Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Entwicklung einer flexibel konfigurierbaren Build-und Integrationtest-Pipeline der SAP HANA Development Infrastruktur

Kelvin Glaß
Matrikelnummer: 4659865
kelvin.glass@fu-berlin.de

Betreuer: Prof. Dr. Lutz Prechelt

Berlin, 12. Januar 2015

Zusammenfassung

In der aktuellen Infrastruktur der SAP HANA zur Erstellung, Qualitätsicherung und Verteilung von SAP HANA Komponenten, sind die Build-und Integrationtest-Pipelineschritte fest vorgegeben. Eine Pipeline meint in dem Kontext dieser Arbeit eine Verkettung von Schritten der Erstellung, Qualitätssicherung oder Verteilung. Für jede HANA-Komponente müssen die einzelnen Build-und Integrationtest-Schritte in dem Continuous-Integration Server Jenkins neu implementiert werden und sind dann erneut fest. Diese Arbeit hat das Ziel die Infrastruktur zu erweitern, so dass eine Build- und Integrationtest-Pipeline der SAP HANA Development-Infrastruktur frei konfiguriert und ausgeführt werden kann. Diese Erweiterung würde die Bereitstellung der Infrastruktur für neue Komponenten wesentlich vereinfachen und beschleunigen. Zudem würde sie den Komponenteninhabern die Möglichkeit eröffnen, ihre Build- und Integrationtest-Schritte jederzeit individuell gestalten zu können. In dem ersten Teil dieser Arbeit wird beschrieben, wie Pipelinekonfigurationen und Änderungen an diesen versioniert in einer Datenhaltung persistiert werden können. Zudem wird dargestellt wie eine Pipeline anhand einer Konfiguration in dem Jenkins-Server erstellt und ausgeführt werden kann. Anschließend wird die Architektur des Systems, das diese Funktionalität bereitstellt entworfen. In dem zweiten Teil wird das zuvor abstrakt beschriebene System implementiert und bewertet, inwiefern verständlich und wartbar diese Implementierung ist.

Kelvin Glaß

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis oder in den Fußnoten angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

12. Januar 2015

Kelvin Glaß

Inhaltsverzeichnis

1 Einführung

Die aktuelle SAP HANA Development-Infrastruktur ist eine Umgebung, welche ermöglicht, dass Produkte und Komponenten für die hybride In-Memory-Datenbank HANA in unterschiedlichen Quellsprachen (aktuell vertreten: Java, C/C++) beliebig viele Schritte der Erstellung, Qualitätssicherung und Verteilung (Build, Codescan und Deploy) durchlaufen. Diese Schritte werden im Folgenden als Pipelineschritte bezeichnet und eine Verkettung dieser Schritte wird als Pipeline betitelt.

Das Starten eines Schrittes wird definiert als die Übergabe von Informationen eines Schrittes an einen Scheduler. Diese Informationen müssen es dem Scheduler ermöglichen, den Schritt auszuführen. Der Scheduler muss dabei zusichern, dass der Schritt nach endlicher Zeit ausgeführt wird.

Die Abarbeitungsweise einer Pipeline kann vereinfacht wie folgt dargestellt werden:
Der erste Pipelineschrittdurchlauf wird durch eine Abgabe im Versionsverwaltungssystem[1] Git[1] ausgelöst oder in einem spezifizierten Intervall (durch die Ausführung eines Continuous-Build[1]) aufgerufen. Dabei werden dem ersten Pipelineschrittdurchlauf zusätzliche Daten übergeben. Ist ein Pipelineschrittdurchlauf erfolgreich abgeschlossen und ein Nachfolge-Pipelineschritt existiert, startet dieser den Nachfolge-Pipelineschritt und gibt ihm beim Startaufruf die initial erhaltenen Daten mit eventuellen Änderungen mit. Sollte ein Pipelineschrittdurchlauf fehlschlagen, so wird kein Nachfolge-Pipelineschrittdurchlauf aufgerufen. Sobald keine Nachfolge-Schritte mehr zu starten und alle Schritte durchlaufen sind, gilt der gesamte Pipelinedurchlauf als terminiert.

Die Abgabe einer Änderung wird aktuell durch das Code-Review-System[1] Gerrit[1] registriert. Der erste Pipelineschrittdurchlauf startet, sobald das Code-Review-System signalisiert, dass eine neue Änderung in einem bestimmten Git-Repository[1] auf einem zugehörigen Git-Branch[1] eingegangen ist.

Für jedes Produkt (bzw. jede Komponente) existiert in dem Continuous Integration (CI) Server [1] Jenkins[1] eine Pipeline. Die einzelnen Pipelineschritte werden innerhalb des Servers durch einzelne konfigurierbare ausführbare Einheiten (sogenannte Jobs), welche Anweisungen für den Jenkins oder die

[1] Erklärung im Glossar
[2] Herkunft der Logo-Grafiken:
Jenkins-Logo: `https://wiki.jenkins-ci.org/download/attachments/2916393/logo.zip?version=3&modificationDate=1305324183000`
Gerrit-Logo: `http://upload.wikimedia.org/wikipedia/mediawiki/a/a7/Gerrit.svg`
Git-Logo: `http://git-scm.com/images/logos/downloads/Git-Logo-1788C.png`

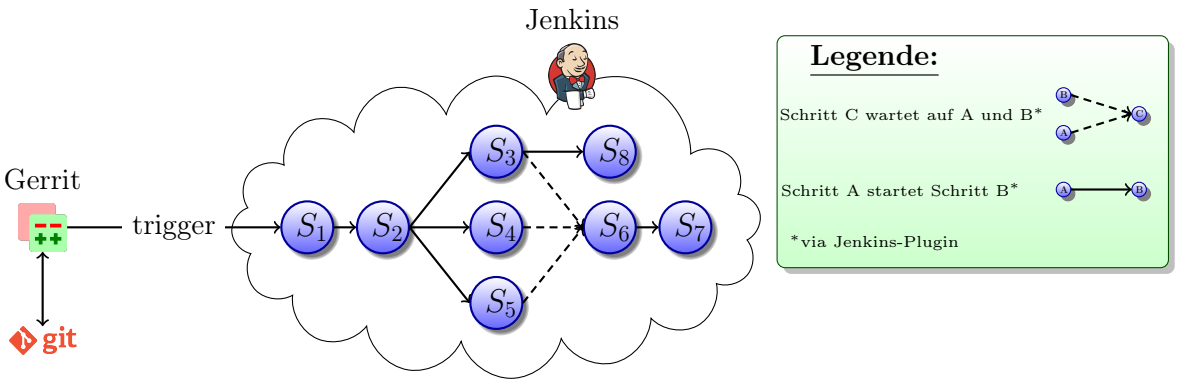

Abbildung 1: Visualisierung der Abarbeitungsweise [2]

Jenkins-Plugins enthalten, abgebildet. Die Jobs werden jedoch nicht alle auf dem System, auf welchem der Server eingerichtet wurde, ausgeführt sondern auf Systemen, welche sich für diese Aufgabe im Jenkins registriert haben (sogenannte Nodes) verteilt. Alle Systeme liegen in Form von automatisch erstellbaren und konfigurierbaren virtuellen Maschinen vor. Dies sorgt für eine dynamische und skalierbare Hardware-Infrastruktur.

Der erste Job eines Pipelinedurchlaufs ist immer ein Job, welcher auf ein Signal des Gerrit wartet, das durch eine Abgabe in dem produktspezifischen Repository auf einem bestimmten Branch ausgelöst wurde. Während der Ausführung schreibt ein Job alle relevante Informationen in Form von Log-Dateien auf einen Speicherpool. Ein Job kann vor oder nach seiner Ausführung einen Nutzer im Gerrit impersonifizieren und als dieser ein Voting abgeben. Dabei wird der Voting-Mechanismus des Gerrit jedoch nicht wie vorgesehen zur Meinungsabgabe zu einer Codeänderung genutzt. Stattdessen wird anhand des Nutzers, welcher das Voting abgegeben hat und des Werts des Votings, signalisiert, dass ein bestimmter Pipelineschritt in einem bestimmten Zustand ist. Somit kann dem Nutzer beispielsweise signalisiert werden, dass ein Schritt gestartet, bzw. mit einem bestimmten Resultat terminiert ist. Beispielsweise kann der Nutzer 'Build' genutzt werden, um mit einem Review-Wert von -2 zu signalisieren, dass der Erstellungsschritt fehlgeschlagen ist. Im Fall eines Continuous Build kann eine Mail an jemanden versendet werden um den Status des Durchlaufs zu kommunizieren.

In der aktuellen Infrastruktur sind die Pipelineschritte fest vorgegeben. Für jedes HANA Produkt (bzw. jede Komponente) muss die Verkettung der einzelnen Pipelineschritte neu im CI-Server implementiert werden.
Zudem existiert keine Möglichkeit, um eine Änderungshistorie der Pipeline eines jeden Produkts zu erhalten. Anhand der Änderungshistorie der Pipeline

könnten die Komponenteninhaber die Entwicklung ihrer Pipeline zurückverfolgen.

Des Weiteren wird der Zustand der Daten eines Pipelineschrittdurchlaufs nicht persistiert. Diese Eigenschaft führt dazu, dass ein Pipelineschrittdurchlauf, welcher von einem Ereignis außerhalb der SAP HANA Development-Infrastruktur abhängig ist und aufgrund dessen scheitert, nur durch das Durchlaufen der gesamten Pipeline erneut gestartet werden kann.[3] Dieses Verhalten kostet unnötig viel Zeit und Rechenressourcen.

Eine flexibel konfigurierbare Pipeline der SAP HANA Development Infrastruktur würde die Bereitstellung der Infrastruktur für neue Komponenten wesentlich vereinfachen und beschleunigen. Zudem würde sie den Komponenteninhabern die Möglichkeit eröffnen, ihre Pipeline jederzeit individuell gestalten zu können und neue Pipelines aufzusetzen.

Ein langfristiges Ziel ist die Entkopplung der Infrastruktur von dem Jenkins-Server, so dass die Möglichkeit besteht, diesen durch ein anderes System zu ersetzen.

Diese Arbeit soll darlegen, wie die beschriebenen Probleme der aktuellen SAP HANA Development-Infrastruktur durch ein möglichst leicht verständliches System gelöst werden können.

In diesem Kapitel wurden die Probleme der SAP HANA Development Infrastruktur erläutert. Das zweite Kapitel beschreibt die Anforderungen an eine Lösung der zuvor dargestellten Probleme. In dem dritten wird anhand dieser Anforderungen ein Modell zur Lösung entworfen. Es folgt die Beschreibung für eine Implementierung dieses abstrakten Modells. Zum Schluss wird das Fazit gezogen und in einem Ausblick dargelegt, welche Erweiterungen der Lösung sinnvoll sind.

[3] Auflistung der Abhängigkeiten der Infrastruktur zu externen Systemen vgl. Anhang A.1

2 Anforderungen

In dem letzten Kapitel wurde beschrieben, wie die aktuelle Infrastruktur aufgebaut ist. Anschließend wurden die Probleme, die es in dieser Arbeit zu lösen gilt abstrakt beschrieben. In diesem Kapitel werden die Anforderungen an die Lösung dieser Probleme genauer beschrieben und erläutert.

2.1 Anforderungen

Die Anforderungen an die Lösung, lassen sich in Anforderungen an die Persistenz von Konfigurationen und Anforderungen an das Verhalten und die Steuerung von Pipelinedurchläufen unterteilen. Die grundlegende Persistenzanforderung ist, dass eine Pipelinekonfiguration hinterlegt werden kann und die Möglichkeit der Modifikation der Konfiguration besteht. Zudem sollte es möglich sein, auf aktuelle und bereits überarbeitete Versionen einer Konfiguration zurückgreifen zu können.
Damit ein Pipelinedurchlauf gestartet werden kann, muss zuvor anhand der hinterlegten Konfiguration eine lauffähige Pipeline erstellt werden. Diese Pipeline muss die Funktionalität bereitstellen, dass sie nach dem initialen Durchlauf, welcher durch einen Trigger (z.B. einen Gerrit-Trigger) angestoßen wurde, anschließend ohne Trigger manuell neu gestartet werden kann. Dies ist notwendig, wenn ein Pipelinedurchlauf aufgrund von Ereignissen außerhalb der Infrastruktur scheitert. Die Pipeline sollte ab jedem Pipelineschritt, der bereits durchlaufen wurde, neu gestartet werden können.

Zudem existiert die Anforderung, dass das entwickelte System im Hinblick auf leichte Verständlichkeit konstruiert wird, damit spätere Änderungen mit möglichst wenig Aufwand vorzunehmen sind.

2.2 Verständlichkeit

Damit es nach der Implementierung des Systems oder des Teilsystems möglich ist zu reflektieren, ob das System verständlich ist, wird dieser Begriff im Kontext dieser Arbeit genauer definiert und die Faktoren, welche die Verständlichkeit verbessern, herausgearbeitet.

Die Anforderung der Verständlichkeit sollte auf eine bestimmte Zielgruppe eingegrenzt werden, da somit ein bestimmtes Vorwissen angenommen werden kann. Eine sinnvolle Zielgruppe ist in diesem Fall die Menge der SAP HANA Development-Infrastruktur-Entwickler, weil sie mit den Aufgaben der Infrastruktur vertraut sind. Zudem gehören sie zu den Personen, welche bei der produktiven Nutzung des Systems für die Wartung und Erweiterung des Systems zuständig wären.

2.2.1 Faktoren zur Verbesserung der Verständlichkeit

Essentiell für die Verständlichkeit ist, dass eine Sprache verwendet wird, die von allen Entwicklern gesprochen wird. Innerhalb aller systemrelevanten Dokumente sollte daher die englische Sprache Verwendung finden, welche auch in der unternehmensinternen Kommunikation verwendet wird. Ein wichtiger Faktor zum Verständnis des Gesamtsystems ist eine differenzierte Benennung, welche über alle Architekturschichten und Abstraktionsebenen konsistent ist. Diese wird gefordert, um Mehrdeutigkeiten zu vermeiden. Ferner wäre es von Vorteil, wenn in den Entwürfen des Systems bekannte Architektur- und Entwurfsmuster verwendet werden. Die Verwendung von Mustern ermöglicht es den Entwicklern, auf bereits vorhandenes Wissen zurückzugreifen und auf der Ebene der Muster zu kommunizieren.[1, S. 666]
Damit es möglich ist, das System durch die Analyse einzelner Teile zu verstehen, ist eine starke Kohäsion und eine geringe Kopplung anzustreben. Die starke Kohäsion sorgt dafür, dass Nebeneffekte minimiert werden und somit klar erkennbar ist, welche Funktionalität eine Klasse oder Methode besitzt[2, S. 27-28]. Die geringe Kopplung ermöglicht es, eine Klasse unabhängig von den anderen Klassen des Systems zu analysieren.
Kohäsion und Kopplung lassen sich jedoch nicht eindeutig quantifizieren. Daher wird die Wartbarkeit bei der Bewertung der Verständlichkeit anhand von Standards, wie der Einhaltung der Namenskonvention bewertet. Zudem wird versucht die Komplexität mittels Metriken zu quantifizieren. Eine weitere Möglichkeit ist es anhand eines Praxistests zu ermitteln, wie verständlich das System für Entwickler ist.

Zusätzlich sollten Tests implementiert werden, da diese die Wartbarkeit erhöhen und exemplarisch das Verhalten und die Funktionalität eines (Teil-) Systems darstellen und dadurch das Verständnis für das System erleichtern.[3, S. 175]

In dem nächsten Kapitel wird dargelegt, welche Form ein System hat, dass die in diesem Kapitel beschriebenen Anforderungen erfüllt.

3 Entwurf des neuen Systems

In dem letzten Kapitel wurden die Anforderungen an die neue Infrastruktur formuliert. Dieses Kapitel beschreibt die Architektur eines Systems, welches diese Anforderungen erfüllt. Zudem wird dargestellt, wie die Arbeitsweise des System ist.

3.1 Struktur der Pipeline

Ein essentieller Unterschied bei der Betrachtung des zu modellierenden Systems im Vergleich zu dem aktuellen System ist, dass die ausführbare Ausprägung von der Konfiguration getrennt wird. Im aktuellen System ist die Konfiguration der Pipeline[4] nur in den Jobkonfigurationen des Jenkins enthalten und somit ist die Pipelinekonfiguration[4] und die ausführbare Ausprägung dessen eine Einheit. In dem neuen System ist die Pipelinekonfiguration losgelöst von ihrer Ausprägung. Die Pipelineschrittkonfiguration[4] ist dabei eine Teilmenge der Pipelinekonfiguration. Die Ausprägung einer Pipelinekonfiguration (bzw. Pipelineschrittkonfiguration) wird im Folgenden Pipeline (bzw. Pipelineschritt[4]) genannt. Eine solche Pipeline (bzw. ein Pipelineschritt) kann ausgeführt werden, wodurch ein Pipelinedurchlauf[4] (bzw. ein Pipelineschrittdurchlauf[4]) startet.

Ein Pipelineschritt besitzt immer einen Status. Dieser Status gibt an, ob ein Schritt vor der Ausführung steht oder ausgeführt wird. Zudem wird über den Status angegeben, wie ein Schrittdurchlauf nach seiner Beendigung bewertet wird. Ein Schritt wird als erfolgreich bewertet, wenn während der Ausführung keine Probleme[5] aufgetreten sind. Sollte ein Versagen der Infrastruktur der Auslöser für ein Problem sein, so wird dies entsprechend im Status dargestellt. Sind jedoch Probleme während der Ausführung eines Schrittes aufgetreten, welche unabhängig von der Infrastruktur aufgetreten sind, so wird der Schritt als fehlerhaft bewertet. Ein Status der angibt, wie ein Schritt terminierte, wird im Folgenden Resultat genannt.

Somit entspricht in der neuen Infrastruktur ein Jenkins-Job einem Pipelineschritt.

3.1.1 Abarbeitungsweise der neuen Pipelinestruktur

In der neuen Infrastruktur wird die Reihenfolge der Ausführung von Pipelineschritten dynamisch anhand einer Menge von Vorbedingungen ermittelt. Eine Vorbedingung meint in diesem Kontext, dass ein Pipelineschritt voraussetzt, dass ein anderer Schritt des selben Pipelinedurchlaufs mit einem spezifizierten Resultat durchlaufen wurde. Alternativ hätte eine Relation zwischen zwei Pipelineschritten durch die Angabe eines Nachfolgers unter der Bedingung eines bestimmten Resultats dargestellt werden können. Es ist durch die Angabe zwar möglich, dass ein Schritt n ($n \in \mathbb{N}$) Nachfolgeschritte startet, jedoch ist es nicht möglich abzubilden, dass ein Schritt nach der

[4] Erklärung im Glossar

[5] Ein Problem meint ein unerwartetes Ereignis.
Beispiel: Ein ausgeführter Prozess terminiert mit einem unerwarteten Rückgabewert.

Vollendung von m ($m \in \mathbb{N}$) Schritten startet. Daher wurde die Angabe einer Vorbedingung bevorzugt.

3.1.1.1 Abarbeitungsweise der Pipeline im Jenkins

Wird die durch Vorbedingungen definierte Struktur der Pipeline mit Hilfe des Jenkins-Servers umgesetzt, so ergibt sich die folgende Abarbeitungsweise der Pipeline:
In dem Jenkins wartet wie in der aktuellen Infrastruktur ein Job auf ein Trigger des Gerrits. Sobald eine Änderung in dem spezifizierten Projekt, in einem angegebenen Branch eingegangen ist, ruft der Job ein Skript. Dieses Skript ermittelt alle Pipelinekonfigurationen, welche für das Repository und den Branch spezifiziert wurden. Anschließend werden alle Pipelinedurchläufe und die ersten Pipelineschritte initial mit den Triggerinformationen in dem Datenhaltungssystem angelegt und die ersten Jobs werden gestartet. Die ersten Jobs sind die Pipelineschritte, welche eine leere Menge von Vorbedingungen besitzen.

Jeder Job wird dementsprechend, sobald er ausgeführt wird, ein Skript nutzen, um den eigenen Status in der Datenhaltung entsprechend zu aktualisieren. Direkt vor der Beendigung des Jobs wird ein Skript ausgeführt, welches den Schrittstatus des aufrufenden Jobs in dem Datenhaltungssystem aktualisiert und die nächsten Pipelineschritte startet. Die nächsten Pipelineschritte sind dabei die Schritte, bei denen nach der Vollendung des Jobs, welcher das Skript aufruft, alle Vorbedingungen erfüllt sind.

Die Abarbeitung eines Pipelinedurchlaufs, welcher aufgrund eines Continuous-Builds gestartet wurde, verläuft analog zu der Abarbeitung einer durch einen Gerrit-Trigger ausgelösten Durchlauf.

Bevor diese Abarbeitungsweise genutzt werden kann, müssen zuvor die entsprechenden Jenkins-Jobs erstellt werden. Solch ein Job wird anhand einer XML-Konfiguration erstellt. Diese Konfiguration muss daher aus der Pipelinekonfiguration generiert und an den Server übermittelt werden.

3.2 Architektur des Systems

Durch die Anforderung, Informationen persistent zu speichern, wird ein System zur Datenhaltung benötigt. Daher muss die Architektur sowohl die Verwaltung des Datenhaltungssystems als auch die Verwaltung des Jenkins kapseln und die Kommunikation zwischen den Systemen ermöglichen.
Für die Verwaltung der Pipelinekonfigurationen wird ein Konfigurations-Manager benötigt, welcher die Möglichkeit bereitstellt, existierende Konfigurationen zu verändern und neue Konfigurationen anzulegen.

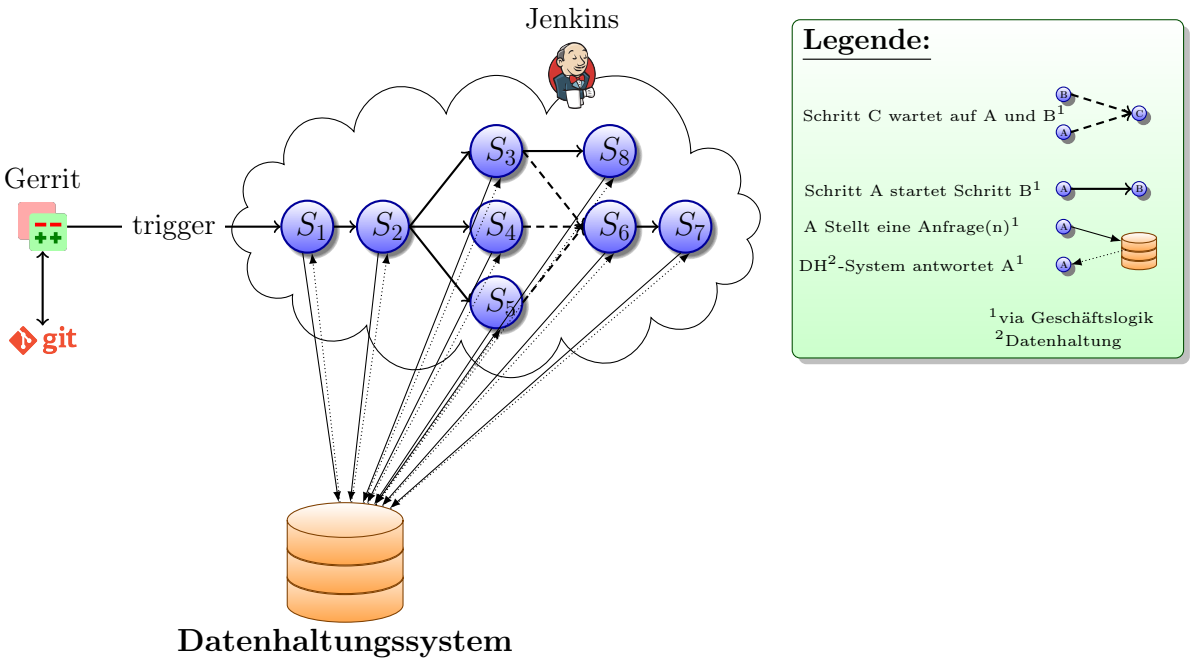

Abbildung 2: Visualisierung der neuen Abarbeitungsweise

Der Jenkins besitzt in diesem System potentiell die Rolle des Servers und des Client. Über die Geschäftslogik[6] werden Operationen und Anfragen an den Jenkins abgesetzt. Somit besitzt der Jenkins in diesem Fall die Rolle des Servers. Andererseits kann ein Jenkins-Job die Rolle eines Client einnehmen, indem er die Geschäftslogik verwendet, um zum Beispiel einen nachfolgenden Schrittdurchlauf zu starten.

Sobald ein Job als Client des Systems fungiert und die Funktionalität zur Erstellung eines neuen Jobs nutzt, werden die entsprechenden Informationen aus dem Datenhaltungssystem bezogen und anhand dieser wird im Jenkins-Server einen neuer Job erstellt.

Das System, welches es zu konstruieren gilt, besteht aus mehreren Systemen (Datenhaltungssystem, Jenkins) und besitzt unterschiedliche Arten von Nutzerschnittstellen (GUI des Konfigurations-Managers, Skripte für den Aufruf durch einen Job). Um die Systeme voneinander und von der Geschäftslogik zu entkoppeln, bietet sich der Einsatz einer hexagonalen Architektur an.

[6]Erklärung im Glossar

3.2.1 Hexagonale Architektur

Die Hexagonale Architektur (auch 'Ports and Adapters') sorgt für die klare Trennung der Geschäftslogik von den anderen Schichten und Systemen. Die Kommunikation findet nur über Ports statt, welche durch die Geschäftslogik angeboten und verwendet werden. Ein Port ist dabei eine Schnittstelle, welche durch ein externes System umgesetzt werden muss. Um die Kommunikation zwischen der Schnittstelle des externen Systems und der Schnittstelle des Ports zu ermöglichen, wird das Entwicklungsmuster Adapter [4, S. 139] genutzt. Somit entsteht eine Geschäftslogik, welche in Abhängigkeit zu ihren Ports jedoch nicht in Abhängigkeit der Art der Implementierung der Adapter steht.

Die Ports und Adapter werden in driving-Ports/Adapters (auch primary-Ports/Adapters) und driven-Ports/Adapters (auch secondary-Ports/Adapters) unterteilt. Ein driving-Port/Adapter löst die Ausführung des Systems aus, während ein driven-Port/Adapter nur an der Ausführung und Kommunikation der System beteiligt ist, diese jedoch nicht in Gang setzt. [5]

3.3 Anwendung des Architekturmusters

In den letzten Abschnitten wurde beschrieben, welche Struktur eine Pipeline in dem neuen System hat. Zudem wurde dargelegt, welches Architekturmuster für die Umsetzung des Systems zu empfehlen ist. Dieser Abschnitt stellt dar, wie dieses Architekturmuster für das System angewendet werden kann.

3.3.1 Ports

Zur Anbindung des Konfigurations-Managers wird ein Port bereitgestellt, welcher eine Funktion einfordert, die eine Konfiguration in einer bestimmten Auszeichnungssprache in die interne Repräsentation umwandelt. Zusätzlich wird die inverse Funktion der zuvor beschriebenen Funktion eingefordert. Zudem wird ein Port für die Nutzung eines Datenhaltungssystems benötigt. Dieser verlangt bestimmte Methoden zur Verwaltung der zu speichernden Daten unter bestimmten Atomizitätsanforderungen, welche in der Portdokumentation aufgezeigt werden müssen. Für das Starten von Pipelines und Pipelineschritten wird ein Port angeboten, welcher fordert, dass Methoden existieren, die anhand der internen Repräsentationen der Konfigurationen und Informationen, Pipelineschritte in dem Scheduling-Server (z.B. Jenkins) verwalten.
Die Kommunikation der Systeme über ihre Ports mit der Geschäftslogik werden in Abbildung 3 visualisiert. Die Skripte implementieren keinen Port der Geschäftslogik, da die von ihnen genutzte Geschäftslogik unabhängig von dem Aufrufenden ist.

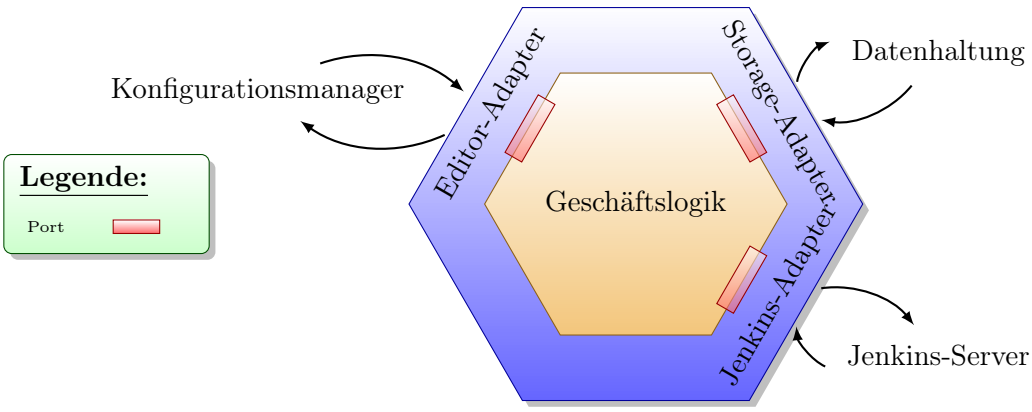

Abbildung 3: Hexagonale Architektur

In diesem Fall ist der Konfigurations-Port der einzige driving-Port (bzw. primary-Port) und die anderen Ports sind die driven-Ports (bzw. secondary-Ports).

3.3.2 Geschäftslogik

Die Geschäftslogik hat in diesem Fall die Aufgabe, Operationen für die Nutzung von internen Repräsentationen bereitzustellen und ähnlich einer Fassade [4, S. 185], die Nutzung der durch die Ports geforderten Methoden zu vereinfachen. Dabei werden Funktionen zum starten, weiterführen und neu starten von Pipelines bereitgestellt.
Im Zuge späterer Erweiterungen ist immer zu beachten, dass generische Logik (d.h. Logik welche unabhängig von dem Adapter ist) nicht in die Implementierung der Adapter fließt.

3.3.2.1 Interne Repräsentation der Konfiguration

In der Geschäftslogik wird statt der Auszeichnungssprache, welche durch die äußeren Systeme genutzt wird eine alternative Darstellung genutzt, die möglichst kompatibel mit den unterschiedlichen Formaten der Pipelinekonfiguration ist.

3.3.2.1.1 Entwurf der Pipeline-Konfiguration

Eine Pipeline-Konfiguration wird durch eine Hashmap repräsentiert. Die Werte sind dabei rekursiv definiert. Jeder Schlüssel liegt dabei in Form einer Zeichenkette vor und der Wert jeder Hashmap ist eine Zeichenkette, null, eine Liste oder eine Hashmap. Ein Objekt in einer Liste kann dabei die gleichen Datentypen, wie ein Hashmapwert besitzen.

Dieses Format findet Verwendung, da es leicht ist es in andere Formate, wie YAML, XML oder JSON umzuwandeln und vice versa[7].

3.3.3 Entwurf der äußeren Systeme

Die Systeme, welche über Adapter in das System integriert werden sollen, müssen zuvor spezifiziert werden. Der Jenkins-Server ist bereits vorhanden und besitzt eine Schnittstelle in Form der Remote Access API, dem Command Line Interface und der dazugehörigen Dokumentation. Somit müssen das Datenhaltungssystem und der Konfigurations-Manager genauer spezifiziert werden.

3.3.3.1 Konfigurationsmanager

Der Konfigurationsmanager bildet für die Entwickler die Schnittstelle zur Definition und Modifikation einer Pipeline. Der Manager muss die in Abschnitt 2.1 definierten Anforderungen an die Konfigurationsadministration erfüllen. Die Modellierung der grafischen Benutzeroberfläche wird in dieser Arbeit nicht behandelt. Im Fokus steht hingegen der Entwurf der Pipelinekonfiguration. Diesem Entwurf entsprechend soll später die interne Repräsentation der Konfiguration, welche die Geschäftslogik verwendet, modelliert werden.

3.3.3.2 Entwurf einer Pipelinekonfiguration

In der aktuellen Infrastruktur werden vorwiegend XML-Dateien für die Konfiguration von Installations- und Testprozessen genutzt. Es wurde jedoch explizit gewünscht, diese Konfiguration in einer Auszeichnungssprache zu verfassen, die leichter zu lesen ist.
Die Sprache Yaml bietet sich für diese Anforderung an, da sie im Hinblick auf diese Eigenschaft entwickelt wurde [6, S. 2].

3.3.3.2.1 Grobstruktur der Konfiguration

Die Konfiguration der Pipeline sollte mit einer Versionsangabe beginnen, um das Verhalten des Systems an die Spezifikationsversion anzupassen.
Der zweite Abschnitt der Konfiguration enthält Informationen, welche die Pipeline genauer spezifizieren.
Der letzte Bereich ist für die Konfiguration der Schritte reserviert.

3.3.3.2.2 Spezifikationsversion

Die Version der Konfigurationsspezifikation wird wie folgt angegeben:

```
1 Version: 1.0
```

[7] Exemplarische Pipelinekonfiguration in der internen Repräsentation vgl. Anhang A.2.1

3.3.3.2.3 Pipeline-spezifische Konfigurationen

Eine Pipeline benötigt einen Namen, welcher im Namensraum aller persistierten Pipelines eindeutig ist. Zudem müssen ein entsprechender Produktname, ein Git-Repository und eine Menge von Branches angegeben werden. Die aktuellen Kunden der Infrastruktur besitzen nur Code einer Programmiersprache in dem Repository ihres Produkts. Daher kann die Quellsprache des Produkts global für die gesamte Pipeline angegeben werden. Die Information ist über das Mapping 'language' anzugeben. Sobald Produkte mit Code in unterschiedlichen Quellsprachen die Pipeline nutzen möchten, muss eine Liste von Sprachen angegeben werden können. Somit hat eine Beispielkonfiguration die folgende Form:

```
Pipeline:
   name: "example-pipeline"
   product:
       name: "example-product"
       language: "java"
       repository: "example.product.project"
       branches:
           - "master"
           - "feature"
```

3.3.3.2.4 Konfigurationen der Pipelineschritte

Die Konfiguration der Pipelineschritte beginnt mit dem Key 'Steps'. Die Konfigurationen der Schritte werden dabei als Liste übergeben. Für jeden Schritt muss ein Name angegeben werden, welcher eindeutig innerhalb des Namensraums der Pipelineschritte dieser Pipelinekonfiguration ist. Die wichtigsten Teilkonfigurationen werden an der folgenden Beispielkonfiguration erläutert:

```
Steps:
    - name: 'Build'
      pre: ~
      build:
          type: maven
      post:
          gerrit:
              voting:
                  verified:
                      failure:
                          value: -1
                          text: "build of example failed"
```

```
23                      success:
24                          value: +1
25                          text: "successfully built product"
26
27      - name: 'Test'
28        preconditions:
29            - result: SUCCESS
30              stepname: 'Build'
31        test:
32            testtype: JUnit
33        post:
34            conti:
35                mail:
36                  addresses:
37                      - "example1@mail.de"
38                      - "example2@mail.de"
39                  failure:
40                      text: "test of example failed"
41                  success:
42                      text: "successfully built example"
```

preconditions-Konfiguration
Ein Schritt gibt immer an, welche Pipelineschritte er mit welchem Endresultat als Startvoraussetzung hat. Diese Angaben, werden unter 'preconditions' gemacht. In diesem Beispiel setzt 'Build' keinen Schritt voraus, da er der erste Pipelineschritt ist. Der Schritt 'Test' soll hingegen erst gestartet werden, wenn der 'Build'-Schritt erfolgreich durchlaufen ist. Die möglichen Resultate entsprechen in dieser Konfigurationsversion den möglichen Jenkins-Ergebnissen.

Aktionskonfiguration
Jede Schrittkonfiguration sollte genau eine Aktionskonfiguration besitzen. Eine Aktionskonfiguration beginnt mit einem Key, wie 'test', 'build', 'integrationtest' oder 'deploy' und einer Menge von zugehörigen Informationen, welche Details zur Ausführung der jeweiligen Aktion enthalten.[8]

pre- und post-Konfiguration
Ein Schritt kann eine post- (bzw. pre-) Konfiguration besitzen. In dieser Teilkonfiguration wird angegeben, was nach (bzw. vor) der eigentlichen Aktion auszuführen ist. Es kann beispielsweise in der post-Konfiguration angegeben werden, was bei einem Continuous-Build oder einem durch das Gerrit aus-

[8] Genauere Informationen sind in dem Yaml-Schema enthalten vgl. Anhang A.2.2

gelösten Pipelineschrittdurchlauf ausgeführt werden soll. Somit kann konfiguriert werden, dass bei einem Continuous-Build eine Mail entsprechend dem Resultat der Aktion versendet wird. Für den Fall, dass die Pipeline aufgrund eines Gerrit-Triggers gestartet wird, kann konfiguriert werden, dass ein Gerrit-Voting entsprechend dem Resultat der Aktion versendet wird. Dabei kann zwischen den Voting-Varianten 'verified' und 'code-review' unterschieden werden. Eigentlich ist vorgesehen, dass nur der Wert von 'verified' durch einen automatisierten Prozess gesetzt wird und der Wert von 'code-review' nur bei einem manuellen Review verändert werden darf[9]. Allerdings hat es sich in der Praxis der Infrastruktur etabliert, dass dieser Wert von einem künstlichen Nutzer gesetzt wird, welchen der ensprechende Pipelineschritt impersonifiziert um ein code-review-Voting abzugeben. Dementsprechend wird diese Konfigurationsmöglichkeit übernommen. In der pre-Konfiguration kann hingegen festgelegt werden, was in einem Schritt als erstes ausgeführt werden soll. Für einen Continuous-Build kann angegeben werden, dass eine Mail mit einem bestimmten Text versendet wird. Zudem kann konfiguriert werden, ob bei einem Schrittdurchlauf, welcher durch einen Gerrit-Trigger ausgelöst wurde ein Voting im Gerrit abgegeben wird.

3.3.3.3 Anforderungen an das Datenhaltungssystem

In den beschriebenen Anforderungen[10] wird die Persistenz unterschiedlicher in Beziehung stehender Informationen gefordert. Es soll eine Pipelinekonfiguration zusammen mit ihrer Historie gespeichert werden. Zudem müssen die Informationen zu den Durchläufen der Pipelines persistiert werden, wie auch die Resultate und Startinformationen aller Pipelineschrittläufe eines Pipelinedurchlaufs, damit es möglich ist, die Anforderung des erneuten Startens der Pipeline zu erfüllen. Die Startinformationen haben die Form eines geordneten Paars aus Schlüssel und Wert.

3.3.3.3.1 Zu speichernde Informationen der Pipelinedurchläufe

Zu einem Pipelinedurchlauf müssen die Start- und Endzeit (d.h. der Zeitpunkt, zu welchem der letzte Pipelineschritt terminierte) persistiert werden. Zudem müssen Informationen zu dem Grund, aus welchem die Pipeline gestartet wurde, gespeichert werden. Im Gerrit werden Änderungen anhand einer numerischen Change-Id und einer Patchset-Nummer eindeutig identifiziert. Dementsprechend werden diese beiden Werte im Falle eines durch das Gerrit ausgelösten Durchlaufs als Startgrund hinterlegt. Eine gängige Zeichenketten-Repräsentation dieser Werte ist:
`<Change-Id>.<Patchset-Nummer>`.
Wenn der Pipelinedurchlauf im Zuge eines Continuous-Builds gestartet wurde, wird dieser innerhalb des Systems durch eine sogenannte Make-Id iden-

[9] https://git.eclipse.org/r/Documentation/intro-quick.html
[10] Erläuterung der Persistenzanforderungen vgl. Abschnitt 2.1

tifiziert. Diese Id ist in der aktuellen Infrastruktur eine Zeichenketten-Repräsentation der Startzeit der Ausführung des Continuous-Builds auf die Sekunde genau. Es wird jedoch vorausgesetzt, dass diese Id eindeutig ist.[11] Der Make-Id kann jedoch auch jede beliebige eindeutige Form gegeben werden.

Durch die Möglichkeit, dass Pipelineschritte nebenläufig ausgeführt werden können, wird potentiell konkurrierend auf die Daten zugegriffen. Es bietet sich daher an, die Informationen in einer relationalen Datenbank zu verwalten, da diese die ACID-Eigenschaften [7, S. 299] für Transaktionen garantiert und für die Persistenz von in Beziehung stehenden Informationen, wie sie hier vorliegen, konzipiert wurde.

3.3.3.4 Entwurf für eine Relationale Datenbank

Anhand der beschriebenen Relationen zwischen den zu speichernden Informationen, lässt sich ein Entity-Relationship-Diagramm erstellen.

[11] In der Infrastruktur wird der Continuous-Build durch einen Jenkins-Job gestartet. In der Konfiguration des Jenkins ist festgelegt, dass alle Jobs mit einem zeitlichen Unterschied von mindestens zwei Sekunden ihre Ausführung beginnen. Somit können keine doppelten Make-Ids entstehen.

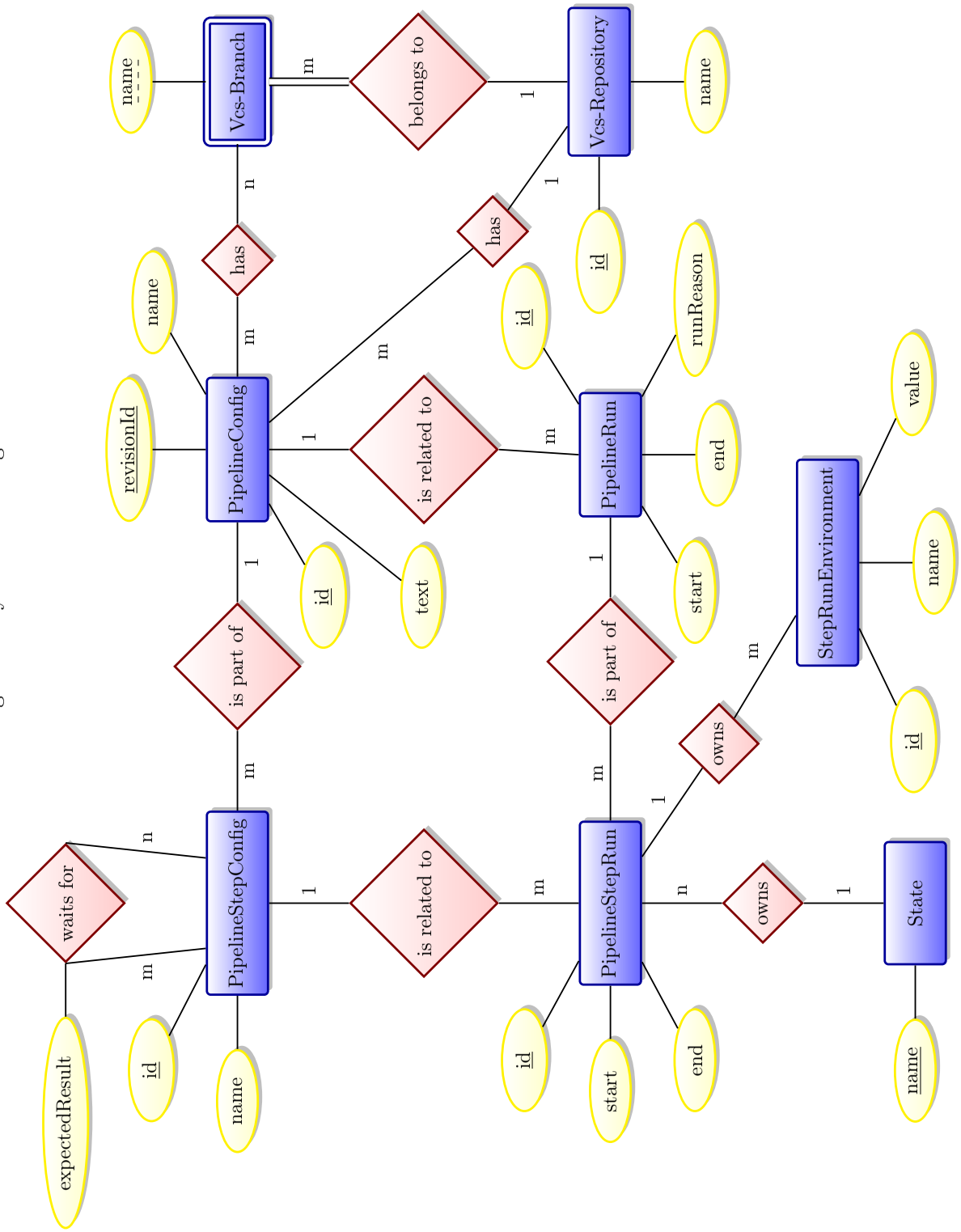

Abbildung 4: Entity-Relation-Diagramm

3.3.3.4.1 Erläuterung des Diagramms

In der Entität 'PipelineConfig', wird die Pipeline-Konfiguration in textueller Form (z.B. JSON) in dem Attribut 'text' gespeichert. Zudem besitzt diese Entität eine 'id' und eine 'revisionId'. Jeder neue Eintrag einer Pipelinekonfiguration erhält eine neue id und eine initiale revisionId. Sobald die Konfiguration geändert wurde, muss ein neuer Eintrag in 'PipelineConfig' erstellt werden. Dieser Eintrag muss die gleiche id, wie der angepasste Eintrag besitzen, jedoch eine neue revisionId erhalten. Somit kann ein Eintrag eindeutig durch id und revisionId bestimmt werden. Zusätzlich ist eine 'PipelineConfig' immer einem Repository (durch die Entität 'Vcs-Repository' abgebildet) und mindestens einem Branch (abgebildet durch die Entität 'Vcs-Branch') des Repositories zugeordnet.

Die Entität PipelineStepConfig speichert hingegen die Konfiguration eines Pipelineschrittes, welche aus der Pipelinekonfiguration generiert wurde. In dem Attribut 'text' wird die textuelle Repräsentation der Konfiguration gespeichert und in dem Attribut 'name' wird der Name des Pipelineschrittes hinterlegt.

Die Relation 'is part of' (zwischen PipelineStepConfig und PipelineConfig) gibt an, welcher Pipelinekonfiguration eine Pipelineschrittkonfiguration zugeordnet ist. Die Abhängigkeiten zwischen den Pipelineschritten wird durch die Relation 'waits for' abgebildet, indem dort für einen Pipelineschritt hinterlegt wird, welche Pipelineschritte er als Voraussetzung besitzt.

Um einen Pipelinedurchlauf abzubilden, wird die Entität 'PipelineRun' verwendet, welche den Start- und Endzeitpunkt ('start' und 'end') in ihren Attributen vorhält. Zudem wird der Grund für den Lauf der Pipeline in 'runReason' gespeichert, d.h. es wird (wie in Paragraph 3.3.3.3.1 beschrieben) die Zeichenketten-Repräsentation persistiert. Die entsprechenden Informationen über den Durchlauf der Pipelineschritte, aus welchem ein Pipelinedurchlauf besteht, werden in 'PipelineStepRun' gespeichert. Ein Eintrag in 'PipelineStepRun' besitzt einen Status (abgebildet in 'State') und eine Menge von Startinformationen (abgebildet durch Einträge in 'StepRunEnvironment').

3.3.3.5 Adapter

In den vorherigen Abschnitten wurde dargelegt, welche Methoden die Ports erwarten, was vornehmlich die Aufgaben der Geschäftslogik sind und wie die äußeren Systeme spezifiziert sind. In den folgenden Abschnitten wird erläutert welche Aufgaben die Adapter erfüllen müssen, damit die Systeme von der Geschäftslogik genutzt werden können.

3.3.3.5.1 Yaml-Konfigurationsadapter

Der Konfigurations-Manager muss den Konfigurations-Port für die Sprache Yaml umsetzen, d.h. es muss eine Methode zur Umwandlung von Yaml in die interne Repräsentation geben und vice versa.

3.3.3.5.2 DB-Adapter

Der Port für die Datenhaltungssysteme muss für die relationale Datenbank umgesetzt werden, so dass die geforderten Methoden die entsprechenden SQL-Aufrufe kapseln und sicherstellen, dass die Atomizitätsanforderungen erfüllt sind.

3.3.3.5.3 Jenkins-Adapter

Der Jenkins-Adapter muss den Port für den Scheduling-Server umsetzen. Somit muss die interne Repräsentation der Pipelinekonfiguration in Jobkonfigurationen überführt werden.

3.3.3.6 Dynamisches Handling von Jenkins-Jobs

Nachdem in den vorherigen Abschnitten beschrieben wurde, wie das Architekturmuster der Hexagonalen Architektur umgesetzt werden kann, werden in den folgenden Abschnitten einige Strategien zur Nutzung des Systems eingeführt.

3.3.3.6.1 Erstellungsstrategie

Ein Job sollte immer unmittelbar vor der ersten Ausführung erstellt werden. Die Alternative ist, alle Schritte einer Pipeline direkt nach der Abgabe der entsprechenden Konfiguration zu erstellen. Dieses Handling benötigt allerdings ebenfalls die Möglichkeit zu prüfen, ob ein Job existiert und einen Job zu erstellen, da die Möglichkeit besteht, dass mit dem Jenkins-Server nicht kommuniziert werden kann[12] und daher der Job nachträglich angelegt werden muss. Es wäre möglich die beiden Strategien zu kombinieren, um Sonderfälle zu behandeln und möglichst viel zusätzliche Laufzeit durch das Erstellen der Jobs einzusparen. Jedoch ist die zusätzliche Zeit in Relation zu der durchschnittlichen Laufzeit eines Pipelineschrittes zu vernachlässigen[13]. Zudem ist es sinnvoll die Strategie möglichst einfach, mit möglichst wenigen Sonderfällen zu gestalten, damit diese leicht verständlich ist.

Für die Betitelungen der Jenkins-Jobs wird eine Konvention benötigt, welche das mehrfache Auftreten des gleichen Namens verhindert. Es bietet sich an,

[12] Dies kann aufgrund von technischem Versagen auftreten.

[13] Im Durchschnitt benötigt ein Job in der Infrastruktur $24,5$ Minuten (vgl. Anhang A.4). Die Erstellung via Remote access API benötigte für eine durchschnittlich große Konfiguration 193ms (vgl. Anhang A.5).

den Namen aus dem Pipelineschrittnamen, der Pipeline-Id und der Pipeline-Revision-Id abzuleiten. Ein Jobname hat somit die folgende Form:
`<Pipelineschrittname>.<Pipeline-Id>.<Pipeline-Revision-Id>`
Dadurch ist sichergestellt, dass der Name in dem Jenkins einmalig ist. Dies kann jedoch nur garantiert werden, wenn die Voraussetzung erfüllt ist, dass nur das in dieser Arbeit beschriebene System den Jenkins-Server nutzt und keine Jobs manuell erstellt werden. Es bietet sich allerdings an den Namen der Pipeline mit in das Namensschema zu integrieren, damit die manuelle Identifikation eines Jobs im Jenkins vereinfacht wird, sollte dies notwendig werden.

3.3.3.6.2 Löschstrategie

Bei einer längeren Laufzeit des Systems, ist es sinnvoll die Jobs, welche nicht mehr benötigt werden zu entfernen, da sonst unnötig viel Speicherplatz auf dem Server verbraucht wird.[14] Neben der Größe der Dateien ist die Anzahl[15] dieser von Bedeutung, da es dazu kommen kann, dass keine INodes mehr in dem System zur Verfügung stehen und somit die Erstellung weiterer Dateien auf dem Dateisystem verhindert wird.

Ein Job sollte entfernt werden, sobald keine nicht vollendeten Pipelinedurchläufe für die entsprechende Piplinekonfiguration existieren und die Pipelinekonfiguration welcher der Pipelineschritt, der durch den Job abgebildet wird zugehörig ist, obsolet ist. Das heißt, eine neue Konfigurationsversion liegt vor.

3.3.3.7 Verhalten im Fehlerfall

Es ist möglich, dass die Datenbank oder der Jenkins-Server zeitweise nicht erreichbar ist, daher wird eine Strategie für das Verhalten in diesem Fall benötigt. Es ist sinnvoll die Strategie durch eine Applikation umzusetzen, welche in regelmäßigen Abständen die Verfügbarkeit der Systeme prüft. Damit dieser Dienst von den Systemen, welche er prüfen soll entkoppelt wird, sollte der Port für das Datenhaltungssystem und der Port für die Anbindung des Scheduling-Servers eine Methode einfordern, welche den Verfügbarkeitsstatus prüft.
Selbiger Dienst kann die Umsetzung des dynamischen Entfernens von Jenkins-Jobs übernehmen.[16]

Sollte die Datenbank nicht erreichbar sein, so laufen alle Jobs weiter, bis

[14] Analyse des durchschnittlichen Speicherbedarfs einer Konfigurationsdatei vgl. Anhang A.6

[15] Neben der Konfigurationsdatei erstellt der Jenkins pro Durchlauf unterschiedliche Protokolldateien.

[16] Dynamisches Handling von Jenkins-Jobs vgl. Paragraph 3.3.3.6

sie sich aufgrund der fehlenden Datenhaltung mit dem Status `FAILURE` beenden. Sobald die Datenhaltung wieder erreichbar ist, werden alle Schritte ermittelt, welche den Status `QUEUED` oder `RUNNING` in der Datenhaltung besitzen. Diese Schritte werden anschließend neu gestartet.

In dem Fall, dass der Jenkins-Server nicht erreichbar ist, werden alle Schritte, die in diesem Moment laufen, in der Datenbank mit `IERROR` markiert, und Sobald der Server wieder erreichbar ist, werden alle Jobs welche in der Datenbank mit `IERROR` markiert sind, neu gestartet.

Das nächste Kapitel beschreibt, wie die in diesem Kapitel beschriebene Architektur des System und seine Abarbeitungsweise implementiert worden ist.

4 Implementierung des Systems

In dem letzten Kapitel wurde ein System modelliert, welches es ermöglicht eine Pipeline flexibel zu konfigurieren und auszuführen. Im Folgenden wird erläutert, wie dieses System implementiert werden kann. Durch diese Implementierung wird gezeigt, dass das zuvor abstrakt beschriebene System umgesetzt und dementsprechend in der Praxis eingesetzt werden kann.

Aufgrund der hexagonalen Architektur ist es möglich die einzelnen Bestandteile des Systems separat zu implementieren und zu testen. Die Voraussetzungen dafür sind, dass die interne Repräsentation der Daten, welche zwischen Geschäftslogik und Ports ausgetauscht werden existieren und die Ports bereits implementiert sind. Die Implementierung der Ports wird benötigt, da es dadurch möglich ist, dass entweder die Geschäftslogik oder die Adapter vorerst durch Mocking umgesetzt werden. Somit muss als erstes die interne Repräsentation implementiert werden und anschließend die Ports.

4.1 Standards bei der Implementierung

Bei der Implementierung des Systems wurde die Sprache Groovy verwendet, es wurde jedoch bewusst darauf verzichtet bestimmte Sprach-Funktionalität zugunsten der Verständlichkeit in der Implementierung zu nutzen. Auf folgende Funktionalität wurde verzichtet: Die Verwendung von anonymen Funktionen und impliziten Return-Statements, um den gewohnten Lesefluss zu erhalten. Zudem wurde auf die dynamische Typisierung verzichtet, damit die Relationen zwischen den Klassen und die Verwendung der eigenen Klassen deutlich wird.

Die verwendete Formatierung des Codes entspricht den Standards des Infrastruktur-Teams. Zudem wurden diverse andere Standards des Teams eingehalten.[17] Diese wurden eingehalten, um eine mögliche Übernahme des Codes durch das Infrastruktur-Team zu vereinfachen. Zudem wurde zur statischen Codeanalyse die Analyseplattform SonarQube[18], mit dem Groovy-Plugin[19] genutzt. Dieses Plugin integriert das statische Codeanalyse-Werkzeug CodeNarc.[18] Für dieses Werkzeug kann eine bestimmte Regelmenge definiert werden, welche Regeln mit den Prioritäten 1,2 oder 3 angibt, die auf den Code angewendet werden sollen. Die Implementierung sollte keine Regelverletzungen in der SonarQube-Standardregelmenge vorweisen, damit sichergestellt ist, dass der Code gegen keine allgemeinen Standards[20] verstößt und

[17] Ein Beispiel für ein Teamstandard ist, dass alle Klassennamen in Import-Anweisungen mit voll qualifizierten Name angegeben werden müssen.

[18] Erklärung im Glossar

[19] http://docs.sonarqube.org/display/SONAR/Groovy+Plugin

[20] Ein Beispiel für solch einen Standard ist, dass niemals die Exception 'Exception' geschmissen werden soll.

Defekte, welche aufgrund der dynamischen Eigenschaften von Groovy sonst erst zur Laufzeit auffallen, minimiert werden.

4.2 Implementierung der internen Repräsentationen

Die interne Repräsentation der Konfiguration wird entsprechend der Ausführung in Paragraph 3.3.2.1.1 umgesetzt.[21] Damit sichergestellt ist, dass diese die erwartete Form besitzt wird sie mittels eines Schemas validiert. Es existiert bisher keine Möglichkeit eine Mapstruktur in Groovy mit einem Schema zu validieren, daher wird in der aktuellen Implementierung der Umweg über das Format Yaml genommen, für welches ein Schema für das Validierungsframework Kwalify angefertigt wurde.[22]

Ferner existieren weitere Informationen, welche zwischen den Ports und der Geschäftslogik kommuniziert werden. Es wird eine Repräsentation für den Status eines Schrittlaufs, sowie die Informationen zu einem Versionsverwaltungssystem und die Informationen eines Schrittdurchlaufs benötigt. Zudem werden Strukturen benötigt, welche die Informationen der Pipelineschritte und ihre Beziehungen untereinander darstellen und Informationen eines Triggers vorhalten. Zusätzlich wird eine Repräsentation von Identifikatoren der Datenhaltung benötigt. Dabei werden Identifikatoren für die (Schritt-) Konfigurationen und (Schritt-) Durchläufe benötigt.

4.2.1 Status eines Schrittlaufs

Der Status wird unter anderem für die Statusabfrage eines Schrittes und für die Ermittlung, ob alle Vorbedingungen eines Schrittes erfüllt sind benötigt. Das Resultat eines Schrittes (hier wird dieses durch den Jenkins bestimmt) ist dabei immer eine Teilmenge der Menge der Status.
Die Menge der Status wird durch eine Enumeration implementiert.[23] Die Semantik eines Status' wird in Tabelle 1 erläutert.

`QUEUED`	Der Schritt befindet sich noch in der Queue des Schedulers
`RUNNING`	Der Schritt ist aktuell in der Ausführung
`SUCCESS`	Der Schritt ist mit einem positiven Resultat terminiert
`FAILURE`	Der Schritt ist mit einem negativen Resultat terminiert
`UNKNOWN`	Der Status des Schrittes ist unbekannt (z.B. durch Probleme bei der Kommunikation mit dem Jenkins)
`IERROR`	Der Status signalisiert, dass der Schritt aufgrund der Infrastruktur nicht positiv terminieren konnte

Tabelle 1: Semantik der Schrittstatus

[21] Implementiert in `pipeline.internal.representation.config.Config`
[22] YAML-Schema vgl. Anhang A.2.2
[23] Implementiert in `pipeline.internal.representation.StepState`

4.2.2 Informationen eines Versionsverwaltungssystems

Damit ein Schritt die Sourcen eines Produkts von dem Versionsverwaltungssystem beziehen kann, ist es notwendig, dass das Repository und der Branch des Produkts bekannt sind. Somit wird dies durch eine Klasse realisiert, welche einen Repository-Namen als Zeichenketten-Attribut vorhält und eine Menge von Branches als Zeichenkette enthält.[24]

4.2.3 Struktur zur Darstellung der Relationen zwischen den Pipelineschritten

Für die Darstellung der Relationen der Pipelineschritte wurden zwei unterschiedliche Strukturen implementiert. Zentral für beide Darstellungsformen ist allerdings eine Klasse, welche die Informationen, die global für die gesamte Pipeline gelten müssen beinhaltet und die Möglichkeit bereitstellt aus der internen Repräsentation der Pipelinekonfiguration die beiden Strukturen zu generieren[25].

Zum einen können die Relationen zwischen den Schritten, wie in der Konfiguration durch eine Liste von Vorbedingungen pro Schritt dargestellt werden.[26]

Zum anderen ist es möglich die Pipelineschritte als Graph darzustellen, mit Schritten als Knoten und Relationen als Kanten.[27] Dieser Graph wird dazu genutzt, um zu ermitteln, ob die Pipeline fehlerhaft konfiguriert wurde und dadurch Zyklen entstanden sind, welche bei der Ausführung zu einer Endlosschleife führen und somit zu dem unnötigen Verbrauch von Ressourcen führen würden. Zudem kann diese Repräsentation von dem Konfigurationseditor später zur Visualisierung der Pipeline genutzt werden.

In einer Pipeline, welche als Graph dargestellt wird, sind die ersten Pipelineschritte die Schritte mit einem Ausgrad[28] von null. Damit es jedoch möglich ist jeden Knoten des Graphen ausgehend von den ersten Pipelineschritten/Knoten zu erreichen, wird zusätzlich zu der Kante, welche die Relation angibt eine Hilfskante gespeichert, welche die invertierte Richtung der Relationskante besitzt und somit auf den Nachfolger zeigt. Der Test auf Zyklenfreiheit wurde durch eine modifizierte Tiefensuche unter Nutzung der Nachfolgerkante umgesetzt. Somit wurde die Zyklenfreiheit für den Graph mit invertierten Kanten gezeigt. Ein Zyklus ist definiert als ein Weg in einem Graph, der Länge größer gleich zwei, bei dem Start und Endpunkt gleich sind. Somit wurde in diesem Test der Weg invertiert, wodurch lediglich der

[24] Implementiert in `pipeline.internal.representation.VcsInformation`
[25] Implementiert in `pipeline.internal.representation.config.PipelineConfig`
[26] Implementiert in `pipeline.internal.representation.config.PreconditionStepConfig`
[27] Implementiert in `pipeline.internal.representation.config.graph.PipelineConfigGraph`
[28] Erklärung im Glossar

End- und der Startpunkt vertauscht wurden. Somit ist der Graph zyklenfrei genau dann, wenn der Graph mit invertierten Kantenrichtungen zyklenfrei ist. Diese Art des Tests auf Zyklenfreiheit ist daher korrekt.

4.2.4 Informationen eines Triggers

Der Grund für den Durchlauf einer Pipeline wird durch Trigger-Informationen identifiziert.[29] Damit es möglich ist diese Informationen, in der Datenhaltung zu persistieren, ohne dass die Geschäftslogik oder das Datenhaltungssystem Annahmen zu dem Inhalt der Informationen treffen müssen, was zu Abhängigkeiten von konkreten Adaptern führen würde, wird der Grund für die Durchführung nur als Zeichenkette repräsentiert und kann daher in dem Datenhaltungssystem hinterlegt werden. Ein Trigger-Objekt enthält zudem immer Informationen zu dem entsprechenden Versionsverwaltungssystem, um anzugeben für welche Codebasis der Durchlauf vollzogen werden muss. Zudem muss ein Trigger die Funktionalität bereitstellen, um aus sich selbst Startinformationen zu generieren, d.h. einen Trigger in eine Hashmap-Form zu überführen.

4.2.5 Identifikatoren für Konfigurationen und Durchläufe

Es musste die Möglichkeit bestehen, dass anhand eines Identifikators eine Pipelinekonfiguration, eine Pipelineschrittkonfiguration, ein Pipelinedurchlauf und ein Pipelineschrittdurchlauf eindeutig identifiziert werden kann.[30] Damit es möglich ist, dass ein Pipelineschritt seine eigene Identität und die Identität des Pipelinedurchlaufs während des Durchlaufs vorhalten kann und anhand dieser Identifikatoren die nächsten Schritte der Pipeline starten oder seine Startinformationen verwalten kann, mussten diese Identifikatoren in eine abstrakte Form überführt werden, dazu wurde die Form einer Hashmap gewählt.[31] Dabei sind der Schlüssel und der Wert Zeichenketten. Diese Informationen werden einem Schritt direkt zum Start mitgegeben. Zudem ist es notwendig, dass ein Identifikator in eine Zeichenketten-Repräsentation überführt werden kann, welche dazu genutzt werden kann innerhalb des Schedulers einen garantiert eindeutigen rekonstruierbaren Schrittnamen zu erstellen.[32]

[29] Implementiert in `pipeline.internal.representation.trigger.Trigger`

[30] Implementiert durch die Interfaces der Form: `pipeline.ports.storage.identify.I*Identifier`

[31] Implementiert in `pipeline.ports.storage.identify.IdentifierEnvironment`

[32] exemplarisch dargestellt in 3.3.3.6.1

4.3 Implementierung der Ports

Die einzelnen Ports konnten jeweils durch ein Interface umgesetzt werden.[33] Für diese zentralen Elemente des Systems ist es sinnvoll als Dokumentation einen Schnittstellen-Kontrakt zu verfassen, welcher die Semantik, der durch die Ports geforderten Methoden erläutert. Durch diese Dokumentation soll sichergestellt werden, dass bei der Neuimplementierung eindeutig ist, welches Verhalten des Ports die Geschäftslogik annimmt. Der Schnittstellen-Kontrakt wurde in Form einer Groovy-Doc-Dokumentation verfasst.

4.4 Implementierung der Geschäftslogik

Die Geschäftslogik wurde in drei Klassen unterteilt, damit es möglich ist, dass ein Nutzer ohne alle Adapterinformationen zu besitzen, einen Teil der Geschäftslogik nutzen kann. Die erste Klasse benötigt nur den Datenhaltungs-Adapter und stellt die Funktionalität bereit, um Pipelinekonfigurationen und Historien von Konfigurationen aus dem Datenhaltungssystem zu beziehen.[34] Die Funktionalität der ersten Klasse wird von der zweiten Klasse um Funktionen zum Speichern von Konfigurationen ergänzt, wofür der Konfigurations-Adapter benötigt wird, da dieser ermöglicht die zuspeichernde Konfiguration von ihrer Stringrepräsentation in die interne Repräsentation zu wandeln.[35] Die Geschäftslogik stellt in diesen beiden Fällen lediglich eine Fassade für die Nutzung der Ports dar. Die dritte Klasse ermöglicht es unter Zuhilfenahme des Scheduler-Ports und des Datenhaltungs-Ports Pipelines zu starten, eine laufende Pipeline weiterzuführen, d.h. dass ein Schritt diese Funktion nutzen kann, um die nächsten Schritte der Pipeline aufzurufen, von welcher er ein Teil ist.[36] Zudem kann ein Nutzer mithilfe dieser Klasse den Schritt einer Pipeline neu starten.

4.4.1 Starten von Pipelines

Der Funktion zum Starten von Pipelines werden Startinformationen mitgegeben, welche die Informationen enthalten müssen, die notwendig sind, damit durch die Geschäftslogik die Trigger erstellt werden können. Als erstes wird der entsprechende Trigger erstellt. Anhand dieses Triggers werden alle Identifikatoren der Pipelinekonfigurationen ermittelt, welche Interesse an dem Trigger haben, d.h. dass der Trigger für ein Repository und einen Branch gilt, für die die Pipeline konfiguriert wurde. Für jede ermittelte Konfiguration wird nun anhand des Identifikators ein Pipelinedurchlauf gestartet.

[33] Die Ports wurden in den folgenden Klassen implementiert:
Datenhaltungs-Port in `pipeline.ports.StoragePort`
Scheduler-Port in `pipeline.ports.StepSchedulerPort`
Konfigurations-Port in `pipeline.ports.ConfigPort`

[34] Implementiert in `pipeline.PipelineConfigAccess`

[35] Implementiert in `pipeline.PipelineConfigManager`

[36] Implementiert in `pipeline.PipelineRunManager`

Das Starten einer Pipeline beginnt damit, dass für den Identifikator der Pipelinekonfiguration die in Abschnitt 4.2.5 beschrieben Struktur (im Folgenden Identifikatorenumgebung genannt) erstellt wird. Der Pipelinedurchlauf muss anschließend in dem Datenhaltungssystem initialisiert werden, dabei wird der Identifikator des Pipelinedurchlaufs zurückgegeben. Dieser Identifikator wird der Identifikatorenumgebung hinzugefügt. Für die Pipelinekonfiguration müssen nun die Identifikatoren der initialen Pipelineschrittkonfigurationen über die Datenhaltung ermittelt werden. Jeder dieser Schritte muss nun gestartet werden.

Wie bereits bei dem Start der Pipeline, muss der Pipelineschrittdurchlauf in der Datenhaltung initialisiert werden, sollte der Schritt bereits initialisiert sein, so wird ein leerer Identifikator zurückgegeben. Wenn der Identifikator des Schrittdurchlaufs leer ist, dann soll nichts gestartet werden und die Methode bricht ab, sonst werden der Identifikator des Schrittdurchlaufs und der Identifikator der Schrittkonfiguration zu der Identifikatorenumgebung hinzugefügt. Damit es möglich ist den Pipelineschritt eindeutig innerhalb des Schedulers (hier Jenkins) zu identifizieren, wird mithilfe des Datenbank-Ports ein eindeutiger Name, wie in Paragraph 3.3.3.6.1 beschrieben anhand der Identifikatoren erstellt. Anhand dieses Namens wird der Schritt im Schedulersystem erstellt, falls er noch nicht existiert. Anschließend wird der Schritt durch den Scheduler-Port in dem Scheduler in der Warteschlange mit der entsprechenden Identifikatorenumgebung eingereiht.

4.4.2 Weiterführen einer Pipeline

Damit eine Pipeline weitergeführt wird, muss der Funktion die Identifikatorenumgebung des aufrufenden Schrittes und das Resultat des eigentlichen Schrittes übergeben werden. Der Pipelinedurchlauf des rufenden Schrittes muss als erstes in der Datenhaltung um das Resultat und die Endzeit ergänzt werden. Anschließend müssen alle Pipelineschritte der Pipelineschrittkonfiguration ermittelt werden, bei denen alle Vorbedingungen erfüllt sind, die jedoch noch nicht in der Datenhaltung als initialisiert aufgeführt werden.

Dazu werden alle noch nicht initialisierten Schritte zusammen mit ihren Vorbedingungen und alle initialisierten Schritte zusammen mit ihrem aktuellen Zustand aus der Datenhaltung bezogen. Anschließend wird der aktuelle Zustand, mit dem erwarteten Zustand verglichen und anhand des Ergebnisses bewertet, ob eine Vorbedingung erfüllt ist. Für jeden Schritt wird nun eine Identifikatorenumgebung erstellt, mit den Pipelinekonfigurations und Pipelinedurchlaufidentifikatoren der initial übergebenen Umgebung, da diese für jeden Pipelineschrittdurchlauf gleich sind. Alle Schritte, bei denen die Vorbedingungen erfüllt sind werden nun, wie in Abschnitt 4.4.1 mit ihrer Identifi-

katorenumgebung gestartet. Schritte, welche bereits gestartet wurden, jedoch in der Menge der zu startenden Schritte sind, sollten kein Problem darstellen, da diese nicht gestartet werden, sollten sie bereits initialisiert worden sein.

4.4.3 Neustarten eines Pipelineschrittes

Um einen Pipelineschritt erneut zu starten, muss der Name des Schrittes bekannt sein, sowie der Identifikator des Pipelinedurchlaufs. Anhand des Identifikators des Pipelinedurchlaufs wird der Identifikator für die Pipelinekonfiguration ermittelt. Anhand dieser Identifikatoren wird die Identifikatorenumgebung des Schrittes erstellt. Zudem muss anhand des Identifikators der Konfiguration und des Schrittnamens der Identifikator für den Pipelineschrittkonfiguration ermittelt werden. Anhand des Pipelinekonfigurationsidentifikators kann nun die Pipelinekonfiguration aus der Datenhaltung bezogen werden. Diese wird in die in Abschnitt 4.2.3 beschrieben Graphform umgewandelt. Anhand dieser Form werden alle transitiven Nachfolger des Schrittes, welchen es neu zu starten gilt ermittelt. Die transitiven Nachfolger eines Schrittes sind die Menge der Schrittknoten, welche ausgehend von dem Schritt über einen beliebig langen Weg nur unter Nutzung der Nachfolgerkante erreicht werden können. Anschließend werden die Durchlaufinformationen zu dem startenden Schritt und aller transitiven Nachfolgeschritte aus der Datenhaltung entfernt. Zuletzt wird der Pipelineschritt mit der Identifikatorenumgebung wie in Abschnitt 4.4.1 beschrieben gestartet.

4.5 Implementierung der Adapter

In den letzten Abschnitten wurde erläutert, wie die interne Repräsentation, die Ports und die Geschäftslogik implementiert wurden. In den folgenden Abschnitten wird beschrieben, wie die Forderungen der Ports mit den Schnittstellen der äußeren Systeme erfüllt wurden.

4.5.1 Implementierung des Jenkins-Adapters

Der Jenkins-Adapter, welcher den Scheduler-Port implementiert, bildet eine Fassade für die Schnittstelle zur Ansteuerung des Jenkins-Servers. Jenkins bietet zur externen Steuerung seines Systems die Remote-Access API und ein Command-Line-Interface (Jenkins-CLI) an.[37] Die Remote-Access API wird ähnlich einer REST-API durch HTTP-Anfragen bedient. Das Command-Line-Interface liegt als JAR-Datei vor. Zu Beginn wurde versucht die Klassen, welche in der JAR vorliegen direkt zur Ansteuerung des Jenkins zu nutzen, allerdings hat sich herausgestellt, dass dies nicht von den Entwicklern des Command-Line-Interface vorgesehen ist. Um sichergehen zu können,

[37] Implementiert in `pipeline.adapters.jenkins.JenkinsAdapter`

dass die Implementierung betriebssystemunabhängig ist, wurde die Remote-Access API verwendet, da der Aufruf von Kommandozeilenbefehlen nicht in allen Betriebssystemen einheitlich ist.[38]

Die Identifikatorenumgebung wird als Menge von Job-Parametern zum Starten des Jobs verwendet. Dabei ist das Risiko, dass ein Job während seiner Ausführung die Informationen zu seiner Identität verfälscht nicht vorhanden, da der Wert eines Parameters in einem Job nicht überschrieben werden kann. Der wichtigste Bestandteil des Jenkins-Adapters ist die Klasse, welche die Pipelinekonfiguration in eine Jenkins-Job-Konfiguration umwandelt. Die Klasse wurde initial durch Mocking implementiert und später für die Durchführung von realitätsnahen Tests um die notwendige Funktionalität erweitert. Es wurde darauf verzichtet die gesamte Klasse zu implementieren, da es dazu notwendig ist alle Pipelineschritte aller aktuell existierender Pipelines auf ihre Gemeinsamkeiten zu analysieren und anhand dieser Ergebnisse die Umsetzung der abstrakten Konfiguration auf eine Job-Konfiguration zu implementieren. Dieser Mehraufwand ist jedoch nicht notwendig, um eine Aussage darüber treffen zu können, ob der Entwurf des Systems umsetzbar und einsetzbar ist. Sollte das System produktiv genutzt werden, dann müsste diese Klasse so erweitert werden, dass alle Konfigurationsmöglichkeiten der Pipelinekonfiguration auf einen Bestandteil einer Jenkins-Job-Konfiguration abgebildet werden.

4.5.2 Implementierung des Yaml-Adapters

Für die Implementierung des Yaml-Adapters, welcher eine Ausprägung des Konfigurations-Ports darstellt, wurde zur Überführung der Yaml-Konfiguration und der internen Repräsentation ineinander, die SnakeYAML-Bibliothek[39] verwendet.[40]

4.5.3 Implementierung des Datenbank-Adapters

Bei der Wahl des relationalen Datenbankmanagementsystems (DBMS) wurde gegen den unternehmensinternen Vorschlag entschieden, mit SAP HANA ein Produkt des Unternehmens zu verwenden und somit die Möglichkeit zu haben bei Problemen direkt in persönlichen Kontakt mit den Entwicklern treten zu können. Zudem wurde der Vorschlag gemacht eine NoSQL MongoDB zu verwenden, um neuartige Technologie einzusetzen. SAP HANA wurde nicht verwendet, da lediglich die Standardfunktionalität für die Verwaltung von vergleichsweise wenigen Datensätzen benötigt wird und daher die erhöhten Ressourcenkosten[41] nicht zu rechtfertigen sind. Auf die Verwendung

[38] Erläuterung zur Nutzung der Remote-Access API vgl. Anhang A.7
[39] https://code.google.com/p/snakeyaml/
[40] Implementiert in `pipeline.adapters.editor.YamlAdapter`
[41] Eine SAP HANA benötigt mindestens 32GB RAM

von MongoDB wurde verzichtet, da sich die zu persistierenden Daten für die Verwendung einer relationalen Datenbank anbieten[42], um radikales Vorgehen[43] zu minimieren und somit das Risiko zu minimieren. Der Datenbank-Adapter wurde stattdessen für das PostgreSQL-DBMS implementiert, für welches bereits Erfahrungswerte vorhanden waren.[44] Dafür wurde ein Schema verwendet, welches dem ER-Diagramm in Paragraph 3.3.3.4 entspricht.[45]

Die Anpassungen, um den Adapter für ein anderes relationales DBMS abzuändern, sollten sich lediglich auf Anpassungen des Schemas, eventuell geringe Änderungen an den SQL-Anfragen und den Wechsel des JDBC-Treibers beschränken.

4.6 Test der Implementierung

Die Implementierung wurde sowohl durch Unittests, als auch durch Systemtests erfolgreich getestet. Es wurden zwei unterschiedliche Arten von Systemtests zum Testen der Funktionalität zum Pipeline abarbeiten und neu starten durchgeführt.

Zum einen wurden komplexe Pipelinekonstrukte getestet, welche innerhalb ihrer Schritte lediglich eine Standardmeldung ausgegeben und die nächsten Schritte gestartet haben. Ein solches Konstrukt wird in Abbildung 5 dargestellt. Jeder Schritt ruft daher im Wesentlichen nur ein Skript auf, welches den Epilog eines Schrittes ausführt, das heißt die Aktualisierung des Schrittstatus und das Starten der nächsten Schritte.

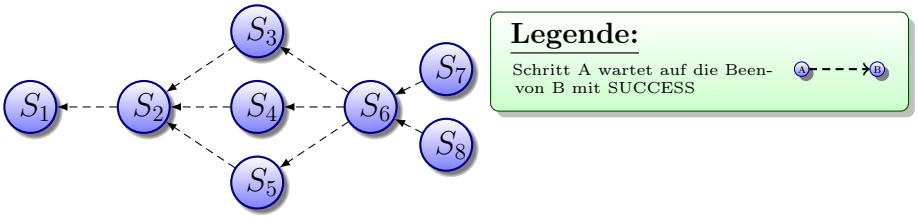

Abbildung 5: Beispiel-Pipelinekonstrukt

Zum anderen wurde eine Pipeline aus dem aktuellen Produktivsystem zur Erstellung des in der Sprache Java geschriebenen Produkts 'hdbstudio' mit Hilfe der Implementierung abgebildet. Diese Pipeline besteht aus einem Erstellungs- und einem Testschritt. Der Erstellungsschritt verwendet Git um die entsprechenden Sourcen zu beziehen und Maven[43], um diese zu bauen.

[42] Beschreibung der zu persistierenden Daten vgl. Abschnitt 3.3.3.3.1
[43] Erklärung im Glossar
[44] Implementiert in `pipeline.adapters.postgredb.DatabaseAdapter`
[45] SQL-Schema vgl. Anhang A.3

Zudem werden unterschiedliche Groovy-Skripte genutzt, um das Erstellungsergebnis zu validieren und es in ein bestimmtes Verzeichnis zu kopieren. Die Angaben, welche Git, Maven und die Groovy-Skripte[46] benötigen wurden in der aktuellen Infrastruktur über Umgebungsvariablen, welche von den vorherigen Schritte geerbt wurden, übermittelt. Da die Schritte in der neuen Infrastruktur voneinander entkoppelt sind und daher keine Umgebungsvariablen vererbt werden können, wurde ein Skript implementiert, welches zu Beginn gerufen wird und alle Startinformationen des Schrittes aus der Datenhaltung bezieht und diese so aufbereitet, dass der Jenkins-Job diese Informationen in seine Umgebung laden kann. Der Testschritt ruft durch ein Groovy-Skript eine Menge von JUnit-Tests auf und verwendet das gleiche Skript wie der Buildschritt, um die Startinformationen aus der Datenhaltung zu beziehen. Zudem verwenden alle Schritte das Skript zum Durchlaufen des Schritt-Epilogs.

Diese Tests haben gezeigt, dass es möglich ist die Groovy-Skripte und große Teile aus den Jenkins-Konfigurationen der aktuellen Infrastruktur in dem neuem System zu nutzen. Zudem wurde gezeigt, dass die Pipeline wie erwartet abgearbeitet wird.

4.7 Analyse der Verständlichkeit

Zur Analyse und Verbesserung der Verständlichkeit der Implementierung des Systems, wurde initial versucht die zyklomatische Komplexitätsmetrik des SonarQube Groovy-Plugins zu verwenden und dies somit auf quantitativer Ebene zu bewerten. Sobald jedoch ein iterativer Prozess entsteht, bei dem der Code angepasst wird, die entsprechende Metrik zu der Änderung ausgewertet wird und dieser Prozess von neuem beginnt, wird oftmals lediglich der Code in soweit angepasst, dass der Wert der Metrik dekrementiert wird (angenommen ein möglichst niedriger Wert würde als besser gelten) und ein geringer Wert wird äquivalent zu verständlichem, simplen Code gesetzt. Die Metrik betrachtet allerdings elementare Faktoren der Komplexität einer Implementierung nicht, wie die konsistente Benennung. Im Fall der zyklomatischen Komplexität wird die Komplexität anhand der bedingten Sprünge berechnet, jedoch bleiben unbedingte Sprünge, wie goto, continue oder break unbeachtet. Daher sollte ein geringer Wert der Metrik nicht mit Verständlichkeit oder Wartbarkeit gleichgesetzt werden.

Aus diesem Grund wurde die Verständlichkeit und Wartbarkeit anhand der

[46] Git benötigt unter anderem Informationen zur URL des Remote-Repositories und des Branches.
Maven benötigt den Pfad zur POM-Datei.
Die Groovy-Skripte benötigen einen Zielpfad, an welchen die Erstellungsergebnisse nach der Validierung kopiert werden sollen.

Ergebnisse einer kleinen Studie zum Umgang der Entwickler mit dem System bewertet.

4.7.1 Studie zum Umgang der Entwickler mit dem System

An der Entwicklung der Infrastruktur-Software sind hauptsächlich drei Entwickler beteiligt. Aufgrund des hohen zeitlichen Aufwands für die Teilnahme an dieser Studie, konnte nur die Arbeitsweise eines Entwicklers bewertet werden.

Der Versuch wurde wie folgt durchgeführt:
Als erstes wurde die Architektur und Arbeitsweise des Systems erklärt. Während dieser Phase konnte der Entwickler Verständnisprobleme klären. Zudem wurde die Package-Struktur der Implementierung erläutert. Anschließend musste der Entwickler eigenständig zwei Aufgabe in dem System lösen.
In der ersten Aufgabe musste der Entwickler einen Trigger für einen Continuous-Build implementieren und in das System integrieren. Die Aufgabe wurde als korrekt bewertet, sobald es möglich war eine Pipeline im Zuge eines Continuous-Build zu durchlaufen.
Die zweite Aufgabe hatte zum Ziel, dass der Port der Datenhaltung und somit auch der Adapter der Datenbank um eine Methode erweitert wurde. Die Methode soll es ermöglichen alle Start- und Endzeiten der Pipelinedurchläufe, welche einer Pipelinekonfiguration zugeordnet sind zu erhalten. Diese Aufgabe wurde als korrekt bewertet, wenn die Methode die erwarteten Werte zurück gab.
Nach der Lösung der Aufgabe wurde ein kurzes Interview durchgeführt. In diesem Interview wurde hinterfragt, wie verständlich das System für den Entwickler war und welche Faktoren dies seiner Erfahrung nach unterstützen.

Für die Erläuterung des Systems und der Package-Struktur benötigten wir 40 Minuten. Die Problematik dabei war, dass das verwendet Architekturmuster der hexagonalen Architektur dem Entwickler unbekannt war. Für die Lösung der ersten Aufgabe benötigte der Entwickler 31 Minuten. Die zweite Aufgabe wurde innerhalb von 47 Minuten gelöst. Ich habe für die erste Aufgabe 13 Minuten benötigt und für die zweite Aufgabe 18 Minuten.

Der Entwickler hat zum ersten Mal mit dem System gearbeitet und benötigte etwas mehr als doppelt so viel Zeit wie ich. Da ich das System implementiert habe kann dies als schnelle Lösung der Aufgaben angesehen werden. In dem nachfolgenden Interview wurde deutlich, dass das System für den Entwickler verständlich war, weil die bereits implementierten Methoden, Tests und Klassen als Beispiele verwendet werden konnten. Die Implementierung ist somit schnell verständlich. Die Voraussetzung dafür ist jedoch das Wissen um das Architekturmuster der Hexagonalen Architektur.

In dem folgenden Kapitel wird über die Ergebnisse der Arbeit resümiert und weitere mögliche Erweiterungen erläutert.

5 Fazit

Das Hauptziel dieser Arbeit war es die bestehende SAP HANA Development Infrastruktur insoweit zu verbessern, dass die bisher fest implementierten Pipelines flexibel modifiziert und erstellt werden können. Jede geänderte beziehungsweise modifizierte Version der Pipeline sollte dabei dokumentiert werden. Zudem sollte es möglich sein, fehlgeschlagene Pipelinedurchläufe ab den fehlerhaft durchlaufenen Schritten den Durchlauf weiter zu führen.

Die Flexibilität der Pipeline wurde durch die Einführung einer Pipelinekonfiguration erreicht. Durch diese Konfiguration kann eine Pipeline als eine Einheit betrachtet und modifiziert werden. Anhand dieser Konfiguration werden zur Laufzeit der Pipeline die notwendigen Jenkins-Jobs erstellt. Ferner wird ein Pipelineschritt nicht länger fest als Nachfolger eines Schrittes definiert, sondern ein Schritt besitzt eine Menge von Vorbedingungen und sobald diese Vorbedingungen erfüllt sind, wird der Schritt gestartet. Diese Konfiguration der Pipeline wird in einer Datenhaltung persistiert und jede Änderung der Konfiguration wird als neuer Eintrag hinterlegt, so dass die alte Version erhalten bleibt. Durch diesen Umgang mit den Konfigurationen kann aus der Datenhaltung eine Historie bezogen werden.

Um zu ermöglichen, dass ein Pipelinedurchlauf ab einem beliebigen bereits durchlaufenen Schritt weitergeführt werden kann, wurden die Daten eines Pipelineschrittes in einer Datenhaltung persistiert und einem Pipelineschrittdurchlauf eindeutig zugeordnet. Die Architektur der beschriebenen Lösungen wurde anhand des Architekturmusters der Hexagonalen Architektur erstellt. Bei der Implementierung dieser war es besonders schwierig die Abhängigkeiten von den äußeren Systemen (z.B. dem Jenkins-Server) zu minimieren. Durch die Annahme, dass eine Pipeline immer aufgrund einer Codeabgabe im Gerrit oder eines Continuous-Build ausgelöst wird, besitzt die Geschäftslogik weiterhin eine Abhängigkeit zu dem Code-Review-System Gerrit. Es ist somit zu hinterfragen, ob das Muster der Hexagonalen Architektur vollständig korrekt umgesetzt wurde. Allerdings ist Gerrit ein System welches sich als fester Bestandteil etabliert hat. Vor der produktiven Nutzung des Systems sollte diese Abhängigkeit jedoch durch die Mittel der Hexagonalen Architektur entfernt werden.

Abschließend lässt sich die Frage stellen, welchen Mehrwert und Mehraufwand die Nutzung der Ergebnisse dieser Arbeit für das Team und die Kunden der Infrastruktur darstellen würden. Durch die Nutzung des Frameworks würde ein Beitrag zu der Erfüllung des Fernziels geleistet. Das Fernziel ist dabei die Unabhängigkeit der Infrastruktur von dem Jenkins-Server zu erreichen. Es wäre nicht mehr notwendig die Funktionalität des Jenkins zum Spezifizieren und Starten von nachfolgenden Jobs zu nutzen. Zudem müss-

te nicht länger die Umgebung des Jenkins zur Weitergabe von Daten an Groovy-Skripte und Nachfolge-Jobs genutzt werden. Der Jenkins hätte vornehmlich die Funktion eines Schedulers, welcher koordiniert wann die Jobs auf welchem Jenkins-Node auszuführen sind.

Es ist nun möglich, dass ein Kunde eine Pipeline selber anlegt und administriert. Somit fällt dieser zeitintensive Aufgabenbereich für das Infrastruktur-Team weg. Die Zeitersparnis durch die Weiterführung eines fehlgeschlagenen Pipelinedurchlaufs kann abhängig davon, in welchem Schritt die Pipeline fehlgeschlagen ist sehr groß sein. Dabei handelt es sich jedoch um einen manuellen Prozess. Somit kann dieser Vorteil selten genutzt werden.

In dem folgenden Kapitel werden Vorschläge zur Erweiterung des Systems gemacht.

6 Ausblick

Bereits während der Arbeit wurde deutlich, dass diverse Erweiterungen des Systems sinnvoll sind. Zusätzlich wurden innerhalb des Infrastruktur-Teams Erweiterungswünsche geäußert. Während der Implementierung und des Entwurfs des Systems wurde daher beachtet, dass das System auf die Umsetzung dieser Anforderungen ausgelegt ist.

6.1 Grafischer Editor

Für einen neuen Nutzer des Systems ist es trotz der intuitiv verständlichen Auszeichnungssprache Yaml nicht möglich ohne ein Konfigurationsbeispiel oder das Schema der Konfiguration eine Pipelinekonfiguration zu erstellen. Daher wäre es sinnvoll einen grafischen Editor zu implementieren, welcher es ermöglicht anhand von parametrisierten Konfigurations- und Pipelineschrittbausteinen eine Pipeline zu erstellen. Diese grafische Darstellung kann dann in eine Konfiguration abgebildet werden.

6.2 Authentifizierung durch den Konfigurationsmanager

Es ist sinnvoll, dass der Schreibzugriff auf die Konfigurationen nur für eine bestimmte Menge von Nutzern möglich ist. Aus diesem Grund sollte es möglich sein Nutzer mit bestimmten Rechten zu verwalten und einen Authentifizerungsmechanismus anzubieten, welcher es dem realen Nutzer ermöglicht einen Systemnutzer zu impersonifizieren. Für diese Funktionalität kann sowohl ein neuer Port für ein Authentifizierungssystem bereitgestellt werden oder die entsprechenden Methoden in der Geschäftslogik untergebracht werden, als auch die vorhandenen Authentifizierungsmechanismen des Datenhaltungssystems genutzt werden. Für die letzte Variante müsste der Daten-

haltungsport erweitert werden und es müsste im Falle eines relationalen Datenbankmanagementsystems sichergestellt werden, dass es row-level-security unterstützt, d.h. das Zugriffsberechtigungen für bestimmte Datensatzmengen statt für Tabellen vergeben werden können.

6.3 Entwicklung eines Dashboards

Zusätzlich sollte ein Dashboard implementiert werden, welches die Informationen der Pipelinedurchläufe darstellt und die Möglichkeit bietet, einen Durchlauf neu zu starten. Somit können Nutzer der Infrastruktur den Status ihres Pipelinedurchlaufs überwachen.

6.4 Branchspezifische Konfigurationen

Es soll möglich sein, für einen Branch eines bestimmten Repositories anzugeben, wieviele Pipelinedurchläufe gleichzeitig für den Branch laufen dürfen. Im Hinblick auf diese Anforderung wurden die Branch und Repository-Informationen in dem Datenbankschema in Paragraph 3.3.3.4 in eigenen Entitäten hinterlegt. Es ist somit einfach möglich die Entität um diese Informationen zu erweitern. Allerdings muss ein Dienst aufgebaut werden, welcher die Informationen über die Pipelines, die aufgrund der Ressourcenbeschränkung nicht gestartet wurden, vorhält. Zudem sollte dieser Fairness garantieren, damit es nicht dazu kommt, dass ein Pipelinedurchlauf zu lange auf seine Ausführung wartet.

6.5 Schemavalidierung der internen Repräsentation

Aktuell wird die interne Repräsentation der Konfiguration durch die Umwandlung in das Yaml-Format anhand eines Schemas validiert. Damit es jedoch möglich ist unabhängig von einer Auszeichnungssprache die Konfiguration zu validieren, sollte ein Schemavalidierungsmechanismus für die Konfiguration in der internen Repräsentation implementiert werden.

6.6 Erweiterung der Pipelinemodelierungsmöglichkeiten

Es ist gewünscht, dass eine 'Wenn-Dann'-Struktur innerhalb der Pipelinemodellierung genutzt werden kann, d.h. dass ein Schritt A startet sobald Schritt B oder Schritt C mit dem erwarteten Resultat terminiert. Dies kann durch die Einführung einer binären 'oder'-Beziehung implementiert werden. Dazu müssen das Datenbankschema der Relation 'waits_for', die Geschäftslogik zur Ermittlung der erfüllten Vorbedingungen und das Konfigurationsschema modifiziert werden.

A Anhang

Bei der Berechnung von Durchschnittswerten werden im Folgenden immer die Werte des Produkts 'hdbstudio' verwendet, da es sich dabei um das Produkt handelt, welches Hauptkunde der Infrastruktur ist und somit realistischere Werte besitzt, als ein Produkt für welche es nur wenige Codeänderungsabgaben pro Woche gibt.

A.1 Externe Abhängigkeiten der aktuellen Infrastruktur

Die aktuelle Infrastruktur besitzt Abhängigkeiten zu den folgenden externen Systemen:

- Gerrit Code-Review-System
- Git Versionsverwaltung
- Sonatype Nexus Server
- QADB (Datenbank, welche diverse Informationen zu Build-Ergebnissen, Bugzillas u.v.m. persistiert)
- via NFS eingebundene Verzeichnisse

A.2 Pipelinekonfiguration

A.2.1 Interne Repräsentation

Die Beispielkonfiguration aus 3.3.3.2 besitzt in der internen Repräsentation die folgende Form:[47]

```
{
    Version: 1.0
    Pipeline:{
       name: "example-pipeline"
       product:{
           name: "example-product"
           language: "java"
           repository: "example.product.project"
           branches:[
               "master",
               "feature"
           ]
```

[47] Die Hashmap wurde in einer JSON-ähnlichen Notation dargestellt (Hashmaps werden in geschweiften Klammern, Listen in eckigen Klammern dargestellt)

```
13          }
14      }
15      Steps:[
16          {
17              name: 'Build'
18              pre: ~
19              build:{
20                  type: maven
21              }
22              post:{
23                  gerrit:{
24                      voting:{
25                          verified:{
26                              failure:{
27                                  value: -1
28                                  text: "build of example failed"
29                              }
30                              success:{
31                                  value: +1
32                                  text: "successfully build
33                                              product"
34                              }
35                          }
36                      }
37                  }
38              }
39          },
40          {
41              name: 'Test'
42              preconditions:[
43                  {
44                      result: SUCCESS
45                      stepname: 'Build'
46                  }
47              ]
48              test:{
49                  testtype: JUnit
50              }
51              post:{
52                  conti:{
53                      mail:{
54                          addresses:[
55                              "example1@mail.de",
56                              "example2@mail.de"
```

```
                        ]
                        failure:{
                            text: "test of example failed"
                        }
                        success:{
                            text: "successfully build example"
                        }
                    }
                }
            }
        }
    ]
}
```

A.2.2 Yaml-Spezifikation der Pipelinekonfiguration

Das Yaml-Schema wurde für das Yaml-Validierungsframework Kwalify[48] erstellt.

```
type: map
mapping:
  "Version":
    type: number
    required: yes
  "Pipeline":
    type: map
    mapping:
      "name":
        type: str
        required: yes
      "product":
        type: map
        required: yes
        mapping:
          "name":
            type: str
            required: yes
          "language":
            type: str
            required: yes
          "repository":
```

[48] http://www.kuwata-lab.com/kwalify/

```
            type: str
            required: yes
          "branches":
            type: seq
            required: yes
            sequence:
               - type: str
  "Steps":
    type: seq
    required: yes
    sequence:
       - type: map
         required: yes
         mapping:
           "name":
             type: str
             required: yes
           "preconditions":
             type: seq
             sequence:
               - type: map
               required: yes
               mapping:
                 "result":
                   type: str
                   required: yes
                 "stepname":
                   type: str
                   required: yes
           "pre":
             type: map
             mapping:
               "gerrit":
                 type: map
                 mapping:
                   "voting":
                    type: map
                    mapping:
                      "verified":
                        type: map
                        mapping:
                          "value":
                            type: int
                            range: { max: 1, min: -1 }
```

```
                        required: yes
                      "text":
                        type: str
                  "review":
                    type: map
                    mapping:
                      "value":
                        type: int
                        range: { max: 2, min: -2 }
                        required: yes
                      "text":
                        type: str
          "conti":
            type: map
            mapping:
              "mail":
                type: map
                mapping:
                  "text":
                    type: str
                    required: yes
                  "addresses":
                    type: seq
                    required: yes
                    sequence:
                      - type: str
                      required: yes
      "build":
        type: map
        mapping:
          "type":
            type: str
            required: yes
      "test":
        type: map
        mapping:
          "type":
            type: str
            required: yes
          "timeout":
            type: int
            range: { min: 0 }
      "integrationtest":
        type: map
```

```
111             mapping:
112               "type":
113                 type: str
114                 required: yes
115               "engine":
116                 type: str
117                 required: yes
118           "codescan":
119             type: map
120             mapping:
121               "type":
122                 type: str
123                 required: yes
124           "consolidate":
125            type: seq
126            sequence:
127              - type: str
128               required: yes
129           "post":
130             type: map
131             mapping:
132               "gerrit":
133                 type: map
134                 mapping:
135                   "voting":
136                     type: map
137                     mapping:
138                       "verified":
139                         type: map
140                         mapping:
141                           "failure":
142                             type: map
143                             mapping:
144                               "value":
145                                 type: int
146                                 required: yes
147                                 range: { max: 1, min: -1 }
148                               "text":
149                                 type: str
150                           "success":
151                             type: map
152                             mapping:
153                               "value":
154                                 type: int
```

```
155                                 required: yes
156                                 range: { max: 1, min: -1 }
157                               "text":
158                                 type: str
159                         "review":
160                           type: map
161                           mapping:
162                             "failure":
163                               type: map
164                               mapping:
165                                 "value":
166                                   type: int
167                                   required: yes
168                                   range: { max: 1, min: -1 }
169                                 "text":
170                                   type: str
171                             "success":
172                               type: map
173                               mapping:
174                                 "value":
175                                   type: int
176                                   required: yes
177                                   range: { max: 1, min: -1 }
178                                 "text":
179                                   type: str
180                 "conti":
181                   type: map
182                   mapping:
183                     "mail":
184                       type: map
185                       mapping:
186                         "addresses":
187                           type: seq
188                           required: yes
189                           sequence:
190                             - type: str
191                             required: yes
192                         "success":
193                           type: map
194                           mapping:
195                             "text":
196                               type: str
197                               required: yes
198                         "failure":
```

```
                                type: map
                                mapping:
                                  "text":
                                    type: str
                                    required: yes
```

A.3 Datenbankschema in SQL

Schema in SQL für das PostgreSQL-Datenbankmanagementsystem, entsprechend Abbildung 4.

```
DROP TABLE IF EXISTS waits_for CASCADE;
DROP TABLE IF EXISTS has_branches CASCADE;
DROP TABLE IF EXISTS Step_Run_Environment CASCADE;
DROP TABLE IF EXISTS Pipeline_Step_Run CASCADE;
DROP TABLE IF EXISTS State CASCADE;
DROP TABLE IF EXISTS Pipeline_Run CASCADE;
DROP TABLE IF EXISTS Pipeline_Step_Config CASCADE;
DROP TABLE IF EXISTS Pipeline_Config CASCADE;
DROP TABLE IF EXISTS Vcs_Branch CASCADE;
DROP TABLE IF EXISTS Vcs_Repository CASCADE;

CREATE TABLE Vcs_Repository(
  repository_id serial,
  repository_name varchar(100) NOT NULL,

  UNIQUE(repository_id),
  UNIQUE(repository_name),
  PRIMARY KEY (repository_id)
);
CREATE TABLE Vcs_Branch(
  branch_name varchar(100),
  repository_id Integer NOT NULL,

  UNIQUE (branch_name, repository_id),
  FOREIGN KEY (repository_id)
    REFERENCES Vcs_Repository(repository_id),
  PRIMARY KEY (branch_name, repository_id)
);
CREATE TABLE Pipeline_Config(
  config_id Integer,
  config_revision_id Integer,
  config_name varchar(40) NOT NULL,
```

```
33    config_json_text text,
34    repository_id Integer NOT NULL,
35
36    UNIQUE(config_id, config_revision_id),
37    FOREIGN KEY (repository_id)
38      REFERENCES Vcs_Repository(repository_id),
39    PRIMARY KEY (config_id, config_revision_id)
40  );
41
42  CREATE TABLE Pipeline_Step_Config(
43    step_config_id serial,
44    step_config_name varchar(100) NOT NULL,
45    step_config_job_exists boolean,
46    config_id Integer NOT NULL,
47    config_revision_id Integer NOT NULL,
48
49    FOREIGN KEY (config_id,config_revision_id)
50      REFERENCES Pipeline_Config (config_id,
51        config_revision_id),
52    PRIMARY KEY (step_config_id)
53  );
54
55  CREATE TABLE Pipeline_Run(
56    run_id serial,
57    run_start timestamp NOT NULL,
58    run_end timestamp,
59    config_id Integer NOT NULL,
60    config_revision_id Integer NOT NULL,
61    run_reason varchar(40) NOT NULL,
62
63    UNIQUE(run_id),
64    FOREIGN KEY (config_id, config_revision_id)
65      REFERENCES Pipeline_Config (config_id,
66        config_revision_id),
67    PRIMARY KEY (run_id)
68  );
69  CREATE TABLE State(
70    state_id serial,
71    state_name varchar(40) NOT NULL,
72
73    UNIQUE(state_id),
74    UNIQUE(state_name),
75    PRIMARY KEY(state_id)
76  );
```

```
77  CREATE TABLE Pipeline_Step_Run(
78    step_run_id serial,
79    state_id Integer,
80    step_run_start timestamp,
81    step_run_end timestamp,
82    step_config_id Integer NOT NULL,
83    run_id Integer NOT NULL,
84
85    UNIQUE(step_run_id),
86    FOREIGN KEY (step_config_id)
87      REFERENCES Pipeline_Step_Config(step_config_id),
88    FOREIGN KEY (run_id)
89      REFERENCES Pipeline_Run(run_id),
90    FOREIGN KEY (state_id)
91      REFERENCES State(state_id),
92    PRIMARY KEY (step_run_id)
93  );
94  CREATE TABLE Step_Run_Environment(
95    run_environment_id serial,
96    run_environment_name varchar(70) NOT NULL,
97    run_environment_value varchar(100) NOT NULL,
98    step_run_id Integer NOT NULL,
99
100   UNIQUE(run_environment_id),
101   FOREIGN KEY (step_run_id)
102     REFERENCES Pipeline_Step_Run(step_run_id),
103   PRIMARY KEY (run_environment_id)
104 );
105 CREATE TABLE has_branches(
106   config_id Integer NOT NULL,
107   config_revision_id Integer NOT NULL,
108   branch_name varchar(100) NOT NULL,
109   repository_id Integer NOT NULL,
110
111   FOREIGN KEY (config_id, config_revision_id)
112     REFERENCES Pipeline_Config(config_id, config_revision_id),
113   FOREIGN KEY (branch_name, repository_id)
114     REFERENCES Vcs_Branch(branch_name, repository_id),
115   PRIMARY KEY (config_id,config_revision_id, branch_name,
116    repository_id)
117 );
118
119 CREATE TABLE waits_for(
120   waiting_step_config_id Integer NOT NULL,
```

```
121   step_config_id Integer NOT NULL,
122   expected_result integer NOT NULL,
123
124   FOREIGN KEY (expected_result)
125     REFERENCES State(state_id),
126   FOREIGN KEY (waiting_step_config_id)
127     REFERENCES Pipeline_Step_Config(step_config_id),
128   FOREIGN KEY (step_config_id)
129     REFERENCES Pipeline_Step_Config(step_config_id),
130   FOREIGN KEY (expected_result)
131     REFERENCES State(state_id),
132   PRIMARY KEY (waiting_step_config_id,step_config_id)
133 );
```

A.4 Berechnung der durchschnittlichen Laufzeit eines Jenkins-Jobs in der aktuellen Infrastruktur

Zur Berechnung der durchschnittlichen Laufzeit wurden von dem Jenkins-Server jeweils für den Build-, Test- und Integrationtest-Schritt der Pipeline für das Produkt 'hdbstudio' die letzten 100 Laufzeiten in XML-Form extrahiert. Für jeden Schritt wurde das arithmetische Mittel der Laufzeit gebildet (bereinigt um die in diesem Moment laufenden Jobs, da diese mit 0 Millisekunden aufgeführt werden). Aus diesen drei arithmetischen Mitteln wurde anschließend das arithmetische Mittel gebildet.

A.4.1 Werte (in Millisekunden):

Schrittname	Anzahl der Werte	Summe der Werte	arithmetisches Mittel
build	95	124738049	$1313032,1$
test	100	2179164	$21791,64$
integrationtest	87	266769407	$3066315,02$

Somit ergibt sich eine durchschnittliche Laufzeit von $1467046,25$ Millisekunden bzw. $24,5$ Minuten

Die Zeit welche eine Pipeline zur Abarbeitung benötigt ist tendenziell deutlich höher, als die Summe, der Laufzeiten ihrer Schritte, da die einzelnen Ausführungen der Jobs, in einer Warteschlange darauf warten müssen, dass ihnen Ressourcen zugeteilt werden.

A.5 Betrachtung der Zeit für die Erstellung eines Jenkins-Jobs via Remote access API

Für die Tests wurde in dem Firefox-Browser der Version 33.0.1 wurde das Plugin HttpRequester 2.0 verwendet (Stand 04.11.2014) um die entsprechenden Http-POST-Anfragen abzusetzen. Zudem wurde ein Jenkins der Version 1.530 verwendet.

Für die Erstellung eines Jobs ist eine Konfiguration notwendig, welche an den Server übermittelt wird, damit ein Szenario mit einer realistischen Konfigurationsgröße getestet werden kann, sollte zuvor die maximale-, minimale- und durchschnittliche-Konfigurationsgröße ermittelt werden.

A.5.1 Größe der XML-Jenkins-Job-Konfigurationen

Die Größe aller Konfigurationsdateien wurde durch ein Bash-Kommando (`find -name config.xml | sed "s/\\ /\\\\ /g" | xargs -L 1 wc -c | cut -d\  -f1,4`), welches im Job-Stammverzeichnis des Jenkins ausgeführt wurde ermittelt. In diesem Fall ist die Betrachtung der maximalen Größen sinnvoll, um zu ermittelt, ob die Job-Erstellung mit Konfigurationen dieser Größe eine signifikant längere Laufzeit erfordert, als die Anfragen, welche eine durchschnittlich große Konfiguration besitzen.

A.5.1.1 Werte (in Byte):

max	min	arithmetisches Mittel
10300	1625	5205

A.5.2 Zeit für die Erstellung eines Jenkins-Jobs

Die Erstellung eines Jobs für eine Konfiguration der Größe 10300 Byte benötigt 202 ms und die Konfiguration der Größe 5205 benötigt 193 ms im arithmetischen Mittel aus 10 Versuchen. Somit besitzt der Aufruf, mit der größeren Konfiguration keine signifikant längere Laufzeit.

A.6 Größe eines Jobeintrags im Jenkinsverzeichnis

Die Größe der Einträge werden erst durch die Tatsache relevant, dass in dem Unterverzeichnis `build/` die Informationen über die bisherigen Durchläufe persistiert werden. Die Log-Dateien der aktuellen Schritte des Produkts hdbstudio benötigen bis zu 9 Megabyte (im Schnitt 8.7 Megabyte). Durch die zum Teil tausendfache Ausführung[49] ergeben sich daher Jobeinträge, die

[49]Zum aktuellen Zeitpunkt (05.11.14 11:06Uhr) ist in der Pipeline des Produkts hdbstudio der Build-Job 7651 mal, der Test-Job 6933 mal und der Integrationttest-Job 6882 mal durchlaufen worden.

Größen im zweistelligen Gigabyte-Bereich besitzen.

In dem Fall des Build-Jobs ergibt sich somit ein Jobverzeichnis mit einer Größe von ca. 66,6 Gigabyte. In der Praxis erreichen die Verzeichnisse jedoch nicht diese Größe, da die Log-Dateien durch die Umstrukturierung des Systems verloren gehen. Aktuell (05.11.14 11:10 Uhr) besitzt das Verzeichnis des Buildjobs der hdbstudio-Pipeline eine Größe von 17 Gigabyte.

A.7 Jenkin Remote-Access API

A.7.1 Erstellung eines Jobs

Zur Erstellung eines neuen Jobs muss eine POST-Anfrage an die URL `<Jenkins-Url>/createItem` mit dem Query-String `name=<Jobname>` abgesetzt werden. Die XML-Konfiguration des zu erstellenden Jobs muss dabei als Dateninhalt mitgeschickt werden.

A.7.2 Eingliedern eines Jobs in die Scheduler-Queue

Um einen Job mit n ($n \in \mathbb{N}$) Parametern in die Queue einzugliedern, muss eine POST-Anfrage an `<Jenkins-Url>/<Jobname>/buildWithParameters`, mit dem Query-String `token=<Jenkinsnutzer-Token>&<Name-1>=<Wert1>&...&<Name-n>=<Wert-n>` gesendet werden. Der Jenkinsnutzer-Token wird dabei zur Identifizierung des Nutzers, welcher den Job startet genutzt. Somit muss für dieses System ein Nutzer angelegt werden, welcher die Berechtigung besitzt die Jobs zu starten.

A.7.3 Entfernen eines Jobs

Damit in dem Jenkins-Server ein Job entfernt wird, muss eine POST-Anfrage an `<Jenkins-Url>/<Jobname>/doDelete` abgesetzt werden.

A.7.4 Existenz eines Jobs ermitteln

Für den Test, ob ein Job bereits existiert, kann eine GET-Anfrage an `<Jenkins-Url>/<Jobname>`. Anschließend muss der Wert der Antwort ausgewertet werden. Ist der Rückgabecode kleiner als 400 wird angenommen, dass der Job existiert. Diese Auswertung kann jedoch weiter verfeinert werden um mögliche 'false positives' auszuschließen.

Glossar

A

Ausgrad

Der Begriff Ausgrad ist in der Graphentheorie bei dem Umgang mit gerichteten Graphen relevant. Der Ausgrad ist eine natürliche Zahl, welche angibt, wieviele Kanten von einem Knoten ausgehen.

B

Branch

Ein Entwicklungszweig, welcher durch den Beginn eines neuen Projekts oder eine alternative Fortführung eines Softwareprojekts entsteht und anschließend in das Urprojekt zurückgeführt werden kann.

C

Code-Review System

Ein System, welches vor das Versionsverwaltungssystem geschaltet ist und es ermöglicht Abgaben vor der Integration in die Datenbasis zu bewerten, Verbesserungen einzufordern oder die Abgabe abzulehnen, so dass diese nicht in die Datenbasis integriert werden darf.

CodeNarc

Werkzeug zur statischen Codeanalyse für Code in der Programmiersprache Groovy.

Continuous Build

Intervall gesteuertes starten eines Pipelinedurchlaufs für eine bestimmte Codebasis.

Continuous Integration Server

Ein Server zur automatisierten Durchführung von Erstellungs-, Qualitätssicherungs- und Verteilungsprozessen.

G

Gerrit

Ein Code-Review System für Git. Ein Nutzer bewertet in diesem System eine Codeänderung mit einem Wert zwischen -2 und 2 im Bereich der natürlichen Zahlen. Zudem kann durch einen automatisierten Verifikations-Prozess eine Codeänderung mit $-1{,}0$ oder 1 bewerten. Ein Review Wert von -2 verhindert, dass eine Codeänderung in das

Repository eingepflegt werden kann und der Wert 2 ermöglicht es die Änderung einzupflegen. Eine 2 kann jedoch keine −2 außer Kraft setzen..

Geschäftslogik

Die generische Logik eines Systems, welche unabhängig von äußeren Systemen ist. Die Geschäftslogik stellt alle Operationen und Methoden bereits, welche essentiell für die Erfüllung der Aufgabe sind, für welche das System erstellt wurde [50].

Git

Eine Open-Source-Software zur verteilten Versionsverwaltung.

J

Jenkins-CI

Ein Continuous-Integration Server dessen Vorteile in der leichten Erweiterbarkeit liegen.

Jenkins-Node

Ein Computersystem, welche in einem Jenkins-Server als zusätzliche Rechenressource aufgeführt wird. Auf diesem Node werden Prozesse ausgeführt um den Server zu entlasten.

M

Maven

Ein für die Sprache Java entwickeltes Werkzeug zur Automatisierung des Vorgangs des Kompilierens und des Auflösens der Abhängigkeiten zwischen den zu kompilierenden Dateien. Die Informationen zu einem Projekt, welches erstellt werden soll, werden in einer POM-Datei gespeichert. Diese POM ist in der Auszeichnungssprache XML verfasst und hat dementsprechend den Namen 'pom.xml'.

Mocking

Eine Methodik zum Testen eines Systems, welche verwendet wird, wenn ein System getestet werden soll, dass in Abhängigkeit von einem System/einer Klasse steht, welches nicht zur Verfügung steht. Das System/die Klasse wird dabei durch eine Vertreterklasse imitiert, welche die angenommene Funktionalität in minimalen Umfang erfüllt.

P

Pipeline

Ausprägung einer Pipelinekonfiguration in einem System, welche ausgeführt werden kann, so dass ein Pipelinedurchlauf in Gang gesetzt wird[50].

Pipelinedurchlauf

Ein Pipelinedurchlauf besteht aus einer Menge von Pipelineschrittdurchläufen, welche alle zu einer Definition einer Pipeline gehören[50].

Pipelinekonfiguration

Beschreibung einer Prozessstruktur, welche als Prozessdefinitionen unterschiedliche Pipelineschrittkonfigurationen besitzt[50].

Pipelineschritt

Ausprägung einer Pipelineschrittkonfiguration, welche ausgeführt werden kann, so dass ein Pipelineschrittdurchlauf in Gang gesetzt wird[50].

Pipelineschrittdurchlauf

Eine Ausführung des Prozesses, welcher in einem Pipelineschritt definiert wurde[50].

Pipelineschrittkonfiguration

Eine Pipelineschrittkonfiguration ist eine Prozessdefinition, welche beschreibt, wie eine Software Erstellt, die Qualität gesichert wird oder die erstellte Software verteilt wird[50].

R

radikales Vorgehen

Das Verhalten, welches dafür sorgt, dass ein Aspekt eines Softwareprojekts außerhalb der bisherigen Erfahrung liegt. Durch diesen Aspekt wird zwangsläufig das Risiko des Projekts erhöht[51].

Repository

Datenablage für ein Softwareprojekt innerhalb eines Versionsverwaltungssystem.

S

[50] Definition im Kontext der Arbeit

[51] `http://www.inf.fu-berlin.de/inst/ag-se/teaching/S-BSE/023_normal-radikal.pdf`

SonarQube

Eine Plattform, welche unterschiedliche Werkzeuge zur statischen Codeanalyse integriert und somit über eine Schnittstelle zur Verfügung stellt.

V

Versionsverwaltungssystem

Ein System zur Erfassung von Änderungen in einem Repository. Eine Änderung erhält dabei eine eindeutige Kennung.

Literatur

[1] J. Goll, *Methoden und Architekturen der Softwaretechnik.* Wiesbaden: Vieweg + Teubner, 2011.

[2] A. Shalloway and J. R. Trott, *Entwurfsmuster verstehen. Eine neue Perspektive auf objektorientierte Software-Entwicklung.* Bonn: Mitp, 2003.

[3] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship.* Boston: Prentice Hall, 2009.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianapolis: Prentice Hall, 1995.

[5] A. Cockburn. (2005) The pattern: Ports and adapters (''object structural"). [Online]. Available: http://alistair.cockburn.us/Hexagonal+architecture

[6] O. Ben-Kiki and C. Evans. (2009) Yaml ain't markup language (yaml)(tm) version 1.2. [Online]. Available: http://yaml.org/spec/cvs/spec.pdf

[7] A. Kemper and A. Eickler, *Datenbanksysteme: eine Einführung.* München: Oldenbourg, 2013.

Abbildungsverzeichnis