoPortletFactory extends MonitorPortletFactory {
```

```
3
4       @Override
5       public Collection<MonitorPortlet> getPortlets(final Run<?, ?> build
            ) {
6           List<MonitorPortlet> portlets = new ArrayList<>();
7           portlets.add(new DemoPortlet("Good First Portlet", "first-demo-
                portlet"));
8           portlets.add(new DemoPortlet("Another Portlet", "second-demo-
                portlet"));
9           return portlets;
10      }
11
12      @Override
13      public String getDisplayName() {
14          return "Pull Request Monitoring (Demo)";
15      }
16
17  }
```

4.2.2 Das Konzept *ExtensionPoint* des *Jenkins*

Ein wesentliches Konzept des *Jenkins* besteht darin, die Kernfunktionalität durch *Plugins* erweitern zu können. Dafür stellt der *Jenkins* Erweiterungspunkte in Form von Schnittstellen oder abstrakten Klassen zur Verfügung, die die *Plugins* nutzen können, um eine Implementierung beizusteuern. Ein *Plugin* selbst kann dann wiederum eigene *ExtensionPoints* definieren, die von weiteren *Plugins* verwendet werden können, wie in dem vorliegenden Beispiel der Klasse *MonitorPortletFactory*. Die Markierungsschnittstelle (*Marker Interface)* ExtensionPoint, die von der abstrakte Klasse *MonitorPortletFactory* implementiert wird (vgl. Listing 4.5), dient dazu, dem *Jenkins* zur Laufzeit Informationen über die implementierende Klasse der *MonitorPortletFactory* zu liefern. Durch das Scannen aller verfügbaren Klassen innerhalb des *Java Classpaths* werden alle Klassen gefunden, die die abstrakte Klasse und somit den *ExtensionPoint* implementieren. Die entsprechende Implementierung wird registriert und dem *Jenkins* bekannt gemacht. Durch die Annotation der Klasse

mit *@Extension* (vgl. Listing 4.6, Zeile 1) wird eine Instanz der Klasse erzeugt und in der *ExtensionList* des *Jenkins* registriert. Sofern die Abhängigkeit zu dem *Pull Request Monitoring Plugin* in der *pom.xml* aus Listing 4.1 als optional deklariert ist, muss auch die Annotation *Extension* als optional markiert sein. Listing 4.7 zeigt die Anwendung der *ExtensionList*, um die registrierten Instanzen der *MonitorPortletFactory* Klasse auszulesen. Diese Klasse *PortletService* stellt diverse statische Methoden zur Verfügung, um mit der *ExtensionList* des *Jenkins* zu interagieren und die Implementierungen der abstrakten Klassen aus Unterabschnitt 4.2.1 anderer *Plugins* auszulesen. Die Verwendung der Klasse wird in Abschnitt 4.2.5 näher beschrieben.

Listing 4.7: Auszug der Klasse *io.jenkins.plugins.monitoring.util.PortletService.java* [**?**].

```
1  public final class PortletService {
2
3      public static List<? extends MonitorPortletFactory> getFactories()
           {
4          return ExtensionList.lookup(MonitorPortletFactory.class);
5      }
6
7      public static List<? extends MonitorPortlet> getAvailablePortlets(
           final Run<?, ?> build) {
8          return getFactories()
9                  .stream()
10                 .map(factory -> factory.getPortlets(build))
11                 .flatMap(Collection::stream)
12                 .collect(Collectors.toList());
13     }
14 }
```

4.2.3 Konfiguration

Die Konfiguration des *Dashboards*, also die in einem *Dashboard* verwendeten *Portlets*, wird pro Projekt gespeichert. Ein Projekt meint dabei in der Regel immer ein *SCM-Repository*. Für einen Nutzer werden sämtliche Konfigurationen

aller *Dashboards* zusammen mit einer eindeutigen ID des zugehörigen Projekts als Liste in einer *UserProperty*, der *MonitorConfigurationProperty*, gespeichert. Diese Klasse stellt einige Methoden zur Verfügung, um die Liste der Konfigurationen abzufragen oder zu bearbeiten. Durch einen statische *Factory*-Methode [**?**] kann die für den aktuellen Nutzer vorhandene *MonitorConfigurationProperty* erfragt werden.

Listing 4.8: Auszug der Klasse *io.jenkins.plugins.monitoring.MonitorConfigurationProperty.java* [**?**].

```
1 public static Optional<MonitorConfigurationProperty> forCurrentUser() {
2     final User current = User.current();
3     return current == null ? Optional.empty() : Optional.ofNullable(
          current.getProperty(MonitorConfigurationProperty.class));
4 }
```

Sollte kein Nutzer eingeloggt sein, so liefert die Methode aus Listing 4.8 kein Ergebnis. Die *MonitorConfigurationProperty* hält stets eine *default* Konfiguration in der Liste, die geladen wird, wenn der Nutzer das jeweilige *Dashboard* das erste Mal öffnet oder für das Projekt keine anderweitige Konfiguration vorhanden ist. Die *default* Konfiguration enthält dabei immer diejenigen *Portlets*, deren Methode aus Listing 4.2 Zeile 7 *true* zurückliefert. Diese *default* Konfiguration des *Dashboards* kann durch den Nutzer auf zwei unterschiedliche Wege verändert werden: Entweder über das *UI* des *Dashboards* oder die *Pipeline*.

4.2.4 *Pipeline*

Eine Pipeline ist eine Sammlung von Schritten, die für ein bestimmtes Projekt definiert und ausgeführt werden. Üblicherweise wird eine *Pipeline* in *Jenkins* als *Jenkinsfile* benannt. Die *Pipeline* ist eine typische Charakteristik des *CI* Prozesses, die dafür sorgt, dass die Software oder allgemeiner das Projekt von dem *SCM Repository* in ein an den Kunden auslieferbares Produkt überführt wird [**? ?**]. Verschiedene Schritte, sogenannte *Build-Jobs* werden dadurch zu einem *Workflow* zusammengefasst und in einer Datei, dem *Jenkinsfile*, als *Groovy*-Code beschrieben [**?**]. Das *Pull Request Monitoring Plugin* stellt einen

Build-Job zur Verfügung, womit der Nutzer das *Dashboard* vorkonfigurieren kann. Listing 4.9 zeigt lediglich die entsprechende *Stage* des *Jenkinsfiles*, in der die Konfiguration für das *Dashboard* definiert wird. *Stages* in *Jenkinsfiles* dienen dazu, den *Code* zu strukturieren und konzeptionell gegenüber anderen *Stages* abzugrenzen. Üblicherweise findet man in einem *Jenkinsfile* eine *Build-*, *Test-* und *Deploy-Stage* [**?**].

Listing 4.9: *Pull Request Monitoring Stage* eines *Jenkinsfiles* [**?**].

```
1  stage ('Pull Request Monitoring - Dashboard Configuration') {
2      monitoring (
3          '''
4          [
5              {
6                  "id": "first-demo-portlet",
7                  "width": 400,
8                  "height": 400,
9                  "color": "#FF5733"
10             },
11             {
12                 "id": "second-demo-portlet"
13             }
14         ]
15         '''
16     )
17 }
```

Bei dieser Konfiguration werden die beiden durch das *Pull Request Monitoring Plugin* ausgelieferten Demo-*Portlets* konfiguriert. Der Nutzer hat die Möglichkeit von der Breite, der Höhe und der Farbe der Implementierung des *Portlets* abzuweichen und diese zu überschreiben. Die Konfiguration wird dabei als *JSON* beschrieben. Durch Zeile 2 wird der *Code* im *Jenkinsfile* mit der entsprechenden *Java*-Klasse, dem *Monitor* verknüpft. Dieser *Step* wird im Kontext des *Workflows* ausgeführt und dadurch zu gegebenem Zeitpunkt die zugehörige *Java*-Klasse aufgerufen. Die *Java*-Klasse fügt eine *MonitoringCustomAction* mit

der definierten Konfiguration dem entsprechenden *Run* hinzu, was Listing 4.10 zeigt.

Listing 4.10: Auszug des *Steps io.jenkins.plugins.monitoring.Monitor.java* [**?**].

```
1  @Override
2  public Void run() throws Exception {
3
4      if (PullRequestFinder.isPullRequest(run.getParent())) {
5          getContext().get(TaskListener.class).getLogger()
6              .println("[Monitor] Build is part of a pull request. Add '
                   MonitoringCustomAction' now.");
7
8          run.addAction(new MonitoringCustomAction(monitor.getPortlets())
                );
9      }
10     else {
11         getContext().get(TaskListener.class).getLogger()
12             .println("[Monitor] Build is not part of a pull request.
                   Skip adding 'MonitoringCustomAction'.");
13     }
14
15     return null;
16 }
```

4.2.5 *Actions*

Actions sind im Allgemeinen Objekte, die es ermöglichen Informationen zu speichern und *UI*-Elemente dem *Jenkins* hinzuzufügen. Jede *Action* definiert dabei einen Unterbereich innerhalb des aktiven *ModelObjects* des *Jenkins*. Ein *ModelObject* ist eine Schnittstelle, die von *Actions* implementiert wird und definiert, dass die implementierende *Action* eine *URL* zur Verfügung stellen muss. Über die definierte *URL* kann der Nutzer mit dem *UI* interagieren und den durch die *URL* definierten Unterbereich aufrufen. Eine *Action* zu einem *ModelObject* wird immer auch in der Menüleiste des *Jenkins* angezeigt. Die für das *Pull Request Monitoring Plugin* wichtigen *ModelObjects* beschränken sich

auf den *Job* und den *Run*. Das *Pull Request Monitoring Plugin* nutzt insgesamt vier solcher *Actions*. Das *ModelObject* System lässt sich am besten an den durch die *Actions* definierten *URL*s veranschaulichen. Abbildung 4.2 zeigt die gesamte *URL* einer *MonitoringDefaultAction*, anhand derer die einzelnen *ModelObjects* erkennbar sind.

$$\overbrace{\overbrace{localhost:8080/jenkins/}^{1.}\overbrace{job/}^{2.}\overbrace{PullRequestMonitoringPlugin/}^{3.}}^{}$$
$$\underbrace{view/change-requests/}_{4.}\underbrace{job/}_{5.}\underbrace{PR-2/}_{6.}\underbrace{3/}_{7.}\underbrace{pull-request-monitoring/}_{8.}$$

Abbildung 4.2: *URL* einer *MonitoringDefaultAction*, Quelle: Eigene Darstellung.

1. Root *URL* des *Jenkins*-Servers.
2. Typ des *ModelObjects* erster Ebene (*Job*).
3. Name des *Jobs* erster Ebene. Dieser *Job* ist das *MultiBranchProject*.
4. Spezifische *URL*s des *MultiBranchProjects*.
5. Typ des *ModelObjects* zweiter Ebene (*Job*).
6. Name des *Jobs* zweiter Ebene. Dieser *Job* repräsentiert beispielsweise einen *Pull Request* und entspricht einem *WorkflowJob*.
7. Typ des *ModelObjects* dritter Ebene (*Run*). Dem Namen des *Runs* (hier der dritte *Run*) steht anders als den *ModelObjects* erster und zweiter Ebene nicht der Typ voran.
8. Spezifische *URL* der *MonitoringDefaultAction*. Darüber ist das *Dashboard* erreichbar.

Das verknüpfte *SCM-Repository* wird als *MultiBranchProject* dem *Jenkins* hinzugefügt und bildet den hierarchischen Knotenpunkt (2, 3). Für dieses *ModelObject* vom Typ *Job* definiert das *Pull Request Monitoring Plugin* eine *MonitoringMultibranchProjectAction*. Diese gibt einen ersten Überblick über das Projekt und wird in Unterabschnitt 5.1.1 näher vorgestellt. Ein *MultiBranchProject* zeichnet sich dadurch aus, dass jeder *Branch* und jeder *Pull Request* als

eigenes *ModelObject* (*Job*) angesehen wird und hierarchisch unter dem *ModelObject* des *MultiBranchProjects* geführt wird (3 - 6). Auf dieser Ebene stellt das *Pull Request Monitoring Plugin* die *MonitoringWorkflowJobAction* zur Verfügung. Diese *Action* referenziert lediglich den aktuellsten *Run* des zugehörigen *Jobs* der zweiten Hierachiebene. Ein *Job* dieser Ebene (5), welcher z. B. einen bestimmten *Pull Request* aus dem verknüpften *SCM-Repository* abbildet (6), besitzt in der Regel eine Menge an *ModelObjects* des Typs *Run* (7). Ein *Run* definiert dabei die unterste Ebene des *ModelObjetcs*. Hier wird das eigentliche *Dashboard* für jeden *Run* in Form einer *MonitoringDefaultAction* bereitgestellt (8).

MonitoringCustomAction

Eine *MonitoringCustomAction* speichert die nutzerspezifische Konfiguration aus dem *Jenkinsfile*. Diese ist eine unsichtbare *Action* und wird nicht in dem *UI* des *Jenkins* angezeigt. Bestehende *Actions*, die einem *Run* hinzugefügt wurden, können nicht mehr verändert werden. Da die *MonitoringDefaultAction* in jedem Fall einem *Run* hinzugefügt wird, sobald dieser geöffnet wird, bedarf es dieser zusätzlichen *Action* im Falle der Konfiguration über eine *Pipeline*. Diese *Action* wird dem *Run* erst hinzugefügt, wenn der entsprechende *Step* der *Pipeline* ausgeführt wurde.

MonitoringDefaultAction

Jedem *Run* innerhalb eines *Multibranch* Projekts wird die *MonitoringDefaultAction* zugeordnet, sofern es sich um einen *Run* im Kontext eines *Pull Requests* handelt. Diese ist über das Menü erreichbar und stellt das eigentliche *Dashboard* bereit.

Die jeweils gültige Konfiguration zu dem *Dashboard* wird zur Laufzeit ermittelt. Listing 4.11 zeigt die darin involvierten Methoden der *MonitoringDefaultAction*.

Listing 4.11: Auszug der Klasse *io.jenkins.plugins.monitoring.MonitoringDefaultAction.java* [**?**].

```
1  public class MonitoringDefaultAction implements RunAction2,
       StaplerProxy {
2
3      private transient Run<?, ?> run;
4
5      public Run<?, ?> getRun() {
6          return run;
7      }
8
9      public String getConfiguration() {
10
11         MonitorConfigurationProperty monitorConfigurationProperty =
               MonitorConfigurationProperty
12                 .forCurrentUser().orElse(null);
13
14         return monitorConfigurationProperty == null
15                 ? resolvePortlets()
16                 : monitorConfigurationProperty.getConfiguration(
                       getConfigurationId()).getConfig();
17     }
18
19     public String getConfigurationId() {
20         String id = getRun().getParent().getParent().getDisplayName();
21         return StringUtils.toRootLowerCase(id).replaceAll(" ", "-");
22     }
23
24     public String resolvePortlets() {
25         MonitoringCustomAction action = getRun().getAction(
               MonitoringCustomAction.class);
26         return action == null
27                 ? getPortlets()
28                 : action.getPortlets();
29     }
30
31     public String getPortlets() {
```

```
32          return PortletService.getDefaultPortletsAsConfiguration(getRun
                ());

33      }
34  }
```

Existiert eine entsprechende Konfiguration für das aktuelle Projekt in der *MonitoringConfigurationProperty* (vgl. Zeile 16), so wird diese angewendet. Besteht keine Konfiguration für das Projekt, so liefert die Abfrage der Konfiguration die *default*-Konfiguration.

Wenn keine *MonitoringConfigurationProperty* für den aktuellen Nutzer vorhanden ist (vgl. Zeile 15), weil dieser zum Beispiel nicht eingeloggt ist, so wird entweder die *default*-Konfiguration angewendet (vgl. Zeile 27 und 32) oder falls eine *MonitoringCustomAction* existiert (vgl. Zeile 28), die darin enthaltene Konfiguration verwendet. Die *UI*-Komponente einer *Action* wird durch die zugehörige *Jelly*-Datei beschrieben.

Listing 4.12: Auszug der Datei *io/jenkins/plugins/monitoring/MonitoringDefaultAction/index.jelly* [**?**].

```
1  <?jelly escape-by-default='true'?>
2
3  <j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler" xmlns:l="/lib/
       layout"
4          xmlns:modals="/modals" xmlns:portlet="/portlet" xmlns:alerts="/
               alerts" xmlns:metadata="/metadata">
5
6      <st:header name="Content-Type" value="text/html;charset=UTF-8"/>
7              <st:adjunct includes="io.jenkins.plugins.select2"/>
8              <st:adjunct includes="io.jenkins.plugins.muuri"/>
9              <st:adjunct includes="io.jenkins.plugins.bootstrap5"/>
10             <div class="grid">
11
12                 <j:set var="portlets" value="${it.getAvailablePortlets(
                       it.run)}"/>

13
```

```
14              <j:forEach var="portlet" items="${portlets}">
15
16                  <portlet:portlet portlet="${portlet}"/>
17
18              </j:forEach>
19
20          </div>
21          <script>var run = <st:bind value="${it}"/></script>
22          <script type="text/javascript" src="${resURL}/plugin/pull-
                request-monitoring/js/pull-request-monitoring.js"/>
23
24      </l:main-panel>
25  </j:jelly>
```

Zeile 7 und 8 binden zwei weitere *Plugins* ein, die die externen *JavaScript* Bibliotheken *Muuri* und *Select2* über ein *Jenkins Plugin* zur Verfügung stellen [**?** **?**]. *Muuri* liefert ein sortierbares, filterbares und verschiebbares Layout, in das die *Portlets* eingebettet werden [**?**]. *Select2* sorgt dafür, dass die Liste der verfügbaren *Portlets* durchsucht und gefiltert werden kann [**?**]. Das Ergebnis wird in Unterabschnitt 5.1.2 vorgestellt.

Zeile 10 dient der Einführung des *Muuri* Layouts und definiert die äußere Form des *Dashboards*. *Muuri* erlaubt es, in dieses Layout einzelne *Items* einzubetten.

In Zeile 12 werden alle verfügbaren *Portlets* durch den *PortletService* aus Listing 4.7 ermittelt. Dazu werden alle *MonitorPortletFactory* Instanzen, die in der *ExtensionList* registriert sind, abgerufen und die jeweiligen *MonitorPortlets* erfragt. Für jedes dieser *MonitorPortlet* Objekte (vgl. Zeile 14) wird eine weitere *Jelly*-Datei inkludiert und die jeweilige *MonitorPortlet* Instanz als Attribut übergeben (vgl. Zeile 16). Listing 4.13 zeigt die zu inkludierende *Jelly*-Datei.

Listing 4.13: Auszug der Datei *portlet/portlet.jelly* [**?**].

```
1  <?jelly escape-by-default='true'?>
2
3  <j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler">
4      <st:documentation>
```

```
5         A portlet for the dashboard.
6
7         <st:attribute name="portlet" use="required">
8             The portlet (instance of MonitorPortlet).
9         </st:attribute>
10    </st:documentation>
11
12    <st:header name="Content-Type" value="text/html;charset=UTF-8"/>
13
14    <div class="muuri-item hidden" data-id="${portlet.id}"
15         data-color="#000000" data-title="${portlet.title}" default-
               width="${portlet.preferredWidth}"
16         default-height="${portlet.preferredHeight}" default-color="
               #000000">
17
18        <div class="muuri-item-content">
19
20            <div class="card" style="color: #000000">
21
22                <div class="plugin-card-id">
23
24                    <j:if test="${portlet.detailViewUrl.isPresent()}">
25                        <a class="plugin-link" style="color: #000000"
                               href="../${portlet.detailViewUrl.get()}">
26                            ${portlet.title}</a>
27                    </j:if>
28
29                    <j:if test="${!portlet.detailViewUrl.isPresent()}">
30                        ${portlet.title}
31                    </j:if>
32
33                </div>
34
35                <div class="plugin-card-content">
36
37                    <st:include page="monitor.jelly" it="${portlet}"/>
38
```

```
39                </div>
40
41                <div class="plugin-remove">
42                    <i class="material-icons icon">&#xE5CD;</i></div>
43
44            </div>
45
46        </div>
47
48    </div>
49
50 </j:jelly>
```

Für jedes *MonitorPortlet* wird das eigentliche *Item* erzeugt und dieses dem Layout aus Listing 4.12 Zeile 10 untergeordnet. Die spezifischen Attribute jedes *MonitorPortlets* werden als HTML-Attribute dem *Item* hinzugefügt (vgl. Zeile 14) und die *UI*-Komponete des jeweiligen *Portlets* ebenfalls als *Jelly*-Datei inkludiert (vgl. Zeile 37).

Initial werden alle *Portlets* dem Layout als unsichtbares *Item* durch die Zuweisung der *CSS*-Klasse *hidden* aus Listing 4.13 Zeile 14 hinzugefügt. Die jeweils gültige Konfiguration wird dann mit den in dem Layout vorhandenen *Items* abgeglichen und die Attribute der *Items*, z. B. Höhe, Breite oder Farbe gegebenenfalls angepasst. Anschließend werden die vom Nutzer gewünschten *Portlets* durch das Entfernen der *CSS*-Klasse *hidden* wieder sichtbar gemacht.

Die Verwaltung der *Portlets* geschieht in einer spezifischen *JavaScript*-Datei *pull-request-monitoring.js*. Die *MonitoringDefaultAction* wird ständig über Änderungen informiert und persisitiert diese in der *MonitorConfigurationPrtoperty*. Ähnlich wie mit dem Schlüsselwort *it* einer *Jelly*-Datei, kann auch aus einer *JavaScript*-Datei auf *Java-Klassen* zugegriffen werden und die darin definierten Methoden aufgerufen werden. Dazu wird in Listing 4.12 Zeile 21 die Instanz der *MonitoringDefaultAction* einer Variablen *run* innerhalb eines script-Tags zugewiesen. In Zeile 22 wird dann die spezifische *JavaScript*-Datei eingebunden. Darin wird über die Variable *run* die gültige Konfiguration aus der *Monitoring-*

DefaultAction abgerufen und die bereits bestehenden *Items* des *Muuri* Layouts entsprechend angepasst. Das Abrufen und Inititialisieren erfolgt in Listing 4.14.

Listing 4.14: Auszug der Datei *pull-request-monitoring.js* [**?**].

```
1 /* global jQuery3, run, Muuri, JSONTree */
2 (function ($) {
3
4     run.getConfiguration(function(config) {
5         initDashboard(config.responseJSON);
6     });
7
8 })(jQuery3);
```

5 Ergebnisse

Nachdem in Kapitel 4 bereits die technische Erklärung der einzelnen Bestandteile erfolgte, werden nun die für den Anwender wichtigen Ergebnisse innerhalb des *Jenkins UI* dargestellt.

5.1 *User Interface* des Plugins

Das *Plugin* bietet, wie in Unterabschnitt 4.2.5 eingeführt, zwei *Actions* mit einer *UI*-Komponente. Eine Übersicht über alle *Pull Requests* auf erster *Job*-Ebene und das eigentliche *Dashboard* auf der *Run*-Ebene.

5.1.1 *Job*-Ansicht

Für jeden *Pull Request* werden der Name des *Pull Requests*, der Name des Mitwirkenden und sowohl der Quell- als auch der Ziel-*Branch* aufgelistet, wie Abbildung 5.1 zeigt.

Von hier aus kann entweder zu der *UI*-Komponente des letzten zugehörigen *Runs*, also der *Run*-Ansicht aus Unterabschnitt 5.1.2 mit dem *Dashboard* navigiert werden oder der *Pull Request* im *SCM*-System, z. B. *GitHub* geöffnet werden.

5.1.2 *Run*-Ansicht

Das *Dashboard* besteht aus verschiedenen Einheiten. Einem Titel, den Metadaten eines *Pull Requests*, dem *Muuri* Layout mit den *Portlets* und zwei Schaltflächen, um neue *Portlets* hinzuzufügen und die Einstellungen zu öffnen.

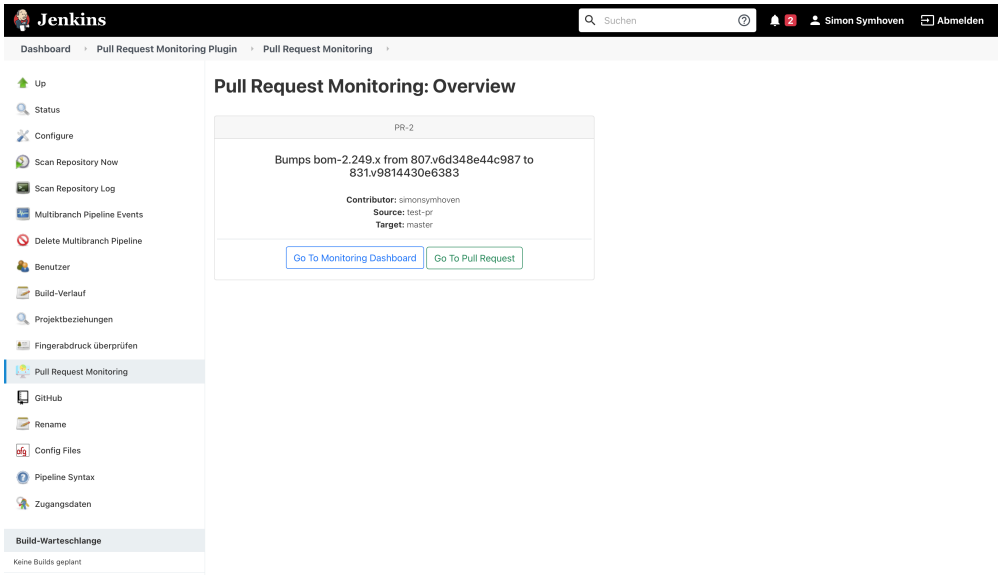

Abbildung 5.1: *MonitoringMultibranchProjectAction* auf erster *Job*-Ebene: Übersicht aller *Pull Requests*, Quelle: Eigene Aufnahme.

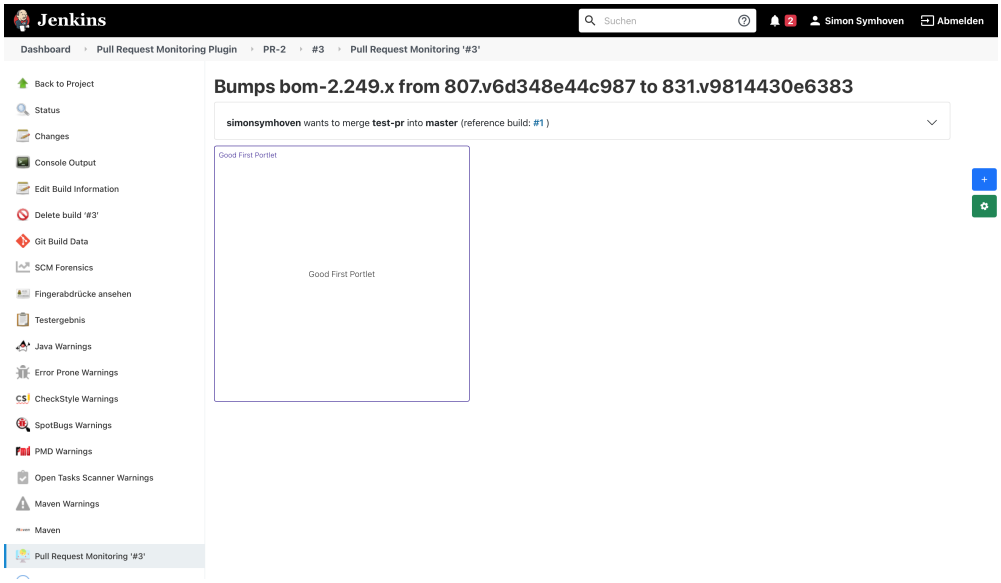

Abbildung 5.2: *MonitoringDefaultAction* auf *Build*-Ebene: Das *Dashboard*, Quelle: Eigene Aufnahme.

Die *Portlets* innerhalb des *Muuri* Layouts aus Abbildung 5.2 können per *Drag & Drop* beliebig verschoben werden oder mit einem Klick auf die rechte obere Ecke des jeweiligen *Portlets* gelöscht werden.

Metadaten eines *Pull-Requests*

Analog zu der Übersicht der *Pull Requests* aus Unterabschnitt 5.1.1 werden auf *Run*-Ebene alle Metadaten des *Pull Requests* angezeigt. Außerdem wird der Referenz-*Run* des zugehörigen Ziel-*Branches* angegeben. Zusätzlich wird, falls vorhanden, die Beschreibung aus dem *SCM*-System ausgegeben, wie Abbildung 5.3 zeigt:

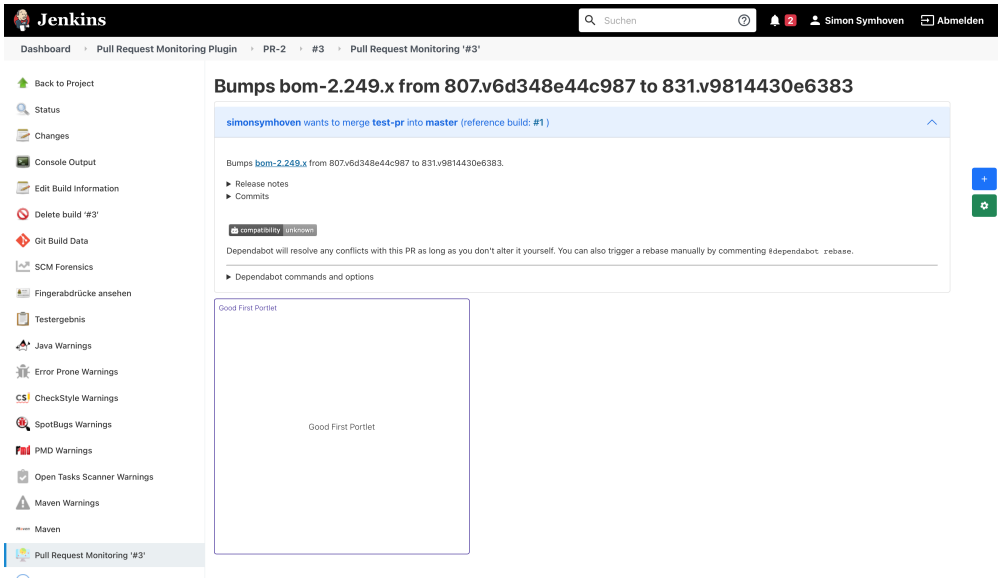

Abbildung 5.3: Metadaten eines *Pull Requests*, Quelle: Eigene Aufnahme.

Portlets hinzufügen

Um ein neues *Portlet* hinzuzufügen, muss zunächst aus der Liste der verfügbaren *Portlets* eines ausgewählt werden, wie Abbildung 5.4 zeigt. Ist dieses bereits im Layout enthalten, so wird das entsprechende *Portlet* ausgegraut. Falls gewünscht, kann die vordefinierte Breite und Höhe des ausgewählten *Portlets*

modifiziert und überschrieben werden. Auch die Farbe kann verändert werden. Abbildung 5.5 zeigt die Einstellungen für das selektierte *Java Portlet* aus dem *Warnings Next Generation Plugin*.

Abbildung 5.4: Ein neues *Portlet* hinzufügen. Auswahl des *Portlets*, Quelle: Eigene Aufnahme.

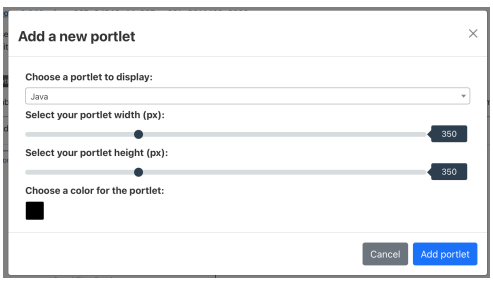

Abbildung 5.5: Ein neues *Portlet* hinzufügen. Auswahl der Größe und Farbe, Quelle: Eigene Aufnahme.

Einstellungen

Die Einstellungen aus Abbildung 5.6 bieten dem Nutzer die Möglichkeit, bestimmte Informationen über das Dashboard einzusehen. Es wird ermittelt, ob es Änderungen an der Konfiguration in der *Pipeline* seit dem letzten *Run* gibt und die Quelle der Konfiguration angezeigt. Diese kann entweder *Default* oder *User-Specific* sein. Außerdem wird berechnet, ob die aktuelle Konfiguration mit der *default*-Konfiguration übereinstimmt. Der Anwender hat die Möglichkeit, die aktuelle Konfiguration zu jeder Zeit auf die *default*-Konfiguration zurückzusetzen. Die beiden Konfiguration werden außerdem als *JSON-Tree* mit ihren Attributen abgebildet.

5.2 Verfügbare *Portlets*

Neben dem *Plugin* selbst wurde während der Entwicklung der vorliegenden Arbeit das entwickelte *API* verwendet, um zwei dieser *Portlets* in anderen *Jenkins*

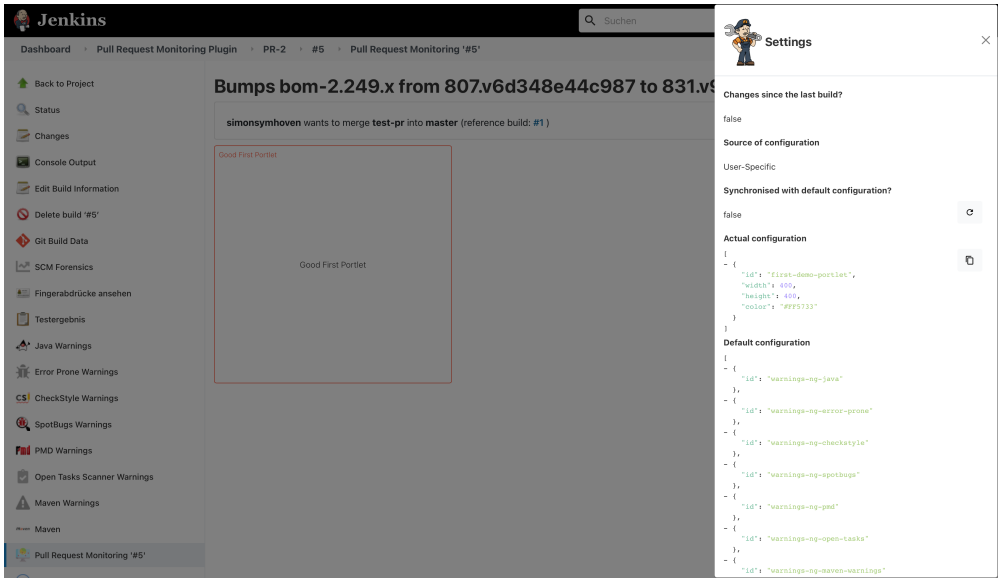

Abbildung 5.6: Ansicht: Einstellungen, Quelle: Eigene Aufnahme.

Plugins zu entwickeln. So konnte das *API* auf Tauglichkeit geprüft werden und erleichtert das Entwickeln weiterer *Portlets* für andere *Plugin* Betreiber.

5.2.1 *Warnings Next Generation Plugin*

«Das *Warnings Next Generation Plugin* sammelt Compiler-Warnungen oder Probleme, die von statischen Analysewerkzeugen gemeldet werden, und visualisiert die Ergebnisse. Es hat eingebaute Unterstützung für mehr als hundert Berichtsformate.», schreibt **?**] über sein *Plugin*. Das *Warnings Next Generation Plugin* liefert für jedes dieser Analysewerkzeuge mithilfe der *MonitorPortlet-Factory* ein *Portlet* aus. Die grafische Darstellung der Ergebnisse unterscheidet sich von *Portlet* zu *Portlet* nicht - lediglich die zugrundeliegenden Daten ändern sich. Somit wird eine Klasse *MonitorPortlet* verwendet und für jedes Analysewerkzeug eine Instanz dieser Klasse mit den spezifischen Ergebnissen ausgeliefert. Dabei werden drei Fälle unterschieden:

1. Es liegen keine Warnungen vor (Abbildung 5.7).

2. Es gibt keine neuen Warnungen, aber ausstehende und eventuell gelöste Warnungen (Abbildung 5.8).
3. Es gibt neue Warnungen. Dazu eventuell ausstehende und gelöste Warnungen (Abbildung 5.9).

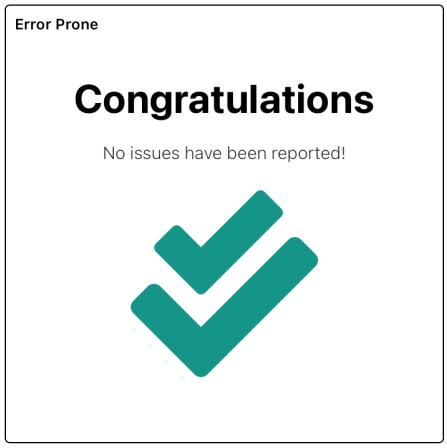

Abbildung 5.7: *Portlet* des *Warnings Next Generation Plugins* ohne Warnungen, Quelle: Eigene Aufnahme.

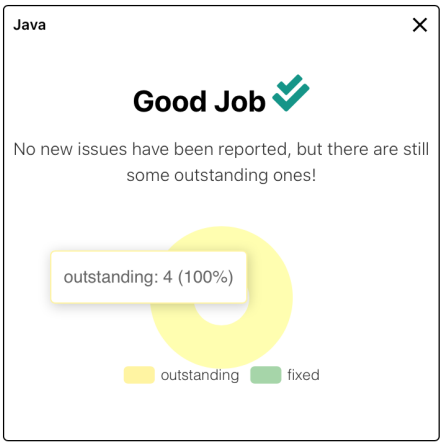

Abbildung 5.8: *Portlet* des *Warnings Next Generation Plugins* ohne neue Warnungen, Quelle: Eigene Aufnahme.

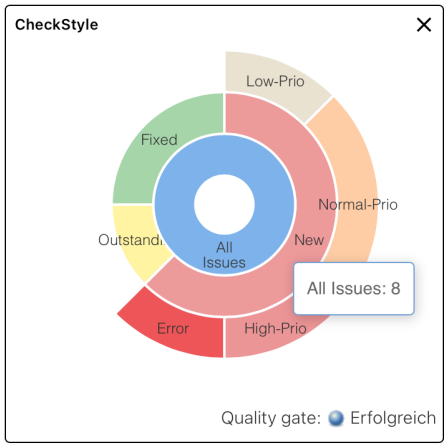

Abbildung 5.9: *Portlet* des *Warnings Next Generation Plugins* mit Warnungen, Quelle: Eigene Aufnahme.

5.2.2 *Code Coverage API Plugin*

Das *Code Coverage API Plugin* dient als *API* zur Integration und Veröffentlichung mehrerer *Coverage Reports* [**?**]. Das *Portlet* visualisiert die *Conditional-* und *Line-Coverage* bezogen auf den aktuellen *Run* und erzeugt aus den Metriken ein Balkendiagramm. Zusätzlich wird die Differenz zu dem Referenz-*Run* kalkuliert und als Delta hinter der jeweiligen *Coverage* ausgegeben. Die absolute *Conditional-* und *Line-Coverage* aus dem *Run* beträgt 78,57 % bzw. 81,82 %. Die Delta-*Coverage* bezogen auf den Referenz-*Run* des Ziel-*Branches* hat sich nicht verändert. Die *Code Coverage* (*Conditional* und *Line*) ist folglich in Quell- und Ziel-*Branch* dieselbe und wird durch den *Pull Request* nicht vermindert oder verbessert.

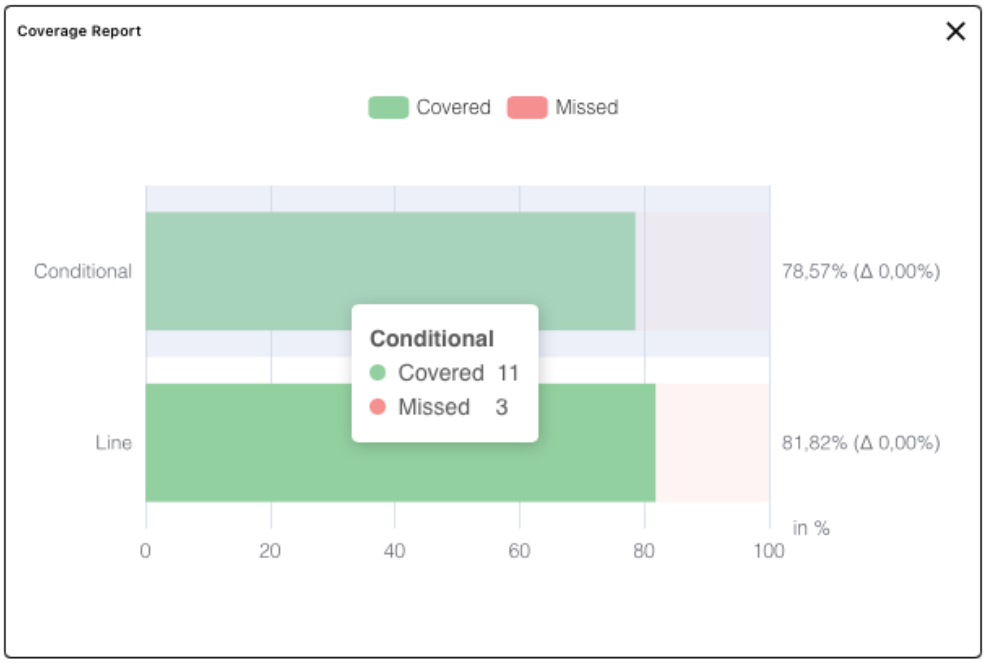

Abbildung 5.10: *Portlet* des *Code Coverage API Plugins*, Quelle: Eigene Aufnahme.

6 Diskussionen

6.1 Allgemeine Voraussetzung

Kernaufgabe des *Jenkins CI*-Servers ist die Ausführungen von *Jobs*. Es gibt *Freestyle-Jobs*, in denen die einzelnen *Build Steps* konfigurierbar definiert werden. *Pipelines* sind spezielle *Jobs*, deren Definition in Form von *Groovy*-Code erfolgt. Diese können in der *Job*-Konfiguration oder im *SCM* abgelegt werden. Durch die Ablage im *SCM* und der Verwendung des *Plugins Pipeline: Mutlibranch* wird das Erkennen von verschiedenen *Branches* in dem verwendeten *SCM-Repository* ermöglicht. Jeder *Branch* wird als ein separater *Job* behandelt, für den unabhängige Arbeitsabläufe als *Pipeline* oder innerhalb der *Job* Konfiguration definiert werden können [**?**]. Das *Pull Request Monitoring Plugin* beschränkt sich auf die Verwendung von *Multibranch* Projekten.

6.2 *Abstract Class* statt *Interface*

Mit Version *1.6.0* wurden die beiden Klassen *MonitorPortlet* und *MonitorPortletFactory* von einem *Interface* zu einer *Abstract Class* geändert [**?**]. Dies bietet den Vorteil, dass bestimmte Methoden mit einer Implementierung belegt werden können, die nur bei Bedarf von der erbenden Klasse überschrieben werden müssen. Des Weiteren bietet die Verwendung einer *Abstract Class* eine deutlich stabilere Schnittstelle nach außen an, da im Falle von einer Erweiterung eine vordefinierte Implementierung der hinzugefügten Methode mitgeliefert werden kann, sodass es bei der erbenden Klasse zu keinen Fehlern während des Kompilierens kommt.

6.3 *User Property* statt *Local Storage*

Bis Version *1.0.3-beta* wurde die Konfiguration des *Dashboards*, also der im *Dashboard* verwendeten *Portlets* im lokalen Speicher, dem *Local Storage* des Browsers persistiert. Darunter die Breite, Höhe und die eindeutige ID jedes *Portlets*, um diese beispielsweise bei der Aktualisierung der Seite erneut zu laden [**?**]. Diese erste lauffähige Beta-Version wurde im *Jenkins UX SIG Meeting*, dem *Jenkins User Experience Special Interest Group Meeting* präsentiert [**?**]. Tim Jacomb, ein aktives Mitglied der *UX SIG* Gruppe, merkte an, dass das Speichern der Konfiguration im lokalen Speicher des Browsers diverse Nachteile mit sich bringt. Konfigurationen sind somit abhängig vom eingesetzten Browser, können also beim Verwenden eines anderen Browsers nicht übernommen werden. Dasselbe gilt, wenn Nutzer sich an unterschiedlichen Rechnern anmelden, um auf eine *Jenkins* Instanz zuzugreifen. Er schlug vor die Konfiguration stattdessen in einer *User Property*, der *MonitorConfigurationProperty* innerhalb der *Jenkins* Infrastruktur zu speichern, um zu jederzeit browser- und rechnerunabhängig auf die Konfiguration zugreifen zu können.

6.4 Auswahl der *JavaScript* Bibliothek *Muuri*

Es existieren diverse *JavaScript* Bibliotheken, welche bereits ein konfigurierbares, verschiebbares und dynamisches Layout für ein *Dashboard* zur Verfügung stellen, darunter *gridstack.js* [**?**] oder *Muuri*. *Muuri* stellte sich als für diesen Zweck einzig brauchbare Bibliothek heraus, da es diese dem Entwickler erlaubt, den Inhalt der einzelnen *Items*, den *Portlets* als HTML zu setzen und über Klassenzuweisungen dem *Dashboard* hinzuzufügen und zu steuern. Bei *gridstack.js* ist dies nicht möglich. Hier muss der zu setzende Inhalt statisch über das *API* der Bibliothek gesetzt werden. Das stellte sich als nicht praktikabel heraus, da die bereitgestellten *Portlets* als *Jelly-View* ausgeliefert werden. Diese können nur in anderen *Jelly-Views* inkludiert werden und nicht im Nachhinein hinzugefügt werden, da das *Jelly* Rendering serverseitig beim Aufruf der entsprechenden *URL* initiiert wird.

6.5 *Default-Portlets* für das *Dashboard*

Seit Version 1.7.0 kann ein *Portlet* entscheiden, ob dieses standardmäßig im *Dashboard* angezeigt werden soll [**?**]. Sofern für das jeweilige *Dashboard* keine nutzerspezifische Konfiguration existiert, werden dem Nutzer beim Öffnen eines zu einem *Run* zugehörigen *Dashboards* alle vorhanden *Portlets* angezeigt, die als *default* markiert sind. Das soll dem Nutzer den Einstieg erleichtern und die manuelle Konfiguration der *Dashboards* vermeiden. Durch lokale Änderungen über das *UI* des *Dashboards* oder die Anwendung einer im *Jenkinsfile* definierten Konfiguration, wird diese *default*-Konfiguration überschrieben.

7 Zusammenfassung und Ausblick

Statt die Resultate für das Gesamtprojekt zu visualisieren, ist eine gefilterte Darstellung der Ergebnisse auf die tatsächlich vorgenommene Änderung an der Codebasis innerhalb eines *Pull Requests* sinnvoll. Diese Arbeit beschäftigte sich mit der Konzeptionierung und Entwicklung eines *Jenkins Plugins* zur Überwachung bestimmter Qualitätskriterien und Metriken eines *Pull Requests*. Diese beziehen sich auf die durch den *Pull Request* veränderte Codebasis - das Delta zwischen Quell- und Ziel-*Branch*. Das Monitoringsystem wird dem Nutzer in Form eines konfigurierbaren und eines durch andere *Plugins* erweiterbaren *Dashboards* innerhalb des *Jenkins UI* zur Verfügung gestellt. Die grundlegenden Funktionen des *Dashboards* basieren dabei auf der externen Bibliothek *Muuri*. Die Konfiguration des *Dashboards* wird browser- und rechnerunabhängig pro Nutzer innerhalb des *Jenkins* gespeichert und ist jederzeit abrufbar. Außerdem wurde ein *API* entwickelt, welches andere *Plugins* konsumieren und ihre Metriken als *Portlets* dem Nutzer bereitstellen können. Diese können somit in dem nutzerspezifischen *Dashboard* hinzugefügt und überwacht werden. Dieses *API* wurde im Rahmen der vorliegenden Arbeit verwendet, um die ersten *Portlets* in dem *Warnings Next Generation* und *Code Coverage API Plugin* zu entwickeln und zu veröffentlichen. Das *Pull Request Monitoring Plugin* ist unter der *MIT*-Lizenz als *Open Source* Projekt veröffentlicht und steht zur Installation innerhalb des *Jenkins* Servers bereit.

Abschließend werden diejenigen Punkte angesprochen, für die die Zeit nicht ausgereicht hat, die aber im Rahmen weiterer Studien oder Erweiterungen des *Plugins* interessant wären zu untersuchen und umzusetzen.

Die Größe eines *Portlets* wird in Pixeln von dem implementierenden *Plugin* festgelegt und verändert diese Größe nicht, wenn das *Dashboard*, respektive

das Browserfenster verkleinert oder vergrößert wird. Denkbar ist es, dies durch ein *Grid*-System zu ersetzen, sodass die Größe der *Portlets* sich der Größe des Browserfensters anpassen, statt eine statische Größe zu definieren.

Des Weiteren können die *Portlets* innerhalb des *Dashboards* nicht bearbeitet werden. Dazu müssen diese zuerst entfernt werden und dann über den Dialog aus Abbildung 5.5 mit den neuen Parametern dem *Dashboard* wieder hinzugefügt werden. Praktikabler wäre eine Möglichkeit für den Nutzer, die bereits existierenden *Portlets* über einen weiteren Dialog direkt bearbeiten zu können ohne diese vorher löschen zu müssen.

Ebenfalls sollen für weitere *Plugins* wie dem *JUnit* [**?**] oder dem *Git Forensics Plugin* [**?**] *Portlets* entwickelt werden und weitere Autoren motiviert werden, die zur Verfügung gestellte *API* zu nutzen, um der *Jenkins Community* ein vielfältigeres *Dashboard* bereitstellen zu können.

Durch die Vorstellung der ersten Beta-Version des *Plugins* in einem der *Jenkins UX SIG* Treffen, wurde die Comquent GmbH aus Puchheim auf das *Plugin* und diese Arbeit aufmerksam. Dadurch ergab sich die Möglichkeit, die vorliegende Arbeit auch im Kontext der wirtschaftlichen Anwendung zu diskutieren und voraussichtlich in aktiven Projekten bei Kunden der Comquent GmbH einzusetzen. Dadurch wäre es besonders interessant herauszufinden, welche Anforderungen sich aus der wirtschaftlichen Anwendung ergeben, um diese mit in die Weiterentwicklung des *Plugins* einfließen lassen zu können. In einem ersten Gespräch mit dem Geschäftsführer der Comquent GmbH, Herrn Andreas Schönfeld, wies dieser darauf hin, dass sich die Anforderungen der Wirtschaft nicht selten von denen aus der Entwickler-Szene unterscheiden, da Trends, Prozesse und Arbeitsweisen häufig sehr viel später antizipiert werden. Als mögliche Weiterentwicklung des *Plugins* wurden die Einflussfaktoren der Delta-Resultate diskutiert: Welcher Entwickler hat durch welche Änderungen welche Metriken wie verändert? Aufgrund der zeitlichen Einschränkungen können die Ergebnisse der praktischen Anwendung nicht in der vorliegenden Arbeit diskutiert werden. Besonders bedanke ich mich bei Herrn Andreas Schönfeld für die Kooperation und die Zusammenarbeit. Diese hat mir wertvolle Einblicke gewährt und eine praxisnahe Forschung ermöglicht, die über diese Arbeit hinaus anhalten wird.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings