Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MAXIMILIAN FISCHER
MASTER THESIS

A DSL FOR CONFIGURING CONTINUOUS DEPLOYMENT PIPELINES FOR ANDROID APPS

Eingereicht am 29. Januar 2018

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 29. Januar 2018

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 29. Januar 2018

Abstract

Application development for mobile platforms, such as Android or iOS, plays an increasingly important role as the consumption of digital content largely takes place on these platforms compared to desktop ones. Developers rely on Continuous Delivery (CD) infrastructures in order to be agile and responsive to customer requirements and feedback. However, the provision of a suitable CD infrastructure requires considerable effort and know-how to setup this infrastructure. Automated implementation of a CD solution could save developers a lot of time and effort, so they can focus on application development.

To enable generalization, we analyze a large number of deployment pipelines for the Android platform, followed by creation of a uniform semantic model. Based on this, we develop the Android Continuous Delivery DSL Engine, which generates a suitable CD infrastructure using a domain-specific language (DSL). The resulting deployment pipeline can be generically applicable to a multitude of Android applications.

For evaluation phase, we created various CD infrastructures using the DSL Engine. We test and evaluate their desired behavior using several open source applications on Android. Finally, we demonstrate that the developed DSL engine is a useful support construct for mobile platform developers.

Zusammenfassung

Die Anwendungsentwicklung für mobile Plattformen wie Android oder iOS spielt heutzutage eine immer wichtigere Rolle, da der Konsum von digitalen Inhalten zum großen Teil auf diesen stattfindet. Damit Entwickler agil auf Kundenanforderungen und Feedback reagieren können, setzen sie während der Entwicklung häufig auf Continuous Delivery (CD) Infrastrukturen. Das Bereitstellen einer angemessenen CD-Infrastruktur verursacht jedoch einen erheblichen Aufwand und setzt ein entsprechendes Know-how bei der Realisierung dieser Infrastruktur voraus. Die automatisierte Umsetzung einer CD-Lösung könnte Entwicklern einen hohen Zeit- und Arbeitsaufwand ersparen, sodass ihr Fokus bei der eigentlichen Applikationsentwicklung verbleiben kann.

Um eine Generalisierung zu ermöglichen, werden eine Vielzahl von Deployment Pipelines für die Android Plattform analysiert und daraus ein einheitliches semantisches Modell gebildet. Darauf aufbauend wird die sogenannte Android Continuous Delivery DSL Engine entwickelt, die unter Verwendung einer domänenspezifischen Sprache (DSL), eine entsprechende CD-Infrastruktur generiert. Die hierdurch erzeugte Deployment Pipeline soll generisch auf eine Vielzahl von Android-Applikationen anwendbar sein.

Zur Evaluierung werden verschiedene CD-Infrastrukturen unter Verwendung der DSL Engine erzeugt. Dabei wird deren gewünschte Verhaltensweise anhand von unterschiedlichen Open Source Android-Applikationen geprüft und bewertet. Zum Abschluss wird gezeigt, dass die entwickelte DSL Engine eine sinnvolle Unterstützung für Entwickler von mobilen Plattformen darstellt.

Inhaltsverzeichnis

1 Einleitung

Im Zeitalter der Digitalisierung bieten sich enorm große Chancen und Möglichkeiten für IT-Unternehmen. Dabei müssen sie sich neuen Herausforderungen stellen. Eine stetige Innovationsfähigkeit und schnelle Handlungsfähigkeit zählen zu den Grundvoraussetzungen, um mit der Konkurrenz mitzuhalten. Als wichtiger Schlüssel zum Erfolg haben sich Methoden und Techniken der agilen Softwareentwicklung herausgestellt, die den Entwicklungsprozess flexibel und schlank gestalten. Eine Kernfrage bleibt jedoch. Wie können wir Software schnellstmöglich zu unseren Kunden ausliefern? Die Antwort darauf lautet *Continuous Software Engineering*. Während diese Praktik im Umfeld der cloudbasierten Dienste längst gelebt wird, hinken Bereiche, wie die Entwicklung von mobilen Applikationen, hinterher. Dabei stellen mobile Applikationen einen beliebten und wachsenden Markt dar. Allein in Google Play, dem zentralen App Store von Android, waren Anfang 2018 rund 3,57 Millionen Apps gelistet (Statista, 2018). Gerade hier spielt es eine große Rolle, schnell auf Feedback zu reagieren und neue Features und Designänderungen in kurzen Zyklen an seine User auszuliefern, um mit den sich schnell ändernden Benutzerwünschen Schritt zu halten. Ein durchschnittlicher User wird eine Applikation bereits nach einem Tag vom Startbildschirm löschen, wenn diese nicht überzeugend ist (Tappforce, 2017).

Es stellt sich die Frage, warum trotz der wachsenden Nutzung von mobilen Applikationen im Bereich des Continuous Software Engineering häufig nur auf cloudbasierte Anwendungen Bezug genommen wird.

1.1 Zielsetzung

In dieser Arbeit soll eine Software entworfen und implementiert werden, die eine domänenspezifische Sprache (DSL) anbietet, um eine Continuous Delivery (CD) Lösung für Android-Applikationen zu generieren.

Zunächst wird festgestellt, inwieweit sich die Auslieferung von mobilen Applikationen im Vergleich zu cloudbasierter Software unterscheidet und welchen Herausfor-

derungen sich App-Entwickler hinsichtlich des Continuous Software Engineering stellen müssen. Um sich dieser Problemstellung anzunähern, werden verschiedene Deployment Pipelines aus dem Bereich Android gesucht, analysiert und zu einer generischen Android Deployment Pipeline aggregiert. Auf Basis dieses Modells wird im Hauptteil der Arbeit eine eigene DSL definiert, um die Konfiguration der entwickelten Software so einfach wie möglich zu gestalten.

Am Ende dieser Arbeit sollen die Fragen beantwortet werden, wie weit sich die Erstellung einer Continuous Deployment Pipeline für den Bereich Android automatisieren lässt, bis zu welchem Grad die entwickelte Software generisch für alle Android-Applikationen anwendbar ist und welche Anpassungen innerhalb der Android-Applikation notwendig sind, um sie für Continuous Software Engineering vorzubereiten.

1.2 Aufbau der Arbeit

Diese Arbeit gliedert sich in sieben Kapitel. Das erste Kapitel, das mit diesem Abschnitt endet, beinhaltet die Einführung in das behandelte Themengebiet sowie die Beschreibung der Zielsetzung der Arbeit.

In Kapitel zwei werden Grundlagen, die für das weitere Lesen der Arbeit benötigt werden, erklärt. Zunächst erfolgt eine allgemeine Vorstellung der Continuous Software Engineering Praktiken, bevor anschließend auf die Erstellung domänenspezifischer Sprachen eingegangen wird. Weiterhin werden benutzte Kerntechnologien wie *Docker*, *Jenkins* und das *Android* Betriebssystem genauer betrachtet.

Im dritten Kapitel wird eine Literaturrecherche durchgeführt, um die Anforderungen an eine Deployment Pipeline aus dem Bereich Android zu spezifizieren.

Kapitel vier und fünf befassen sich mit dem Design und der anschließenden Implementierung der Software. Dabei wird eine domänenspezifische Sprache definiert, um eine Android Deployment Pipeline auf möglichst einfache Weise zu konfigurieren. Die Software nutzt diese Sprache, um eine Continuous Software Engineering Umgebung inklusive der Pipelines automatisiert zu generieren.

Im vorletzten Kapitel wird die Funktionalität der entwickelten Lösung anhand von verschiedenen Open Source Android-Applikationen nachgewiesen. Darüber hinaus wird erklärt, wie sich die Veröffentlichung einer Android-Applikation in eine Deployment Pipeline integrieren lässt.

Diese Arbeit schließt mit einer Zusammenfassung der gewonnenen Erkenntnisse und gibt einen Ausblick auf mögliche Verbesserungen.

2 Grundlagen

In diesem Kapitel wird zunächst der Continuous Software Engineering Ansatz mit seinen Vorteilen für das Software Engineering vorgestellt. Anschließend werden Domain Specific Languages (DSL) definiert und deren Aufbau erläutert. Um Verständnisprobleme in nachfolgenden Kapiteln vorzubeugen, werden die im Laufe dieser Arbeit verwendeten Tools wie Jenkins, Jenkins Pipeline und Docker präsentiert. Abgeschlossen wird das Grundlagenkapitel mit einer kurzen Einführung in Android, in der auch die Herausforderungen bei der Entwicklung von Android-Applikationen hervorgehoben werden.

2.1 Continuous Software Engineering

Das Aufkommen und die Beliebtheit von agilen Methoden zur Software-Entwicklung liefert den Beweis, dass eine hohe Flexibilität und schnelle Anpassung im Bereich der Software-Entwicklung gefordert ist. Sehr komplexe, geschäfts- und sicherheitskritische Software wird häufig durch verteilte Teams entwickelt. Um die Qualität und Robustheit der Software zu erhalten und weiter zu verbessern wurden in den letzten zwei Jahrzehnten zwei Annahmen getroffen. Zum einen ist eine enge Verbindung zwischen Entwicklung und Ausführung erforderlich, um Fehler so schnell wie möglich erkennen und beheben zu können. Zum anderen gab es die weit verbreitete Anerkennung, dass die Erhöhung der Häufigkeit bestimmter kritischer Aktivitäten, wie beispielsweise dem Release der Software, dazu beitragen kann, viele Herausforderungen zu überwinden (Fitzgerald & Stol, 2014). Diese Ideen führten zu Praktiken wie Continuous Integration, Continuous Delivery und Continuous Deployment, welche im Folgenden genauer erläutert werden.

2.1.1 Continuous Integration

Duvall, Matyas und Glover (2007) definieren Continuous Integration (CI) in ihrem gleichnamigen Buch als eine Praktik der Software Entwicklung, bei der ein

Teammitglied seine Arbeit mindestens einmal pro Tag integriert. Das führt zu mehreren Integrationen pro Tag, welche durch einen automatisierten Buildprozess und automatisierte Tests verifiziert werden, um Integrationsprobleme so schnell wie möglich zu erkennen. Duvall et al. (2007) referenzierten bei dieser Definition einen Artikel von Fowler und Foemmel (2006).

Um Continuous Integration anzuwenden, muss mindestens eine Verbindung zu einem Versionsverwaltungssystem, ein Build Skript, ein Feedback Mechanismus und ein Prozess, welcher die Veränderungen im Quellcode integriert[1] existieren.

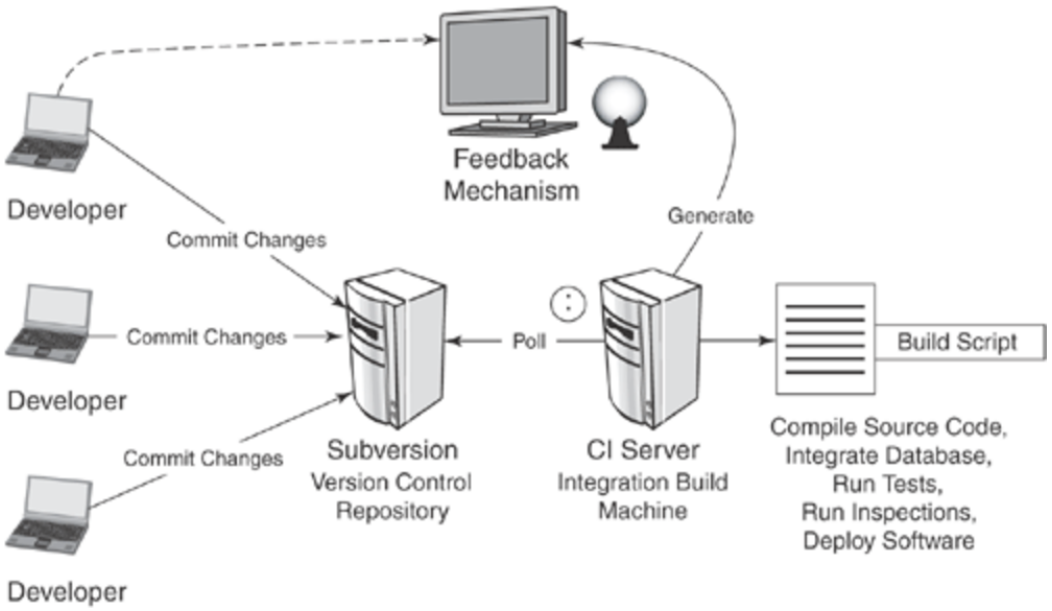

Abbildung 2.1: Übersicht der Komponenten eines CI-Prozesses (Duvall, Matyas & Glover, 2007).

Abbildung 2.1 zeigt die Komponenten eines CI-Prozesses, dessen Ablauf meist wie folgt stattfindet. Ein Entwickler ändert etwas am Quellcode und commitet seine Änderung in das Repository des Versionsverwaltungssystem. Währenddessen prüft der CI-Server das Repository zyklisch nach Änderungen. Sobald ein Commit im Versionsverwaltungssystem auftaucht, holt sich der CI-Server den aktuellen Stand des Quellcodes aus dem Repository und führt anschließend das Build-Skript aus, welches die Software integriert. Anschließend generiert der CI-Server Feedback, zum Beispiel durch das Versenden von E-Mails, und fährt mit der zyklischen Überprüfung des Repositories fort (Duvall et al., 2007).

Die anschließenden Konzepte bauen auf Continuous Integration auf und bringen dieses einen Schritt weiter. Sie beinhalten neben den Konzepten von CI zusätz-

[1] Der Integrationsprozess kann durch einen CI-Server oder manuell durchgeführt werden.

lich ein Deployment in Test- und Produktivumgebungen sowie das Testen von funktionalen und nicht funktionalen Anforderung der Software.

2.1.2 Continuous Delivery

Continuous Delivery (CD) bezeichnet eine Sammlung von Techniken, Prozessen und Werkzeugen, bei denen Software in kurzen Zyklen entwickelt und zu jedem Zeitpunkt auf verlässliche Weise veröffentlicht werden kann. Es zielt darauf ab, Software schneller und häufiger zu bauen, zu testen und zu veröffentlichen. Im Mittelpunkt dieses Ansatzes, mit dem Kosten, Zeit und das Risiko von Softwareauslieferungen reduziert werden, steht ein einfacher und wiederholbarer Prozess für die Softwareauslieferung.

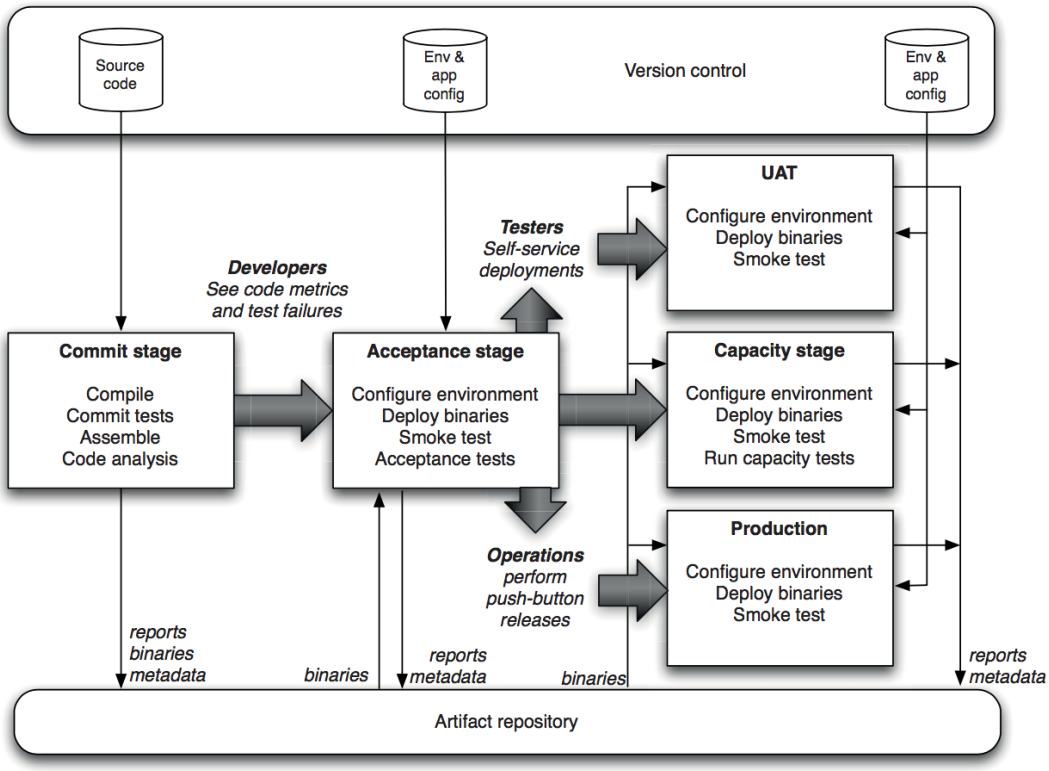

Abbildung 2.2: Beispiel einer Deployment Pipeline (Humble & Farley, 2010).

Continuous Delivery wird durch eine Deployment Pipeline realisiert, die den kompletten Prozess von einer Änderung in der Versionsverwaltung bis hin zur Veröffentlichung der Software modelliert. Diese ist in mehrere Stages unterteilt, die jeweils einem eigenen Build-Prozess entsprechen und Validierungen beinhalten, die die Software bestehen muss, um als veröffentlichungsfähig bezeichnet werden

zu können. Abbildung 2.2 zeigt eine Deployment Pipeline, wie sie Humble und Farley (2010) darstellen.

Jede neue Änderung am Programmcode erzeugt eine neue Instanz der Deployment Pipeline und führt die erste Stage aus. Diese erzeugt ein oder mehrere Artefakte, die an nachfolgende Stages weitergereicht werden. Sobald die erste Stage erfolgreich beendet wurde, werden weitere Stages sequentiell nacheinander ausgeführt, bis eine Stage fehlschlägt oder das Ende der Pipeline erreicht wurde. Die letzte Stage beinhaltet typischerweise ein Deployment der Änderung in die Produktivumgebung, welches manuell getriggert wird und von einem Beteiligten des Projektes autorisiert werden sollte. Jede erfolgreich durchlaufene Stage erhöht das Vertrauen in die getätigte Änderung und somit die Erwartung, dass daraus eine neue, funktionierende Version der Software entstehen wird. Das Sequenzdiagramm in Abbildung 2.3 zeigt, wie sich eine Änderung des Quellcodes durch die Pipeline bewegt.

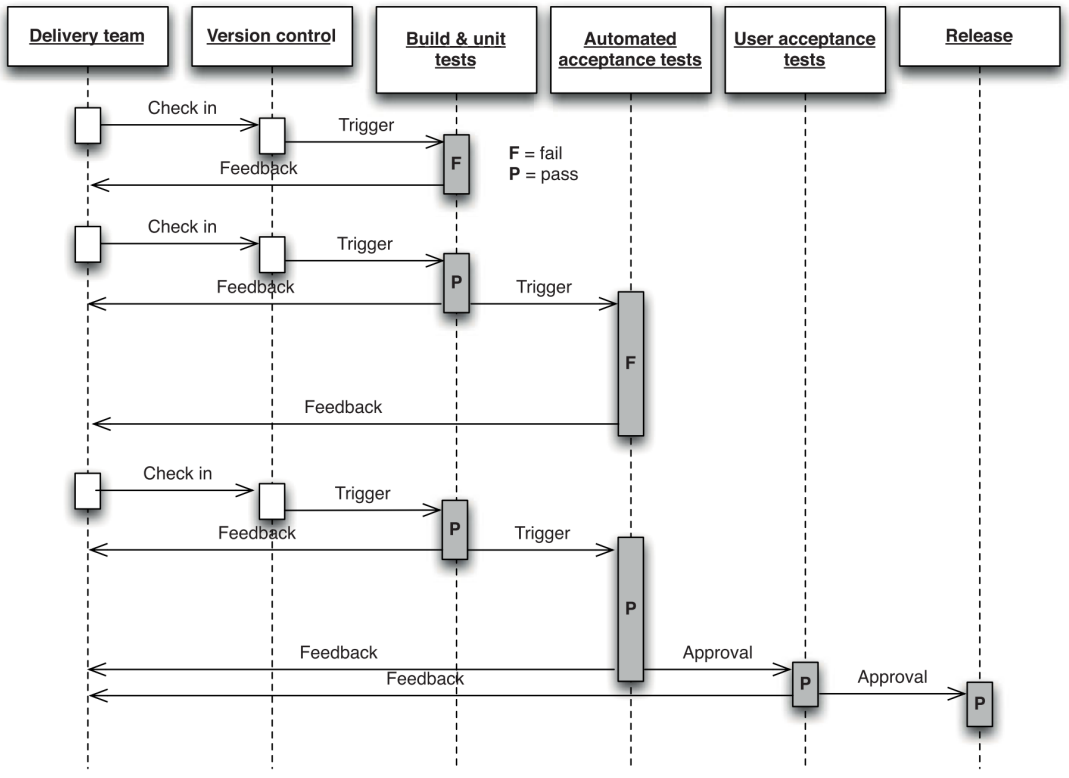

Abbildung 2.3: Eine Deployment Pipeline in Ausführung (Humble & Farley, 2010).

Eine Deployment Pipeline erfüllt im Wesentlichen drei Ziele. Sie macht alle Aspekte des Softwareauslieferungsprozess, wie Build, Test, Deploy und Veröffentlichung der Software für alle Teammitglieder sichtbar und fördert so die Zusammenarbeit

des Teams. Darüber hinaus sorgt sie für verbessertes Feedback, indem Probleme so früh wie möglich erkannt und diese im Anschluss so schnell wie möglich behoben werden. Zuletzt ist es möglich, durch einen vollautomatisierten Prozess eine beliebige Softwareversion in jede beliebige Umgebung zu installieren und zu veröffentlichen. Zusammengefasst soll durch eine Deployment Pipeline hochqualitative Software in schneller und verlässlicher Weise erzeugt werden (Humble & Farley, 2010). Nachfolgend werden die einzelnen Stages einer Deployment Pipeline genauer betrachtet.

Commit Stage

Die Commit Stage stellt den Einsteig in die Deployment Pipeline dar und wird bei jeder Änderung des Quellcode Repositories angestoßen. In dieser Stage wird geprüft, ob das Produkt fehlerfrei kompiliert und es werden Binaries für nachfolgende Stages erzeugt. Gegebenenfalls werden neben Binaries weitere Artefakte wie Testdaten erzeugt. Darüber hinaus werden Unit-Tests sowie Code-Analysen durchgeführt. Nur ein erfolgreiches Durchlaufen dieser Stage triggert die nachfolgenden Pipeline-Prozesse.

Ziel ist, sicherzustellen, dass das System auf einer technischen Ebene fehlerfrei funktioniert. Dabei soll die Commit Stage möglichst schnell ablaufen, um zeitnahes Feedback bezüglich einer fehlerhaften Software-Version an das Entwicklerteam zu berichten (Humble & Farley, 2010).

Folgende Aktionen werden in dieser Pipeline Stage durchgeführt:

- Kompilieren des Quellcodes.
- Ausführung der Unit-Tests.
- Analyse des Quellcodes.
- Erstellen und Archivierung der Binaries.

Automated Acceptance Test Stage

Die Automated Acceptance Test Stage stellt sicher, dass das System die Spezifikation der Kunden erfüllt und somit den Anforderungen seiner Benutzer entspricht. Die Tests dieser Stage dienen zur Verifikation des gesamten Produktes aus Sicht des Kunden. Zur Ausführung der Akzeptanztests muss die Software zunächst in eine Testumgebung deployed werden. Dieser Schritt geht oft mit einer Konfiguration der Testumgebung sowie Ausführung der Smoke-Testing[2] Suite einher. Anschließend werden die Akzeptanztest ausgeführt, welche funktionale und

[2]Smoke-Tests sollen die grundlegende Funktionalität einer (Test-)Umgebung prüfen.

nicht funktionale Anforderungen der Software aus Kundensicht testen. Wird die Testsuite der Akzeptanztests zu umfangreich, kann eine eigene sogenannte Performance Test Stage in die Pipeline eingefügt werden. Die Akzeptanztest Suite sollte auch Feedback zu Regressionen der aktuell getesteten Version an die Entwickler liefern.

Diese Stage der Pipeline läuft typischerweise um ein Vielfaches länger als die Commit Stage. Daher empfiehlt sich oft die parallele Ausführung der Akzeptanztests, um schnelleres Feedback zu bekommen.

In dieser Stage der Pipeline werden folgende Schritte durchgeführt:

- Konfiguration der Testumgebung.
- Auslieferung von Binaries oder Installer.
- Ausführung der Smoke Tests.
- Ausführung der Akzeptanztests.

Smoke Tests stellen einfache Testfälle dar, die überprüfen, ob das Deployment in die Umgebung erfolgreich durchgeführt wurde und die Applikation läuft. Beispielsweise könnte ein Ping, der sicherstellt, dass die Applikation sowie alle wichtigen Ressourcen erreichbar sind, einen Smoke-Test darstellen. Diese Smoke-Test-Suite kann auch bei nachfolgenden Stages, welche einen Deployment beinhalten, verwendet werden. Bei Continuous Delivery zweigt die Pipeline hier ab, so dass nachfolgende Stages manuell getriggert werden (Humble & Farley, 2010).

Manual Test Stage

In iterativen Prozessen folgen den Akzeptanztests immer manuelle Tests in Form von explorativen oder Usability-Tests. Automatisierte Akzeptanztests sollen Zeit schaffen, damit sich das Testpersonal auf hochwertige Aktivitäten konzentrieren kann, statt nur Test-Skripte auszuführen.

Die Manual Test Stage stellt sicher, dass das System seine Anforderungen erfüllt und Mängel erkannt werden, die nicht durch automatisierte Tests erfasst werden. Die Tester prüfen in dieser Stage die Benutzerfreundlichkeit, führen explorative Tests durch und checken das Look-and-feel der Applikation auf verschiedenen Plattformen (Humble & Farley, 2010).

In der Release Stage werden folgende Tätigkeiten durchgeführt:

- Konfiguration der Produktionsumgebung.
- Auslieferung von Binaries oder Installer.
- Ausführung der Smoke Tests.

Performance Test Stage

Jedes System hat neben den funktionalen auch nicht-funktionale Anforderungen, wie beispielsweise Sicherheit, Performanz sowie Kapazität. In dieser Pipeline Stage werden automatisierte Tests ausgeführt, um zu messen, inwieweit das System diesen Anforderungen genügt.

Dabei können die Ergebnisse dieser Stage eine Barriere zu nachfolgenden Stages darstellen oder lediglich Informationen zur Entscheidungsfindung liefern. Eine sehr leistungsstarke Anwendungen gilt normalerweise als nicht einsetzbar, sobald die Performance eine gewisse Grenze unterschreitet. Für andere Anwendungen ist es sinnvoller, die nicht funktionalen Anforderungen lediglich im Blick zu behalten und einen Menschen entscheiden zu lassen, ob der Release Candidate besteht oder nicht (Humble & Farley, 2010).

In der Release Stage werden folgende Tätigkeiten durchgeführt:

- Konfiguration der Produktionsumgebung.
- Auslieferung von Binaries oder Installer.
- Ausführung der Smoke Tests.
- Ausführung der Performance Tests.

Release Stage

Die Release Stage liefert die Software, sofern sie alle vorherigen Stages erfolgreich durchlaufen hat. Die Auslieferung kann als Paket oder über ein Deployment in den Produktionsbetrieb erfolgen. Diese Stage sollte immer der letzte Schritt der Deployment Pipeline darstellen (Humble & Farley, 2010).

In der Release Stage werden folgende Tätigkeiten durchgeführt:

- Konfiguration der Produktionsumgebung.
- Auslieferung von Binaries oder Installer.
- Ausführung der Smoke Tests.

2.1.3 Continuous Deployment

Im Gegensatz zu Continuous Delivery erfolgt bei Continuous Deployment ein automatisiertes Deployment jeder Änderung in die Produktivumgebung, ohne

die Autorisierung eines Mitarbeiters des Projektes. Abbildung 2.4 veranschaulicht den Unterschied zwischen Continuous Delivery und Continuous Deployment (Humble & Farley, 2010).

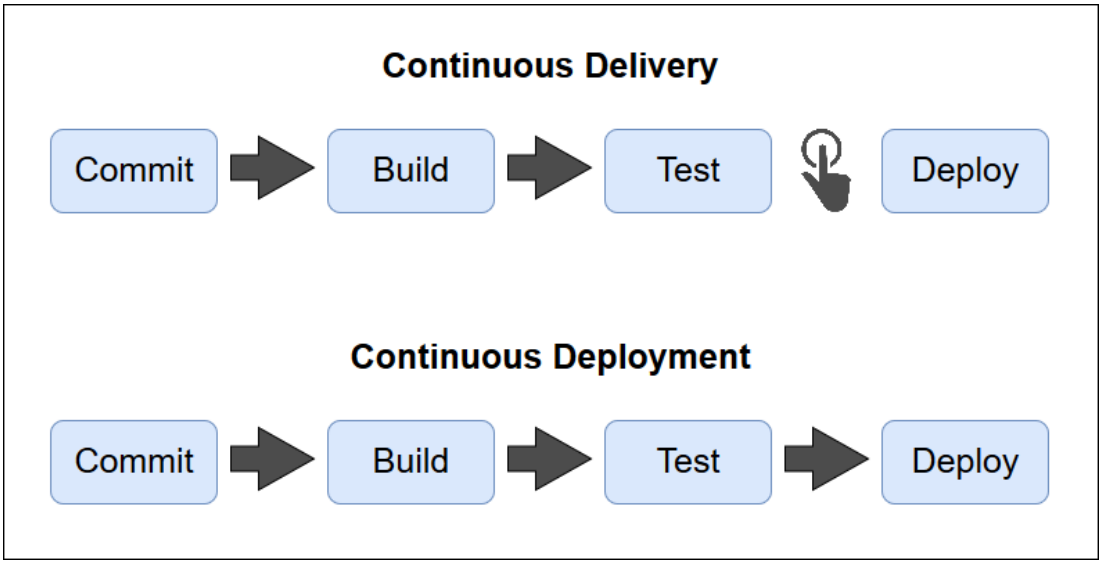

Abbildung 2.4: Continuous Delivery und Continuous Deployment im Vergleich.

2.2 Domain-Specific Languages

(Fowler, 2010) definiert eine Domain-Specific Language (DSL) als:

> „A computer programming language of limited expressiveness focused on a particular domain."

Er stellt vier Schlüsselelemente dieser Definition in den Vordergrund.

- *Computer programming language*: Eine DSL wird von Menschen genutzt, um einer Maschine Befehle zu erteilen. Wie bei allen modernen Programmiersprachen sollte sie so entworfen werden, dass sie möglichst einfach von Menschen verstanden wird, auch wenn sie schlussendlich von einem Computer ausgeführt wird.
- *Language nature*: Eine DSL ist in erster Linie eine Programmiersprache, sollte sich aber in Anlehnung an eine natürliche Sprache fließend lesen und schreiben lassen. Ihre Ausdruckskraft entsteht nicht nur aus einzelnen Anweisungen, sondern vielmehr durch die Art und Weise, wie diese zusammengesetzt werden.

- *Limited expressiveness*: General-purpose Programmiersprachen[3] wie Java bieten viele Möglichkeiten wie Variablendeklaration, Kontrollstrukturen und Abstraktionen. Auch wenn diese Eigenschaften nützlich und notwendig sind, verkomplizieren sie aber gleichermaßen das Lernen und die Benutzung. Domänenspezifische Sprachen stehen im Gegensatz zu diesen Universalsprachen und bieten nur eine Teilmenge von deren Eigenschaften. Sie sind nicht dafür ausgelegt, ein komplettes System zu implementieren, sondern sollten vielmehr für einen speziellen Aspekt des Systems genutzt werden.
- *Domain focus*: Eine limitierte Sprache ist nur dann nützlich, wenn sie eine klare Sicht auf eine kleine Domäne bietet. Der Domänenfokus ist eine Konsequenz aus dem vorherigen Punkt und macht die limitierte Ausdruckskraft einer Sprache lohnenswert.

Domänenspezifische Sprachen lassen sich in drei Hauptkategorien unterteilen. Externe DSLs, interne DSLs und language workbenches.

Eine *externe DSL* ist von der Hauptsprache der Anwendung getrennt. Normalerweise hat eine externe DSL eine benutzerdefinierte Syntax, es wird jedoch häufig auch die Syntax einer anderen Sprache, wie beispielsweise XML[4], für eine externe DSL hergenommen. Ein externes DSL-Skript wird durch Quellcode in der Host-Anwendung mit Hilfe von Text-Parsing Techniken analysiert. Bekannte Beispiele für externe DSLs sind SQL, AWK oder auch XML.

Eine *interne DSL* ist in die Hauptsprache der Anwendung eingebettet. Sie stellt eine besondere Art dar, wie diese Universalsprache genutzt wird. Ein internes DSL-Skript ist valider Code in seiner Universalsprache, verwendet aber nur eine Untermenge der Funktionen dieser Sprache in einem gewissen Stil, um einen kleinen Aspekt des Gesamtsystems abzudecken. Eine interne DSL sollte sich eher wie eine benutzerdefinierte Sprache, statt wie eine Universalsprache anfühlen. Ein klassisches Beispiel hierfür ist Lisp.

Eine *language workbench* ist eine spezialisierte integrierte Entwicklungsumgebung zur Definition und zum Aufbau von domänenspezifischen Sprachen. Insbesondere wird eine language workbench nicht nur verwendet, um die Struktur einer DSL zu bestimmen, sondern auch als benutzerdefinierte Bearbeitungsumgebung für DSL-Skripte. Die resultierenden Skripte vereinen die Bearbeitungsumgebung und die Sprache.

Fowler (2010) zeigt noch eine andere Sichtweise auf DSLs, in der die Notwendigkeit eines *semantischen Modells* betont wird. Unter der Prämisse eine vernünftige DSL zu erstellen gibt es nur wenige Ausnahmen, in denen ein semantisches Modell nicht notwendig ist. Die Grundidee ist, das semantische Verhalten in einem

[3] auf Deutsch werden general-purpose languages als Universalsprachen bezeichnet.
[4] Extensible Markup Language

Modell zu erfassen, welches durch die DSL im Laufe eines Parse-Vorganges mit Daten befüllt wird. Im Bereich des Software Engineering werden ständig Abstraktionen in Form von Programmbibliotheken oder Frameworks gebildet, die unter Verwendung einer Command-Query API manipuliert werden können. Im Kontext von domänenspezifischen Sprachen entspricht das semantische Modell einer Programmbibliothek und die DSL selbst einem Front-End, das einen eigenen Stil zur Manipulation des semantischen Modells bietet. Die meiste Arbeit beim Erstellen einer DSL liegt in der Entwicklung des semantischen Modells. Die eigentliche DSL dient nur als Zugriffsschicht und deren Erstellung stellt die leichtere Aufgabe dar.

Durch die Trennung von semantischem Modell und DSL ergeben sich einige Vorteile. Typischerweise soll durch eine DSL ein komplexer Sachverhalt einer spezifischen Domäne repräsentiert werden. Innerhalb des semantischen Modells kann über die Semantik dieser Domäne nachgedacht werden, ohne sich mit den Details einer DSL-Syntax oder einem Parser zu befassen. Zusätzlich ist ein unabhängiges Testen des Modells und Parsers möglich, indem Objekte direkt mit Hilfe der Command-Query API erzeugt und manipuliert werden. Somit können Probleme des Modells isoliert betrachtet werden, ohne genau verstehen zu müssen, wie der Parser funktioniert.

Ein explizites semantisches Modell macht zudem die Verwendung mehrerer DSLs innerhalb einer Domäne ohne Code-Duplizierung möglich. Man kann zunächst mit einer einfachen internen DSL beginnen und später eine alternative DSL[5] anbieten. Einige Nutzer und Skripte werden noch auf die erste Variante der DSL zurückgreifen, die einfach bestehen bleibt, da beide Sprachen trotz unterschiedlichem Parser und unterschiedlicher Syntax das semantische Modell auf die gleiche Art und Weise befüllen. Zudem können sich Sprache und Modell unabhängig voneinander entwickeln. Erweiterungen des Modells können ohne Änderung an der DSL erfolgen und sobald diese Änderungen fertiggestellt sind, können die nötigen Strukturen in der DSL nachgepflegt werden. In umgekehrter Weise kann auch mit der Syntax experimentiert werden, solange der Parser die gleichen Objekte des semantischen Modells erzeugt.

Ein weiteres Thema, das oft in Verbindung mit domänenspezifischen Sprachen betrachtet wird, ist die *Codegenerierung*. Fowler (2010) unterscheidet zwischen *Model-Aware Generation* und *Model Ignorant Generation*, je nachdem, ob eine explizite Repräsentation des semantischen Modells in der Zielumgebung vorliegt oder dieses Modell in den Kontrollfluss des generierten Codes eingebettet ist. Beide Ansätze greifen auf das semantische Modell zurück, indem Daten daraus gelesen werden, um den Zielcode zu erzeugen. Auch hier wird von der Flexibilität, die durch Trennung von Modell und DSL entsteht, profitiert.

[5] Beispielsweise in Form einer externen DSL.

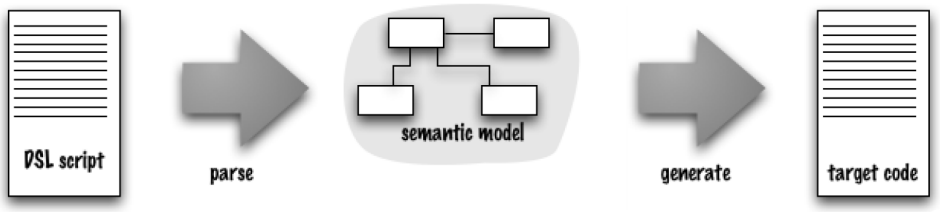

Abbildung 2.5: Architektur einer DSL-Abarbeitung (Fowler, 2010).

Abbildung 2.5 zeigt die Gesamtarchitektur einer DSL-Abarbeitung in einfacher Weise. Dabei wird vor allem die Trennung von Parser, semantischem Modell und Codegenerator hervorgehoben. Der Schritt der Codegenerierung ist optional, wird aber dennoch gezeigt, da Codegenerierung in eine andere Zielumgebung auch Bestandteil dieser Arbeit sein wird (Fowler, 2010).

2.3 Der Jenkins Buildserver

Jenkins ist ein erweiterbares und webbasiertes Open Source Tool, um Continuous Integration, Continuous Delivery und Continuous Deployment Prozesse abzubilden. Es ist komplett in Java geschrieben und wird von Teams beliebiger Größen sowie für eine Vielzahl von Projektarten eingesetzt, die unterschiedliche Technologien wie .NET, Ruby, Groovy, PHP, Java und vieles mehr beinhalten.

Die folgenden drei Argumente haben zu dem großen Erfolg von Jenkins beigetragen und sind mitunter die Hauptgründe, weshalb Jenkins auch in dieser Arbeit als Buildserver dient, auf dem die im Kontext dieser Arbeit entwickelte Software aufsetzt. Zunächst fällt die einfache und intuitive Benutzung von Jenkins durch seine Weboberfläche auf. Trotzdem ist Jenkins extrem flexibel und kann an eigene Bedürfnisse, wie in unserem Fall die Android Domäne, angepasst werden. Es existieren Hunderte von Open Source Plugins und jede Woche kommen neue hinzu. Ein weiterer Grund für die Verwendung von Jenkins ist seine aktive und große Community, welche für die Pflege von Jenkins selbst sowie dessen Plugins verantwortlich ist (Smart, 2011).

2.3.1 Jenkins Pipeline

Beim Jenkins Pipeline Plugin handelt es sich um eine Sammlung von Plugins, mit deren Hilfe Continuous Delivery Pipelines in Jenkins modelliert und implementiert werden können.

Während es in Jenkins schon immer die Möglichkeit gab, mehrere sogenannte Freestyle Jobs zu verketten, macht Pipeline dieses Konzept zu einem zentralen Bestandteil von Jenkins.

Zur Definition einer Jenkins Pipeline muss eine Textdatei, welche als *Jenkinsfile* bezeichnet wird, erstellt werden. Dabei sind zwei Arten von Syntax zu unterscheiden. Zum einen die neuere und vereinfachte *Declarative Pipeline*-Syntax und die mächtigere *Scripted Pipeline*-Syntax. Beide Varianten setzen auf das Pipeline Subsystem auf und verwenden Groovy, jedoch nutzt die Declarative Pipeline nur eine Untermenge der Möglichkeiten einer Scripted Pipeline, weshalb im Kontext dieser Arbeit ausschließlich Scripted Pipeline Code generiert wird. Um ein Pipeline-Skript in Jenkins einzufügen, gibt es zwei Wege. Einerseits kann das Jenkinsfile in ein Versionsverwaltungssystem commitet werden und Jenkins holt sich die Datei automatisch beim Build des Projektes. Alternativ kann das Skript auch innerhalb der Jenkins Weboberfläche in einen Pipeline Editor eingetragen werden.

Mit Hilfe von Jenkins Pipeline können von einfachen bis hin zu komplexen Szenarien alle Arten von Anwendungsfällen abgedeckt werden. Abbildung 2.6 zeigt einen praxisnahen Pipeline Flow, der mit Jenkins Pipeline realisiert wurde.

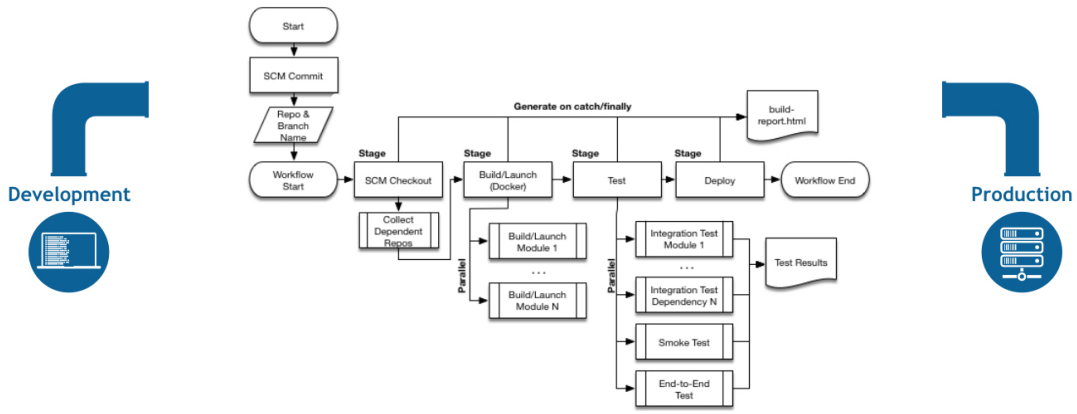

Abbildung 2.6: Jenkins Pipeline Flow Beispiel (Jenkins, 2017a).

Will man in seinem Projekt einen Continuous Delivery oder Continuous Deployment Prozess modellieren, ist Jenkins Pipeline gegenüber der herkömmlichen Variante, Freestyle Jobs zu verketten, meist die bessere Alternative. Die Pipeline Plugin Suite ist beständig und überlebt sowohl geplante als auch ungeplante

Neustarts des Buildservers. Ein weiteres nützliches Feature ist das Pausieren der Pipeline, um auf menschliche Interaktionen zu warten, da, wie in Abschnitt 2.1 beschrieben, bei Continuous Delivery nicht jeder erfolgreiche Build auch ein Deployment mit sich zieht. Darüber hinaus unterstützt Jenkins Pipeline komplexe Anforderungen, indem Fork und Join Operationen, Schleifen und Parallelität zugelassen werden. Zuletzt ist Jenkins Pipeline leicht erweiterbar, so dass es viele Plugins gibt, die sich in die Pipeline integrieren lassen und die Pipeline-Syntax erweitern (Jenkins, 2017a).

Die im Folgenden beschriebenen Plugins lassen sich mit Jenkins Pipeline nutzen.

2.3.2 Jenkins Plugins

Das enorme Angebot von Plugins war einer der Gründe, warum Jenkins in dieser Arbeit als Continuous Delivery Tool genutzt wird. Mittlerweile werden von der Community mehr als 1300 Plugins gepflegt, die Jenkins mit fast jeder gängigen Technologie verknüpfen. Die verschiedenartigen Plugins decken alle gängigen Versionsverwaltungssysteme, nahezu jede Programmiersprache und alle Bereiche des Continuous Software Engineerings ab (CloudBees, 2017).

Da wir uns im Kontext dieser Arbeit in der Android Domäne bewegen, bekommen Android Plugins für Jenkins eine besondere Bedeutung. Das *Google Play Android Publisher Plugin*[6] ermöglicht Jenkins, eine Android-Applikation sowie dazugehörige Informationen im Google Play Store zu veröffentlichen. Das *Android Lint Plugin*[7] parst die Ergebnisse einer statischen Code-Analyse und präsentiert diese auf anschauliche Weise innerhalb der Jenkins-Benutzeroberfläche. Für die statische Code-Analyse wird das Lint Tool von Android verwendet.

Zum Signieren einer Android-Applikation werden Passwörter sowie private Schlüssel benötigt. Das *Credentials Plugin*[8] sorgt für die sichere Speicherung dieser Informationen in Jenkins, während das *Credentials Binding Plugin*[9] diese Informationen während des Build Vorganges zur Verfügung stellt.

Weiterhin kann das *Lockable Resources Plugin*[10] zum Zugriff auf eine gemeinsam genutzte Ressource verwendet werden, um Wettstreitigkeit zwischen parallelen Build-Strängen zu vermeiden. In unserem Fall stellen Android Geräte oder Emulatoren, auf denen Tests ausgeführt werden, eine gemeinsam genutzte Ressource dar.

[6] https://wiki.jenkins.io/display/JENKINS/Google+Play+Android+Publisher+Plugin
[7] https://wiki.jenkins.io/display/JENKINS/Android+Lint+Plugin
[8] https://wiki.jenkins.io/display/JENKINS/Credentials+Plugin
[9] https://wiki.jenkins.io/display/JENKINS/Credentials+Binding+Plugin
[10] https://wiki.jenkins.io/display/JENKINS/Lockable+Resources+Plugin

Dieser Abschnitt zeigt, wie flexibel Jenkins mit Hilfe von Plugins an eigene Projektbedürfnisse angepasst werden kann. Zum Abschluss der Jenkins-Grundlagen sei noch erwähnt, dass auch ein *Docker Plugin* existiert, welches bereits in Jenkins Pipeline integriert ist, um einzelne Schritte der Pipeline innerhalb eines Docker Containers auszuführen (Jenkins, 2017b). Weitere Ausführungen zu Docker sind Bestandteil des kommenden Abschnittes.

2.4 Die Docker Containervirtualisierung

Docker ist eine Open-Source-Software, um Anwendungen in Containern zu isolieren. Innerhalb eines Containers befindet sich eine isolierte Instanz eines Betriebssystems, mit dessen Hilfe die Anwendungen laufen. Dies erinnert an eine abgespeckte Version einer virtuellen Maschine, jedoch bieten Container einige Vorteile gegenüber klassischen virtuellen Maschinen.

Da sich Container-Ressourcen mit dem Host-Betriebssystem teilen, sind sie sehr effizient und können binnen Sekunden gestartet und gestoppt werden. Anwendungen, die innerhalb von Containern laufen, haben nur einen geringen Overhead im Vergleich zu Anwendungen, die direkt auf dem Host-Betriebssystem gestartet werden. Im Vergleich zu herkömmlichen virtuellen Maschinen können sehr viele Container durch ihre leichtgewichtige Natur parallel auf einem Host laufen. Entwickler nutzen die Containertechnologie, um von der Portierbarkeit zu profitieren. Sie brauchen sich nicht mehr um Unterschiede zwischen ihrer und der Benutzerumgebung und um eventuelle Abhängigkeiten zu kümmern. Somit lassen sich auch Fehler, die durch subtile Änderungen der Laufzeitumgebung entstehen, vermeiden. In umgekehrter Weise entsteht für Endanwender der Vorteil, dass sie komplexe Anwendungen einfach herunterladen und starten können, ohne sich mit der Konfiguration und Installation zu befassen.

Allerdings sei erwähnt, dass sich die Ziele beider Technologien unterscheiden. Während virtuelle Maschinen fremde Umgebungen vollständig emulieren sollen, sind Container dafür gedacht, Anwendungen in sich geschlossen und portabel zu machen.

Die Docker Plattform besteht im Wesentlichen aus der Docker Engine, die sich um das Erstellen und Ausführen von Containern kümmert und einem Cloud Service namens Docker Hub, um Container-Images zu verteilen (Mouat, 2016).

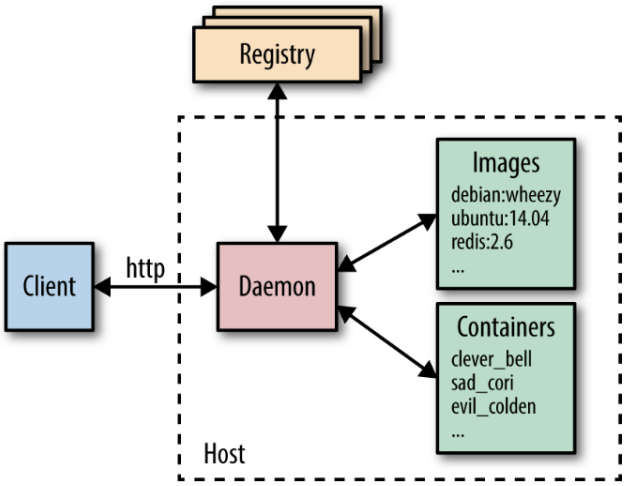

Abbildung 2.7: Docker Architektur (Mouat, 2016).

Docker Architektur

Abbildung 2.7 zeigt eine Übersicht der im Rahmen dieser Arbeit verwendeten Docker Komponenten. Der Docker Daemon stellt das Herzstück der Architektur dar, dessen Aufgabe die Erstellung, Ausführung und Überwachung der Container ist. Auch das Verwalten der Images und das Bauen der Container fallen in seinen Zuständigkeitsbereich.

Die Kommunikation zwischen Docker Client und Docker Daemon läuft per HTTP-Request ab. Standardmäßig wird hierzu ein Unix Domain Socket verwendet. Auch die Nutzung eines TCP-Sockets ist möglich, falls sich Client und Daemon auf unterschiedlichen Systemen befinden. Für die Kommunikation mit dem Docker Daemon wird eine wohldefinierte REST-API verwendet, so dass selbst geschriebene Programme direkt mit dem Daemon arbeiten können, ohne den Umweg über den Client zu nutzen.

In Docker Registries werden Images gespeichert, verwaltet und verteilt. Dabei ist Docker Hub die Standard Registry, auf der neben offiziellen Images auch tausende andere öffentlich verfügbare Images existieren. Der Docker Daemon lädt Images automatisch aus Registries herunter, falls diese lokal nicht vorhanden sind.

In Abbildung 2.7 sind auf der rechten Seite Images und Container zu sehen. Ein Container ist eine laufende Instanz eines Images, der mit Einstellungen wie Name, ID oder IP-Adresse initialisiert wird. Er besitzt im Gegensatz zu einem Image einen Status. Ein Image dient somit lediglich als Vorlage für einen Container (Mouat, 2016).

2.5 Das mobile Betriebssystem Android

Android ist ein Software Stack für mobile Geräte, wie Smartphones, Mediaplayer, Netbooks und Tabletcomputer, der ein Betriebssystem, Middleware und Kernapplikationen beinhaltet. Dabei bilden Applikationen die oberste und ein Linux Kernel die unterste Schicht des Stacks. Dies ist die beliebteste mobile Plattform und betreibt etwa hundert Millionen mobile Geräte in 190 verschiedenen Länder (Android Developers, 2017a). Bei Android handelt es sich um freie Software, die quelloffen entwickelt wird (Google, 2017).

Um selbst Applikationen für Android zu entwickeln, wird eine Java Entwicklungsumgebung und ein Framework namens Android SDK benötigt. Applikationen werden somit komplett in Java geschrieben. Die fertige Anwendung muss in eine Android Package (APK) Datei verpackt werden und kann anschließend auf App Stores bereitgestellt oder direkt mit Hilfe des Paketmanagers auf den Endgeräten installiert werden (Android Developers, 2017a).

Der Standard App Store für Android-Applikationen ist Google Play. Anwender können dort Android-Applikationen suchen, installieren und upgraden. Entwickler können ihre Applikationen in Google Play veröffentlichen, indem sie einen Eintrag erstellen und eine oder mehrere APK-Dateien hochladen. Dabei werden drei getrennte Tracks angeboten: alpha, beta und production. Durch diese ist eine schrittweise Veröffentlichung möglich. Die Benutzergruppen für alpha und beta Tracks können selbst gewählt werden, um eine Applikation beispielsweise firmenintern zu evaluieren, bevor sie im production Track freigegeben wird. Zusätzlich kann Google Play Daten über die Applikation erfassen, den Vertrieb und die Preisgestaltung auf den verschiedenen Android-Plattformen verwalten und die Leistung der App im Store analysieren (Android Developers, 2017c).

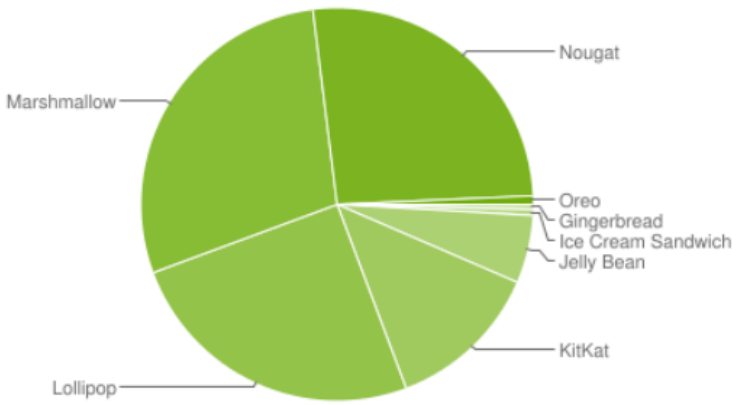

Abbildung 2.8: Verteilung der Android-Versionen (Android Developers, 2018).

Auf der anderen Seite gibt es auch Herausforderungen, denen sich Android Entwickler stellen müssen und die ein Continuous Software Engineering Ansatz gerade im Bereich Android sehr lohnenswert machen. Aufgrund der vielen mobilen Geräte und der Offenheit der Plattform hat Android mit einer hohen Software- und Hardware-Fragmentierung des Marktes zu kämpfen. Abbildung 2.8 veranschaulicht die unterschiedlichen Android-Versionen, die Anfang 2018 zeitgleich im Umlauf waren. Fokussieren sich Entwickler lediglich auf die aktuellste Version, wird nur ein kleiner Teil der Android-Nutzer erreicht. Continuous Software Engineering Praktiken können dabei helfen, mit den unterschiedlichen Versionen und den damit einhergehenden Optionen mitzuhalten.

Neben den verschiedenen Android-Versionen gibt es auch eine Vielzahl unterschiedlicher Geräte auf dem Markt. Dies wird als Hardware-Fragmentierung bezeichnet und auf Abbildung 2.9 dargestellt (Stand 2015). Dabei besitzt jedes Gerät einzigartige Features, wie beispielsweise die Größe und Auflösung des Displays. Auch hier müssen Applikationen umfassend getestet werden, um sicherzustellen, dass diese auf allen gängigen Smartphones und Tablets fehlerfrei laufen.

Abbildung 2.9: Android Geräte Fragmentierung (Open Signal, 2015).

Eine weitere Herausforderung nicht nur für Entwickler, sondern auch für Continuous Software Engineering stellen langsame Unit-Tests in Android dar. Häufig benötigen diese Android-Komponenten und können nur mit Hilfe eines Emulators

oder Android-Gerätes ausgeführt werden, obwohl gerade die erste Stage einer Pipeline möglichst schnelles Feedback liefern sollte.

Des Weiteren wird mit Veröffentlichung der APK in einen App Store die Kontrolle über die Applikation abgegeben. Der Store ist für die Bekanntmachung, Installation und das Updaten der App verantwortlich. Dies ist ein wesentlicher Unterschied zu Bereichen außerhalb des mobile Developement, da dort meist ein direktes Deployment in die Produktivumgebung stattfindet und man selbst entscheidet, wie ein Rollout in die Produktivumgebung durchgeführt wird (Summers, 2016).

Zum Verifizieren der im Verlauf dieser Arbeit entwickelten Software, wurde eine Android-Applikation namens *OSR Playground* erstellt. Sie addiert lediglich zwei Zahlen und zeigt einen Informationstext an. Ziel dieser App ist es, Unit-, UI-, und Performance-Tests beispielhaft durchzuführen sowie den Signiervorgang und den Veröffentlichungsprozess einer Applikation zu analysieren, um diese Aktivitäten automatisiert mit Hilfe einer DSL ausführen zu können.

3 Anforderungen an eine Android Deployment Pipeline

Nur selten wird in der Literatur zu Continuous Software Engineering die Auslieferung von mobilen Applikationen betrachtet. Aus diesem Grund wurde eine Literaturrecherche durchgeführt, um zu sehen, ob und vor allem wie Continuous Integration, Continuous Delivery und Continuous Deployment Prozesse im Kontext der Android Entwicklung durchgeführt werden. Die gesammelten Ergebnisse sind in diesem Kapitel zusammengefasst und werden im Laufe dieser Arbeit zur Entwicklung einer Domain Specific Language inklusive Codegenerator eingesetzt.

Wie bereits im vorherigen Kapitel kurz erwähnt, unterscheiden sich mobile Applikationen stark von herkömmlichen Desktop- oder Web-Applikationen. Neben der hohen Betriebssystem- und Gerätefragmentierung unter Android, müssen mobile Applikationen eine Vielzahl von Eingaben wie beispielsweise Gestiken, Spracheingaben und Sensordaten handhaben. Auch Ressourcen wie CPU, Speicher und die verfügbare Akkuleistung sind auf mobilen Geräten vergleichsweise knapp (Rabenhorst & Steffens, 2016).

Rossi et al. (2016) beschreiben neben den bereits genannten Punkten weitere Gründe, die das Deployment von mobiler Software erschweren. Da Software Updates nicht transparent für den Nutzer ablaufen und ein App Store wie Google Play Zeit für Reviews der Applikation braucht, sind häufige Updates nicht praktikabel. Deshalb wird meist ein Continuous Delivery Ansatz gegenüber Continuous Deployment bevorzugt. Neben der Tatsache, dass zu häufige Updates den Benutzer stören, kann dieser selbst entscheiden, wann und ob er die mobile Software auf seinem Gerät upgradet. Dadurch können viele unterschiedliche Versionen der Applikation zeitgleich im Umlauf sein und die Entwickler müssen sicherstellen, dass weiterhin alle Versionen korrekt funktionieren. Zusätzlich muss stets eine komplette APK-Datei veröffentlicht werden, sodass kein schrittweises Deployment der Applikation erfolgen kann. Dies stellt laut Rossi et al. (2016) ein großes Risiko dar. Auch ein Hotfix oder Rollback können nur in seltenen Fällen und ausschließlich mit Hilfe des jeweiligen App Stores realisiert werden.

Diese Unterschiede machen deutlich, dass eine Deployment Pipeline, wie sie in Kapitel 2 auf Seite 5 gezeigt wurde, nicht eins zu eins übernommen werden kann. Jedoch wurde diese Deployment Pipeline als Ausgangspunkt genutzt, um in Verbindung mit den gefundenen Anforderungen ein einheitliches Modell für Continuous Deployment Pipelines, wie sie im Bereich Android genutzt werden können, zu erstellen. Um die nachfolgenden Abschnitte anschaulich zu gestalten, wird dieses Modell in Abbildung 3.1 visualisiert. Darauf aufbauend werden zunächst die abgebildeten Pipelines sowie deren Konfigurationsmöglichkeiten abstrakt betrachtet, bevor anschließend im Detail auf die gefundenen Anforderungen und die Herleitung der einzelnen Stages eingegangen wird.

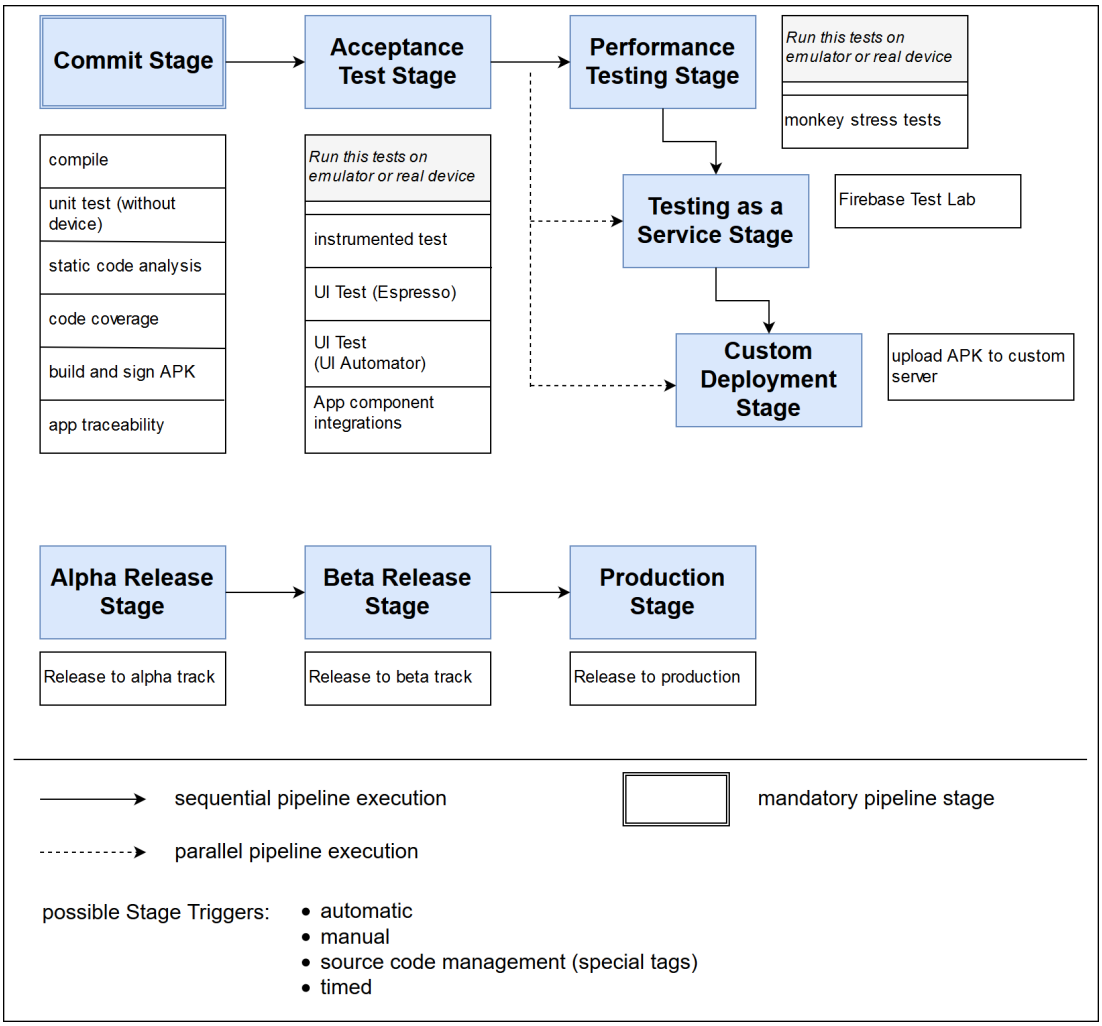

Abbildung 3.1: Modell einer Continuous Delivery Lösung für den Bereich Android.

Die Aufteilung in zwei Pipelines sticht bei diesem Schaubild als erstes ins Auge. Die Alpha Release, Beta Release und Production Stage sind für die Veröffentli-

chung der erzeugten Artefakte in Google Play verantwortlich und wurden in eine eigene Pipeline ausgelagert. Zum einen hat dies die zu Beginn dieses Kapitels genannten Gründe, durch die ein Continuous Deployment von mobilen Applikationen zu risikoreich ist und den Nutzer zu sehr überfordert. Im Laufe der Entwicklung wurde zusätzlich festgestellt, dass maximal 15 Uploads nach Google Play möglich sind. Hierzu zählen auch Verschiebungen der APK beispielsweise von Alpha in den Beta Track. Eine Quelle für diese Limitierung wurde nicht gefunden, jedoch bestätigten mehrere Foren[1] dieses Verhalten. Bei großen Projekten sind 15 Commits pro Tag schnell überschritten, was zu einem Fehlschlagen der Pipeline führen würde. Auch das manuelle Anstoßen der Stages ist nicht zufriedenstellend, da alle Pipeline-Instanzen, die nicht getriggerten wurden, in einen vordefinierten Timeout laufen würden, oder schlimmer noch, niemals terminieren. Das Resultat sind erfolgreiche Builds, die als fehlgeschlagen markiert werden. Da laut Humble und Farley (2010) die Visualisierung des Build-Status ein wichtiges Konzept des Continuous Software Engineering ist, wurden die Google Play Deployment Stages in eine eigene Pipeline ausgelagert.

Das doppelt umrahmte Rechteck zeigt, dass lediglich die Commit Stage zwingend erforderlich ist, da hier Artefakte für nachfolgende Stages erzeugt werden und sichergestellt wird, dass die Applikation fehlerfrei kompiliert (Humble & Farley, 2010). Alle weiteren Stages, die Abbildung 3.1 zeigt, sind optional und können nach den jeweiligen Anforderungen ausgelassen werden.

Parallel ausführbare Stages sind durch die gestrichelten Pfeile gekennzeichnet. In unserer Android Deployment Pipeline können Performance Testing, Testing as a Service und Custom Deployment Stage parallel ausgeführt werden. Es geht dabei weniger um die zeitgleiche Ausführung der Stages, sondern vielmehr darum, diese Stages auf eine Ebene zu stellen, um die Reihenfolge der Ausführung dynamisch je nach Belieben festzulegen. So kann bei einem Build die Custom Deployment Stage manuell gestartet werden und die Applikation beliebig lange getestet werden, bevor anschließend nicht funktionale Anforderungen durch die Performance Testing Stage geprüft werden. Für den nächsten Build könnte die Reihenfolge wie in Abbildung 3.1 durch die durchgezogenen Pfeile angedeutet festgelegt werden. Für die dynamische Änderung der Reihenfolge müssen manuelle Trigger verwendet werden.

Die gezeigte Android Deployment Pipeline gibt eine Standardreihenfolge mit Hilfe der durchgezogenen Pfeile vor. Die parallel ausführbaren Stages haben bereits gezeigt, dass eine abweichende Reihenfolge pro Pipeline-Instanz möglich ist. Es soll möglich sein, die Reihenfolge dieser Stages zu fixieren. So kann beispielsweise die Ausführung der Custom Deployment Stage bei jedem Build vor Ausführung der Performance Testing Stage erzwungen werden. Die Acceptance Test Stage sollte immer nach der Commit Stage ausgeführt werden. Diese zwei Stages in-

[1] https://stackoverflow.com/questions/34872860/15-apk-upload-daily-limits-in-google-play

nerhalb der Pipeline zu verschieben würde keinen Sinn ergeben, da die Commit Stage wichtige Artefakte liefert, während die Akzeptanztests so bald wie möglich laufen sollten, sofern sie vorhanden sind. Die Stages der zweiten Pipeline laufen immer sequentiell ab. Dieses Verhalten ist durch den Play Store vorgegeben. Dieser identifiziert eine APK eindeutig mit Hilfe einer Application-ID und eines Versionscodes, der bei jeder Veröffentlichung in den App Store inkrementiert werden muss. Befindet sich eine gewisse Version der Android-Applikation bereits in einem Track, so kann diese lediglich in nachfolgende Tracks mit größeren Benutzergruppen verschoben werden. Eine Rückführung in vorherige Tracks ist nicht möglich. Eine APK im Beta Track kann somit lediglich nach Production und nicht in den Alpha Track verschoben werden. Dies gilt auch, falls diese Version noch nie im Alpha Track lag. Wird eine neuere Applikation hochgeladen, werden alle älteren Applikationen aus den vorherigen Tracks verdrängt. Wird beispielsweise eine neue Applikation in den Beta Track geladen, wird die alte Applikation automatisch aus dem Alpha Track entfernt. Die Option, eine spezifische Reihenfolge für die zweite Pipeline festzulegen, ist demnach überflüssig.

Jeder Stage kann genau ein Trigger zugewiesen werden. Dieser legt fest, wie die Ausführung einer Stage gestartet wird. Einzige Ausnahme stellt hier die Commit Stage dar, die, wie im Continuous Software Engineering üblich, bei jeder Änderung des Quellcodes angestoßen wird. Aus der Literaturrecherche ließen sich vier verschiedene Arten von Trigger identifizieren:

- Automatische Trigger.
- Manuelle Trigger.
- Source Code Management (SCM) Trigger.
- Zeitgesteuerte Trigger.

Ein automatischer Trigger führt die Stage aus, sobald alle vorherigen Stages erfolgreich durchlaufen wurden. Ein manueller Trigger wartet auf User Input, um die Ausführung der Pipeline fortzusetzen. Diese beiden Trigger sind vor allem innerhalb der ersten Pipeline sinnvoll. Source Code Management Trigger werden unter anderem von Orr (2014) verwendet, um mit Hilfe von bestimmten Git Tags eine Stage auszuführen. Dies wird typischerweise für Release Stages ausgeführt. Jeder Tag, der beispielsweise mit dem Schlüsselwort *beta* startet, führt zur Ausführung der Beta Release Stage und stellt die Applikation Beta Testern zur Verfügung.

Zeitgesteuerte Trigger führen Stages periodisch, nach Ablauf einer vorgegebenen Zeitspanne, aus. Auch dies ist besonders für die Release Stages der zweiten Pipeline interessant. Hier kann zum Beispiel einmal wöchentlich die aktuellste erfolgreich gebaute Applikation in einen gewissen Track veröffentlicht werden. Durch die erste Pipeline wird sichergestellt, dass diese Applikation releasefähig

ist. Einen ähnlichen Ansatz benutzen Rossi et al. (2016). Hier wird einmal wöchentlich vom Master Branch in einen Release Branch gemerged. Anschließend folgt eine Stabilisierungsphase des Release Branches, bevor die finale Applikation veröffentlicht wird. Auch Alpha und Beta Versionen der Applikation werden bei Facebook für das Deployment von mobiler Software eingesetzt. Dabei wird die Alpha Version täglich vom Master Branch und die Beta Version täglich vom Release Branch veröffentlicht (Rossi et al., 2016).

Klepper, Krusche, Peters, Bruegge und Alperowitz (2015) beschreiben mehrere Release Management Workflows für mobile Applikationen in Abhängigkeit von Projektgröße, Komplexität, Personaleinsatz, Zeitachse und Prioritäten. Dabei fällt auf, dass die Aktivitäten, in unserem Fall die Schritte einzelner Stages, immer gleich sind und lediglich die Trigger variieren. Um eine möglichst generische Lösung zu schaffen, sollen die Trigger daher je nach Anliegen frei konfigurierbar sein, auch wenn manche Trigger in gewissen Stages unzweckmäßig erscheinen.

3.1 Commit Stage

Der erste Schritt der Commit Stage ist das *Kompilieren* und stellt sicher, dass die Applikation fehlerfrei übersetzbar ist (Humble & Farley, 2010). Hierfür wird Java Quellcode zunächst mit einem Java-Compiler übersetzt und anschließend von einem Cross-Assembler für die Dalvik virtuelle Maschine angepasst. Im Bereich Android wird aktuell ausschließlich Gradle als Build Management Tool genutzt (Android Developers, 2017b).

Als nächstes folgt die Ausführung der Unit-Tests. Android unterscheidet zwischen Unit-Tests, die ohne Abhängigkeiten direkt innerhalb der Java virtuellen Maschine ausgeführt werden und sogenannten *Instrumented Unit-Tests*, welche ein reales Gerät oder einen Emulator benötigen. Da die Android Build Tools lediglich zwischen diesen beiden Testarten unterscheiden, werden Instrumented Unit-Tests in nachfolgende Stages verlagert, auch wenn Humble und Farley (2010) kleinere Integrationstests, welche die Funktionalität das System auf einer technischen Ebene sicherstellen, eher in der Commit Stage ansiedeln würden. Ein nützliches Testing Framework unter Android ist Robolectric[2], mit dessen Hilfe das Android Framework emuliert wird und Unit-Tests direkt in der Java virtuellen Maschine auf dem lokalen Rechner ausgeführt werden (Android Developers, 2017e).

Ein weiterer Schritt ist die statische Code-Analyse, die zur Übersetzungszeit stattfindet (Humble & Farley, 2010). Android Lint ist ein bekanntes Tool, das bereits in den Android Build Tools enthalten ist. Im Internet finden sich weitere Tools

[2]http://robolectric.org/

wie PMD, Checkstyle oder FindBugs. Humble und Farley (2010) sowie Niederwieser (2013) nehmen zusätzlich die Auswertung der Testabdeckung in die Commit Stage auf.

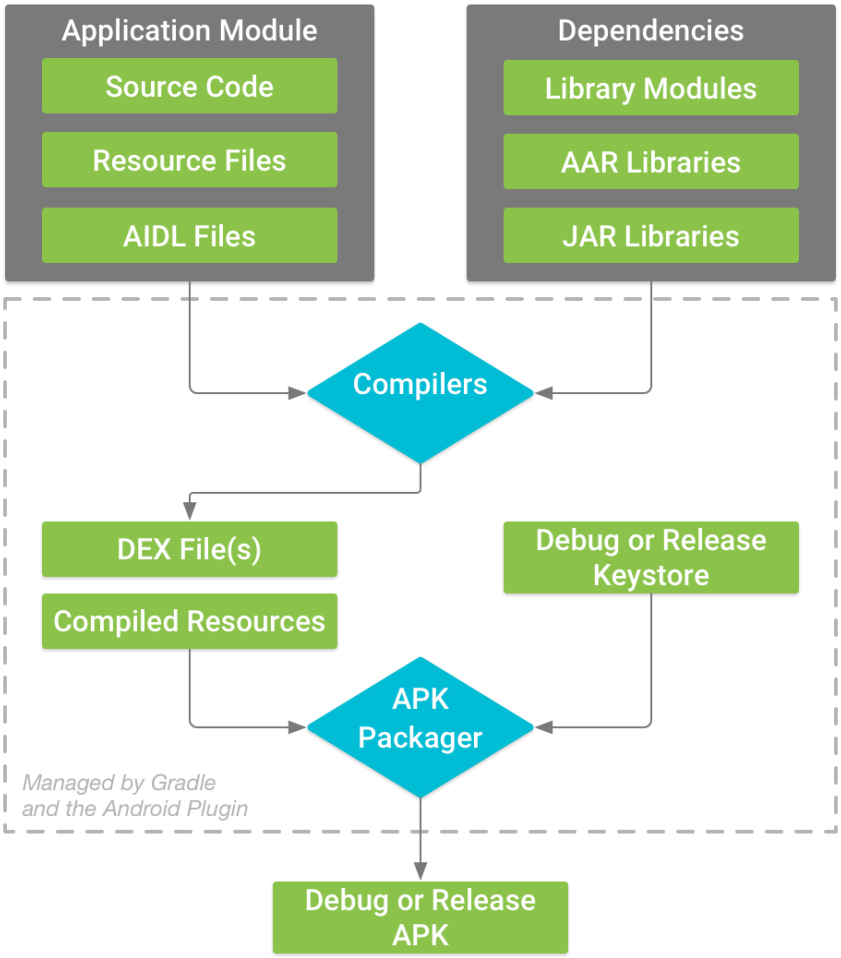

Abbildung 3.2: Android Build Prozess (Android Developers, 2017b).

Ein wichtiger Schritt der Commit Stage ist die Erstellung von APK-Dateien, die archiviert und für nachfolgende Stages oder auch Pipelines verwendet werden. Abbildung 3.2 zeigt den Build Prozess eines Android App Modules. Der Kompiler übersetzt den Quellcode der Applikation zu Dalvik Executable (DEX) Dateien und alle weiteren Abhängigkeiten zu kompilierten Ressourcen. Anschließend kombiniert der APK Packager diese Kompilate zu einer APK-Datei. Um eine APK auf einem Android Gerät zu installieren oder diese in einen App Store zu veröffentlichen, muss sie zunächst signiert werden. Dabei verwendet der APK Packager entweder einen Debug oder einen Release Keystore. Android Studio konfiguriert neue Projekte automatisch mit einem Debug Keystore. Dieser wird zu Test- und Analysezwecken verwendet. Soll die Applikation veröffentlicht wer-

den, so muss sie mit einem Release Keystore signiert werden. Wie ein Release Keystore erzeugt wird und welche unterschiedliche Möglichkeiten existieren, um eine Applikation zu signieren, wird in Android Developers (2017d) erklärt. Bevor die finale APK erstellt wird, nutzt der Packager ein Optimierungstool namens zipalign, welches im Android SDK enthalten ist, sodass die Applikation weniger Speicher beansprucht, wenn sie auf einem Android Gerät ausgeführt wird. Das Ergebnis dieses Build Prozesses ist mindestens eine Debug APK. Wird die Applikation signiert, entsteht zusätzlich eine Release APK. Darüber hinaus wird von Gradle eine Test-APK erstellt, sobald Instrumented Unit-Tests in den Quellen der Applikation vorhanden sind. Diese wird zusammen mit der Debug APK auf ein reales Gerät oder einen Emulator geladen, um die enthaltenen Tests auszuführen (Android Developers, 2017b).

Der letzte Schritt unserer Commit Stage ist das Sicherstellen der *App Traceability*. Zum einen muss jede im Play Store veröffentlichte Version einer Android-Applikation einen höheren Versionscode als seine Vorgänger aufweisen, zum anderen sollen alle Versionen der Applikation einem Build der Continuous Delivery Pipeline zugeordnet werden können. Dadurch wird sichtbar, wann eine spezielle APK gebaut wurde und was sie enthält. Im Android System ist die Versionsnummer einer installierten Applikation einsehbar, so dass Beta-Tester beispielsweise die exakte Versionsnummer einer Testapplikation ablesen können und diese einem Build zugeordnet werden kann. Ab hier kann der Stand dieser Applikation bis zu einer gewissen Commit-Id zurückverfolgt werden. Orr (2014) und Niederwieser (2013) nutzen die Jenkins-Build-Nummer, eine Umgebungsvariable die bei jedem Build inkrementiert wird, um einen rückverfolgbaren Versionscode für eine APK zu erzeugen.

3.2 Acceptance Test Stage

Zur Ausführung der Acceptance Test Stage müssen die zuvor erzeugten Debug- und Test-APKs auf ein mobiles Endgerät oder einen Emulator installiert werden. Anschließend werden die funktionalen Tests ausgeführt. Funktionale Tests sollen die Applikation aus Sicht der Kunden beziehungsweise der Endnutzer testen. Im Bereich von mobilen Applikationen sind dies meist UI-Tests. Aber auch Unit-Tests, die das Android Framework benötigen, oder Integrationstests von Komponenten, die nicht direkt mit dem Benutzer interagieren, werden hier getestet. Services und Content Provider stellen typische Beispiele für diese Komponenten dar.

Wichtig ist bei den funktionalen Tests, dass sie in verschiedenen Umgebungen, also auf verschiedenen mobilen Endgeräten ausgeführt werden. Sobald eine Applikation veröffentlicht wird, steht sie mehreren tausend Endgeräten zur Verfügung

und sollte auf all diesen Geräten fehlerfrei funktionieren. Um eine möglichst hohe Testabdeckung zu erreichen, beschränkt sich Orr (2014) auf drei Größen in einem Android System und testet deren Kombinationsmöglichkeiten. Diese sind die Android Betriebssystem Version, die Bildschirmgröße und Auflösung sowie die Sprachumgebung einer Applikation.

Ein beliebtes Framework zum Durchführen von UI-Tests ist Espresso. Falls die Applikation mit dem Android System interagiert, kann UI Automator, ein weiteres UI-Testing Framework, verwendet werden (Android Developers, 2017e).

3.3 Performance Testing Stage

In dieser Stage sollen alle nicht funktionalen Anforderungen getestet werden. Rossi et al. (2016) fukussieren sich bei den Performance-Tests auf die Geschwindigkeit, den Speicherverbrauch und die Batterieeffizienz einer Applikation.

In den Android Build Tools ist bereits ein Kommandozeilenwerkzeug namens Monkey enthalten, um Anwendungen auf Abstürtze zu prüfen. Es läuft auf einem Emulator oder realen Gerät und erzeugt Pseudozufallsströme von Benutzerereignissen wie beispielsweise Klicks, Berührungen oder Gesten. Dabei kann Monkey auf eine bestimmte Application-ID beschränkt werden, um nur die eigene App zu testen (Geiss, 2012).

Ein weiterer Schritt dieser Stage könnte die Messung der UI-Performance sein. Dies soll sicherstellen, dass Benutzerinteraktionen mit der Applikation als flüssig wahrgenommen werden. Ein objektves Maß stellen hier die Frames per second dar, die bei einer Android-Applikation konstant 60 betragen sollte. Android Developers (2017f) beschreiben Tools zur Messung der Benutzeroberflächenleistung und zusätzlich noch einen Ansatz zur Integration dieser Messungen in eigene Testpraktiken.

3.4 Testing as a Service Stage

Diese Stage lagert die funktionalen und nicht funktionalen Tests einer Android-Applikation auf externe Infrastrukturen aus. Testing as a Service Stage ist kein Begriff, der in der Literatur zu findet ist, sondern wurde innerhalb dieser Arbeit so getauft. Für die beiden vorherigen Stages wurden jeweils Emulatoren oder reale Android-Geräte benötigt. Im Gegensatz dazu führt diese Stage die bisherigen Testaktivitäten auf speziell dafür ausgelegten Servern aus. Dadurch lässt sich neben den Anschaffungskosten für Android Hardware und der Rechenleistung für Emulatoren auch viel Aufwand und Zeit für Testaktivitäten unter Android sparen

(Zachow, 2016). Einen beliebten cloud-basierten Testservice bietet das Firebase Test Lab, mit dessen Hilfe die Applikation auf echten Geräten, die in Rechenzentren von Google laufen, getestet werden können. Dieser Service kann auch in Anspruch genommen werden, falls keine Tests für eine Android-Applikation vorliegen (Firebase, 2017). Eine cloudbasierte Teststrategie wird auch von Rabenhorst und Steffens (2016), neben einer Reihe von anderen Stages und Testaktivitäten, eingesetzt.

Die Aktivitäten dieser Stage beschränken sich auf den Upload einer Debug und gegebenenfalls auch einer Test APK auf eine externe Testinfrastruktur sowie dem anschließenden Download der Testergebnisse, wie Logs, Videos oder auch Screenshots.

3.5 Custom Deployment Stage

Zusätzlich zu den automatisierten Tests findet häufig eine manuelle Qualitätssicherung statt. Hierzu wird die APK-Datei auf einen lokalen Server hochgeladen und kann anschließend von dem Validierungspersonal heruntergeladen und über den Android Paketmanager installiert werden (Klepper et al., 2015). Der Unterschied zur vorherigen Stage liegt im manuellen Testen der Applikation sowie dem Upload der Applikation auf einen internen Server. Durch die Custom Deployment Stage kann ebenso die Usability einer App untersucht werden.

3.6 Alpha Release, Beta Release und Production Stages

Diese drei Stages gehen einen Schritt weiter und veröffentlichen eine Android-Applikation in den Alpha, Beta oder Production Track von Google Play. Zuvor muss allerdings ein Eintrag für die Applikation in Google Play erstellt werden. Das Feedback von Nutzern der Alpha und Beta Versionen einer Applikation hat keinerlei Auswirkungen auf die öffentliche Bewertung dieser App. Die Alpha und Beta Release Stages stellen die Applikation einer frei wählbaren Nutzergruppe zur Verfügung. Meistens wird eine Alpha Version Testnutzern innerhalb einer Organisation zur Verfügung gestellt, während ein Beta Release die Applikation auf eine größere Anzahl von Testnutzern ausweitet. Anschließend wird die Applikation in der finalen Production Stage allen Android Nutzern verfügbar gemacht. Dies geschieht mit Hilfe von Google Play, dem Standard App Store von Android (Google Developers, 2017).

Alpha und Beta Release Stages müssen zur Veröffentlichung nicht Google Play verwenden, da die Applikation während der letzten Stage aber ohnehin auf Google Play landet und die Applikation somit einfach in den Production Track verschoben werden kann, wird die Nutzung der Alpha und Beta Tracks in Google Play empfohlen. Zusätzlich werden frühzeitig Fehler erkannt, die den Upload zu Google Play tangieren, wie beispielsweise Größenbeschränkungen der APK oder auch die Geräte- und Bildschirmfilterung (Hambrick, 2015).

4 Architektur und Design

Dieses Kapitel beschreibt das Design der entwickelten Software, namens *Android Continuous Delivery DSL Engine*. Im Folgenden wird sie auch als *Continuous Delivery Engine* oder *DSL Engine* bezeichnet. Dabei wird zunächst die Gesamtarchitektur grob betrachtet und anschließend detailreicher auf einzelne Schichten dieser DSL Engine eingegangen. Die oberste Schicht nutzt die zuvor gezeigten Anforderungen, um daraus eine domänenspezifische Sprache abzuleiten. Für darunterliegende Schichten werden Entwurfsmuster gezeigt, die zu einem späteren Zeitpunkt der Arbeit zur Implementierung des semantischen Modells sowie des Codegenerators genutzt werden sollen.

4.1 Die Gesamtarchitektur

Dieser Abschnitt des Design-Kapitels soll zunächst einen abstrakten Überblick über die Arbeitsweise der DSL Engine sowie deren Aufgaben vermitteln. Die Continuous Delivery Engine soll in Java entwickelt werden und als domänenspezifische Sprache wird eine interne DSL verwendet. Das heißt, Anwender dieser DSL Engine beschreiben ihre Continuous Delivery Pipeline mit Hilfe einer Java API, die im Laufe dieses Kapitels genauer spezifiziert wird. Die vom Anwender erzeugte Beschreibung wird im Folgenden auch als *DSL-Skript* bezeichnet. Als Buildserver für die Android-Applikationen soll Jenkins zum Einsatz kommen. Um eine möglichst flexible Lösung zu schaffen, läuft Jenkins innerhalb eines leichtgewichtigen Containers. Als Containertechnologie wird Docker verwendet.

Abbildung 4.1 visualisiert den Ablauf, wie aus einem DSL-Skript unter Verwendung der Continuous Delivery Engine ein Buildserver erstellt und mit den erzeugten Artefakten konfiguriert wird. Dabei stellt das Rechteck mit den abgerundeten Ecken die DSL Engine dar, während die blauen Rechtecke Docker Container zeigen. Die Continuous Delivery Engine bekommt als Input ein DSL-Skript übergeben, welches intern verarbeitet wird. Innerhalb der DSL Engine gibt es eine Komponente namens Generator, die XML Artefakte erzeugt. Diese XML Files lassen sich in Pipeline Jobs und Credentials für den Jenkins Buildserver katego-

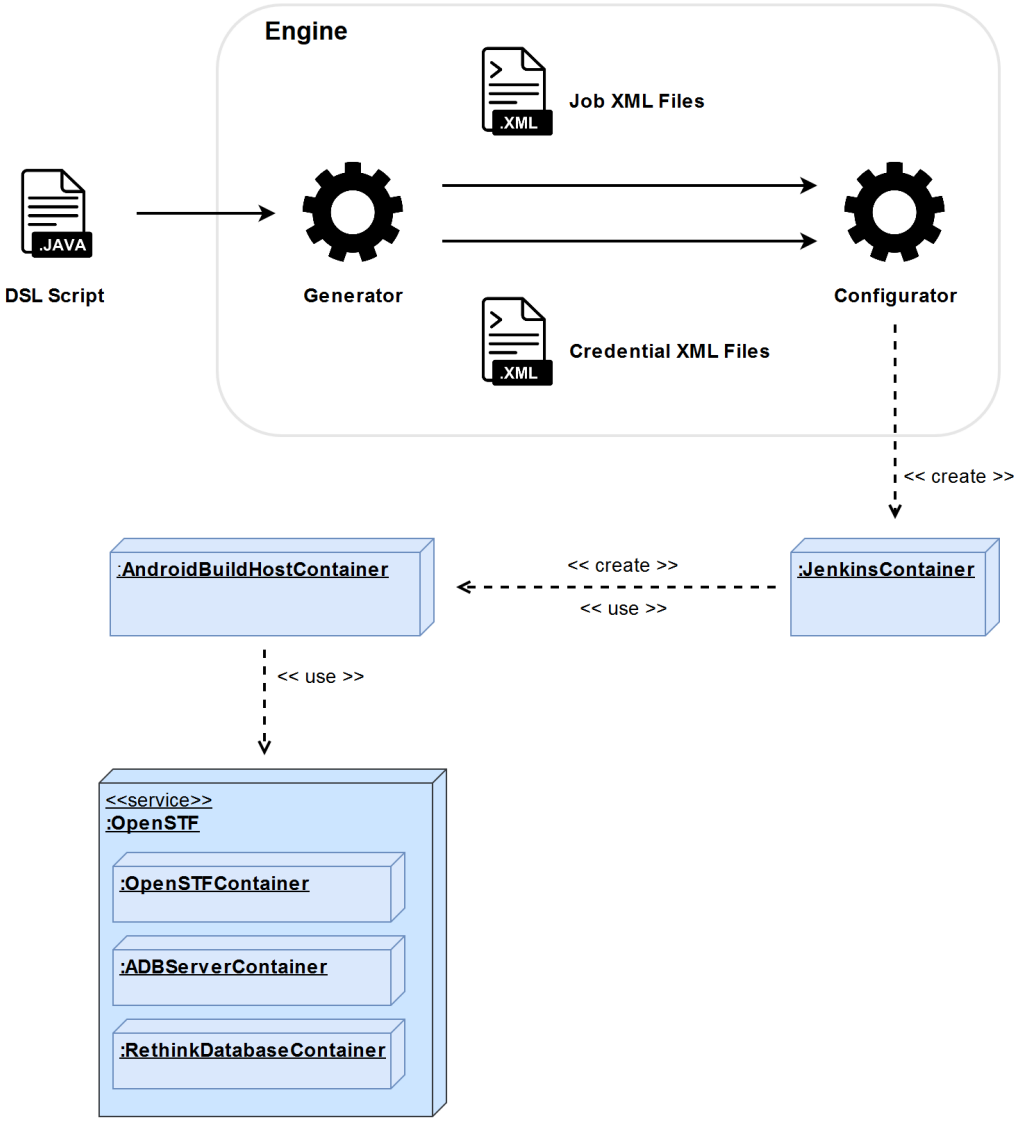

Abbildung 4.1: Arbeitsablauf der Continuous Delivery Engine.

risieren. Anschließend erzeugt eine weitere Komponente, der Konfigurator, einen Jenkins Docker Container und startet diesen. Sobald der Jenkins Container läuft, fügt der Konfigurator die erzeugten Artefakte in Jenkins ein. Zu diesem Zeitpunkt ist Jenkins voll konfiguriert und es kann bereits auf die Jenkins Weboberfläche zugegriffen werden.

Wird während des Build Vorganges Android SDK Funktionalität benötigt, wird ein eigener Docker Container namens Android Build Host erzeugt, der die Android SDK Build Tools und die für die Applikation notwendigen SDK Platform Tools bereitstellt. Dieser Container wird nach Abarbeitung der Pipeline-Instanz wieder gelöscht und für jeden Build neu erzeugt.

Werden reale Endgeräte zum Testen der Applikation genutzt, so können diese mit openSTF in das System eingebunden werden. STF steht für Smartphone Test Farm und ist ein Open Source Projekt, um reale Android-Geräte über den Browser fernzusteuern. Mit STF werden alle Android Betriebssystemversionen unterstützt (Kinnunen & Brunner, 2017). Oftmals ist es physikalisch nicht möglich, reale Android Geräte an den Continuous Delivery Server anzuschließen. Deshalb soll der Android Build Host openSTF zur Anbindung entfernter mobiler Endgeräte nutzen.

4.2 Die Schichten der Android Continuous Delivery DSL Engine

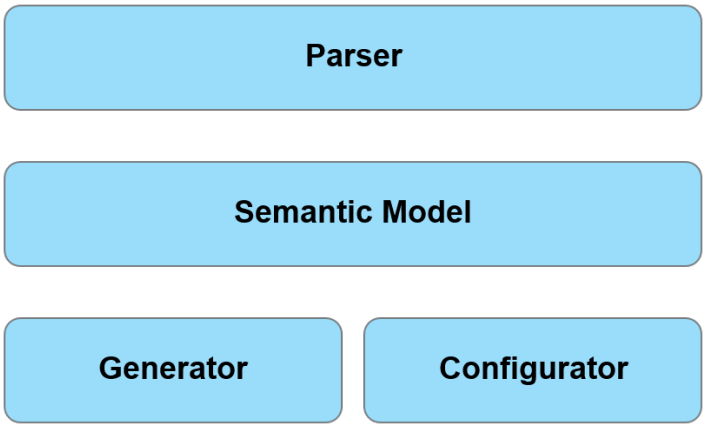

Abbildung 4.2: Schichten der DSL Engine.

Die Android Continuous Delivery DSL Engine selbst besteht aus vier Hauptkomponenten, die sich auf drei Schichten aufteilen lassen. Abbildung 4.2 zeigt

diese Schichtenarchitektur. Die oberste Ebene stellt der Parser dar, der das DSL-Skript liest und daraus ein Objektmodell erzeugt. Eine grundlegende Designentscheidung ist es, den Parser völlig losgelöst von darunter liegenden Schichten zu entwickeln. Aktuell soll eine interne DSL zur Problembeschreibung verwendet werden. Falls die Syntax dieser DSL grundlegend verändert oder sogar eine externe DSL angeboten wird, sollen andere Schichten der DSL Engine nicht davon betroffen sein. So wird es möglich, mehrere unterschiedliche domänenspezifische Sprachen parallel zu unterstützen. Die einzige Anforderung an die neuen Parser ist, die gleichen Objekte des semantischen Modells zu erzeugen.

Die mittlere Ebene besteht aus dem semantischen Modell, welches eine logische Struktur des Buildserver inklusive der Credentials und Continuous Delivery Pipelines mit Hilfe von Java Objekten abbildet.

Von dieser Schicht losgelöst existiert eine weitere Schicht, die sowohl den Generator, welcher XML Artefakte erzeugt, als auch den Konfigurator, welcher für die Handhabung der Docker Container und die Konfiguration des Buildservers verantwortlich ist, enthält. Auch diese beiden Komponenten sind vom Herzstück der Continuous Delivery Engine, dem semantischen Modell, entkoppelt, um die genutzten Technologien wie Docker und Jenkins einfach ersetzbar zu machen und Unabhängigkeit von bestehenden Technologien zu gewährleisten. Denkbar wäre ähnlich wie in der obersten Schicht auch die Unterstützung eines zweiten, alternativen Codegenerators, der die Declarative-Pipeline-Syntax innerhalb von Jenkins unterstützt, oder auch einen weiteren Konfigurator, der eine andere Containertechnologie verwendet.

Um einen anderen Buildserver zu unterstützen, müsste somit lediglich die untere Schicht der Applikation, genauer gesagt der Generator, getauscht werden und ein anderer Docker-Container genutzt werden.

4.3 Der DSL-Parser

Wie bereits im vorherigen Abschnitt erläutert, soll der Parser möglichst unabhängig von anderen Komponenten der DSL Engine sein und eine interne DSL, basierend auf Java, unterstützen. Die interne DSL soll ihre Funktionalität über ein Fluent Interface bereitstellen. Fluent Interface ist ein Konzept zur Erstellung von objektorientierten APIs, bei dem sich der geschriebene Quellcode an Sätzen der natürlichen Sprachen anlehnt und dadurch verständlicher wird. Im Gegensatz zu herkömmlichen Command-Query APIs, welche Objekte isoliert betrachten, wird bei Fluent Interface sehr linguistisch gedacht, um Objekte durch natürliche Satzbildung miteinander zu verweben. Unsere interne DSL soll *Method Chaining* zur Realisierung des Fluent Interfaces nutzen. Bei Method Chaining gibt jede modi-

fizierende Methode sein Host Objekt zurück, so dass Methodenaufrufe zu einer einzelnen Anweisung verkettet werden können, ohne Variablen zur Speicherung von Zwischenergebnissen zu benötigen. Es gibt allerdings noch andere Muster, wie beispielsweise *Nested Functions* oder *Function Sequence*, zur Umsetzung einer Fluent Interface API (Fowler, 2010).

Die unterhalb des Parsers liegende Schicht, das semantische Modell, nutzt Command-Query Interface Objekte. Um die unterschiedlichen Eigenschaften von Fluent Interface und Command-Query API nicht zu vermischen und die Parsing Schicht von dem semantischen Modell klar zu trennen, wird ein Entwurfsmuster namens Expression Builder verwendet. Dieses bietet ein Fluent Interface als separate Schicht über einer regulären API an. Abbildung 4.3 zeigt beispielhaft wie mit Hilfe eines Expression Builder aus einer internen DSL die Objekte eines semantischen Modells erzeugt werden.

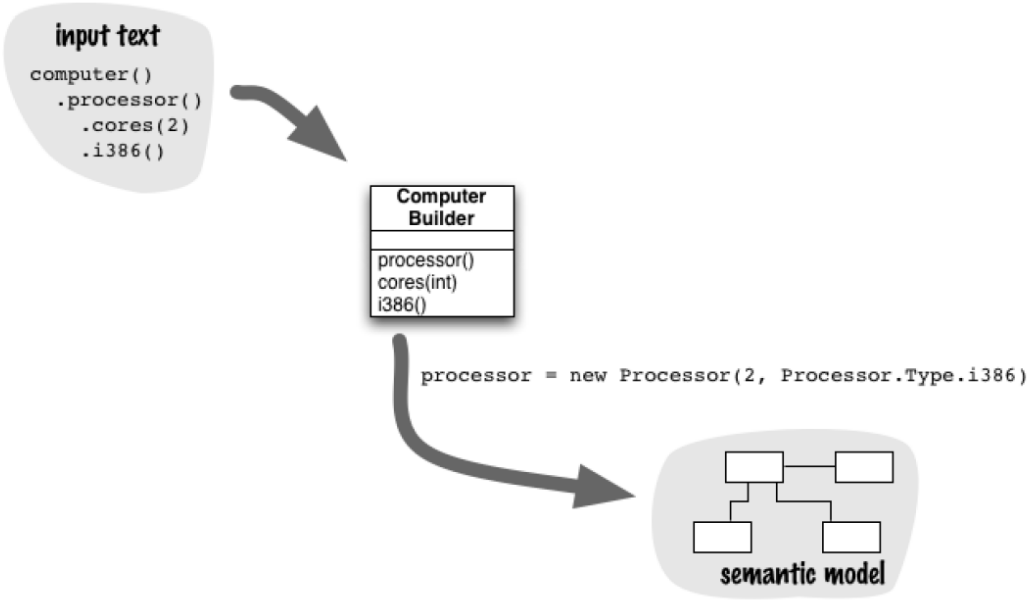

Abbildung 4.3: Das Expression Builder Entwurfsmuster (Fowler, 2010).

Kommen wir nun zur Syntax, die für die DSL Engine genutzt werden soll. Zunächst muss das Repository angegeben werden, in dem sich die Android-Applikation befindet. Häufig gibt es bei Method Chaining das sogenannte Finishing Problem. Das heißt, es gibt keinen natürlichen Endpunkt der Methodenkette. Abhilfe schafft hier eine Marker-Methode, in unserem Falle `generate()`, um das Ende des DSL-Skriptes zu kennzeichnen. Listing 4.1 zeigt den äußeren Rahmen unseres DSL-Skriptes, welches mit der Angabe des Repositories startet und mit Aufruf der `generate`-Methode endet.

```
1 .repo("https://github.com/M-Fischer/android-playground.git")
2 .doFurtherConfiguration() // for Buildserver and Pipeline
3 .generate()
```

Listing 4.1: DSL-Syntax zur Bestimmung des App-Repositories.

Als nächstes werden die Eigenschaften des Buildserver beschrieben. Listing 4.2 zeigt die Syntax zur Buildserver-Konfiguration. Hierfür gibt es eine vorgeschriebene `buildServer`-Methode, in der die Zugangsdaten für den Administrator des Buildservers festgelegt werden. Alle weiteren Methoden des Listings sind optional. Mit `signing` werden sowohl das Keystore als auch das Key Passwort zur Signierung einer Applikation festgelegt. `ftpCredentials` beschreibt Benutzername und Passwort des FTP Servers, der in der Custom Deployment Stage verwendet werden soll. `openSTF` spezifiziert letztendlich, wo der openSFT Service läuft. Zusätzlich muss hier noch ein Access Token angegeben werden, um innerhalb des Android Build Host Containers auf den openSTF Service zuzugreifen zu können.

```
4 .buildServer("serverUser", "serverPassword")
5     .signing("keystorePw", "keyPw")
6     .ftpCredentials("ftpUser", "ftpPw")
7     .openSTF(OPEN_STF_URL, BEARER_TOKEN)
```

Listing 4.2: DSL-Syntax zur Buildserver-Konfiguration.

Das Kernstück der internen DSL stellt die Pipeline Konfiguration dar. Die Konfigurationsmöglichkeiten leiten sich aus den gesammelten Anforderungen von Kapitel 3 ab. Listing 4.3 zeigt beispielhaft alle Methoden, die zur Konfiguration einer Continuous Delivery Pipeline verwendet werden können.

Den Einstiegspunkt in die Pipeline Konfiguration stellt die `pipeline`-Methode dar, in der der Pipeline ein Namen gegeben wird. Die Commit Stage ist verpflichtend und legt im Gegensatz zu allen anderen Stages keinen Trigger fest. Diese erste Stage wird bei jeder Änderung im Repository der Applikation angestoßen. Defaultmäßig wird in der ersten Stage eine Debug APK gebaut. Es können jedoch weitere Schritte, wie beispielsweise Unit-Testing, statische Code-Analyse oder das Erstellen einer releasefähigen Applikation beschrieben werden. Alle weiteren Stages werden mit Hilfe der `stage`-Methode spezifiziert und müssen sich zwischen einem manuellen, automatischen oder zeitgesteuerten Trigger für die jeweilige Stage entscheiden. Die Reihenfolge der Stages wird durch die Reihenfolge der `stage`-Methodenaufrufe festgelegt, unterliegt jedoch den Beschränkungen, die im Anforderungskapitel genannt wurden. Sowohl die Custom Deployment Stage als auch die Performance Testing Stage haben Folgemethoden, in denen zusätzliche Konfigurationen wie die Adresse des FTP Servers oder die Anzahl der

```
8  .pipeline("pipelineName")
9      .stage(COMMIT_STAGE).
10         .step(COMPILE_RELEASE)
11         .step(UNIT_TEST)
12         .step(LINT)
13     .stage(ACCEPTANCE_TEST_STAGE, Trigger.AUTOMATIC)
14     .parallel()
15         .seqStage(PERFORMANCE_TESTING_STAGE, Trigger.AUTOMATIC)
16             .eventCount(1500)
17         .seqStage(CUSTOM_DEPLOYMENT_STAGE, Trigger.AUTOMATIC)
18             .location("localhost")
19         .stage(TESTING_AS_A_SERVICE_STAGE, Trigger.MANUAL)
20     .endParallel()
21     .stage(ALPHA_RELEASE_STAGE, Trigger.AUTOMATIC)
22     .stage(BETA_RELEASE_STAGE, Trigger.MANUAL)
23     .stage(PRODUCTION_STAGE, Trigger.TIMETRIGGER, 7, DAYS)
```

Listing 4.3: DSL-Syntax zur Beschreibung der CD Pipeline.

zufälligen generierten Events festgelegt werden können. Mit Hilfe der `parallel` und `endParallel`-Methoden wird ein Block gekennzeichnet, innerhalb dessen alle Stages parallel ablaufen. Sollen innerhalb des Parallelblockes einzelne Stages, wie in Listing 4.3 die Performance Testing und Custom Deployment Stages, streng nacheinander abgearbeitet werden, kann dies mit der Methode `seqStage` und den entsprechenden Übergabeparametern realisiert werden. Auf die Schachtelung von sequenziellen Stages innerhalb des Parallelblockes wird im nachfolgenden Abschnitt, dem Design des semantischen Modells, genauer eingegangen. Die Aufteilung in zwei Pipelines soll transparent für den Anwender erfolgen und muss nicht explizit im DSL-Skript gekennzeichnet werden.

Für mögliche Stages und Trigger werden Konstanten definiert, welche die Autovervollständigung dieser durch moderne integrierte Entwicklungsumgebungen ermöglicht. Mit Hilfe von Interfaces und generischen Typen sollen die Möglichkeiten der Methodenaufrufe fixiert werden, sodass mögliche Fehlkonfigurationen bereits zur Compile-Zeit erkannt und somit die Applikation nicht übersetzt werden kann. Natürlich müssen weitere Prüfungen, wie beispielsweise die Reihenfolge der Stages während des Parsevorganges, also zur Laufzeit, stattfinden. Ein komplettes Listing zur Beschreibung der internen DSL-Syntax befindet sich im Anhang.

4.4 Das semantische Modell

Die Architektur des semantischen Modells lässt sich am besten mit einem Klassendiagramm visualisieren. Abbildung 4.4 zeigt den Entwurf des semantischen Modells in Form eines UML-Klassendiagramms. Da dieses Modell weitestgehend aus Getter- und Setter-Methoden bestehen wird, wurden diese zur Vereinfachung weggelassen. Auch die gezeigten Attribute können unvollständig sein und spiegeln nur die gesammelten Informationen aus den vorherigen Abschnitten wider.

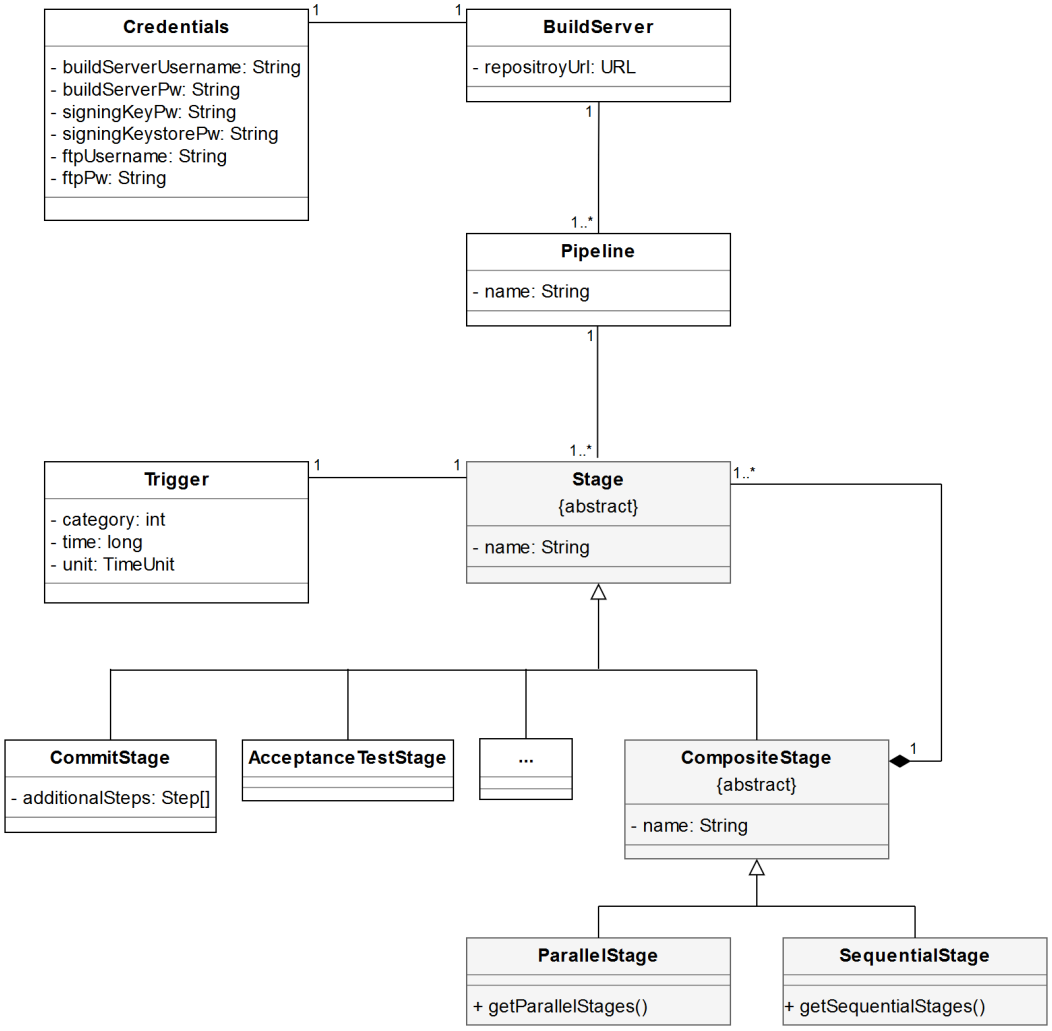

Abbildung 4.4: Klassendiagramm des semantischen Modells nach UML 2.0.

Das Wurzelelement ist die `BuildServer`-Klasse, welche alle weiteren Klassen des semantischen Modells enthält. Hier besteht eine direkte Assoziation zur `Credentials` Klasse, welche alle benötigten Zugangsdaten der DSL Engine enthält, so-

wie zu mindestens einer Pipeline. Das semantische Modell soll möglichst generisch sein, um später neben Android auch weitere Domänen zu unterstützen. Daher kann die `BuildServer`-Klasse beliebig viele Pipelines enthalten. Für die Aufteilung in zwei Pipelines und das korrekte Erzeugen dieser Objekte ist der Parser verantwortlich.

Jede Pipeline-Instanz besitzt wiederum mindestens eine Stage und jede Stage hat genau einen Trigger. Auch die Commit Stage besitzt einen Trigger, selbst wenn dieser nicht explizit im DSL-Skript niedergeschrieben wird. Hier soll ein nach außen hin nicht sichtbarer Trigger zum Einsatz kommen. Für jede Stage existiert eine konkrete Unterklasse, in der spezifische Attribute und Methoden für diese Stage enthalten sind. Die Commit Stage besitzt beispielsweise zusätzlich ausführbare Schritte. Zudem kann das Modell leicht um neue Stages erweitert werden, indem diese einfach von `Stage` erben.

Die grau gekennzeichneten Klassen benutzen das Kompositum Entwurfsmuster, ein Strukturmuster von Gamma, Helm, Johnson und Vlissides (1995), um die Stage Objekte zu Baumstrukturen zusammenzufügen. Somit können parallel ablaufende Stages modelliert werden. Auch hier ist das semantische Modell wieder flexibel ausgelegt, da alle Stages parallel ablaufen könnten, der Parser jedoch bei Erzeugung und Verdrahtung der Stage Objekte prüft, ob diese für den aktuellen Anwendungsfall wirklich parallel ausführbar sind.

Um die Möglichkeiten der Stage-Anordnungen zu beschreiben, wird in Abbildung 4.5 eine Pipeline gezeigt, in der die dritte, vierte, fünfte und sechste Stage parallel ablaufen, die vierte und fünfte Stage jedoch sequentiell innerhalb dieses Parallelblockes ausgeführt werden. Es soll möglich sein, sequenzielle Ausführungen von Stages innerhalb eines Parallelblockes zu verwenden. Diese sequenzielle Ausführung kann beispielsweise durch Datenabhängigkeiten zwischen den beteiligten Stages begründet sein. Darunter ist ein Objektdiagramm zu erkennen, welches die Struktur der oberen Pipeline widerspiegelt. Für das Objektdiagramm wurden die Klassen des semantischen Modells auf Seite 38 hergenommen. Auch wenn dieser Anwendungsfall für die Android Domäne sehr konstruiert wirkt, könnten spätere Erweiterungen ein solches Verhalten benötigen und sollten deshalb sowohl im Design als auch der Umsetzung der DSL Engine berücksichtigt werden.

4.5 Der Codegenerator

Die Aufgabe des Codegenerators besteht darin, das semantische Modell zu lesen und daraus einen Zielcode für den Jenkins Container zu generieren. Dabei gibt es zwei Artefakt-Typen, die generiert werden müssen. Zum einen Credentials, die innerhalb der Pipeline genutzt werden sollen, und zum anderen die Jenkins

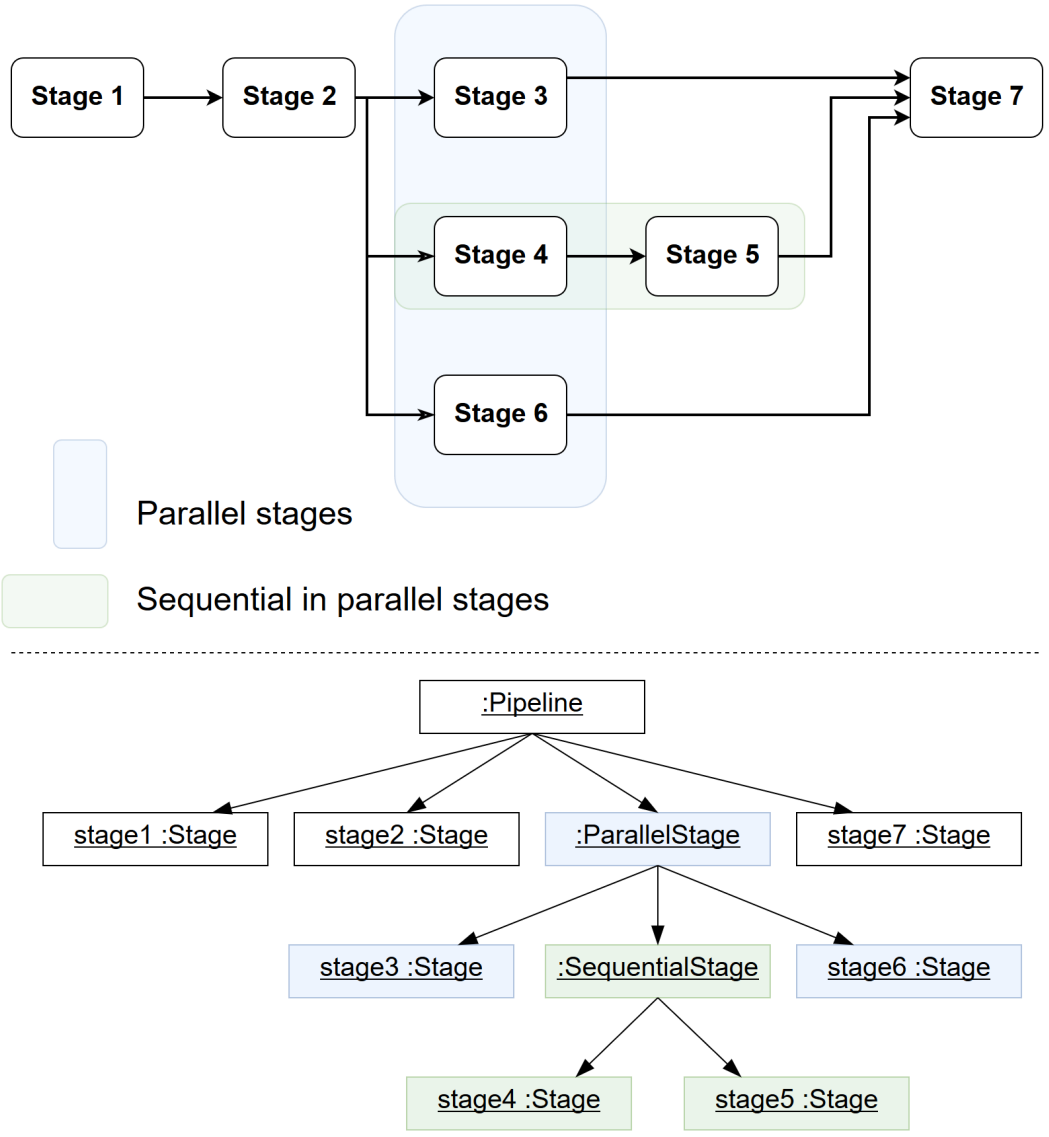

Abbildung 4.5: Pipeline und zugehöriges Objektdiagramm des Kompositum Entwurfsmusters.

Job Beschreibungen. Beides wird in Jenkins mit Hilfe von XML Datenstrukturen repräsentiert. Da keine explizite Repräsentation des semantischen Modells in unserer Zielumgebung vorliegen soll, wird Model Ignorant Generation als Ansatz für die Zielcodeerstellung gewählt.

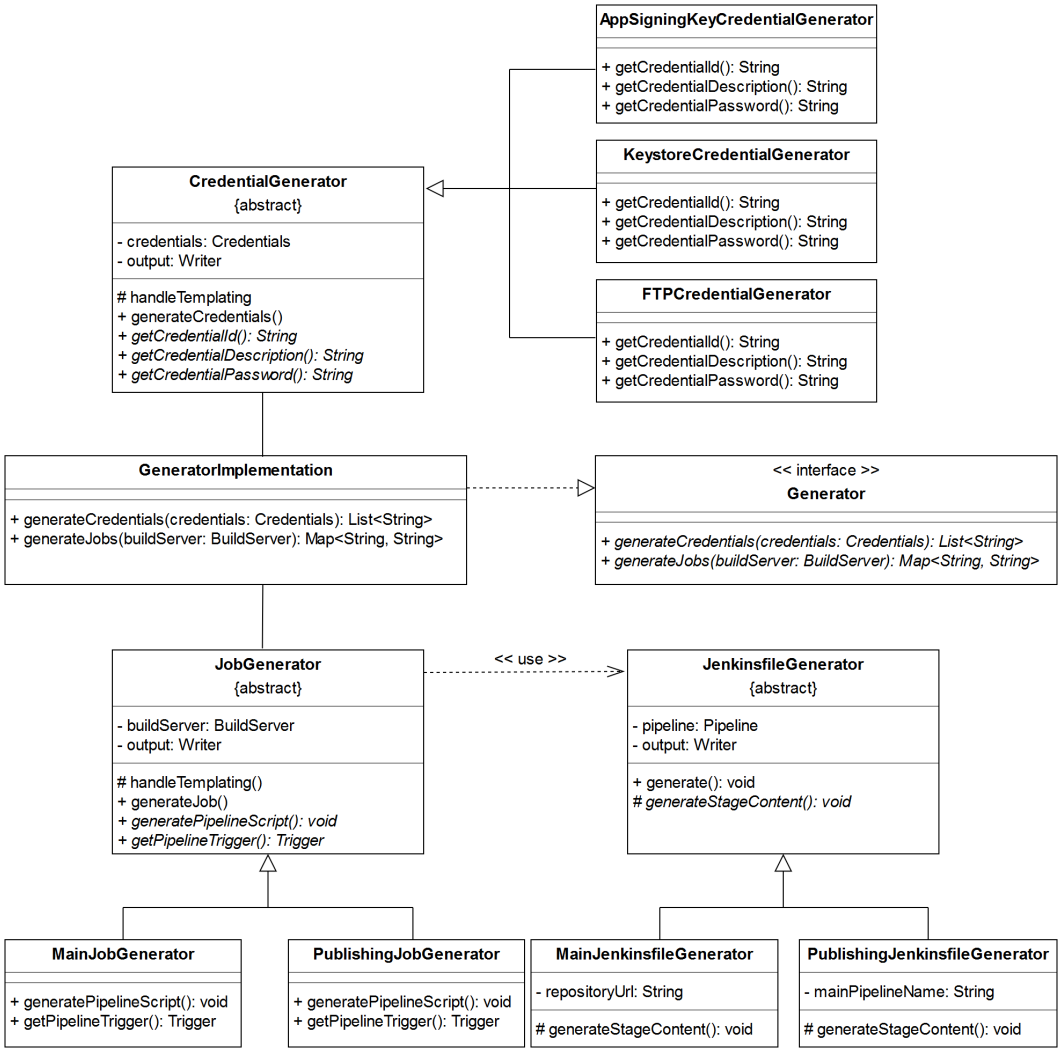

Abbildung 4.6: Klassendiagramm des Codegenerators nach UML 2.0.

Abbildung 4.6 zeigt ein Klassendiagramm, das die Generator-Komponente beschreibt. Den Einstiegspunkt stellt das Generator Interface, welches zwei Methoden, eine zur Erzeugung der Credentials und eine zur Joberstellung, spezifiziert. Beide Methoden bekommen das semantische Modell oder wie im Falle der Credentials eine Untermenge davon übergeben und liefern eine Collection von Strings, welche die XML Dateien repräsentieren, zurück.

Zur Implementierung dieses Interfaces werden die abstrakten Klassen `CredentialsGenerator` und `JobGenerator` genutzt. Beide Klassen funktionieren in ähnlicher Weise und nutzten Templated Generation als Methode zur Generierung der XML Files. Dabei wird Zielcode erzeugt, indem das Ausgabefile manuell geschrieben und mit Hilfe einer speziellen Template-Sprache die variablen Teile dieses Ausgabefiles generiert werden. Templated Generation wird eingesetzt, falls der Zielcode zum einen aus viel statischem Inhalt besteht und zum anderen der dynamische Teil eine geringe Komplexität, also nur wenige Schleifen oder Verzweigungen aufweist (Fowler, 2010). Beides ist bei Erzeugung der XML-Dateien gegeben, weshalb hier Templated Generation zum Einsatz kommen soll.

Die Wahl der richtigen Template Engine muss noch getroffen werden, jedoch sollen sich die abstrakten Klassen `CredetialsGenerator` und `JobGenerator` um die Initialisierung und Handhabung der Template Engine kümmern, während deren Unterklassen als Embedment Helper fungieren. Dieses Entwurfsmuster kapselt komplexen Template-Code in eine Helper Klasse, sodass lediglich einfache Methodenaufrufe im Template selbst zurückbleiben. Ein wesentlicher Vorteil von Templated Generation gegenüber anderen Ansätzen der Codegenerierung ist, dass man bereits am Template erkennen kann, wie der Zielcode aussieht. Das Embedment Helper Muster soll dabei helfen, diese Klarheit zu bewahren und wird in Abbildung 4.7 visualisiert.

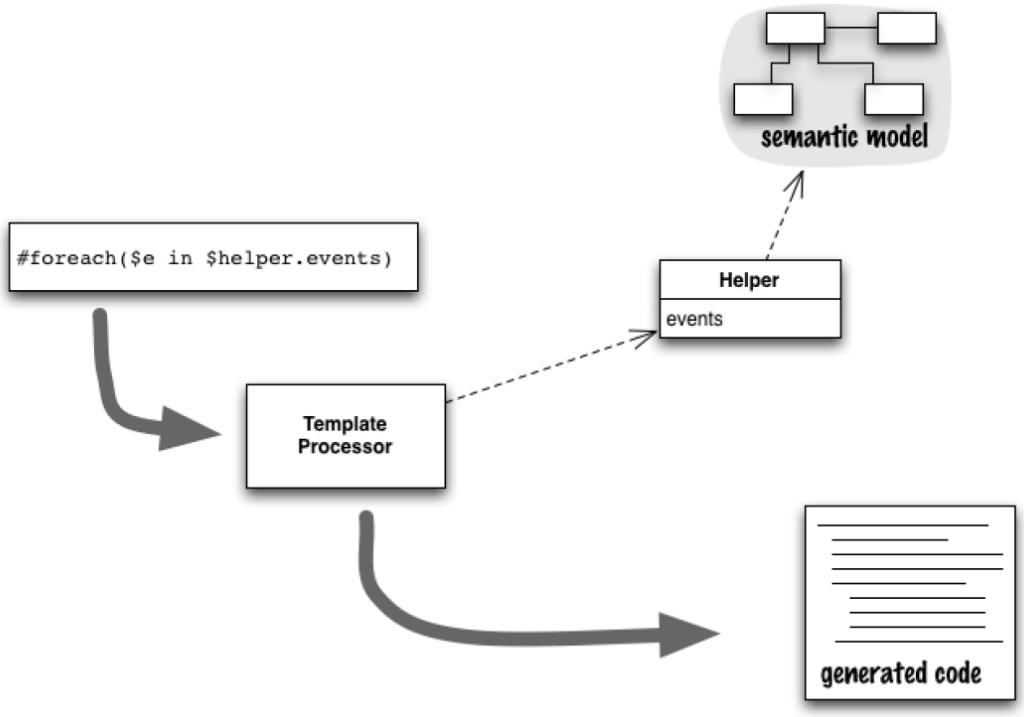

Abbildung 4.7: Das Embedment Helper Entwurfsmuster(Fowler, 2010).

Die Unterklassen des `CredentialGenerator` spiegeln jeweils einen gewissen Credential-Typ wider und bieten Methoden zur Einbettung von Id, Beschreibung und Passwort in das Template an.

Die `JobGenerator` Klasse teilt sich in zwei konkrete Job-Generatoren auf, welche die beiden unterschiedlichen Pipeline-Jobs in Jenkins abbilden. Beide müssen die Methoden `getPipelineTrigger` und `generatePipelineScript` ihrer abstrakten Oberklasse implementieren. Als Pipeline Trigger wird der Trigger der ersten Stage dieser Pipeline bezeichnet. Dieser muss genau wie das Jenkinsfile in die XML-Jobbeschreibung eingebettet werden.

Zur Erzeugung des Jenkinsfiles gibt es eine eigene abstrakte Klasse namens `JenkinsfileGenerator` innerhalb des Generators. Diese Klasse wird von den konkreten Job-Generator-Klassen genutzt, um die `generatePipelineScript`-Methoden zu realisieren. Auch hier wird der pipelinespezifische beziehungsweise stagespezifische Teil in Unterklassen ausgelagert, während der Hauptalgorithmus zur Erstellung des Jenkinsfiles in der Oberklasse verbleibt. Dieses Vorgehen ist ein bekanntes Verhaltensmuster namens *Schablonenmethode* von Gamma et al. (1995). Da innerhalb eines Jenkinsfiles nur wenige statische Codeteile enthalten sind und komplexe Strukturen wie Schleifen benötigt werden, um über die verfügbaren Stages in der angegebenen Reihenfolge zu iterieren, wird hier der Ansatz von Transformer Generator gewählt. Hierbei wird das semantische Modell, in unserem Fall die Pipeline Objekte, als Input verwendet und in Quellcode für die Zielumgebung, also in ein Jenkinsfile, transformiert.

Anhand des Klassendiagrammes lässt sich erkennen, dass der Quellcode des Generators enger als die restlichen Teile der Continuous Delivery Engine an den Zielcode gekoppelt ist, da hier die Aufteilung der Credentials und der Pipelines in konkrete Unterklassen stark an die Struktur der zu generierenden Jenkinsfiles erinnert.

4.6 Der Konfigurator

Der Konfigurator ist für zwei Aufgaben verantwortlich. Mit Hilfe eines Docker Clients für Java muss ein Jenkins-Container erstellt und gestartet werden. Hierfür wird ein Image auf Docker Hub hochgeladen, das von diesem Konfigurator genutzt werden kann. Die zweite Aufgabe besteht darin, die erzeugten XML-Artefakte des Generators per REST-Schnittstelle an den Jenkins Container zu senden. Für die HTTP-Anfragen soll ein üblicher HTTP-Client für Java genutzt werden. Für beide Aufgaben müssen die richtigen Java Client Tools gefunden und genutzt werden. Die Struktur des Konfigurators soll im Vergleich zu anderen Komponenten der DSL Engine recht trivial aufgebaut werden.

5 Implementierung

Dieses Kapitel hebt die Besonderheiten der Android Continuous Delivery DSL Engine in Bezug auf die Implementierung hervor. Die DSL Engine wurde komplett in Java entwickelt. Zur Unterstützung wurden Tools wie Eclipse[1], Gradle[2] und Git[3] sowie interne Bibliotheken genutzt. Die folgenden Abschnitte stellen keine vollständige Dokumentation dar, sondern sollen vielmehr einige interessante Aspekte der Realisierung verdeutlichen. Auch die Vollständigkeit von Klassendiagrammen und die syntaktische Richtigkeit von Codebeispielen muss nicht zwingend gegeben sein, da die gezeigten Aspekte lediglich zur Veranschaulichung dienen. Während der Implementierung wurde viel Wert auf die Einhaltung von Clean Code Prinzipien nach Martin (2008) sowie die korrekte Umsetzung der zuvor genannten Entwurfsmuster gelegt. Zusätzlich wurden auf eine hohe Testabdeckung aller wichtigen Klassen geachtet. Zunächst werden in diesem Kapitel Implementierungsdetails der bereits beschriebenen Komponenten gezeigt. Darauf aufbauend wird die Auslieferung der Android Continuous Delivery DSL Engine als Jar-File dargestellt. Abschließend werden die generierten Artefakte genauer in Augenschein genommen.

5.1 Der DSL-Parser

Die Parsing-Schicht der DSL Engine ist für die Übersetzung der Fluent Interface API in konkrete Objekte des semantischen Modells verantwortlich. Da im Laufe der Entwicklung die Anzahl und Komplexität der Expression-Builder-Klassen zur Realisierung einer DSL Parsing-Schicht rasch zunahm, wird in Abbildung 5.1 die Struktur des DSL Parser als UML-Klassendiagramm dargestellt.

Innerhalb der Grafik stellen die blau markierten Klassen die unterschiedlichen Expression Builder dar. Einzige Ausnahme ist hier die `CredentialsBuilder`

[1] https://www.eclipse.org/
[2] https://gradle.org/
[3] https://git-scm.com/

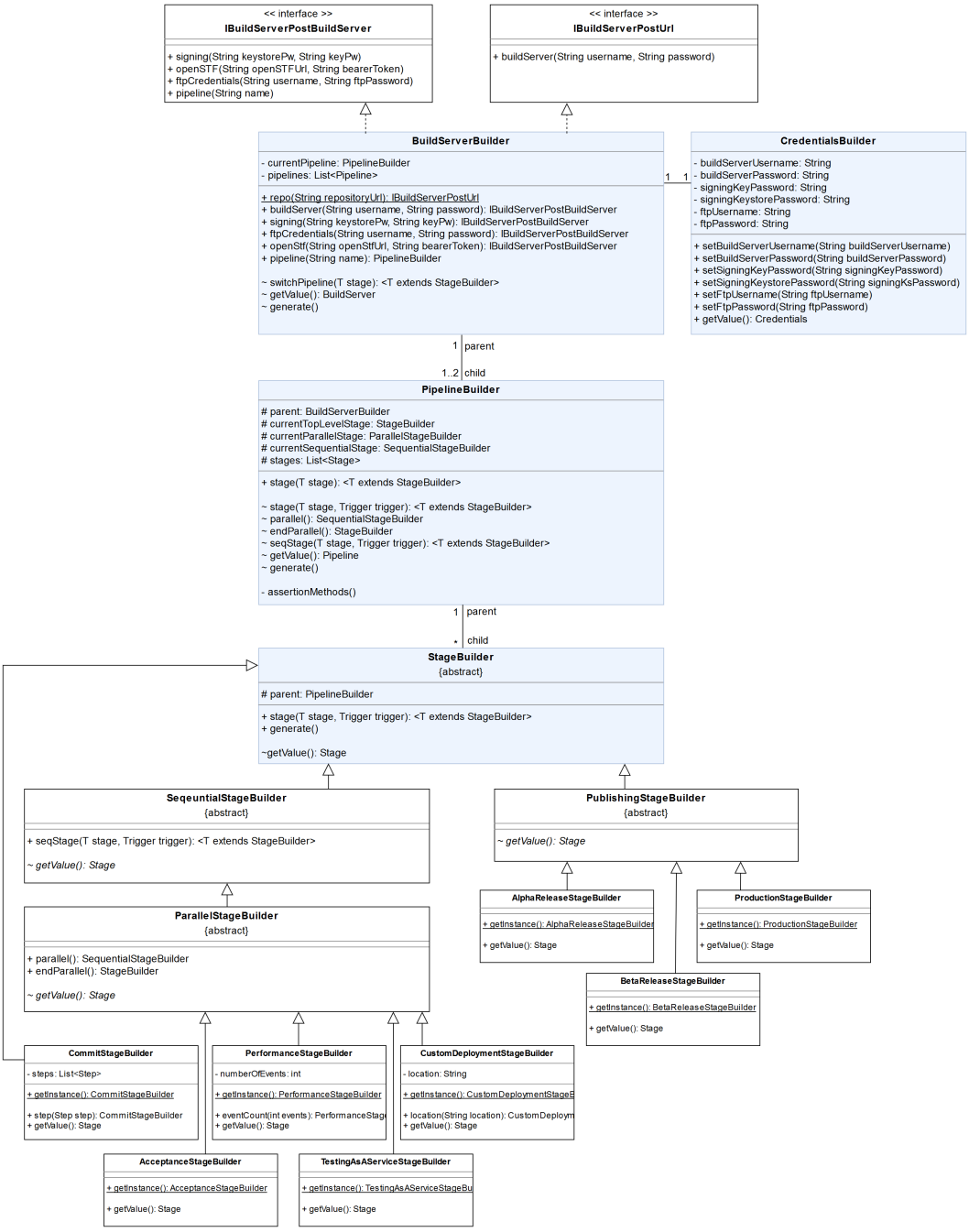

Abbildung 5.1: Klassendiagramm der Parsing-Schicht nach UML 2.0.

Klasse, welche nach Fowler (2010) das *Construction Builder* Entwurfsmuster anwendet. Bei Construction Builder wird ein unveränderliches Objekt schrittweise mit Hilfe einer Builder-Klasse erzeugt. Der Name leitet sich davon ab, dass Konstruktor-Argumente des Zielobjektes in den Attributen des Builders gespeichert werden. Im Folgenden wird der `CredentialsBuilder` auch als eine Builder Klasse angesehen. Jede Builder-Klasse hat eine eigene Verantwortlichkeit. So ist der `BuildServerBuilder` beispielsweise für die spezifische Konfigurationen des Buildservers zuständig, während der `PipelineBuilder` für die korrekte Zusammenstellung einer Pipeline verantwortlich ist. Die Builder-Klassen für Credentials und Stages verhalten sich analog.

```
1  public class PipelineBuilder {
2      // (...)
3      Pipeline getValue() {
4          assertIsPipelineValid();
5          Stage[] pipelineStages = stages.toArray(
6                  new Stage[stages.size()]);
7          return new Pipeline(pipelineName, pipelineStages);
8      }
9
10     protected void assertIsPipelineValid() {
11         if (stages.size() < 1) {
12             throw new PipelineParserException(
13                     "Pipeline must at least have one stage");
14         }
15         if (firstStageIsCommitStage() == false) {
16             throw new InvalidPipelineOrderException(
17                     "First stage in pipeline must be " +
18                     Stage.COMMIT_STAGE_NAME);
19         }
20         if (pipelineContainsAcceptanceStage()) {
21             if (secondStageIsAcceptanceStage() == false) {
22                 throw new InvalidPipelineOrderException(
23                         "Second stage in pipeline must be " +
24                         Stage.ACCEPTANCE_TEST_STAGE_NAME);
25             }
26         }
27         checkForDuplicateStages(this.stages);
28         return;
29     }
30 }
```

Listing 5.1: Die getValue() und Assertion-Methoden des PipelineBuilders.

Eine weitere Gemeinsamkeit aller Builder-Klassen sind die `getValue`-Methoden, die jeweils das korrespondierende Objekt des semantischen Modells erzeugen und zurückliefern. Dabei werden vor Erstellung der Objekte Assertion Methoden aufgerufen, um Prüfungen, wie beispielsweise die Reihenfolge der Stages, zur Laufzeit durchzuführen. Listing 5.1 zeigt die `getValue`-Methode inklusive ihrer Laufzeitprüfungen. Das semantische Modell wird schrittweise während der Abarbeitung der Parsing Schicht Methoden durch Aufruf dieser `getValue`-Methoden erzeugt.

Besteht ein Builder aus mehreren Sub-Builder, wie beispielsweise der Buildserver oder auch die Pipeline, wird das aktuelle Sub-Builder-Objekt in einer Kontextvariable gespeichert. Kontextvariablen sind durch das Schlüsselwort *current* vor dem Variablennamen gekennzeichnet und dienen dazu, den benötigten Kontext während des Parsevorganges festzuhalten. Zusätzlich existiert zwischen Builder und Sub-Builder eine Eltern-Kind-Relation und die Sub-Builder-Klassen enthalten eine Referenz namens `parent` auf das Eltern-Objekt, um Methodenaufrufe an ihr Eltern-Objekt weiterzureichen.

Den Einstiegspunkt des Parsevorganges stellt die statische `repo`-Methode dar. Sie erzeugt ein `BuildServerBuilder`-Objekt und gibt dieses in Form eines Interfaces zurück. Dies dient dazu, nachfolgende Methodenaufrufe zu beschränken. Somit sind nur die Methoden dieses Interfaces aufrufbar, in diesem Falle, die `buildServer`-Methode. Diese gibt ein weiteres Interface namens `IBuild ServerPostBuildServer` zur weiteren Konfiguration des Buildservers zurück. Listing 5.2 zeigt die Deklaration des Interfaces sowie die Implementierung der `buildServer`-Methode.

Sobald die `pipeline` Methode aufgerufen wird, gibt diese ein Objekt vom Typ `PipelineBuilder` zurück, so dass im weiteren Verlauf der DSL lediglich Methoden der Pipeline-Builder-Klasse aufrufbar sind. In gleicher Weise funktioniert der Aufruf der `stage`-Methode, die ein Objekt vom Typ `StageBuilder` zurückliefert.

Ein weiteres Mittel, um den Anwender bei Konfiguration der Pipeline zu unterstützen, war die Nutzung von Java Zugriffsmodifizierer. Der Pipeline-Builder implementiert zwei `stage`-Methoden, jedoch ist nur eine davon, diejenige ohne das Trigger-Argument, öffentlich aufrufbar. Die öffentliche `stage`-Methode des Pipeline-Builder wird zur Beschreibung der Commit Stage genutzt, da sie keinen Trigger benötigt. Anschließend wird ein `StageBuilder` zurückgegeben, der nur eine öffentliche `stage`-Methode mit Trigger als Argument besitzt. Die Logik dieser Methode wird weiterhin im Pipeline Builder implementiert, jedoch reicht der Stage-Builder den Aufruf zu seinem Eltern-Objekt weiter. Ein weiteres Beispiel hierfür ist die `generate`-Methode, welche nur innerhalb der Stage-Builder-Klassen öffentlich aufrufbar ist. Die tatsächliche Implementierung findet jedoch im Buildserver-Builder statt, da hier die Objekte des semantischen Modells verdrahtet werden und der Codegenerator angestoßen wird. Es wird jedoch sichergestellt, dass in der Builder Hierarchie bis zum Stage-Builder hinabgestiegen wird, also

```
1  public interface IBuildServerPostBuildServer {
2      IBuildServerPostBuildServer signing(
3              String keystorePw, String keyPw);
4      PipelineBuilder pipeline(String name);
5      // (...)
6  }
7
8  public class BuildServerBuilder implements
9          IBuildServerPostBuildServer {
10
11     public IBuildServerPostBuildServer buildServer(
12             String username, String password) {
13         credentialsBuilder.setBuildServerUsername(username);
14         credentialsBuilder.setBuildServerPassword(password);
15         return this;
16     }
17     // (...)
18 }
```

Listing 5.2: Die Interface-Deklaration und buildServer()-Methode des Buildserver-Builders.

mindestens eine Pipeline und eine Stage vorhanden sind, bevor das semantische Modell erzeugt werden kann.

Die Klassenhierarchie der Stage-Builder-Klassen wurde aus drei Gründen etwas komplexer und unübersichtlicher als erwartet. Zum einen findet eine Aufteilung in zwei Pipelines statt, weshalb die abstrakte `PublishingStageBuilder`-Klasse benötigt wurde. Der tatsächliche Wechsel zwischen den Pipelines findet an oberster Stelle innerhalb des Buildserver-Builder statt, auch wenn der Sprung in eine andere Pipeline nur an unterster Stelle, dem Stage-Builder, erkannt wird. Dies war nötig, da dieser Wechsel transparent für den Anwender ablaufen soll.

Des Weiteren sollen parallel ausführbare Stages unterstützt werden, weshalb Klassen wie `SequentialStageBuilder` und `ParallelStageBuilder` nötig wurden.

Den dritten Grund für die erhöhte Komplexität der Stage-Builder-Klassenhierarchie stellen die spezifischen Folgemethoden der `stage`-Aufrufe dar. Jede Stage kann individuelle Parameter enthalten, die mit Hilfe einer speziellen Methode gesetzt werden. Beispielsweise kann einer Performance Testing Stage die Anzahl der zufällig generierten Events durch Aufruf der `eventCount`-Methode konfiguriert werden. Zur Umsetzung wurde für jede Stage eine eigene Stage-Builder-Klasse erzeugt, die ihre zusätzlichen Konfigurationsmöglichkeiten in Form von individuellen Methoden anbieten. Die abstrakte Stage-Builder-Klasse bietet eine einheit-

liche Schnittstelle und somit ein gemeinsames Verhalten für konkrete Stage-Builder-Klassen an. Daher erbt beispielsweise die `PerformanceStageBuilder`-Klasse von `StageBuilder` und bietet eine `eventCount`-Methode zur Spezialisierung an. Darüber hinaus werden Java *Generics* innerhalb der `stage`-Methode verwendet, um konkrete Ausprägungen von Stage-Builder-Klassen zurückzuliefern. Dadurch wird dem Anwender die Möglichkeit zur Autovervollständigung des DSL-Skriptes innerhalb seiner Java-Entwicklungsumgebung ermöglicht und fehlerhafte Konfigurationen einer Stage bereits zur Compile-Zeit erkannt.

Listing 5.3 zeigt einen Ausschnitt der `stage`-Methode des Pipeline-Builder, um die Verwendung der generischen Elemente zu veranschaulichen. Dabei wird ein konkreter Stage-Builder als Argument übergeben, der Trigger und Eltern-Builder dieses Stage-Builders gesetzt und am Ende der Methode genau dieser Stage-Builder zurückgeliefert. Anschließend können die spezifischen Methoden dieses Stage-Builder aufgerufen werden.

```
1  <T extends StageBuilder> T stage(T stage, Trigger trigger) {
2      stage.setTrigger(trigger);
3      stage.setParent(this);
4      handleParallelBlockExecution();
5
6      // save new context
7      if (currentStage != null) {
8          stages.add(currentStage.getValue());
9      }
10     currentStage = stage;
11
12     if (isPublishingStage(stage)) {
13         if (inParallelSection()) {
14             throw new PipelineParserException(
15                     "cannot put publishing stage in parallel section");
16         }
17         return parent.switchPipeline(stage);
18     }
19
20     return stage;
21 }
```

Listing 5.3: Generics innerhalb der stage() Methode.

Innerhalb der gezeigten Methode ist auch die Neuzuweisung einer Kontextvariablen und der `switchPipeline` Aufruf des Eltern-Builder zu sehen. Die konkreten Stage-Builder wurden als *Singleton*, entsprechend dem gleichnamigen Entwurfsmuster, implementiert und werden in einer Klasse namens `Stages` erzeugt und

verwaltet. Für eine bessere Übersichtlichkeit wurde auf die Darstellung dieser Klassen in Abbildung 5.1 verzichtet.

Für die beiden möglichen Ausführungsvarianten von Stages, nämlich parallel und sequentiell, stehen entsprechend abstrakte Klassen bereit. Auch eine Verschachtelung von parallelen und sequentiellen Ausführungssträngen ist möglich. Hierbei werden die Methodenaufrufe an das Eltern-Objekt, den Pipeline-Builder, weitergereicht, in dem die Handhabung von parallelen Stages stattfindet. Auch hier sind die Methoden mit Hilfe von Zugriffsmodifizierer nicht direkt innerhalb des Pipeline-Builders aufrufbar. Der `ParallelStageBuilder` stellt eine spezialisierte Form des `SequentialStageBuilder` dar. Dies ist darauf zurückzuführen, dass die Methode `seqStage` nur innerhalb eines Parallelblocks aufrufbar sein soll. Ist eine Stage parallel ausführbar, muss der korrespondierende Stage-Builder lediglich von `ParallelStageBuilder` erben. Hierbei bildet der Acceptance-Stage-Builder eine Ausnahme, da seine korrespondierende Stage nicht parallel ausgeführt werden kann. Um nachfolgenden Stages die parallele Ausführung zu ermöglichen, ist auch die Acceptance-Stage-Builder-Klasse eine Subklasse von `ParallelStage Builder`. Die Syntax der entwickelten internen DSL, parallele Stages in einen Block zu kapseln, bedingt dieses Verhalten.

5.2 Das semantische Modell

Das semantische Modell wurde entsprechend des Entwurfs aus Kapitel 4.4 umgesetzt. Aus diesem Grund wird das semantische Modell innerhalb dieses Kapitels nicht genauer erläutert. Es ist unveränderlich und enthält Zugriffsmethoden, die von der Generator-Komponente genutzt werden, um den Zielcode zu erzeugen.

5.3 Der Codegenerator

Die Kernfunktionalitäten des Codegenerators werden durch die `Jenkinsfile Generator`-Klassen bereitgestellt, die für die Erstellung eines Jenkinsfiles pro Pipeline verantwortlich sind. Diese Jenkinsfiles werden später in XML-Strukturen eingebettet, die einen Jenkins-Job beschreiben. Abbildung 5.2 stellt einen Ausschnitt des bereits gezeigten Klassendiagrammes dar, bei dem die Transformer-Generation-Klassen hervorgehoben sind.

Die `JenkinsfileGenerator`-Klasse bekommt bei ihrer Erstellung einen Teil des semantischen Modelles sowie einen Writer, in den das generierte Jenkinsfile geschrieben wird, übergeben. Innerhalb der `generate`-Methode wird dann über die

Abbildung 5.2: Klassendiagramm der Transformer-Generation-Klassen.

einzelnen Stages der Pipeline iteriert und schrittweise der Zielcode erzeugt. Listing 5.4 demonstriert einen Ausschnitt der `JenkinsfileGenerator`-Klasse.

```
1  public abstract class JenkinsfileGenerator {
2      public void generate() {
3              generateJenkinsfileHeader();
4              generateStages();
5              generateJenkinsfileFooter();
6      }
7      private void generateStages() {
8          Stage[] stages = pipeline.getStages();
9          for (int i = 0; i<stages.length; i++) {
10             if (isParallelStageComposite()) {
11                 generateParallelStages(stages[i]);
12             } else {
13                 generateStage(stages[i]);
14             }
15         }
16     }
17     private void generateStage(Stage stage) {
18         output.printf("stage('%s') {\r\n", stage.getName());
19         generateStageTrigger(stage);
20         generateStageContent(stage);
21         output.println("}");
22     }
23     //(...)
24 }
```

Listing 5.4: Ausschnitt des Jenkinsfile Generators.

Die Methode `generateStageContent` ist abstrakt und wird von den konkreten Jenkinsfile-Generator-Klassen implementiert. Das Kompositum Entwurfsmuster wird durch die Methoden `generateParallelStages` und `generateSequentialStages` aufgelöst. Auf diese Weise wird das semantische Modell in den Kontrollfluss der Jenkinsfiles eingebettet. Statische Teile des Zielcodes, wie beispielsweise der Jenkinsfile Header oder auch Android-Befehle der jeweiligen Stages, wurden in eigenen Klassen als String-Konstanten definiert. Die Android-Befehle wurden zusätzlich in eine Map mit dem Stagenamen als *Key* und dem jeweiligen Android Befehl als *Value* gepackt. Da die Zielcodeerstellung unabhängig von den restlichen Komponenten der Continuous Delivery Engine ist, sind diese Konstanten nur innerhalb des Generator-Paketes sichtbar.

Abbildung 5.3 hebt die Klassen hervor, die das Templated Generation Entwurfsmuster anwenden. Als Template Engine wurde *Apache Velocity*[4] verwendet. Im Folgenden wird lediglich der Job-Generator betrachtet, da sich der Credentials-Generator analog verhält.

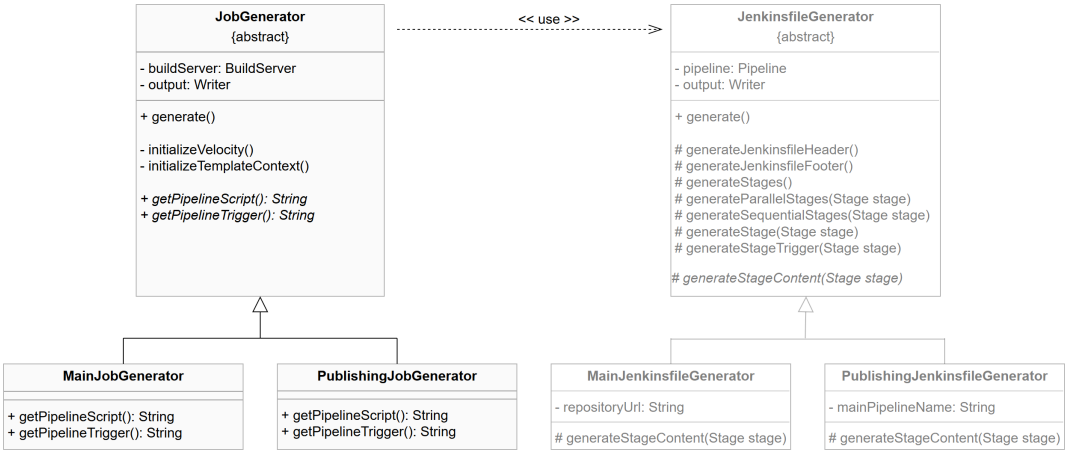

Abbildung 5.3: Klassendiagramm der Templated-Generation-Klassen.

Es existieren zwei Template-Dateien, einerseits für Credentials und anderseits für Jenkins-Jobs. Dabei handelt es sich um XML-Dateien, in denen die dynamischen Elemente durch Methodenaufrufe einer Helper-Klasse ersetzt wurden. Die konkreten Job-Generator-Klassen stellen in diesem Falle die Helper-Klassen dar. Die abstrakte Klasse `JobGenerator` initialisiert die Template Engine und ruft die `merge`-Methode von Velocity auf. Darauffolgend findet eine Ersetzung der im Template spezifizierten Methoden durch Velocity statt.

Im Anhang B befindet sich eine vollständige Darstellung einer generierten Job-XML-Datei, die auf die hier gezeigte DSL-Syntax Bezug nimmt. Zusätzlich ist noch eine generierte Credential-XML-Datei abgebildet.

[4]http://velocity.apache.org

5.4 Der Konfigurator

Der Konfigurator hat innerhalb der DSL Engine zwei Aufgaben. Zum einen muss der Jenkins Docker-Container erzeugt und konfiguriert werden, zum anderen müssen die generierten Artefakte in den Jenkins Buildserver eingefügt werden. Beide Aufgaben wurden getrennt voneinander implementiert, so dass es möglich ist, in eine bereits laufende Jenkins-Instanz die generierten Artefakte einzufügen.

Für das Erzeugen und Konfigurieren des Containers wurde ein Java *Docker Client* von Spotify[5] verwendet. Dieser holt sich von der Online-Plattform *Docker Hub*[6] ein spezifisches Docker-Image, welches auf einem Jenkins Basisimage aufsetzt, konfiguriert das Port Mapping, baut den Jenkins-Container und startet diesen. Notwendige Zugangsdaten für den Jenkins Buildserver, welche im DSL-Skript spezifiziert wurden, werden über Umgebungsvariablen in den Container hineingereicht. Innerhalb des Containers werden sie anschließend von einem Script ausgelesen, welches einen Jenkins Administrator Account erstellt. Um innerhalb des Jenkins-Containers den Android Build Host Container zu erzeugen, wird der Docker Socket des Host-Rechners in den Container gemountet.

Um die Jobs und Credentials an den Jenkins Buildserver zu schicken, wird die Apache *HttpComponents*[7] Bibliothek verwendet. Die erzeugten XML-Dateien werden mittels HTTP-POST an den entsprechenden Jenkins REST-API-Endpunkt geschickt.

5.5 Auslieferung der Continuous Delivery Engine

Die Android Continuous Delivery DSL Engine wird als Java Bibliothek ausgeliefert. Dies hat zur Folge, dass ein neues Java Projekt angelegt und die Jar-Datei eingebunden werden muss, um das Fluent Interface der interne DSL zu nutzen.

Die Bibliothek hat eine Klasse namens `CdEngine`, in der sich eine statische `run`-Methode befindet, welche den Einstiegspunkt innerhalb der Bibliothek darstellt. In dieser Methode wird der Generator angestoßen und die erzeugten Artefakte dem Konfigurator übergeben. Ziel dieser Struktur ist, eine Flexibilität bereitzustellen, um beispielsweise nur den Generator anzustoßen und die erzeugten Artefakte als Dateien abzulegen. Auch das zusätzliche Einfügen von Jenkins-Jobs in eine bereits laufende Instanz eines Buildservers wird dadurch ermöglicht.

[5] https://github.com/spotify/docker-client
[6] https://hub.docker.com/
[7] https://hc.apache.org/

Aktuell wird die `run`-Methode als letzte Anweisung des Parsers aufgerufen und es wird stets ein neuer Jenkins-Container erzeugt, in den die generierten Artefakte eingefügt werden.

5.6 Besonderheiten der generierten Jenkinsfiles

In diesem letzten Abschnitt des Implementierungskapitels werden die erzeugten Jenkinsfiles genauer betrachtet, da sie die Logik der Pipeline abbilden.

Die erste Pipeline nutzt den Android Build Host Container, um die Applikation zu bauen und testen. Jenkins Pipeline unterstützt bereits die Nutzung von Docker-Containern. Dieses Feature wurde verwendet, um innerhalb des Jenkinsfile mit dem Android Build Host zu interagieren. Im Android Build Host Image sind die Android SDK Build Tools installiert. Zusätzlich ist es mit Hilfe von Gradle möglich, die von der Applikation benötigten SDK Platform Tools automatisch, während des Build-Vorganges herunterzuladen. Die Platform Tools sind abhängig von der verwendeten Android Version der App. So wird das Android Build Host Image schlank gehalten und muss nicht bei jeder neuen Android Version angepasst werden. Um Performance-Engpässe während des Build-Vorgangs zu entgegnen, könnte auch ein Volume in den Android Build Host Container gemounted werden, so dass die SDK Platform Tools nur einmalig bei Änderung der Android Version heruntergeladen werden müssen. Dies wird aktuell noch nicht verwendet.

Zum Zeitpunkt der Entwicklung gab es noch keine Möglichkeit, parallel ausführbare Stages mit Hilfe der Declarative Pipeline-Syntax zu spezifizieren. Daher wurde die Scripted Pipeline-Syntax für die erzeugten Jenkinsfiles verwendet.

Der Jenkins Buildserver versucht die verfügbaren CPUs bestmöglich auszulasten. Daher können mehrere Pipeline Instanzen zeitgleich ausgeführt werden. Während der Acceptance Test und Performance Testing Stages werden jedoch eine Applikation auf die Zielgeräte installiert, die jeweiligen Tests durchgeführt und anschließend diese Applikation wieder von den Geräten gelöscht. Damit die Pipeline Instanzen ihre gemeinsam genutzten Ressourcen, die Android-Geräte, nicht gegenseitig beeinflussen, wurde das *Lockable Resources Plugin*[8] verwendet, um einen wechselseitiger Ausschluss der Instanzen zu realisieren.

Um Zugangsdaten und Passwörter, die während eines Android Builds benötigt werden, sicher in Jenkins abzulegen, wurde das *Credentials Plugin*[9] verwendet. Durch dieses Plugin wird der REST-API-Endpunkt verfügbar, an den der Konfigurator die Credential-XML-Dateien schickt. Um die gespeicherten Credentials

[8] https://wiki.jenkins.io/display/JENKINS/Lockable+Resources+Plugin
[9] https://wiki.jenkins.io/display/JENKINS/Credentials+Plugin

während des Build-Vorganges zu nutzen, wird ein weiteres Jenkins Plugin names *Credentials Binding Plugin*[10] benutzt. Dieses injiziert die Credentials als Umgebungsvariablen während des Build-Vorganges. Die Credentials sind nur innerhalb des spezifizierten Blockes verfügbar und erscheinen nicht in den Build-Logs. Dieses Konzept wird zum sicheren Signieren einer Release APK genutzt. Listing 5.5 zeigt die Commit Stage eines generierten Jenkinsfiles.

```
stage('Commit Stage') {
    sh './gradlew assembleDebug'
    pathToDebugApk = sh(script: 'find . -name "*.apk"',
        returnStdout: true).readLines().get(0).trim()
    withCredentials([string(credentialsId: 'app-signing-key-password',
                variable: 'RELEASE_KEY_PASSWORD'),
                string(credentialsId: 'keystore-password',
                variable: 'RELEASE_KEYSTORE_PASSWORD')]) {
        sh './gradlew assembleRelease'
    }
    packageName = sh(script: "getPackage ${pathToDebugApk}",
            returnStdout: true).trim()
    versionCode = sh(script: "getVersionCode ${pathToDebugApk}",
            returnStdout: true).trim()

    archiveArtifacts artifacts: '**/*.apk',
            fingerprint: true
}
```

Listing 5.5: Die Commit Stage eines generierten Jenkinsfiles.

Die Credential-IDs werden vom Generator bei der Credentials-Erstellung festgelegt und bei Erzeugung der Jenkinsfiles wiederverwendet. Die Namen der Umgebungsvariablen sind hart codiert und müssen in der Gradle Build-Konfiguration der Applikation angegeben werden. Im Anhang D findet sich eine Gradle Build-Konfiguration, die diese Umgebungsvariablen nutzt, um eine Android-Applikation automatisiert zu signieren. Neben einer Vielzahl von weiteren Möglichkeiten, die Applikation sicher zu signieren, wurde dieser Ansatz gewählt, da der Entwickler die Kontrolle über seinen *App signing key* behält. Falls der Jenkins Container gelöscht wird, können neue Versionen der Applikation mit Hilfe des privaten Schlüssels weiterhin auf Google Play veröffentlicht werden. Zudem werden in gleicher Weise die FTP-Credentials der Custom Deployment Stage gespeichert und injiziert.

Das Jenkinsfile nutzt Shell-Skripte, die sich im Android Build Host Container befinden, um den Versionscode und die Applikation-ID einer APK-Datei auszu-

[10] https://wiki.jenkins.io/display/JENKINS/Credentials+Binding+Plugin

lesen. Der Versionscode wird von der Custom Deployment Stage genutzt, um die APK-Datei entsprechend ihrer Versionsnummer zu benennen und somit Konflikte bei der Benennung zu vermeiden. Im Rahmen einer Veröffentlichung muss der Versionscode einer Applikation ohnehin für jeden Build inkrementiert werden. Die Applikation-Id wird innerhalb der Performance Testing Stage benötigt, um die zufällig generierten Events auf die jeweilige Applikation zu beschränken. Daneben existiert ein Shell-Skript, um Android-Befehle auf allen per openSTF angebundenen Geräten auszuführen. Wurde openSTF in der DSL nicht spezifiziert, werden die Befehle auf allen lokal angeschlossenen Geräte ausgeführt.

Die zweite Pipeline nutzt das *Google Play Android Publisher Plugin*[11], um die signierte APK-Datei in den jeweiligen Track des Play Stores zu veröffentlichen. Ein umfassendes Beispiel der erzeugten Jenkinsfiles findet sich eingebettet in den Jenkins Job XML-Dateien im Anhang B.

Zum Abschluss dieses Kapitels soll kurz aufgezeigt werden, welche Funktionalitäten nicht implementiert wurden. Zum einen konnten innerhalb des Jenkins-Buildserver zum Zeitpunkt der Entwicklung noch keine Git-Tags erkannt und gebaut werden, weshalb die Continuous Delivery Engine aktuell nur automatische, manuelle und zeitgesteuerte Trigger anbietet. Dieses Problem scheint von der Jenkins Community bereits gelöst zu sein (Jenkins, 2016). Des Weiteren wurde die Testing as a Service Stage nicht realisiert, da hierzu ein Firebase-Account[12] benötigt wird. Die erforderlichen Bibliotheken sollten sich jedoch in den Android Build Host Container integrieren lassen. Zuletzt unterstützt die DSL Engine aktuell keine Emulatoren, da diese häufig der Kritik unterliegen, die Gerätehardware nicht nah genug abbilden zu können. Eine Gegenüberstellung von realen Android Geräten und Emulatoren liefert Hechtel (2016).

[11] https://wiki.jenkins.io/display/JENKINS/Google+Play+Android+Publisher+Plugin
[12] https://console.firebase.google.com/

6 Evaluation

In diesem Kapitel erfolgt eine Evaluierung der entwickelten Android Continuous Delivery DSL Engine. Sie soll unter Verwendung eines einfachen DSL-Skriptes automatisiert eine funktionsfähige Android Deployment Pipeline inklusive Buildserver erzeugen. Zusätzlich soll die erzeugte Deployment Pipeline generisch, für eine Vielzahl von Android-Applikationen, anwendbar sein. Um die Erfüllung dieser Anforderungen nachzuweisen wurden vier Open Source Android-Applikationen als Samples für diese Evaluierung ausgewählt. Im Anschluss wird die automatisierte Veröffentlichung einer APK in den Play Store aufgezeigt und diskutiert.

6.1 Vorgehensweise

Mit Einführung von Android 4.1 im Jahr 2013 wurde das SDK-Build-System auf Gradle umgestellt. Die Gründe waren eine höhere Flexibilität, Funktionen wie die automatische Verwaltung von Abhängigkeiten und der Wunsch nach einem einzigen Build-System für Android (Schmidt, 2013). Aus diesem Grund unterstützt die Android Continuous Delivery DSL Engine nur Android-Applikationen, die Gradle als Build-System verwenden.

Für die Evaluation wurden unter Verwendung der Continuous Delivery Engine mehrere Deployment Pipelines erzeugt und überprüft, inwieweit die Anforderungen aus Kapitel 3 erfüllt werden. Um ein aussagekräftiges Ergebnis zu erhalten, wurden vier Android Open Source Applikationen anhand von drei Kriterien ausgewählt. Sie müssen von unterschiedlichen Entwicklern stammen, unterschiedliche Android-Versionen verwenden und eine hohe Beliebtheit im Play Store aufweisen.

Als mobiles Testgerät wurde ein Nexus 5X mit Android-Version 8.1.0 „*Oreo*“ und den Sicherheitsupdates vom 5. Januar 2018 verwendet. Die im Rahmen dieser Evaluation genutzten DSL-Skripte erzeugten jeweils eine Deployment Pipeline, welche eine Commit Stage, eine Acceptance Test Stage, eine Performance Testing Stage und eine Custom Deployment Stage beinhalteten. Die Commit Stage führte Unit-Tests aus und erzeugte mindestens eine Debug APK-Datei, welche im Laufe

der Custom Deployment Stage an einen lokalen FTP-Server hochgeladen wurde. Innerhalb der Performance Testing Stage wurden jeweils 3500 zufällig generierte Benutzerereignisse erstellt. Zudem wurden verschiedene Trigger sowie parallele und sequenzielle Ausführungsvarianten für die Performance Testing und Custom Deployment Stages konfiguriert.

Im zweiten Teil dieses Kapitels wird der automatisierte Upload nach Google Play betrachtet. Intern wird diese Funktion, wie in der Arbeit geschildert, in eine eigene Pipeline gekapselt. Da für die Veröffentlichung Schlüsselpaare zur Signierung einer Applikation sowie ein Eintrag in Google Play erstellt werden muss, beschränkt sich dieser zweite Teil der Evaluation auf die selbst entwickelte OSR Playground Applikation. Die zweite Pipeline benötigt weder mobile Endgeräte, noch Abhängigkeiten zu den Android SDK Tools und ist für alle korrekt signierten APKs in gleicher Weise anwendbar.

6.2 Evaluierung anhand von Open-Source Applikationen

Die erste Android-Applikation zur Evaluation der Continuous Delivery DSL Engine war *PocketHub*[1], ein Android Client für GitHub. Sie verwendet die API-Level 27, welches der Android Version 8.1 „*Oreo*“ entspricht. Die generierte Deployment Pipeline lief ohne Probleme und meldete einen erfolgreichen Build als Ergebnis.

Die nächste Applikation namens *Voice*[2], ist sowohl in Google Play, als auch in F-Droid, einem weiteren, nicht von Google betriebenen, App Store für das Android Betriebssystem verfügbar. Voice ist ein Hörbuch Player mit dem Ziel, die Bedienung so einfach wie möglich für seine Nutzer zu gestalten. Das im Quellcode spezifizierte API-Level beträgt 26 und unterscheidet sich nur geringfügig von PocketHub. Jedoch ist der Aufbau der Applikation wesentlich komplexer. So wurden während des Build Vorganges zwei Debug APK-Dateien und sechs Test APK-Dateien erzeugt. Weiterhin werden innerhalb der Gradle-Konfiguration von Voice fünf Subprojekte referenziert. Auch hier lieferte die Deployment Pipeline einen erfolgreichen Build. Dieses Beispiel soll zeigen, dass die Android Continuous Delivery DSL Engine auch mit komplexen Gradle-Konfigurationen und mit mehr als einer erzeugten Debug APK-Datei umgehen kann.

Als dritte Applikation zur Evaluation der DSL Engine wurde eine Wetter App hergenommen. Auch sie ist auf Google Play und F-Droid verfügbar, verwendet allerdings API-Level 23, welches der Android Version 6.0 „*Marshmallow*“ ent-

[1] https://github.com/pockethub/PocketHub
[2] https://github.com/PaulWoitaschek/Voice

spricht. Ihr Name lautet *Forecastie*[3]. Der erzeugte Buildserver baute die Debug und Test APK-Dateien und führte die entsprechenden Tests fehlerfrei auf dem Android Gerät aus.

Die vierte und letzte Evaluationsapplikation ist *Fly*[4], ein 3D-Spiel für Android und PC. Sie wurde im Rahmen der *Mobile Application Developement* Vorlesung an der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt. Allerdings ist die letzte Änderung mehr als zwei Jahre her. Es wird API-Level 20, also Android Version 4.4 „*KitKat*“ benutzt. Hier schlug die erzeugte Deployment Pipeline fehl, da die benötigten SDK Platform Tools nicht automatisch von Gradle heruntergeladen werden konnten. Die Wetterapplikation *Forecastie* hat jedoch bewiesen, dass diese Gradle Funktionalität, selbstständig die benötigten SDK Platform Tools herunterzuladen, spätestens ab API-Level 23 verfügbar ist.

Dieser erste Teil der Evaluation hat gezeigt, dass die Android Continuous Delivery DSL Engine generisch für alle Android-Applikationen ab API-Level 23 erfolgreich eingesetzt werden kann.

6.3 Evaluierung der automatisierten Veröffentlichung in den Play Store

Der zweite Teil der Evaluation soll nun darstellen, wie mit Hilfe der DSL Engine eine automatisierte Veröffentlichung nach Google Play erfolgt. Dies wird anhand der in Kapitel 2.5 vorgestellten *OSR Playground*[5] Applikation verifiziert. Um die App in den Play Store hochzuladen, musste zunächst ein Eintrag für sie mittels der Google Play Console[6] erstellt werden. Dieser initiale Schritt wurde bereits während der Entwicklung der Continuous Delivery Engine durchgeführt.

Für eine umfassende Evaluierung wurden verschiedene Android Deployment Pipelines generiert. Alle Pipelines erzeugten innerhalb der Commit Stage eine releasefähige APK-Datei, unterschieden sich jedoch in ihren Veröffentlichungstages. So wurden drei Pipelines generiert, die jeweils nur eine Stage zur Veröffentlichung hatten, um das Hochladen in den Alpha, Beta und Production Track getrennt zu betrachtet. Die vierte erzeugte Pipeline beinhaltete alle drei Veröffentlichungstages zugleich und verifizierte somit die Verschiebung der APK-Dateien von einem Track in den nächsten. Pro Pipeline wurden drei Veröffentlichungen durchgeführt, die alle erfolgreich verliefen. Somit ist mittels der Continuous Delivery Engine auch eine Veröffentlichung der Applikation in den Play Store möglich.

[3]https://github.com/martykan/forecastie
[4]https://github.com/FAU-Inf2/fly-gdx
[5]https://github.com/Maximilian-Fischer/osr-playground
[6]https://play.google.com/apps/publish/

Zum Abschluss sei noch erwähnt, dass zwei Dinge für das Hochladen in den Play Store berücksichtigt werden müssen. Zum einen fordert der Play Store, dass jede veröffentlichte APK-Datei ihren Versionscode erhöht. Dies kann mit Hilfe der Build-Nummer von Jenkins automatisiert werden. Im Anhang D wird die Gradle-Konfiguration von OSR Playground gezeigt, die die `jenkinsBuildNumber` Umgebungsvariable zur Erzeugung des Versionscodes der Applikation nutzt. Zum anderen muss dem Jenkins Buildserver API-Zugriff auf den Google Play Account gewährt werden. Dieser Schritt ist einmalig nach Erstellung des Jenkins Docker-Container notwendig und kann aktuell nur über die Weboberfläche von Jenkins durchgeführt werden. Eine Anleitung hierfür liefert Orr (2018).

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde gezeigt, dass sich eine Continuous Delivery Infrastruktur in der Domäne Android automatisiert erstellen und für eine Vielzahl von Applikationen anwenden lässt. Hierzu wurden verschiedene Android Deployment Pipelines analysiert und zu einem einheitlichen semantischen Modell zusammengefasst. Dieses Modell fungierte als Grundlage für die Entwicklung einer Software namens Android Continuous Delivery DSL Engine. Sie wurde in Java implementiert und bietet eine interne domänenspezifische Sprache zur Nutzung an. Mit Hilfe der angebotenen DSL wird die Beschreibung der gewünschten Continuous Delivery Infrastruktur auf einfache Weise ermöglicht. Die Architektur dieser Software wurde flexible gestaltet, um die verwendete DSL leicht erweiterbar und jederzeit austauschbar zu gestalten

Am Ende dieser Arbeit wurde eine Evaluation der erzeugten Android Continuous Delivery DSL Engine unter Verwendung von vier Android Open Source Applikationen durchgeführt. Dabei stellte sich heraus, dass die erzeugte Continuous Delivery Infrastruktur mit Android-Applikationen, die ein API-Level von 23 oder höher aufweisen, kompatibel ist. Durch eine Anpassung der App-spezifischen Gradle-Konfiguration wird selbst die automatisierte Veröffentlichung nach Google Play möglich.

Die im Hauptteil dieser Arbeit entwickelte DSL Engine soll Android-Entwickler bei der Umsetzung von Continuous Delivery Praktiken unterstützen. So kann der Einstieg in den Bereich Continuous Software Engineering erleichtert werden, indem ein einheitliches und übersichtliches semantischen Modells für Android Deployment Pipelines bereitgestellt wird. Weiterhin muss keine zusätzliche Programmiersprache, wie beispielsweise Groovy, zur Konfiguration einer Deployment Pipeline verwendet werden. Die angebotene interne DSL verwendet die gleiche Programmiersprache, die auch zur Entwicklung von Android-Applikationen genutzt wird. Dies alles führt zu einer verringerten Komplexität und schlussendlich zu einer Einsparung von Zeit und Aufwand bei der Realisierung einer CD-Infrastruktur im Bereich Android.

Zukünftige Arbeiten könnten die DSL Engine auf weitere mobile Plattformen, wie *iOS*, erweitern. Neben dem Bereich mobile Applikationen wäre auch eine Unterstützung von anderen Domänen wie beispielsweise embedded oder cloudbasierten Systemen denkbar.

Anhang A Syntax der entwickelten internen DSL

```
.repo("https://github.com/M-Fischer/android-playground.git")

.buildServer("serverUser", "serverPassword")
    .signing("keystorePw", "keyPw")
    .ftpCredentials("ftpUser", "ftpPw")

.pipeline("My Android Pipeline")
    .stage(COMMIT_STAGE).
        .step(COMPILE_RELEASE)
        .step(UNIT_TEST)
        .step(LINT)
    .stage(ACCEPTANCE_TEST_STAGE, Trigger.AUTOMATIC)
    .parallel()
        .seqStage(PERFORMANCE_TESTING_STAGE, Trigger.AUTOMATIC)
            .eventCount(1500)
        .seqStage(CUSTOM_DEPLOYMENT_STAGE, Trigger.AUTOMATIC)
            .location("localhost")
        .stage(TESTING_AS_A_SERVICE_STAGE, Trigger.MANUAL)
    .endParallel()
    .stage(ALPHA_RELEASE_STAGE, Trigger.AUTOMATIC)
    .stage(BETA_RELEASE_STAGE, Trigger.MANUAL)
    .stage(PRODUCTION_STAGE, Trigger.TIMETRIGGER, 7, DAYS)

.generate()
```

Listing 7.1: Syntax der entwickelten internen DSL.

Anhang B Beispiel einer generierten Jenkins Job XML-Dateien

```
<?xml version='1.0' encoding='UTF-8'?>
<flow-definition>
    <actions/>
    <description/>
    <keepDependencies>false</keepDependencies>
    <properties>
        <org.jenkinsci.plugins.workflow.job.properties.
                PipelineTriggersJobProperty>
            <triggers>
                <hudson.triggers.SCMTrigger>
                    <spec>* * * * *</spec>
                    <ignorePostCommitHooks>false</ignorePostCommitHooks>
                </hudson.triggers.SCMTrigger>
            </triggers>
        </org.jenkinsci.plugins.workflow.job.properties.
                PipelineTriggersJobProperty>
    </properties>
    <definition class="
        org.jenkinsci.plugins.workflow.cps.CpsFlowDefinition">
    <script>
#!/usr/bin/env groovy

node {
    git 'https://github.com/M-Fischer/android-playground.git'
    def packageName
    def versionCode
    def pathToDebugApk
    docker.image('knogl/android-build-host').inside(
            "-v /dev/bus/usb:/dev/bus/usb --privileged") {
    try {
        sh'chmod +x gradlew'
        stage('Commit Stage') {
            sh './gradlew clean'
            sh './gradlew assembleDebug'
            pathToDebugApk = sh(script: 'find . -name "*.apk"',
                returnStdout: true).readLines().get(0).trim()
            sh './gradlew assembleAndroidTest'
            withCredentials([
                    string(credentialsId: 'app-signing-key-password',
                    variable: 'RELEASE_KEY_PASSWORD'),
                    string(credentialsId: 'keystore-password',
```

```
                    variable: 'RELEASE_KEYSTORE_PASSWORD')]) {
                sh './gradlew assembleRelease'
            }
            sh './gradlew test'
            sh './gradlew lint'
            androidLint()
            archiveArtifacts artifacts: '**/*.apk', fingerprint: true
            packageName = sh(script: "getPackage ${pathToDebugApk}",
                    returnStdout: true).trim()
            versionCode = sh(script: "getVersionCode ${pathToDebugApk}",
                    returnStdout: true).trim()
        }
        stage('Acceptance Test Stage') {
            lock('androidDevices') {
                sh 'adb start-server'
                sh 'adb+ shell settings put global
                        window_animation_scale 0'
                sh 'adb+ shell settings put global
                        transition_animation_scale 0'
                sh 'adb+ shell settings put global
                        animator_duration_scale 0'
                sh './gradlew connectedAndroidTest'
            }
        }
        parallel (
            "Sequential Stages" : {
                stage('Performance Testing Stage') {
                    lock('androidDevices') {
                        sh 'adb start-server'
                        sh './gradlew installDebug'
                        sh "adb+ shell monkey -p ${packageName} -v 1500"
                        sh './gradlew uninstallDebug'
                    }
                }
                stage('Custom Deployment Stage') {
                    withCredentials([usernamePassword(
                            credentialsId: 'ftp-username-password',
                            passwordVariable: 'PASSWORD',
                            usernameVariable: 'USERNAME')]) {
                        sh "curl -T ${pathToDebugApk}
                            ftp://localhost/app-debug-${versionCode}.apk
                            --user $USERNAME:$PASSWORD"
                    }
                }
            },
```

```
87              "Testing as a Service Stage" : {
88                  stage('Testing as a Service Stage') {
89                      input 'approve Testing as a Service Stage'
90                      echo 'Testing as a Service not yet implemented ...'
91                  }
92              },
93          )
94      }
95      finally {
96          junit '**/TEST-*.xml'
97          deleteDir()
98      }}
99  }</script>
100         <sandbox>false</sandbox>
101     </definition>
102     <triggers/>
103     <disabled>false</disabled>
104 </flow-definition>
```

Listing 7.2: Beispiel einer generierten Job XML-Datei.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <flow-definition>
3      <actions/>
4      <description/>
5      <keepDependencies>false</keepDependencies>
6      <properties>
7          <org.jenkinsci.plugins.workflow.job.properties.
8                  PipelineTriggersJobProperty>
9            <triggers>
10 <jenkins.triggers.ReverseBuildTrigger>
11     <spec></spec>
12     <upstreamProjects>My Android Pipeline</upstreamProjects>
13     <threshold>
14         <name>SUCCESS</name>
15         <ordinal>0</ordinal>
16         <color>BLUE</color>
17          <completeBuild>true</completeBuild>
18     </threshold>
19 </jenkins.triggers.ReverseBuildTrigger>
20 </triggers>
21         </org.jenkinsci.plugins.workflow.job.properties.
22                 PipelineTriggersJobProperty>
23     </properties>
24     <definition class="
25         org.jenkinsci.plugins.workflow.cps.CpsFlowDefinition">
```

```
26          <script>
27 #!/usr/bin/env groovy
28 node {
29     step([$class: 'CopyArtifact',
30         filter: '**/*release.apk',
31         fingerprintArtifacts: true,
32         projectName: 'My Android Pipeline',])
33     try {
34         stage('Alpha Release Stage') {
35             lock('googlePlay') {
36                 androidApkUpload googleCredentialsId: 'google-play',
37                 apkFilesPattern: '**/*release.apk',
38                 trackName: 'alpha'
39             }
40         }
41         stage('Beta Release Stage') {
42             input 'approve Beta Release Stage'
43             lock('googlePlay') {
44                 androidApkMove googleCredentialsId: 'google-play',
45                 fromVersionCode: false,
46                 apkFilesPattern: '**/*release.apk',
47                 trackName: 'beta'
48             }
49         }
50         stage('Production Stage') {
51             sleep(time: 7, unit: 'DAYS')
52             lock('googlePlay') {
53                 androidApkMove googleCredentialsId: 'google-play',
54                 fromVersionCode: false,
55                 apkFilesPattern: '**/*release.apk',
56                 trackName: 'production'
57             }
58         }
59     }
60     finally {
61         deleteDir()
62     }
63 }</script>
64          <sandbox>false</sandbox>
65     </definition>
66     <triggers/>
67     <disabled>false</disabled>
68 </flow-definition>
```

Listing 7.3: Beispiel einer generierten Publishing Job XML-Datei.

Anhang C Beispiel einer generierten Credentials XML-Datei

```
<org.jenkinsci.plugins.plaincredentials.impl.StringCredentialsImpl>
  <scope>GLOBAL</scope>
  <id>app-signing-key-password</id>
  <description>The app signing key password</description>
  <secret>keyPw</secret>
</org.jenkinsci.plugins.plaincredentials.impl.StringCredentialsImpl>
```

Listing 7.4: Beispiel einer generierten Credentials XML-Datei.

Anhang D Gradle-Konfiguration der OSR Playground Applikation

```
apply plugin: 'com.android.application'

ext.versionMajor = 1
ext.versionMinor = 0
ext.versionPatch = 0
ext.jenkinsBuildNumber = Integer.valueOf(System.env.BUILD_NUMBER ?: 0)

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.0"
    defaultConfig {
        applicationId "de.awesome.osr_playground"
        minSdkVersion 15
        targetSdkVersion 25
        versionName computeVersionName()
        versionCode computeVersionCode()
        testInstrumentationRunner
            "android.support.test.runner.AndroidJUnitRunner"
    }
    signingConfigs {
        release {
            storeFile file("../android.jks")
            keyAlias "AndroidKey"
            storePassword System.env.RELEASE_KEYSTORE_PASSWORD
            keyPassword System.env.RELEASE_KEY_PASSWORD
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile
                    ('proguard-android.txt'), 'proguard-rules.pro'
            signingConfig signingConfigs.release
        }
        debug {
            versionNameSuffix " (build ${jenkinsBuildNumber})"
        }
    }
}


```

```
42 // Returns name based on version values, e.g. '1.4.0'
43 def computeVersionName() {
44     return "${versionMajor}.${versionMinor}.${versionPatch}"
45 }
46 // Returns auto-incrementing value, e.g 140017 for Jekins build #17
47 def computeVersionCode() {
48     return (versionMajor * 100000) + (versionMinor * 10000) +
49         (versionPatch * 1000) + jenkinsBuildNumber
50 }
51
52 dependencies {
53     compile 'com.android.support:appcompat-v7:25.+'
54     // furher android dependencies (...)
55 }
```

Listing 7.5: build.gradle-Datei der OSR Playground Applikation.

Anhang E Inhalt des Datenträgers

Auf dem beigefügten Datenträger befinden sich die im Rahmen dieser Arbeit erstellten Quellcode- und LaTeX-Dateien. Hierfür wurde folgende Ordnerstruktur gewählt.

`source` Dieses Verzeichnis enthält die Java-Quellen der Android Continuous Delivery DSL Engine sowie der OSR Playground Applikation. Zusätzlich sind die verwendeten Dockerfiles, Shell-Skripte und Gradle-Konfigurationen in den jeweiligen Unterordnern zu finden.

`library` In diesem Ordner befindet sich die Android Continuous Delivery DSL Engine in Form einer Jar-Datei.

`latex` Unter diesem Verzeichnis sind alle LaTeX-Dateien zum Erstellen dieser Arbeit vorhanden.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

Android Developers. (2017a). *Android, the world's most popular mobile platform.* Zugriff 1. Oktober 2017 unter https://developer.android.com/about

Android Developers. (2017b). *Configure Your Build.* Zugriff 1. November 2017 unter https://developer.android.com/studio/build/index.html

Android Developers. (2017c). *Google Play Console.* Zugriff 1. Oktober 2017 unter https://developer.android.com/distribute/console

Android Developers. (2017d). *Sign Your App.* Zugriff 18. November 2017 unter https://developer.android.com/studio/publish/app-signing.html

Android Developers. (2017e). *Testing Apps on Android.* Zugriff 1. November 2017 unter https://developer.android.com/training/testing/index.html

Android Developers. (2017f). *Testing UI Performance.* Zugriff 3. Dezember 2017 unter https://developer.android.com/training/testing/performance.html

Android Developers. (2018). *Dashboards.* Zugriff 25. Januar 2018 unter https://developer.android.com/about/dashboards/index.html

CloudBees. (2017). *About Jenkins.* Zugriff 1. Oktober 2017 unter https://www.cloudbees.com/jenkins/about

Duvall, P., Matyas, S. M. & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series).* Addison-Wesley Professional.

Firebase. (2017). *Firebase Test Lab for Android.* Zugriff 3. Dezember 2017 unter https://firebase.google.com/docs/test-lab/

Fitzgerald, B. & Stol, K.-J. (2014). Continuous Software Engineering and Beyond: Trends and Challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering* (S. 1–9). RCoSE 2014. Hyderabad, India: ACM. doi:10.1145/2593812.2593813

Fowler, M. (2010). *Domain Specific Languages* (1st). Addison-Wesley Professional.

Fowler, M. & Foemmel, M. (2006). Continuous Integration. Zugriff unter https://www.martinfowler.com/articles/continuousIntegration.html

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman.

Geiss, M. (2012). Continuous Integration and Testing for Android.

Google. (2017). *Android Open Source Project*. Zugriff 1. Oktober 2017 unter https://source.android.com/source/licenses

Google Developers. (2017). *APKs and Tracks*. Zugriff 3. Dezember 2017 unter https://developers.google.com/android-publisher/tracks

Hambrick, R. (2015). *Publish to Google Play – Continuous Delivery for Android*. Zugriff 3. Dezember 2017 unter https://stablekernel.com/publish-to-google-play-continuous-delivery-for-android-part-4/

Hechtel, E. (2016). *Mobile Device Emulator and Simulator vs Real Device*. Zugriff 10. Januar 2018 unter https://saucelabs.com/blog/mobile-device-emulator-and-simulator-vs-real-device

Humble, J. & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st). Addison-Wesley Professional.

Jenkins. (2016). *Support for building tags*. Zugriff 10. Januar 2018 unter https://issues.jenkins-ci.org/browse/JENKINS-34395

Jenkins. (2017a). *Pipeline*. Zugriff 24. September 2017 unter https://jenkins.io/doc/book/pipeline

Jenkins. (2017b). *Using Docker with Pipeline*. Zugriff 1. Oktober 2017 unter https://jenkins.io/doc/book/pipeline/docker

Kinnunen, S. & Brunner, G. (2017). *STF - Smartphone Test Farm*. Zugriff 22. Dezember 2017 unter https://openstf.io/

Klepper, S., Krusche, S., Peters, S., Bruegge, B. & Alperowitz, L. (2015). Introducing Continuous Delivery of Mobile Apps in a Corporate Environment: A Case Study. In *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering* (S. 5–11). RCoSE '15. Florence, Italy: IEEE Press. Zugriff unter http://dl.acm.org/citation.cfm?id=2820678.2820681

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*.

Mouat, A. (2016). *Docker: Software entwickeln und deployen mit Containern*. Dpunkt.Verlag GmbH.

Niederwieser, P. (2013). *Building a Continuous Delivery Pipeline with Gradle and Jenkins*. Zugriff 18. November 2017 unter https://www.infoq.com/presentations/cd-gradle-jenkins

Open Signal. (2015). *Android Fragmentation Report August 2015*. Zugriff 1. Oktober 2017 unter https://opensignal.com/reports/2015/08/android-fragmentation/

Orr, C. (2014). *Building, Testing and Deploying Android Apps with Jenkins*. Zugriff 5. November 2017 unter https://www.cloudbees.com/sites/default/files/2014-0625-Berlin-Christopher_Orr-Build_Test_Deploy_w-Android.pdf

Orr, C. (2018). *Google Play Android Publisher Plugin*. Zugriff 20. Januar 2018 unter https://wiki.jenkins.io/display/JENKINS/Google+Play+Android+Publisher+Plugin

Rabenhorst, S. & Steffens, A. (2016). The Impact of Context on Continuous Delivery. *Full-scale Software Engineering/Current Trends in Release Engineering*, 51.

Rossi, C., Shibley, E., Su, S., Beck, K., Savor, T. & Stumm, M. (2016). Continuous Deployment of Mobile Software at Facebook (Showcase). In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (S. 12–23). FSE 2016. Seattle, WA, USA: ACM. Zugriff unter http://doi.acm.org/10.1145/2950290.2994157

Schmidt, J. (2013). *Gradle als Googles Build-System für Android*. Zugriff 20. Januar 2018 unter https://www.heise.de/developer/meldung/Gradle-als-Googles-Build-System-fuer-Android-1867576.html

Smart, J. (2011). *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media.

Statista. (2018). *Google Play Store - Anzahl der Apps 2018*. Zugriff 12. Januar 2018 unter https://de.statista.com/statistik/daten/studie/74368/umfrage/anzahl-der-verfuegbaren-apps-im-google-play-store/

Summers, M. (2016). *The Many Challenges of Android Development*. Zugriff 1. Oktober 2017 unter https://magora-systems.com/challenges-of-android-development

Tappforce. (2017). *iOS and Android Continuous Delivery*. Zugriff 12. Januar 2018 unter https://hackernoon.com/ios-android-continuous-delivery-da1366f7e6e5

Zachow, M. (2016). Designing an Android Continuous Delivery pipeline. In *Software Engineering (Workshops)* (S. 160–163).