Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 22

Entwicklung eines Systems zur kontinuierlichen Integration für autonome Roboter

Kevin Wenz

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Dr. Peter Biber *Robert Bosch GmbH* Dipl.-Ing. Jan-Peter Ostberg *Universität Stuttgart*
Beginn am:	15. Januar 2015
Beendet am:	17. Juli 2015
CR-Nummer:	D.2.5, D.2.9, I.2.9

Kurzfassung

Autonome Roboter basieren auf dem komplexen Zusammenspiel vieler Sensoren. Dieses Zusammenspiel muss durch Software beobachtet und geregelt werden. Damit Roboter sich autonom – ohne ständige Überwachung – bewegen können, muss die Software ihre Funktion fehlerfrei ausführen. Um dies zu unterstützen, wurde im Rahmen dieser Arbeit ein *Continuous Delivery*-Prozess entwickelt. Dieser Prozess sieht vor, dass die Software des Roboters „ständig“ und automatisiert geprüft wird. Ein besonderer Fokus lag dabei auf der Entwicklung eines Funktionstestsystems für Robotersoftware.
Dieses Testsystem führt Testfälle aus, die auf Basis von Szenarien, bestehend aus einer Aufgabe, einem Kontext und mehreren Metriken, modelliert werden. Am Ende wurde der Nutzen des Testsystems durch Robotersoftware-Entwickler evaluiert.

Abstract

The behavior of an autonomous robot is determined by many sensors that scan the robot's environment. Data produced by these sensors needs to be accessed by complex software. Testing software is a very important aspect — especially when its target is an autonomous interacting device — to verify if the robot's software behaves in the right manner. *Continuous Delivery* is a process which tries to improve the procedure of writing and verifing the functionality of software.
In this thesis, a test process — based on *Continuous Delivery*— is presented that enables developers to test their software automatically on a regular basis. The functionality of software for autonomous robots can be tested by providing a scenario consisting of a task, a context and some metrics. In the end, the whole process was evaluated by developers.

Danksagung

Mein erster Dank gilt Prof. Dr. Stefan Wagner dafür, dass er die Arbeit ermöglicht hat und sich, obwohl das Thema nicht zu seinen Schwerpunkten gehört, als Prüfer zur Verfügung gestellt hat. Dr. Peter Biber möchte ich für die Idee zur Arbeit, für die gute Betreuung, sein Vertrauen und für die Möglichkeit, meine Masterarbeit im industriellen Kontext durchführen zu können, danken. Ebenfalls ein großer Dank geht an Jan-Peter Ostberg für die Betreuung der Arbeit und sein Feedback in vielfältigen Dingen, obwohl das Thema auch nicht in seinem Fachgebiet liegt. Danke auch an Maximilian Wenger für den regen Austausch zu Ideen und zu Implementierungsdetails. Allen Teilnehmern der Evaluation sei gedankt, dass sie sich die Zeit zum Ausprobieren des Testsystems genommen haben.

Ein großer Dank gilt meinen Eltern die mir über die gesamte Studienzeit viel Rückhalt gegeben, und mich sowohl mit Tipps als auch finanziell unterstützt haben. Meiner Schwester Annika Wenz sei gedankt, da sie mir immer mit ihrem Rat zur Seite steht und mit ihrer Sprachkenntnis viel zur Lesbarkeit dieser – und anderer – Arbeiten beigetragen hat. Außerdem möchte ich mich bei allen Personen bedanken, denen ich im Laufe meines Studiums begegnen durfte und die sowohl an der Universität als auch abseits für Abwechslung gesorgt haben.

Inhaltsverzeichnis

1. Einleitung **9**
- 1.1. Motivation . . . 9
- 1.2. Ziel . . . 10
- 1.3. Vorgehen . . . 10
- 1.4. Aufbau der Arbeit . . . 12

2. Grundlagen **13**
- 2.1. *Continuous Delivery* . . . 13
- 2.2. Jenkins . . . 17
- 2.3. Tests . . . 18
- 2.4. Robot Operating System . . . 23

3. Verwandte Arbeiten **27**
- 3.1. Continuous Delivery . . . 27
- 3.2. Testen . . . 28
- 3.3. Zusammenfassung . . . 30

4. Ist-Zustand **31**
- 4.1. Systeme und Umfeld . . . 31
- 4.2. Zusammenfassung . . . 34

5. *Continuous Delivery*-System **35**
- 5.1. Anforderungen . . . 35
- 5.2. *Continuous Delivery*-Prozess . . . 38
- 5.3. Testsystem . . . 39
- 5.4. Zusammenfassung . . . 50

6. Evaluierung **51**
- 6.1. Befragung . . . 51
- 6.2. Ergebnisse . . . 52
- 6.3. Threats to Validity . . . 54
- 6.4. Diskussion . . . 55

7. Fazit **57**
- 7.1. Zusammenfassung . . . 57
- 7.2. Ausblick . . . 58

A. Fragenbogen **59**
A.1. Ist-Analyse 59
A.2. Evaluation 62

B. Glossar und Abkürzungsverzeichnis **65**
Glossar 65
Abkürzungsverzeichnis 65

Literaturverzeichnis **67**

Abbildungsverzeichnis

1.1. Gantt-Diagramm 11

2.1. Abgrenzung der Begriffe *Continuous Integration*, *Continuous Delivery* und *Continuous Deployment* 14
2.2. Jenkins Konfiguration 18
2.3. Der funktionale Test 19
2.4. Ablauf eines Unittests 20

5.1. *Continuous Delivery*-Prozess 39
5.2. Jenkins Ergebnisausgabe 40
5.3. Testprozess 41
5.4. Testsystem-Architektur 41

6.1. Evaluation: Antworten 1 53
6.2. Evaluation: Antworten 2 53

Tabellenverzeichnis

5.1. Anforderungsliste 36
5.2. Szenariendefinition 37

Verzeichnis der Listings

2.1. Formulierung eines Akzeptanzkriteriums 22
2.2. Definition einer ROS-Message 25
2.3. Definition eines ROS-Service-Types 25
2.4. Veröffentlichen einer ROS-Message 25
2.5. Abbonieren eines Themas 25

5.1. Jenkins-Webhook-URL 39
5.2. YAML-Konfigurationsdatei 43
5.3. C++ „fork() und execvp()“-Konzept 44
5.4. Definition einer ROS-Message 46
5.5. Publish einer ROS-Message 47
5.6. Einbetten von Matlab-Skripten in C++ 49
5.7. Nutzung von Matlab und ROS 49
5.8. Einbettung von Python in C++ 50
5.9. Nutzung von Python und ROS 50

1. Einleitung

Dieses Kapitel bildet eine Einführung in diese Arbeit. Hierzu gehört die Motivation (Abschnitt 1.1), welche die Idee hinter dieser Arbeit darstellt, gefolgt vom Abschnitt 1.2, welcher das Ziel dieser Arbeit beschreibt und die Aufgabenbeschreibung, die Abgrenzung und die verwendete Methodik enthält. Im Abschnitt 1.3 wird das zeitliche und methodische Vorgehen vorgestellt. Im letzten Teil des Kapitels (Abschnitt 1.4) findet sich eine kurze Beschreibung des Aufbaus der restlichen Arbeit.

1.1. Motivation

In jedem produzierenden Gewerbe ist die Qualität und die Funktionstüchtigkeit des entstehenden Produkts eines der wichtigsten Anliegen der Produzenten. Auch in der Softwareentwicklung gilt es, ein qualitativ hochwertiges Produkt zu entwickeln, um zum einen die Wartung der Software in der Zukunft zu erleichtern, zum anderen aber die Funktionalität der Software von Beginn an sicherzustellen. Für die Sicherstellung dieser Funktionalitäten bietet die Softwareentwicklung, über den Entwicklungszyklus einer Software hinweg, verschiedene Ansätze. Zu Beginn der Entwicklung spielt die Anforderungsanalyse eine wichtige Rolle, um die gewünschten Funktionen und deren Priorität beim Kunden zu ermitteln. Diese Analyse endet in der Spezifikation. Auf Basis dieser Spezifikation werden Tests definiert, welche sicherstellen, dass eine entwickelte Software die nötigen Anforderungen erfüllt. Der Softwaretest stellt somit eine direkte Kontrolle der Wünsche des Kunden dar.

In der Robotik spielt die Funktionalität der Roboter eine sehr wichtige Rolle. Im Besonderen in der Entwicklung von autonomen Robotern, welche selbständig und ohne ständige Kontrolle durch Menschen arbeiten, muss die volle und sichere Funktionsfähigkeit gewährleistet sein. Um selbständig tätig sein zu können, benötigt ein autonomer Roboter viele Sensoren, die die Umgebung wahrnehmen. Diese Sensoren müssen in der richtigen Weise zusammenspielen und bilden daher eine komplexe Software, welche nur schwer modularisiert und daher nur schwer in Teilen getestet werden kann. Sich häufig verändernde Bedingungen in der Umgebung ergeben insbesondere im Außenbereich stark schwankende Werte an der Sensorik. So liefert beispielsweise eine Kamera bei veränderten Lichtverhältnissen deutlich andere Ergebnisse.

In klassischen Vorgehensmodellen der Softwaretechnik, wie dem Wasserfallmodell oder dem V-Modell, findet die Durchführung der Tests und somit die Überprüfung, ob eine Software der Spezifikation entspricht, erst in den späten Teilen der Entwicklung – nach Fertigstellung der Implementierung – statt. Die Tests finden daher erst ihre Anwendung, wenn die Erstellung der Software bereits erfolgt ist. Agile Methoden versuchen, dem in verschiedenen Ansätzen entgegenzutreten und die harte Trennung der Entwicklungsphasen aufzulösen. So bietet das Themenfeld der „Continuous“-Prozesse die Möglichkeit, Softwaretests von Beginn an mit zu entwickeln und kontinuierlich einzusetzen.

1.2. Ziel

In dieser Arbeit soll ein Prozess erstellt werden, welcher die Entwicklung der Software eines autonomen Rasenmähers unterstützt. Als Vorbild zu diesem Prozess soll *Continuous Delivery*, welches ein kontinuierliches Testen von Software ermöglicht, dienen. Hierzu gehört die Entwicklung eines Vorgehens für das Erstellen neuer Softwarefunktionalitäten sowie Softwareverbesserungen. Eine der Hauptaufgaben ist die Entwicklung eines Systems zur Durchführung von Funktionstests.

Abgrenzung

In diesem Abschnitt folgt die Abgrenzung der Dinge, die in der vorliegenden Arbeit bearbeitet werden sollen zu denen, die über die vorliegende hinaus gehen.

Continuous Delivery Es wird ein *Continuous Delivery*-System entwickelt. Die verwendete Definition des Begriffs *Continuous Delivery* stützt sich dabei auf das Buch „Continuous Delivery“ von Humble und Farley [HF10]. Dies bedeutet, dass der zu erstellende Prozess vom Einchecken der Software bis hin zum Funktionstest – im *Continuous Delivery* der Akzeptanztest – unterstützt werden soll (Abschnitt 2.1). Einzelne Teile, welche für den Prozess benötigt werden sind bereits vorhanden. So kann zum Beispiel ein Jenkins-Integrationsserver bereitgestellt werden. Diese vorhandenen Ressourcen müssen geprüft werden und gegebenenfalls eingearbeitet werden.

Testen Ein wichtiger Bestandteil des *Continuous Delivery* stellt der Akzeptanztest dar. Getestet werden soll nicht mittels Unittests, sondern auf Basis von *Highlevel*-Tests. Trotzdem soll beim Durchführen der Tests keine Hardware zum Einsatz kommen. Statt auf Hardware wird in einer Simulationsumgebung, welche die Umgebung und somit die Sensordaten simuliert und das Simulierte darstellt, getestet. Die Definition der Testfälle erfolgt in Form von konkreten Szenarien, welche der Roboter meistern können muss. Dabei sind Ein- und Ausgabewerte in vielen Fällen keine konkrete Werte, sondern ungefähre Daten.

1.3. Vorgehen

Die gesamte Arbeit erstreckte sich vom 15.01.2015 bis 17.07.2015. Der Großteil der Arbeit wurde am Bosch-Standort in Renningen durchgeführt, da das zu testende System sowie die gebotene Infrastruktur dort vorhanden war. Eine genauere Einteilung der verschiedenen Phasen findet sich in Abbildung 1.1. Die drei größeren Abschnitte – Einarbeitung, Implementierung und Ergebnis – geben die Hauptphasen an.

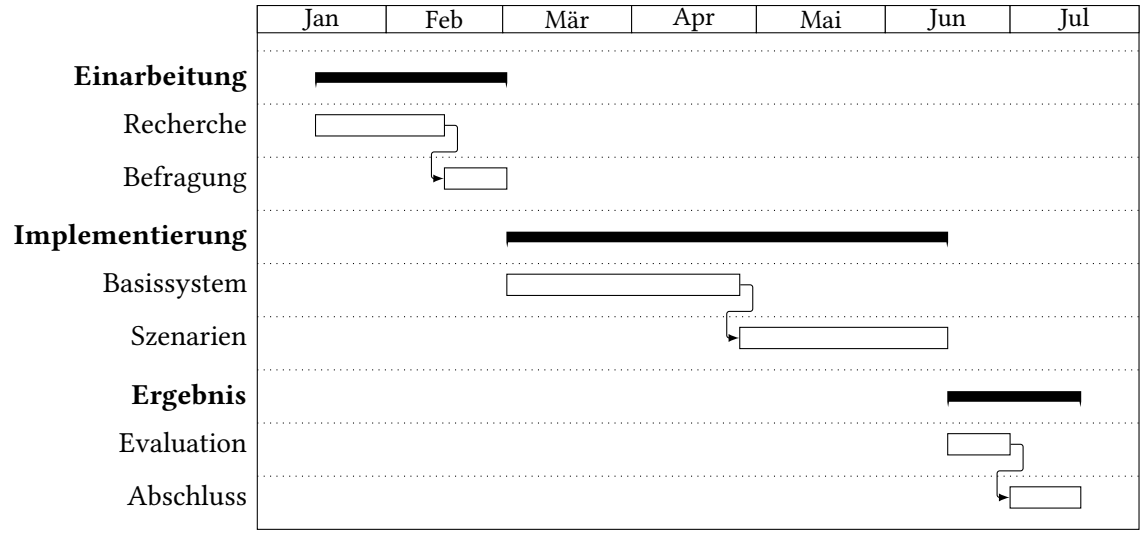


Abbildung 1.1.: Dieses Gantt-Diagramm enthält den Verlauf der Arbeit. Es wird der sequenzielle Ablauf und die Länge der Phasen aufgetragen.

Einarbeitung

Der Einarbeitungsphase lagen zwei Hauptaufgaben zu Grunde. Zum einen war es wichtig, einen Überblick über das gesamte Themenfeld zu erhalten. Die Schwerpunkte lagen dabei bei der generellen Einarbeitung in das Thema „*Continuous Delivery* “. Davon ausgehend schloss sich vor allem eine Recherche zum Thema „Testen von Robotern“ an. Eine detaillierte Literatur-Recherche findet sich im Kapitel 3. Zum anderen lag ein wichtiger Schritt darin, im Allgemeinen das Testen von Robotersoftware und im Speziellen das *System-Under-Test (SUT)* kennen zu lernen. Hierfür wurden Mitarbeiter verschiedener Abteilungen nach ihren Anforderungen an ein Testsystem befragt, und später regelmäßig um Feedback gebeten. Dieser Prozess und dessen Ergebnis finden sich in Kapitel 4.

Implementierung

In der Implementierungsphase wurde zuerst die allgemeine Systemarchitektur entworfen und implementiert. Die dabei getroffenen Entscheidungen basieren auf den Erkenntnissen aus der Befragung. In einem weiteren Schritt wurden die verschiedenen Komponenten des Systems auf ihre Implementierungsmöglichkeiten überprüft und anschließend evaluiert. Das Vorgehen der Implementierungsphase und deren Herausforderungen werden in Abschnitt 1.3 beschrieben.

Ergebnis

Den Abschluss dieser Arbeit bildet die Ausarbeitung der Ergebnisse. Zu den Ergebnissen gehört das im vorherigen Abschnitt beschriebene System, aber auch die Evaluierung des Nutzens des Systems.

Methodik Die Evaluation wird durch eine Befragung von Entwicklern durchgeführt. Die Dauer der Evaluierungsphase betrug zwei Wochen. In dieser Phase wurden fünf Mitarbeiter aus zwei Abteilungen gebeten, reale Testfälle auf Basis echter Robotersoftware zu entwerfen. Die Befragung besteht aus folgenden Phasen: Vorstellung des Systems, Testen des Systems durch die Entwickler und Feedback durch die Entwickler. Der Vorgang der Evaluation und deren Ergebnis finden sich in Kapitel 6.

1.4. Aufbau der Arbeit

Die vorliegende Arbeit ist folgendermaßen aufgebaut: Kapitel 1 enthält das Ziel und das Vorgehen der Arbeit. In Kapitel 2 finden sich Grundlagen, die für das Verständnis der Arbeit nötig sind. Das dritte Kapitel beschreibt Arbeiten, die thematisch verwandt zur vorliegenden Arbeit sind. Kapitel 4 enthält eine Analyse des aktuellen Prozesses der Softwareentwicklung. Im 5. Kapitel werden der entwickelte Prozess sowie das Testsystem vorgestellt und Implementierungsdetails begründet. Anschließend (Kapitel 6) folgt die Evaluation des entwickelten Systems und in Kapitel 7 die Zusammenfassung der Arbeit und den Ausblick für nachfolgende Projekte. Im Appendix befinden sich die zur System-Analyse (Anhang A.1) und Evaluation (Anhang A.2) verwendeten Fragebögen sowie ein Glossar (Anhang B) zur Klärung von Begrifflichkeiten.

2. Grundlagen

Dieses Kapitel behandelt Themen, welche zum Verständnis der Arbeit wichtig sind. Dabei beschränkt sich dieses Kapitel auf Informationen, welche in aller Regel aus Fachliteratur, wissenschaftlichen Papieren oder bei Technologien aus deren Wiki-Seiten entnommen wurden. Es werden „Best-Practices“ und gängige Technologien vorgestellt. An entsprechenden Stellen finden sich Referenzen auf tiefer gehende Literatur.

In Abschnitt 2.1 wird, das Hauptthema der Arbeit, *Continuous Delivery*, behandelt. Ergänzend werden dabei die Begriffe „*Continuous Integration* “ und „*Continuous Deployment* “ besprochen. Ausgehend von diesem Abschnitt lassen sich die darauf folgenden Abschnitte ableiten. So befasst sich Abschnitt 2.2 mit dem *Continuous Integration*-Server Jenkins. Darauf folgt, in Abschnitt 2.3, der umfangreichste Teil dieser Arbeit, das Testen, insbesondere der Akzeptanztest. Abschließend wird im Abschnitt 2.4 das vom *Continuous Delivery* losgelöste Thema *Robot Operating System (ROS)*, das im Kontext dieser Arbeit verwendete Roboterbetriebssystem, behandelt.

2.1. *Continuous Delivery*

Continuous Delivery beschreibt einen automatisierten Prozess zum Verwalten und Bauen von Software. Dieser Abschnitt erklärt diesen Prozess, wie er im weit verbreiteten Buch „Continuous Delivery“ von Humble und Farley [HF10] sowie in „Continuous Integration“ von Duvall et. al [DMG07] diskutiert wird.

Neely und Stolt beschreiben *Continuous Delivery* als „... the ability to release software whenever we want“ [NS13]. Dies drückt aus, dass es bei *Continuous Delivery* darum geht, Quellcode immer in einem Zustand zu halten, der es ermöglicht, eine lauffähige Version der Software zu bauen und auszuführen. Das Konzept kommt aus dem Bereich der Webentwicklung. Dort ist es üblich, die Release-Zyklen einer Software auf wenige Wochen zu setzen. In einigen Firmen wird sogar mehrmals täglich Software freigegeben [Jen11, Chu13]. Ein Beispiel hierfür ist *Facebook Inc.*. Bei Facebook wird Software teilweise täglich freigegeben. Dabei ist jeder Entwickler für den gesamten Quellcode verantwortlich. Um Fehler und Ausfälle gering zu halten dürfen immer nur kleine Quellcode-Änderungen durchgeführt werden. Dies ermöglicht, das schnelle Lokalisieren von Fehlern [FFB13].

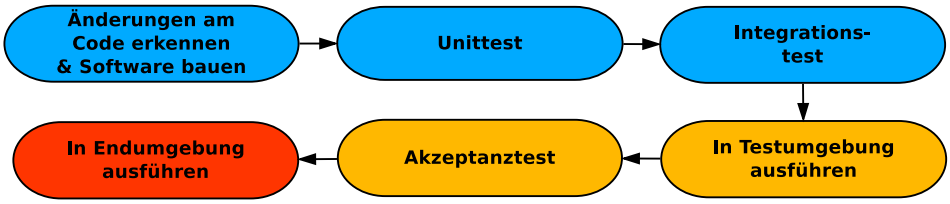


Abbildung 2.1.: Abgrenzung der Begriffe *Continuous Integration*, *Continuous Delivery* und *Continuous Deployment*, basierend auf [Pul13]. *Continuous Integration* besteht aus den Aufgaben in Blau, *Continuous Delivery* aus den blau und orange gefärbten und *Continuous Deployment* aus allen sechs Aufgaben.

Pipeline

Im Allgemeinen werden drei Begriffen unterschieden: *Continuous Integration*, *Continuous Delivery* und *Continuous Deployment*. Diese Begriffe stützen sich jeweils auf Teile des selben Prozesses. Der Hauptunterschied liegt im Grad der Automatisierung. *Continuous Integration* bildet die Grundlage und beschreibt die ersten Schritte des Prozesses. Bei *Continuous Delivery* wird *Continuous Integration* um den Akzeptanztest erweitert. Mit *Continuous Deployment* lässt sich der vollständige automatisierte Prozess beschreiben. Ville Pulkkinen [Pul13] hat diese Unterscheidung visualisiert. Abbildung 2.1 stellt in Anlehnung daran die Unterschiede graphisch dar.

Continuous Integration Bei *Continuous Integration* werden drei Schritte automatisiert durchgeführt.

- Zu Beginn steht das Anstoßen der Pipeline. Dies geschieht in aller Regel durch den *Commit von neuem Quellcode*, kann aber auch manuell durch einen Entwickler durchgeführt werden. Dabei werden Jobs im *Continuous Integration*-Server gestartet, welche die weiteren Aufgaben ausführen.
- Im zweiten Schritt steht der *Unittest*. Unittests werden kleine Einheiten – beispielsweise einzelne Klassen oder Komponenten – einer Software getestet.
- Der *Integrationstest* fasst einige kleine Tests zusammen und stellt sicher, dass die einzelnen Komponenten des komplexen Systems und alle Schnittstellen zusammenarbeiten. Die Ergebnisse der beiden Testarten Unittest und Integrationstest werden daraufhin zusammengefasst und im *Continuous Integration*-Server dargestellt.

Continuous Delivery Im Vergleich zu *Continuous Integration* geht *Continuous Delivery* einen Schritt weiter und testet zusätzlich die Funktion der Software.

- Dazu wird die Software zuerst in eine *Test-Umgebung überführt*. In dieser Umgebung soll festgestellt werden, ob die Software ihre Funktion erfüllt. In verschiedenen Umgebungen verhält sich Software unterschiedlich. Daher ist es wichtig, dass eine Software beim Funktionstest in einer zur Einsatz-Umgebung ähnlichen Umgebung ausgeführt wird. Das heißt, dass Betriebssystems- und Bibliotheksversionen sowie eingesetzte Technologien mit der zukünftigen Umgebung übereinstimmen sollten.

- Diese Funktionstests werden *Akzeptanztests* genannt. Während Unit- und Integrationstests einen starken Fokus auf die Code- beziehungsweise Architekturebene legen, fokussiert sich der Akzeptanztest den Fokus auf die Sicht des Kunden und späteren Nutzers. Dafür werden Szenarien und Nutzer-Stories definiert, welche typische Nutzerinteraktionen mit dem System abbilden. Die Testfälle des Akzeptanztests sollten die Funktionen des Systems so gut abbilden, dass am Ende die volle Funktionalität des Systems sichergestellt ist.

Continuous Deployment Die Abrundung der Pipeline bildet *Continuous Deployment*.

- Zusätzlich zu den vorangegangenen Schritten wird die in *Continuous Integration* und *Continuous Delivery* ausreichend getestete Software in die Ausführungsumgebung überführt und somit für die Nutzer zugänglich gemacht. Aufgrund der Häufigkeit, in der Tests ausgeführt werden, und aufgrund der kleinen Änderungen pro Pipelinedurchlauf kommen Fehler in diesem Zustand nur sehr selten vor.

Ziele

Moderne Software besteht in der Regel aus vielen einzelnen Komponenten, welche miteinander interagieren müssen. Auch wenn jede einzelne Komponente für sich selbst funktioniert, ist es oft nicht möglich, die einzelnen Komponenten gemeinsam und somit die Software als Ganzes zu testen oder zu nutzen. Der Aufwand, der betrieben werden müsste, um dies zu ermöglichen, ist eine zu große Hemmschwelle für Entwickler [HF10]. Dies führt dazu, dass am Ende der Entwicklung alles in einer großen *Big-Bang*-Integration zusammengeführt wird. Dabei werden Probleme sichtbar, welche in einem frühen Stadium bereits hätten ausgemerzt werden können. Außerdem ist es zu keinem Entwicklungszeitpunkt möglich, die entwickelte Software zu testen oder zu präsentieren. Genau diese Probleme will *Continuous Integration* lösen. Software soll während der gesamten Entwicklungszeit für Tests verfügbar sein, ebenso soll die Möglichkeit bestehen, die Software Kunden oder Vorgesetzten zu präsentieren. Carl Caum beschreibt *Continuous Delivery* wie folgt: „Continuous Delivery doesn't mean every change is deployed to production ASAP. It means every change is proven to be deployable at any time"[Cau13].
Humble et al. [HF10] beschreiben in fünf Punkten, welchen Nutzen *Continuous Delivery* für Entwickler bringt:

Befähigen der Entwickler. Eine der Hauptkomponenten ist die Entwicklungspipeline. Sie ermöglicht es, dass jeder Entwickler zu jeder Zeit den aktuellen Stand und die aktuell verfügbaren Ressourcen ermitteln und nutzen kann. Dies führt dazu, dass mehrere Versionen der Software und der Umgebung verfügbar sind. Das wiederum erleichtert das Testen, das Reproduzieren von Fehlern, die Fehlerwiederherstellung und das automatisierte Bauen von Software.

Fehlerreduzierung. Durch ein einheitliches Konfigurationsmanagement werden bestimmte Fehlertypen vermieden. Dafür werden Konfigurationsinformationen (Buildskript, Parametereinstellungen etc.) in einem zentralen Repository gespeichert. So ist es zum Beispiel für neue Mitarbeiter einfacher, gängige Anfängerfehler zu vermeiden.

Stress absenken. Generell wird der Stress, der bei einem neuen Release aufgebaut wird, verringert. Wenn zum Beispiel nach einem neuen Release klar wird, dass die Software zurückgezogen werden muss, kann einfach und schnell eine alte Version der Software nachgezogen werden. Zudem wird durch häufige Releases der Prozess des Releases zur Gewohnheit und Entwickler routinierter in ihren Aufgaben.

Deployment Flexibility. Die Software kann einfach in eine neue Ausführungsumgebung überführt werden. Dazu wird die Konfiguration der neuen Umgebung automatisch bereitgestellt. Dies hat zudem den Vorteil, dass Kunden kürzer auf neue Funktionen und Fehlerkorrekturen warten müssen [NS13].

Praxiserfahrung. Das standardisierte Vorgehen bringt Routine in den Prozessablauf und somit in den Entwicklungsalltag.

Eigenschaften

Die Werke von Humble et al., Fowler et al. und Neely et al. [HF10, FF06, NS13] beschreiben einige Ideen, die hinter *Continuous Delivery* stecken, sowie Voraussetzungen, die erfüllt sein müssen, damit *Continuous Delivery* erfolgreich funktioniert. In diesem Abschnitt werden diese Attribute unter den Stichwörtern *Automatisierung*, *Feedback* und *Philosophie* zusammengefasst.

Automatisierung Der beschriebene Hauptunterschied liegt in der Art, wie eine neue Version einer Software erstellt wird. Während in der traditionellen Softwareentwicklungen das Freigeben einer neuen Version ein bis zwei Tage dauern und viele Nerven kosten kann, findet dieses Freigeben bei *Continuous Delivery* als „non-event“ statt. Das heißt, sie ist keine komplexe Aufgabe, die nur selten erledigt wird. Sie ist im Prinzip „langweilig“ und ohne viel Aufwand zu erledigen. Um dies zu ermöglichen, muss sie einen hohen Grad an Automatisierung aufweisen und einer „One-Click“-Aufgabe entsprechen. Hierzu gehört es, alle Aufgaben klar zu verteilen, alle manuellen Schritte zu identifizieren und in den automatischen Prozess einzubinden. Der Testprozess muss transparent sein und alle Skripte, welche zur Automatisierung nötig sind, müssen zentral und für alle zugänglich, z. B. in einem Repository, gespeichert werden. Dadurch werden Testfälle wiederhol- und reproduzierbar und bilden somit auch eine Grundlage für Regressionstests. Außerdem wird erreicht, dass mit den selben Versionen gearbeitet werden kann und das Verständnis, die Wartung und die Fehlersuche verbessert werden. Automatisierung ermöglicht schnelles Feedback, hohe Qualität und reduziert die Komplexität von z. B. Code-Zusammenführungen.

Feedback Durch die Einführung von *Continuous Delivery* werden Integrationsfehler schnell gefunden und gelöst [FF06]. Um das schnelle Fehler-Finden zu ermöglichen, muss jede Änderung am Quellcode ein Feedback auslösen. Hierfür ist eine schnelle Feedback-Schleife nötig, welche durch Tests abgeschlossen wird. Dadurch werden fehlgeschlagene Builds und Tests sofort an den Entwickler oder das Entwicklerteam gemeldet. Daraufhin müssen die Entwickler dafür sorgen, dass die Fehler beseitigt werden. Unterstützend für das schnelle Feedback wirken kleine Commits, da diese einerseits eine Fehlersuche einfacher machen, und andererseits auch die Möglichkeit eines schnellen Rollbacks bieten, wenn eine neue Version zurückgenommen werden muss. Durch den Feedbackprozess wird die Qualität der Software automatisch, kontinuierlich und frühzeitig verbessert.

Philosophie *Continuous Delivery* ist nicht einfach nur ein veränderter Prozess. Entwickler müssen die „Philosophie“, die hinter *Continuous Delivery* steht, übernehmen. Wenn einer der Entwickler dies nicht tut, wird *Continuous Delivery* schnell frustrierend und scheitert. Ein wichtiges Prinzip sind die kleinen häufigen Commits. Eine Software wird erst als „fertig“ bezeichnet, wenn sie wirklich freigegeben werden kann. Alle Entwickler müssen sich für den *Deliver*-Prozess zuständig fühlen und es darf kein Herumschieben der Verantwortung geben. Wenn anfangs Probleme mit *Continuous Delivery* bestehen, kann es helfen, die Frequenz zu erhöhen, um mehr Routine im Prozess zu bekommen.

2.2. Jenkins

In diesem Abschnitt wird grundlegendes Wissen zum kontinuierlichen Integrationsserver *Jenkins* beschrieben. Diese Informationen basieren auf dem offiziellen Jenkins-Wiki [Jen15].

In seinen Anfängen, im Jahr 2011, wurde Jenkins unter dem Namen *Hudson* durch einen Mitarbeiter der Firma Sun Microsystems entwickelt. Geschrieben wurde Jenkins in Java. Jenkins kann als eigenständiger Server betrieben werden. Das Grundkonzept von Jenkins besteht darin, dass verschiedene Aufgaben – *Jobs* genannt – definiert werden können. Als Bestandteil des *Continuous Integration*-Server richten sich die Jobs hauptsächlich auf das kontinuierliche Bauen und Testen von Software. Dabei unterstützt Jenkins Entwickler beim Einpflegen von Änderungen und Erstellen neuer Builds. Die Funktionalität des Servers kann durch verschiedenste Plugins erweitert werden.

Im Folgenden wird anhand der Jenkins-Konfigurationsseite (Abbildung 2.2) die Hauptfunktionalität des Jenkins-Servers beschrieben:

Source-Code-Management Der Abschnitt Source-Code-Management bietet die Möglichkeit, aktuelle Quelldaten zu erhalten. Als Standard werden hierfür Interfaces zu Versionierungstools wie *Subversion* oder *Git* bereitgestellt.

Build-Auslöser Um einen Job zu starten, können verschiedene Ereignisse gewählt werden. Im Stil von Cron-Jobs können wiederkehrende Ereignisse festgelegt werden. Es ist möglich, über eine API Jobs mittels eines Remote-Aufrufs zu starten. Eine weitere Möglichkeit ist, dass Jenkins aktiv Source-Code-Management-Systeme nach Änderungen prüft. Es ist ebenfalls möglich, dass Abhängigkeiten zwischen Jobs definiert werden, wodurch bestimmte Jobs dann gebaut werden, wenn andere erfolgreich waren.

Buildverfahren Dieser Abschnitt legt fest, was im Build-Schritt gemacht werden soll. In der Standardkonfiguration ist es möglich, Skripte wie Shell oder Batch auszuführen. Dies ermöglicht eine flexible Gestaltung der Ausführung des Jobs. Neben dem Aufrufen von einfachen Build-Befehlen können hier Tests gestartet werden, aber auch das Ausführen von Aufgaben anderer Art ist hier möglich.

Post-Build-Aktionen In diesem Schritt können Aufgaben, die den Buildprozess abschließen, durchgeführt werden. Dies können zum Beispiel Zusammenfassungen der Compiler-Warnungen oder Testergebnissen sein. Es ist aber auch möglich, Dokumentationen zu erstellen, Benachrichtigungen an Entwickler zu schicken oder Artefakte zu speichern, die beim Buildprozess entstanden sind.

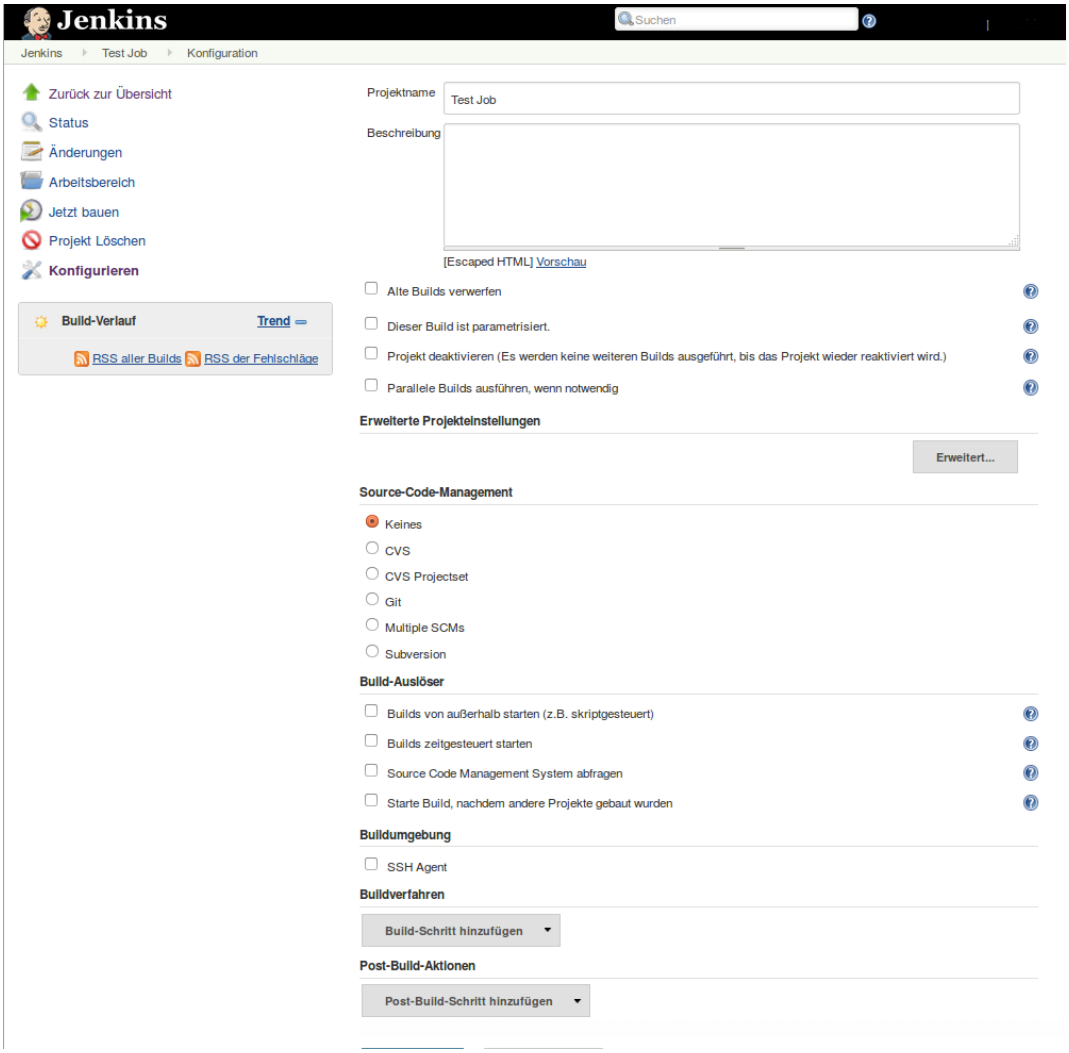


Abbildung 2.2.: Die Konfigurationsseite des Jenkins Integration-Servers. Sie zeigt einen Überblick über die Einstellmöglichkeiten eines Jenkins-Jobs.

2.3. Tests

Testen gehört im Softwareentwicklungsprozess zu den wesentlichen Tätigkeiten, um die Qualität einer Software sicherzustellen. Insbesondere „funktionsorientierte Testmethoden sind dabei unverzichtbar“ [Lig09]. Tests können dabei auf verschiedenen Ebenen – z. B. der Quellcode-, der Komponenten- oder der Systemebene – durchgeführt werden. Auch der Zweck eines Tests kann unterschiedlich sein, beispielsweise Funktions-, Stress- oder Performanztests. Dabei können Tests Maße bereitstellen, welche diese Qualität beschreiben und eine graphische Darstellung unterstützen. Da im *Continuous Delivery*-Prozess der Großteil der manuellen Schritte eliminiert wird, muss die Qualität automatisch und nicht von Menschenhand festgestellt werden. Im *Continuous Delivery*-Prozess spielen Tests deshalb eine wichtige Rolle.

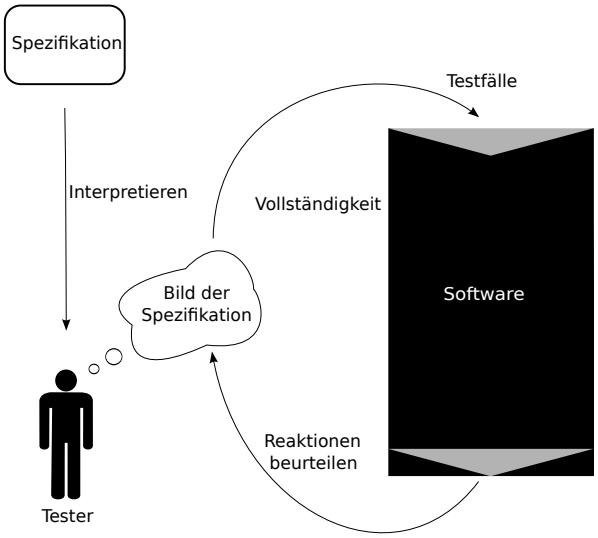


Abbildung 2.3.: Der funktionale Test nach Liggesmeyer [Lig09]. Der Tester leitet konkrete Testfälle aus der Spezifikation ab. Diese werden ausgeführt und müssen im Anschluss auf ihre Richtigkeit geprüft werden.

Nach Liggesmeyer [Lig09] ergeben Komponenten-, Integrations- und Systemtest gemeinsam den funktionalen Test. In Abschnitt 2.1 wurde bereits beschrieben, dass *Continuous Delivery* Unit-, Integrations- und Akzeptanztests erfordert. Dabei ist der Unittest ein Test, welcher Komponenten eines Systems testet, während der Systemtest ähnlich dem Akzeptanztest im *Continuous Delivery* das gesamte System testet. Bei funktionalen Tests bilden die Funktionen, die zu Beginn in der Anforderungsspezifikation festgelegt wurden, die Grundlage. Dabei werden die Software oder deren Teile mit bestimmten Parametern initialisiert und gestartet. Nach Beendigung geht es nun darum, zu prüfen, ob die Ausgabe und das Verhalten den eingegebenen Parametern entsprochen haben. Abbildung 2.3 stellt dies dar. Im folgenden Abschnitt wird auf die Testarten des *Continuous Delivery* eingegangen. Im Besonderen wird der Akzeptanztest als zentrales Element dieser Arbeit betrachtet.

Unittest

Der Unittest beschreibt Tests auf Quellcodeebene. Dabei werden einzelne Funktionen, Klassen oder Komponenten auf ihre Richtigkeit hin überprüft. Der Berufsverband *Institute of Electrical and Electronics Engineers (IEEE)* hat bereits 1987 in seinem *Standard 1008* [IEE87] den Unittest als Test definiert. Er geht dabei von drei Phasen aus, welche zur Durchführung von Unittests nötig sind.

Phase 1: Testfälle planen

- **Testziel** Beschreibt, was das Ziel des Tests ist und wann der Testprozess als abgeschlossen gilt.
- **Testelemente** Festlegung, welche Softwareteile getestet werden sollen.
- **Verfeinerung** Bisher verfügbare Testfälle verfeinern, besondere Anforderungen ermitteln und den Testplan aufstellen.

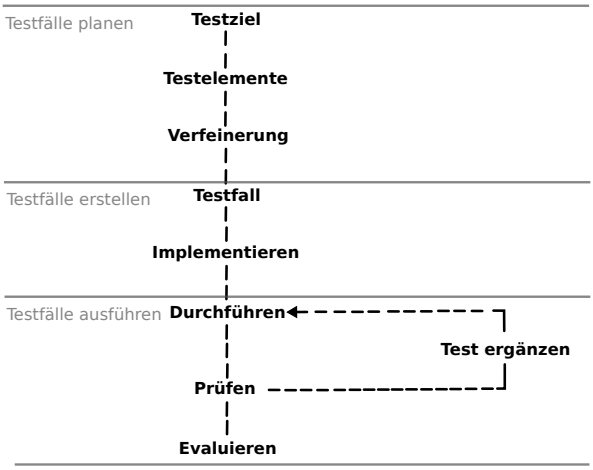


Abbildung 2.4.: Der Ablauf eines Unittests, basierend auf der Definition der IEEE [IEE87]. Im Unittest werden die drei Phasen *Testfälle planen*, *Testfälle erstellen* und *Testfälle ausführen* durchlaufen.

Phase 2: Testfälle erstellen

- **Testfall** Spezifizieren der Testfälle sowie Festlegung der Testarchitektur und der Testprozedur.
- **Implementieren** Festlegen der konkreten Testdaten, sowie aller Ressourcen, die zum Test notwendig sind.

Phase 3: Testfälle ausführen

- **Durchführen** Testfälle ausführen und Ergebnisse abgreifen.
- **Prüfen** Nachvollziehen, welche Testausführungen normal abgelaufen sind und wo es Abweichungen gab.
- **Evaluieren** Status und Ergebnisse der getesteten Komponenten prüfen. Testreport erstellen und Testgegenstände speichern.

Integrationstest

Im V-Modell [Boe84] folgt der Integrationstest auf den Komponententest. Er wird ausgeführt, wenn der Komponententest erfolgreich abgeschlossen wurde. Speziell in großen Softwareprojekten, die aus vielen, von verschiedenen Entwicklern geschriebenen Komponenten bestehen, kann es beim Integrieren – das heißt beim Zusammenfügen dieser Komponenten – zu Fehlern kommen. Der Integrationstest soll daher prüfen, ob diese Komponenten zusammenpassen. Somit werden Fehler in Schnittstellendefinitionen oder deren Implementierung frühzeitig gefunden.

Eine Beschreibung des Integrationstests findet sich im Buch *Basiswissen Software* [SL12, Kapitel 3.3]. Der Integrationstest nutzt als Grundlage das Softwaredesign und die Systemarchitektur. Er kann sich ausschließlich auf Software beziehen, aber auch unter dem Begriff Systemintegrationstest das Zusammenspiel zwischen Hard- und Software prüfen. Eine beispielhafte Fehlerquelle, die beim Integrationstest gefunden wird, sind veraltete oder schwammig formulierte Schnittstellenspezifikationen. Auch ist es möglich, dass der Test scheitert, wenn zwei Komponenten zusammengebaut werden sollen, welche auf verschiedenen Versionen der Schnittstellenspezifikationen basieren. Manche dieser offensichtlichen Fehlerquellen – beispielsweise fehlende Schnittstellen – können jedoch bereits beim Linken der Software erkannt werden. Der Integrationsprozess und somit auch der Integrationstestprozess können vereinfacht gestaltet werden, indem nicht alle Komponenten, wie in der *Big-Bang*-Integration üblich, gleichzeitig zusammengesetzt werden, sondern nacheinander in kleinen Einheiten integriert werden. Die Unterscheidung in Komponentenintegrationstest und Systemintegrationstest liegt nahe. Wenn Komponenten schrittweise integriert werden, werden Testtreiber benötigt, welche nicht integrierte Komponenten simulieren. Gute Testtreiber zeichnen sich dadurch aus, dass sie wiederverwendet werden können und universell einsetzbar sind.

Es gibt drei Fehlertypen, welche bei einem Integrationstest anfallen können:

- Es werden keine oder syntaktisch falsche Daten ausgetauscht
- Ausgetauschte Daten sind syntaktisch korrekt, werden aber vom Gegenüber falsch interpretiert.
- Daten werden korrekt verstanden, sind aber zu einem falschen Zeitpunkt ausgetauscht worden.

Oberflächlich betrachtet kann man zu dem Schluss kommen, dass Integrationstests Komponententests ersetzen können. Dem spricht jedoch entgegen, dass ...

- ... der Zugriff auf einzelne Komponenten in der Integrationstestumgebung nicht vorgesehen und somit erschwert ist.
- ... bei speziellen Komponententests kritische Zustände explizit hergestellt werden können, die bei Integrationstest eventuell nie erreicht werden.
- ... funktionale Fehler zwar auf Fehler in Komponenten zurückzuführen sind, das Finden der Fehler bei Betrachtung großer Teile des Systems jedoch erschwert wird.

Akzeptanztest

In diesem Abschnitt wird der automatisierte Akzeptanztest gesondert beschrieben, da er sowohl in der vorliegenden Arbeit eine zentrale Rolle spielt, als auch im *Continuous Delivery* eine wichtige Bedeutung hat. Der automatisierte Akzeptanztest bildet das Endkriterium des *Continuous Delivery*-Prozesses. Er entscheidet, ob eine Software fehlerfrei ist. Der *Continuous Deployment*-Prozess (Abschnitt 2.1) geht dabei sogar soweit, dass die neue Version einer Software bei erfolgreichem Bestehen des Akzeptanztests automatisch in Betrieb genommen wird.

Spillner et al. [SL12, Kapitel 3.5] beschreiben einige generelle Eigenschaften von Akzeptanztests. Der Akzeptanztest orientiert sich im Gegensatz zum Komponenten- und Integrationstest am Kunden. Das bedeutet, dass die Wünsche des Kunden im Mittelpunkt stehen und Testfälle auf konkreten

Listing 2.1 In der Formulierung eines Akzeptanzkriterium werden sowohl Vor- und Nachbedingung als auch das entsprechende Ereignis angegeben.

```
Gegeben ...
Wenn ...
Dann ...
```

Anforderungen des Kunden aufbauen. Per Definition wird mit dem Bestehen des Akzeptanztests die Richtigkeit einer Software bestätigt. Aufgrund seiner Verständlichkeit für den Kunden eignet sich der Akzeptanztest auch als Test, der die Validierung durch den Kunden mit einschließt. Die Validierung kann dabei zum einen über Interaktionsszenarien auf der Benutzeroberfläche, zum anderen aber auch über die Definition einzelner Funktionsanforderungen vorgenommen werden. In diesem Fall ist es möglich, den Akzeptanztest aus den Anforderungen (automatisch) zu generieren. Andere Quellen für Testfälle eines Akzeptanztests bilden beispielsweise *Use Cases*, Geschäftsprozesse und Risikoanalysen. Es können verschiedene Personen als Kunde fungieren. In einem möglichen Fall – der *Benutzerakzeptanz* – ist der Kunde selbst der Nutzer, der die Anwendung in Auftrag gegeben hat und später nutzen möchte. Bei der *Systembetreiberakzeptanz* ist der Kunde eine Person, die die Software später verwaltet oder bereitstellt, und in Stellvertretung für zukünftige Kunden entscheidet. Die dritte Gruppe bildet der *Feldtest*. Hierbei wird zukünftigen Nutzern in Alpha- und Betatests die Software zum Test überlassen.

Nach Humble et al. [HF10] hat es für Entwickler einige Konsequenzen, wenn sie automatisierte Akzeptanztests einsetzten. In erster Linie wird durch deren Einführung – besonders in einen *Continuous Delivery*-Prozess – schnelles Feedback ermöglicht. Neue Features sowie auch jede Quellcode-Änderung können sofort und vollautomatisiert getestet und die Ergebnisse an den Entwickler zurückgegeben werden. Durch die Automatisierung ist es außerdem möglich, dass Entwickler sich nicht mit der simplen Aufgabe der Durchführung von Tests auseinandersetzten müssen, sondern neue Testfälle schreiben oder sich anderen Aufgaben widmen können. Durch den Einsatz von Akzeptanztests und deren häufige Wiederholung derselben Testfälle werden Regressionstests automatisch mit abgedeckt und müssen somit nicht extra implementiert werden. Dies ist hervorzuheben, da Regressionstests als ein elementarer Bestandteil des Softwaretestens anzusehen sind [L$^+$96]. Bei der Einführung sind Akzeptanztests sehr teuer, z. B. wegen der aufwendigen Pflege. Um dem entgegen zu wirken empfehlen Humble et al. [HF10] gute und wohl bedachte Werkzeuge zu verwenden, welche die Arbeit vereinfachen. Des Weiteren amortisieren sich die Kosten durch eine später geringere Fehlerrate. Auch ist die Automatisierung im Gesamten betrachtet günstiger als die manuelle Durchführung. Die Höhe des Aufwands und somit der Kosten hängt dabei von der Art der Software ab. So sind diese Kosten bei Individualsoftware hoch und bei Standardsoftware eher gering. In extremen Fällen stellt das manuelle Durchführen eines Test jedoch die bessere Alternative dar, da eine Automatisierung zu kostenintensiv wäre oder von Menschen durchgeführte Bewertung der automatischen überlegen ist, z. B. hinsichtlich der Konsistenz einer Nutzeroberfläche oder bezüglich des „Look&Feel“.

Zudem geben Humble et al. einige Hinweise, welche bei der Implementierung der automatischen Akzeptanztests zu beachten sind. Die Testfälle des Tests sollten so entworfen werden, dass man bei erfolgreichen Tests nach dem *Refactoring* sicher sein kann, dass das getestete System immer noch voll funktionsfähig ist. Dies soll insbesondere auch dann sichergestellt sein, wenn ganze Komponenten ausgetauscht werden. Der wichtigste Testfall, der für jede Anforderung implementiert werden sollte,

ist der „Happy Path“. Der „Happy Path“ entspricht dem Standardablauf des Programms, wie es unter normalen Bedingungen ablaufen sollte. Sämtliche alternative Pfade und der „Sad Path“ können den „Happy Path“ später ergänzen. Akzeptanztests spiegeln die Einsatzfähigkeit eines Systems wider. Um dies sicherzustellen, müssen – wie bereits in Abschnitt 2.3 beschrieben – diese Tests in einer Umgebung durchgeführt werden, welche der späteren Ausführungsumgebung sehr ähnlich ist. Testfälle für einen Akzeptanztest sollten in Form eines Akzeptanzkriteriums notiert und formuliert werden. Das bedeutet, dass jeder Testfall wie in Listing 2.1 formuliert sein muss. Bei der Festlegung von Akzeptanzkriterien sind vier Schritte notwendig:

- Akzeptanzkriterium mit dem Kunde besprechen
- In ausführbarem Format (Listing 2.1) aufschreiben
- Formulierung in *domain specific language*
- Schicht zur Kommunikation mit getestetem System bereitstellen

2.4. Robot Operating System

Dieser Abschnitt behandelt das Robot Operation System, welches auf dem in der Arbeit genutzten Rasenmäher-Roboter zum Einsatz kommt. Die Inhalte dieses Abschnitts basieren auf dem offiziellen ROS-Wiki [ROS14], wo darüber hinaus noch tiefergehende Informationen eingeholt werden.

Das *Robot Operating System* (ROS) wird bereits seit 2007, anfangs durch das *Stanford Artificial Intelligence Laboratory* und seit 2012 durch das Robotikinstitut *Willow Garage* und die Organisation *Open Source Robotics Foundation*, entwickelt. Der Quellcode ist frei verfügbar und kann mit einer *Berkeley Software Distribution (BSD)*-Lizenz genutzt werden. ROS stellt Funktionalitäten bereit, welche zum Betreiben eines Roboters benötigt werden. Dazu gehört, dass ROS Hardware-Komponenten abstrahiert und somit auch eine Ausführung auf Nicht-Robotern ermöglicht. Zudem werden Hardwaretreiber bereit gestellt. Im Prinzip kann ROS als eine Sammlung von Werkzeugen und Bibliotheken gesehen werden, mit der das Entwickeln von Software für Roboter erleichtert wird. Beispielsweise können mit Hilfe von Kommandozeilenwerkzeugen aktive Knoten überwacht werden. ROS setzt dabei auf ein Konzept, bei dem die Funktionalität auf verschiedene Knoten aufgeteilt werden kann. Diese Knoten sind lose gekoppelt und bieten somit eine Unabhängigkeit zwischen verschiedenen Funktionen. Zur Kommunikation zwischen den Knoten stehen drei Wege zur Verfügung:

Topics Knoten können sich auf verschiedene Themen – sogenannte *Topics* – abonnieren und werden somit darüber informiert, was andere Knoten über dieses Thema veröffentlichen. Konkrete Veröffentlichungen werden ROS-Messages genannt und sind auf eine bestimmte Struktur inklusive Datentypen festgelegt. Eine Beispieldefinition findet sich in Listing 2.2. Nachrichten können in *ROS-Bag*s gesammelt und abgespeichert werden.

Services Services hingegen implementieren das RPC *request* / *response*-Modell. Dies ermöglicht es Knoten konkrete Anfragen zu starten. Ähnlich wie bei *Topics* werden auch hierfür Nachrichtenstrukturen mit Datentypen definiert, nun mit Anfrage- und Antwort-Teil. Eine beispielhafte Definition findet sich in Listing 2.3.

Parameter-Server Der Parameter-Server ermöglicht es, Variablen zur Laufzeit abzuspeichern und abzufragen. Im Gegensatz zu *Topics* und Services ist der Parameter-Server nicht auf Geschwindigkeit ausgelegt. Er ermöglicht es, Variablen Datentypen und Namensräume zuzuweisen. Der Parameter-Server bietet sich zum Austausch von Informationen zwischen zwei oder mehr Knoten an.

Ziele

Die Entwicklung von ROS hat zum Hauptziel, wiederverwendbaren Quellcode zu schreiben. Damit soll ein schnelles Aufbauen von neuen Systemen ermöglicht werden, was speziell in der Forschung von Bedeutung sein kann. Durch die häufige Wiederverwendung einzelner Komponenten und somit das implizite Testen soll die Zuverlässigkeit der einzelnen Komponenten gesteigert werden. Gleichzeitig soll ROS leichtgewichtig bleiben, was eine einfache Portierung auf andere Plattformen ermöglicht. Bibliotheken, die geschrieben werden, sollen möglichst unabhängig von ROS funktionieren und dafür ein klares Funktionsinterface besitzen. ROS ist in verschiedenen Sprachen implementiert und strebt es an, auch ein Sprachen-unabhängiges Framework zu sein. Zur Zeit werden die Sprachen C++, Python und Lisp unterstützt; Java und Lua stehen neben anderen Sprachen in einer experimentellen Entwicklungsphase zur Verfügung. Durch *rostest* werden Testhelfer für Unit- und Integrationstests angeboten. ROS unterstützt zudem Anwendungen mit langer Laufzeit sowie große Entwicklungsprozesse.

Begriffe und Elemente

Auf Dateisystem-Ebene gibt es einige Elemente, welche bei der Entwicklung von ROS-Software relevant sind. *Packages* bilden den Hauptcontainer einer Anwendung und enthalten zusammengehörende Teile einer Software. Neben der entwickelten Software kann ein Package auch eine Bibliothek oder Konfigurationselemente enthalten. Wenn mehrere logische zusammenpassende Packages zusammengefasst werden sollen, können *Metapackages* verwendet werden. Das *Package Manifest* dient der Beschreibung der Metadaten eines Packages. Die Beschreibung enthält Name, Version, Beschreibung, Lizenzinformationen, Abhängigkeiten und andere Informationen. Das Manifest steht immer in der `package.xml`-Datei. *Repositories* können mehrere Packages bilden, welche gemeinsam in einem Versionsverwaltungswerkzeug verwaltet und veröffentlicht werden. *Message*-Definitionen (Listing 2.2) werden als *.msg-Dateien im msg-Ordner gespeichert, dementsprechend findet man *Service*-Definitionen (Listing 2.3) als *.srv-Dateien im srv-Ordner.

Listing 2.2 Definition einer ROS-Message. Neben dem Header werden der Name der Nachricht als *string* und ein *Integer* als Wert gesendet.

```
Header header
string name
int32 value
```

Listing 2.3 Definition eines ROS-Service-Typs. Ein Service-Typ besteht aus einem *Request* und einem *Response*-Teil, welche durch − − − getrennt werden.

```
string name
int32 request
---
int32 response
```

Listing 2.4 Beispielcode für das Veröffentlichen einer Nachricht. Entnommen aus [ROS14]

```
 ros::init(argc, argv, "talker");
 ros::NodeHandle n;
 ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
 int count = 0;
 while (ros::ok())
 {
  std_msgs::String msg;
  std::stringstream ss;
  ss << "hello world " << count;
  msg.data = ss.str();
  chatter_pub.publish(msg);
  ros::spinOnce();
  ++count;
 }
 return 0;
}
```

Listing 2.5 Beispielcode für das Abbonieren eines Themas. Entnommen aus [ROS14]

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
 ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
 ros::init(argc, argv, "listener");
 ros::NodeHandle n;
 ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
 ros::spin();
 return 0;
}
```

3. Verwandte Arbeiten

In diesem Kapitel werden mit dieser Arbeit verwandte Arbeiten betrachtet. Da im Zuge der Literaturrecherche keine Arbeiten gefunden wurden, die *Continuous Delivery* im Zusammenhang mit automatisierten Robotertests betrachten, werden die einzelnen Teilgebiete der Arbeit separat betrachtet. Ziel dieses Kapitels ist es, einen Überblick, welche Ansätze in den Bereichen dieser Arbeit bereits verfolgt wurden, zu bekommen. Diese Betrachtung dient als Grundlage für die Wahl der Implementierungsentscheidungen im Verlauf der Masterarbeit.

Im Besonderen geht dieses Kapitel dabei auf *Continuous Delivery* (Abschnitt 3.1) und die im Fall von *Continuous Delivery* reichlich vorhanden Erfahrungen im wissenschaftlichen sowie im praktischen Bereich ein. Des Weiteren wird auf automatisches Testen eingegangen, welches sich in die Bereiche *Testfalldefinition* (Abschnitt 3.2) und *Robotertests* (Abschnitt 3.2) unterteilt. Während im Abschnitt zur Testfalldefinition zunächst verschiedene Eingabeformate im Mittelpunkt stehen, werden anschließend Anwendungsfälle des Robotertests beschrieben.

3.1. Continuous Delivery

Continuous Delivery kommt in vielen Projekten und Firmen zum Einsatz. Erfahrungen, die dabei gemacht wurden, werden im Folgenden vorgestellt. Ade Miller beschreibt *Continuous Delivery* als Prozess, bei dem einmal am Tag neuer Quellcode geschrieben wird. Im Anschluss daran werden jedes Mal Unittests ausgeführt. In der Analyse [Mil08] konnten bei 13 % der Tage nicht erfolgreiche Builds festgestellt werden. Gefunden wurden diese in den meisten Fällen beim Kompilieren, Unittesten und bei der statischen Codeanalyse. Durch den Einsatz vom *Continuous Delivery* konnten 40 % der Kosten gespart werden. Anstatt viele Unittests zu erstellen, stellt Ash Maurya [Mau08] den funktionalen Gesamttest, bei dem der Nutzer im Fokus steht, in den Mittelpunkt. Als Vorteil sieht Maurya dabei, dass im Vergleich zum Unittest beim funktionalen Test weniger Testtreiber erstellt werden müssen. Neely und Stolt [NS13] führen an, dass beim Testen im *Continuous Delivery* die Automatisierung eine sehr wichtige Rolle spielt; daher sei es wichtig, dass Tests schnell, verlässlich und zugänglich seien. Ein weiterer wichtiger Punkt sei die Akzeptanz von *Continuous Delivery* unter den Mitarbeitern. Aufgrund der automatisierten Freigabe von neuen Softwareversionen sei es im Zusammenhang mit *Continuous Deployment* auch nötig, dass zwischen Entwicklern und dem Produkt-Management eine gute Zusammenarbeit bestehe [OAB12]. Bei der Einführung von *Continuous Delivery* könne man, wie Ricky Ratnayaken [Rat14] beschreibt, auch auf Probleme stoßen. Zum einen führt der Autor an, dass *Continuous Delivery* nicht einfach in jedem Projekt implementiert werden könne, sondern dabei die Struktur des Projekts oder sogar firmenspezifische Strukturen berücksichtigt werden müssen. Dabei führt auch er an, dass die Akzeptanz und das Verständnis der Mitarbeiter vorhanden und klare Produktziele formuliert sein müssen. Des Weiteren spricht er sich für eine gute Werkzeugintegration,

wie zum Beispiel das Nutzen einer Testumgebung oder eine gute Kodierumgebung, aus. Dadurch solle eine effiziente Entwicklungsumgebung entstehen. Dem widerspricht das Projekt von James Shore [Sho06], welcher im *Continuous Delivery*-Prozess nahezu auf spezielle Tools verzichtet und stattdessen auf einen alten Computer, eine SVN[1]-Instanz und die positive Einstellung seiner Mitarbeiter setzt. In ihrem Bericht über die Entwicklungsgewohnheiten bei Facebook berichten Feitelson et al. [FFB13] über einen *Continuous Deployment*-Prozess der vollständig ohne *Branches* und spezielles Testteam auskommt. Jeder Entwickler ist für alle Aufgaben zuständig und somit auch für das Testen. Mit ihrem Prozess gelingt es ihnen bis zu zwei Mal am Tag neue Softwarefeatures freizugeben. Timothy Fitz [Fit09] beschreibt das Start-Up IMVU, welches mit seinem *Continuous Deployment*-Prozess 50 Software-Freigaben pro Tag erreicht. Für jede Freigabe würden 15.000 Testfälle, auf 30 bis 40 Computern verteilt, jeweils neun Minuten lang ausgeführt.

3.2. Testen

Dieser Abschnitt beschreibt Arbeiten, welche sich mit zwei Teilgebieten des Testens beschäftigen. Der erste Teil handelt von der automatisierten „Testfallgenerierung“, im zweiten Teil werden Arbeiten zum Thema „Robotertest“ vorgestellt.

Testgenerierung

Zur Automatisierung der Testfallausführung müssen Testfälle nach ihrer Definition automatisiert ausgeführt werden können. Methoden zur Testfalldefinition müssen zwei Kriterien entsprechen: Zum einen müssen Testfalldefinitionen durch Entwickler einfach erstellt, zum anderen müssen sie auch in ein für Software interpretierbares Format überführt werden können.

Im Simulationsframework SITAF [PK12] von Park und Kang wird solch eine automatisierte Evaluierung vorgestellt. Testfälle werden dabei mittels Dateien im *Extensible Markup Language (XML)*-Format definiert, diese enthalten Testdateneingabe, Simulationsabhängigkeiten und die benötigten Testtreiber. XML ist ein Datenstrukturierungsformat, welches vornehmlich zum Lesen und Schreiben durch Rechner geeignet ist. Durch die Nutzung von graphischen Modellierungstools ist das XML-Format für Entwickler besser nutzbar. Wittevrongel und Maurer stellen einen Ansatz vor, welcher mit Hilfe eines Modellierungswerkzeugs für die *Unified Modeling Language (UML)* – zum Beispiel Rational-Rose – erstellte Sequenzdiagramme nach XML konvertiert werden. Anschließend können mit dem Werkzeug „SCENTOR“ [WM01] automatisiert Testfälle sowie deren erwartete Ergebnisse generiert werden. Auch Biswal et al. [BNM08] und Hartmann et al. [HVFR05] beschreiben Ansätze, welche die Möglichkeit bieten, aus UML-Diagrammen Testfälle zu generieren. Biswal et al. nutzen hierzu neben Aktivitätsdiagrammen zur Generierung von Szenarien auch Sequenz- und Klassendiagramme, um konkrete Testfälle zu erstellen. In einer weiteren Veröffentlichung von Hartmann et al. [HVFR04] benutzen diese einen ähnlichen Ansatz, um vollständige Systemtests zu erstellen. Eine Definition der Testfälle geschieht bei Micskei [MSOM12] in drei Elementen, einem Kontext, einem Szenario und einem Aktivitätsmodell. Diese werden in Sequenzdiagramm- und Metamodellen definiert und

[1] SVN-Webseite: https://subversion.apache.org

anschließend in Testfälle überführt. Tsai et al. [TYL+03] beschreiben, wie Szenarien bestehend aus Zuständen und Ereignissen zu einem Zustandsmodell zusammengebaut und daraus über eine *depth-first-* oder eine *breadth-first*-Suche Testfälle generiert werden. Mit dem Projekt jSynoPSys [DT09] von Dadeau et al. werden Testdaten ebenfalls über die Zustandsübergänge eines Automaten erstellt. In einer weiteren Arbeit stellen Tsai et al. [TNP+02] ein mit JUnit vergleichbares Testframework vor, bei dem die Definition der Testfälle jedoch ebenfalls als Szenario geschieht. Auch Sarma et al. [SM07] setzen auf eine Traversierung von Graphen mittels *breadth-first*. Als Eingabe dienen Szenarien mit Eingabe-, Ausgabe-, Vorbedingungs- und Nachbedingungsdefinition, sowie in der *Object Constraint Language (OCL)* beschriebene Daten. Diese Eingabe wird zuerst in Sequenzdiagrammgraphen und anschließend mittels Traversierung in Testfälle überführt. Bei Tsai et al. [TBPY01] werden definierte Szenarien mittels Szenario-*Slicing* und der *Ripple Effect Analysis* selektiert und validiert. Eine Alternative zur graphischen Definition von Testfällen bietet die Erstellung von regulären Ausdrücken, welche analysiert werden und dann als Grundlage für Testfälle dienen. Dadeau et al. [DCT12] stellen einen Ansatz vor, bei dem eine Folge von erlaubten Operationen als regulären Ausdruck definiert werden. Aus diesen Definitionen lassen sich im Anschluss Testfälle ableiten. Auch Castillos und Botella [CB11] spezifizieren mittels regulären Ausdrücken Testfälle, welche einer bestimmten Art entsprechen und als Grundlage für Testfallinstanzen dienen.

Robotertests

Bei Robotersoftware handelt es sich in aller Regel um Software, welche auf spezieller Hardware ausgeführt wird und aufgrund vieler Sensoren, die zur autonomen Bewegung nötig sind, recht komplex ist. Tests, welche das Gesamtverhalten eines Roboters prüfen, sind dadurch ebenfalls von komplexer Natur. Nicht immer können diese eindeutig mit „erfolgreich“ oder *fehlgeschlagen* beantwortet werden, da oftmals eine Toleranz der möglichen Ergebnisse vorhanden ist. Durch das zusätzliche Einbinden der Hardware, kommen weitere Fehlerquellen und Ungenauigkeiten hinzu. Um dies zu umgehen, werden in den meisten Fällen Hardwaresimulationen genutzt. Die folgenden beschriebenen Arbeiten versuchen, diese Problematiken zu lösen.

In der Arbeit von Krotkov et al. [KFJ+07] wird unter realen Bedingungen und ohne Simulation getestet. Hierzu muss der Roboter einige Wegpunkte abfahren. Um die Tests zu erstellen werden Metriken, die die Performanz beschreiben, analysiert. Den Autoren gelingt es, durch das System die Anzahl der Notfallstopps des Roboters signifikant um das 22-fache zu reduzieren. Im Gegensatz dazu verfolgen die folgenden Arbeiten den Ansatz, die Robotersoftware über eine Simulation zu testen. So zum Beispiel die Evaluierungsplattform für Rettungsroboter von Shimizu [SYT10], welche die dynamische Umgebung des Roboters und den Roboter selbst simuliert und Bewegungen des Roboters am Überwachungsbildschirm überprüft. Dabei handelt es sich um eine manuelle Methode zur Evaluierung.

Damit Tests sich nahtlos in den *Continuous Delivery*-Prozess einfügen, sollte auch die Ausführung möglichst automatisiert ablaufen. Son et al. [SKPK11] wandten diese Automatisierung auf Unit-, Zustands- und Schnittstellentests an. Dabei wird die Performanz analysiert und die Sicherheit der einzelnen Komponenten an Metriken gemessen. Zhang et al. [ZLZS14] setzen in ihrer Straßenverkehrssimulation ebenfalls auf drei Arten von Tests wobei sie das Gesamtsystem, einzelne Aufgaben und einzelne Algorithmen betrachten. Auch Laval et al. [LFB13] präsentieren ein Konzept zum Testen von Software

für autonome Roboter, welches auf drei Stufen basiert. Sie schlagen die Unterscheidung vor, ob der Roboter sich bewegt oder nicht, ob das Umfeld dynamisch ist oder nicht und ob der Roboter seine Umgebung kennt oder nicht. Daraus wird ein fünfstufiger Prozess generiert, welcher wiederverwendbare, wiederholbare und semi-automatische Tests hervorbringt. Der von Sorton und Hammaker [SH05] entworfene Robotertests bezieht sich auf die Bordelektronik von „Unbemannten Flugobjekten“ und nutzt dabei den Flugsimulator „FlightGear“. Dabei handelt es sich um ein *OpenSource*-Programm, weswegen es möglich ist, auf Schnittstellen direkt zuzugreifen, diese anzusteuern und Ergebnisse beziehungsweise Parameter abzugreifen. Durch die richtige Konfiguration erreichten die Autoren eine sehr gute Übereinstimmung der Simulation mit dem Verhalten der echten Flugobjekte. Während Sorton und Hammaker in die genutzte Software eingreifen und so auf die Metriken im System zugreifen konnten, ist das bei Softwaresystemen nicht immer der Fall. Chung und Hwang [CH07] verfolgen daher den Ansatz, über externe Metriken, wie Antwortzeiten oder erreichte Abdeckung, Schlüsse zu ziehen. Micskei et al. [MSOM12] schlagen vor, anhand der Erfüllung von vorgegebenen Szenarien, das heißt eine konkrete Aufgabe in Kombination mit einer konkreten Umgebung, den Erfolg eines Tests festzustellen. Am Ende werden dabei Überdeckungsmetriken geprüft. Diese enthalten beispielsweise die Information, welche Teile des Kontexts in einem Testfall erreicht wurden. Sun et al. [STC13] gehen dabei noch einen Schritt weiter, indem sie mit Hilfe von „fuzzy-AHP“-Evaluierungsmethoden die Qualität einer Lösung bewerten und somit prüfen, wie „intelligent“ eine Lösung ist.

3.3. Zusammenfassung

Continuous Delivery für Robotersoftware setzt sich aus verschiedenen Themen zusammen, und wird - zusammen mit den verwandten Prozessen *Continuous Integration* und *Continuous Deployment*- häufig eingesetzt. Es gibt viele Erfahrungsberichte und Tipps, die beachtet werden sollten, um *Continuous Delivery* erfolgreich zu implementieren. Dabei werden besonders die Aspekte einer möglichst vollständigen Automatisierung, sowie die Akzeptanz von *Continuous Delivery* unter den Entwicklern häufig aufgeführt [Rat14, NS13]. Durch die Einführung von *Continuous Delivery* lassen sich durchaus Verbesserungen erzielen [Mil08] und die Anzahl der Softwarefreigaben erhöhen [FFB13, Fit09].

Zu einer vollständigen Automatisierung gehört neben einem automatischen Commit- und Build-Prozess auch die Automatisierung der Tests. Um Entwickler in ihrer Arbeit zu unterstützen, muss die Definition der Testfälle in einem ansprechenden Format geschehen. Hierfür bieten sich graphische Darstellungen an, welche in den meisten Fällen auf UML-Diagrammen basieren [WM01, BNM08, HVFR04, HVFR05]. Diese lassen sich mit verfügbaren UML-Modellierungswerkzeugen erstellen und anschließend mittels Traversierung in Testfälle übertragen. Neben der graphischen Notation ist es auch möglich, Testfälle in Textform zu erstellen. Hier kann beispielsweise auf reguläre Ausdrücke [DCT12, CB11], XML [PK12] oder OCL [SM07] zurückgegriffen werden. In der vorliegenden Arbeit wird ein simples textbasiertes Format verwendet, um dem Aufwand der Traversierung zu entgehen. Beim Testen von Robotern ist es üblich, die Tests in einer Simulationsumgebung auszuführen. Dabei wird in einem Teil der Fälle ein mehrstufiger Test angewendet [SKPK11, ZLZS14]. Um der Komplexität der Tests für autonome Roboter gerecht zu werden, werden häufig Metriken verwendet, welche über Schnittstellen abgegriffen werden können oder von außen über externe Metriken [SH05] geschätzt werden müssen [CH07, MSOM12]. In der vorliegenden Arbeit sollen Metriken aus dem SUT mittels Nachrichten an die Validierungskomponente übertragen werden.

4. Ist-Zustand

Im vorliegenden Kapitel wird der Zustand des Entwicklungsprozesses, wie er zu Beginn der Arbeit vorgefunden wurde, beschrieben. Dies beinhaltet zum einen die Zusammensetzung der Entwicklungsteams. Zum anderen wird auch auf den Entwicklungsprozess, verwendete Technologien und verwendete Testmethodiken eingegangen. Als Grundlage für diese Analyse dienten Befragungen, welche in drei Abteilungen der Robert Bosch GmbH durchgeführt wurden.

4.1. Systeme und Umfeld

Das in dieser Arbeit beschriebene Delivery-System wird für eine Robotik-Abteilung der Firma Robert Bosch GmbH entwickelt. Diese Abteilung ist für die Vorausentwicklung des selbständig mähenden Rasenmähers *Indego*[1] zuständig. In diesem Abschnitt wir der Indego sowie die entwickelnde Abteilung vorgestellt.

Die Robert Bosch GmbH ist ein im Jahr 1886 gegründetes Unternehmen und ist in verschiedenen Geschäftsbereichen tätig. Bosch agiert dabei global und hat weltweit knapp 300.000 Mitarbeiter [Bos15]. In immer mehr Produkten – vor allem im Bereich der Automatisierung – spielt Software eine wichtiger werdende Rolle. Das Ergebnis dieser Arbeit soll die Entwicklung des autonomen Rasenmähers unterstützen. Gleichzeitig werden auch die in thematisch benachbarten Abteilungen entwickelten Roboter zur Pflanzenerkennung bzw. Unkrautvernichtung und die autonomen Stapler untersucht.

Eine weitere Abteilung der Robert Bosch GmbH ist dafür zuständig, die zur Entwicklung von Roboterprojekten notwendige IT-Infrastruktur zu unterstützen. Dabei wird eine Umgebung bereitgestellt, die dem *Continuous Integration* ähnelt. Das *Robotic System & Software Framework* (RoSe) möchte ein komplettes Framework für Robotik-Projekte bieten. In diesem Framework soll der gesamte Entwicklungsprozess abgebildet werden. Dazu gehört unter anderem das Bauen von Software auf Jenkins-Instanzen, Testen mit oder ohne Hardware und das Speichern von wichtigen Artefakten wie Testergebnissen, Executables und Dokumentationen. Das Framework befindet sich zur Zeit im Aufbau und bietet noch nicht alle Teilkomponenten; so ist zum Beispiel noch keine Lösung für ganzheitliche automatisierte Akzeptanztests verfügbar. Projekte der Robert Bosch GmbH können jedoch auf bereits verfügbare Dienste der RoSe-Abteilung zurückgreifen.

Zur Evaluierung der Anforderungen an das System, welches den *Continuous Delivery*-Prozess für autonome Roboter implementiert, wurden drei Entwicklungsteams im Bosch-Konzern analysiert. Zur Analyse wurden Interviews mit Entwicklern dieser Teams geführt. Dabei wurden die aktuell

[1] Indego Webseite: http://www.bosch-indego.com

verwendeten Entwicklungsprozesse und Arbeitsweisen, aber auch Schwachstellen und Wünsche ermittelt. Die Interviews wurden in einem semi-strukturierten Interview, mit jeweils einem Mitarbeiter der entsprechenden Abteilung, geführt. Als Grundlage für die Interviews diente ein Fragebogen (Appendix A.1). Neben dem Team, welches den Indego-Rasenmäher entwickelt, wurden zwei Teams aus der „Bosch Start-Up GmbH“ nach deren Entwicklungsmethoden befragt. Im Folgenden werden die Ergebnisse der Interviews und somit die Arbeitsweise der Teams vorgestellt.

Autonomer Rasenmäher

Der Geschäftsbereich *Power Tools* entwickelt neben Werkzeugen für Handwerker auch Gartengeräte. Die Abteilung *CR/AEG* ist dabei für die Vorausentwicklung von Gebrauchsgüter zuständig. In diesen Bereich fällt auch die Entwicklung des autonomen Rasenmähers *Indego*. Der Indego-Rasenmäher ist bereits im Handel erhältlich, wird jedoch noch weiter entwickelt. Zur Nutzung des Indego-Rasenmähers muss ein Begrenzungsdraht installiert werden, welcher den Umriss des Gartens vorgibt. Nach Aufnahme des Gartenumrisses kann der Indego-Rasenmäher den Garten abmähen. Beim Mähen wird der sogenannte *Logicut* Algorithmus angewandt, welcher ein strukturiertes Abmähen ermöglicht. Diese und andere Funktionen sowie das Zusammenspiel von mehreren Sensoren ergeben eine komplexe Navigationssoftware. An der Vorausentwicklung des Rasenmähers sind aktuell drei Mitarbeiter sowie eine wechselnde Anzahl von Studenten tätig.

Zur Unterstützung der Entwicklung werden die Softwareversionierungstools *Git* und *Stash*[2] in Kombination mit dem *Track and Release*-Werkzeug *JIRA*[3] eingesetzt. Wichtige Informationen zu Werkzeugen und dem in der Entwicklung befindlichen System, werden über ein Wiki festgehalten, welches unter anderem Installationsanweisungen enthält. Neben dem `master`- und `develop`-*Branch* werden für neue Features eigene Brachnes in Git benutzt, welche unabhängig voneinander weiterentwickelt werden können und zu einem späteren Zeitpunkt in die Haupt*branches master* und *develop* gemergt werden können. Neue Features und Fehler können zuvor einzelnen Entwicklern zugewiesen werden. Tests werden auf Systemebene durchgeführt und haben bereits in einem hohen Maß zur Fehlererkennung beigetragen. Hierfür werden Log-Dateien, welche durch ein eigenes Format definiert wurden, während der Ausführung der Software geschrieben. Diese Log-Dateien werden im Fehlerfall manuell nach Fehlerursachen durchforstet. Zur Evaluierung werden außerdem Matlab-Skripte verwendet, welche die Log-Dateien prüfen. Die Codeüberdeckung wird mittels Flags im Quellcode geprüft. Es werden keine Tests auf Unitebene verwendet.

Pflanzenerkennungs- und Unkrautvernichtungsroboter

Die Abteilung BOSP/PAA ist ein Teil des zur Robert Bosch GmbH gehörenden Start-ups *Robert Bosch Start-up GmbH* und ist im Bereich der *Deepfield Robotics* tätig. Diese Abteilung hat das Ziel, neue Herausforderungen in der Landwirtschaft mittels Robotersystem anzugehen. Im Fokus stehen dabei gesellschaftliche, ökologische und ökonomische Herausforderungen. Als Teil der Robert Bosch GmbH können hier Synergieeffekte mit dem Gesamtkonzern genutzt werden.

[2] Stash Webseite: http://de.atlassian.com/software/stash
[3] JIRA Webseite: https://de.atlassian.com/software/jira

Die Entwicklung des Teams BOSP/PAA befindet sich noch in einer frühen Phase, weswegen sich noch wenige spezielle Aufgabenverteilungen ergeben haben. Es hat sich jedoch bereits ein Entwicklungsprozess etabliert. Die zwei zentralen Technologien bilden das Versionierungstool *Git* und der *Continuous Integration*-Server *Jenkins*. In Git werden mehrere *Branches* geführt. Einer dieser *Branches* ist der aktuelle *Development-Branch*, welcher den aktuellen Stand der Entwicklung beinhaltet. Ein weiterer *Branch*, *Demo*, enthält immer eine vorführbare Version der entwickelten Software. Sowohl der Development- als auch der Demo-*Branch* sind immer ausführbar. Zusätzlich gibt es für jeden Issue einen *Branch*. Quellcodeneuerungen werden immer nur auf diesen zusätzlichen *Branches* durchgeführt, diese *Branches* werden nach Prüfung in den Development-*Branch* gemergt. Um die Quellcodequalität zu sichern, wird der Codierstyleguide der Firma Google verwendet. Zur Überprüfung wird für jede Neuerung, bevor sie gemergt wird, ein Review durch zwei oder vier Augen durchgeführt. Anschließend werden die Neuerungen mittels eines *Pull-Request*s an den Jenkins-Server übertragen und dort gebaut und getestet. Wird bei einem dieser Builds oder Tests ein Fehler gefunden, werden Tickets im Issue-System angelegt und für genau diesen Fehlerfall ein Unittest erstellt. In der sonstigen Entwicklung werden jedoch kaum explizit Testfälle definiert, sondern durch Beobachtung der laufenden Software das Verhalten überprüft. Um diese durch abweichendes Verhalten erkannte Fehler zu analysieren werden Log-Dateien angelegt und überprüft. Eine weitere Methode, um die Qualität der Software sicherzustellen, sind Checklisten, welche als Leitfaden für die Erstellung neuer Features dienen.

Autonome Stapler

Die Abteilung BOSP/PAR gehört ebenfalls zur *Robert Bosch Start-up GmbH* und arbeitet im Bereich „Intralogistics Robotics“. Sie möchte die Synergien zwischen automatisiertem Fahren und „Industrie 4.0“ nutzen.

Die Abteilung BOSP/PAR nutzt zur Entwicklung das Versionierungstool *Git* und den *Continuous Integration*-Server *Jenkins*. Werden Neuerungen in Git festgestellt, wird ein Job, in dem der neue Quellcode gebaut wird, angelegt. Jenkins wird dabei auf verschiedenen Clients ausgeführt, welche automatisiert durch den Jenkins-Server angesteuert werden. Nach einem erfolgreichen Build werden Unittests ausgeführt und somit einzelne Komponenten der Software geprüft. Dabei ist jeder Entwickler für das Testen seines Quellcodes zuständig. Werden bei der Ausführung des Builds oder beim Unittest Fehler entdeckt, wird zum einen auf *Github*[4] der Quellcode als fehlerhaft markiert, zum anderen wird der Teil des Entwicklerteams, der für diesen Code zuständig ist, per E-Mail informiert. Um einen Überblick über das gesamte System zu erhalten, werden Systemtests durchgeführt, welche das System als Ganzes testen sollen und immer über Nacht angestoßen werden. Diese Systemtests müssen das Verhalten einer komplexen Robotersoftware prüfen und können in der Regel nicht über einen einfachen Vergleich zweier Werte durchgeführt werden. Daher werden verschiedene Daten aus dem laufenden System mitgeloggt. Beispiele hierfür sind Karten der Roboterumgebung, Kennzahlen und viele Rohdaten. Jedoch fehlen Informationen, welche Daten wichtig sind, weshalb Logging-Daten sehr viel Overhead haben. Ergebnisse aus diesen Systemtests werden mittels PHP in einer HTML-Seite angezeigt. Für das Erstellen der Testfälle gibt es keinen konkreten Prozess. Es wird den Entwicklern überlassen, was getestet wird. Neben Funktionentests werden auch Performanztests in Form von Laufzeit und Speicherverbrauch durchgeführt.

[4] Github Webseite: http://github.com

4.2. Zusammenfassung

Die drei analysierten Abteilungen verwenden bereits einen Entwicklungsprozess, welcher an einen *Continuous Integration*-Prozess erinnert. Zwei der Abteilungen haben bereits einen Integrationsserver im Einsatz, und mit dem RoSe-Team gibt es sogar einen hauseigenen Service, welcher die Infrastruktur für einen *Continuous Delivery*-Prozess bieten soll. Dieser wird jedoch nicht in vollem Umfang angeboten. Testen ist zum aktuellen Zeitpunkt noch nicht gängige Praxis. Auch wenn es Bestrebungen gibt, dass Software getestet werden soll, wird diese Praxis noch nicht in vollem Umfang angewandt. Als Grund hierfür wird fehlende Zeit angegeben.

Bei der Befragung äußerten Entwickler einige Wünsche, deren Umsetzung in naher Zukunft angestrebt werden sollte. Diese befassen sich hauptsächlich mit den Softwaretests. Generell sollte mehr Zeit für das Testen und den Softwareverifizierungprozess genutzt werden. Ein Problem ist hierbei, dass eine geeignete Infrastruktur fehlt, welche beispielsweise sicherstellt, dass bei jeder Testausführung der gleiche Systemzustand hergestellt wird. Ein weiterer Aspekt ist die lange Dauer von Tests, während deren Ausführung nicht immer ersichtlich ist was vor sich geht und was die Akzeptanzkriterien sind.

5. *Continuous Delivery*-System

In diesem Kapitel wird das im Rahmen dieser Arbeit entstandene *Continuous Delivery*-System sowie dessen Entstehungsprozess beschrieben. Das Kapitel beginnt, in Abschnitt 5.1, mit der Darstellung der Anforderungen, die an das System gestellt werden. Abschnitt 5.2 beschreibt den vorgeschlagenen *Continuous Delivery*-Prozess und enthält zudem empfohlene „Best-Practices“. Im vierten Teil (Abschnitt 5.3) werden das Testsystem, dessen Architektur und die einzelnen Komponenten vorgestellt. Abschnitt 5.4 schließt das Kapitel mit einer Zusammenfassung ab.

5.1. Anforderungen

In diesem Abschnitt werden die Anforderungen an das zu entwickelnde System beschrieben. Auf die allgemeine *Problemformulierung* folgt eine *Anforderungsliste*, welche konkrete Anforderungen enthält. Im Teil *Szenarien* folgen Testfälle, welche implementiert werden sollen.

Problemformulierung

Autonome Roboter verfügen über viele Sensoren, welche Umfeldinformationen bereitstellen und damit als Grundlage für alle Entscheidungen dienen. Zur Entscheidungsfindung müssen diese Informationen zusammengeführt werden. Daraus ergibt sich ein komplexer Quellcode, dessen Qualität sichergestellt werden muss. Im *Continuous Delivery*-Prozess findet die Endvalidierung, ob ein Programm korrekt funktioniert, im Akzeptanztest (Abschnitt 2.3) statt.

Umgebungsdaten, welche mit Sensoren aufgenommen werden, können in ihrer Struktur stark von Daten abweichen, welche in derselben Umgebung an einer nur leicht abweichenden Stelle aufgenommen werden. In diesen Fällen ist es nicht möglich, über einfache Vergleiche festzustellen, ob die Ausführung der Software korrekt war. Zum Beispiel ist es schwierig, eine aufgenommene Karte mit einer Vergleichskarte zu validieren, da die Diskretisierungsschritte sich in der Regel unterscheiden. Dies tritt auf, da eine Ausführung in der physischen Welt nie exakt am selben Punkt beginnt und daher jedes Mal ein anderer Startreferenzpunkt gewählt wird. Auch im Vergleich zwischen realer und simulierter Karte treten immer Unterschiede auf, da eine Simulation nie perfekt alle Umstände der realen Ausführung abbilden kann. Aus diesem Grund müssen andere Metriken und Validierungsgrundlagen gefunden werden. Aus dieser Problematik ergeben sich Anforderungen, welche im nächsten Abschnitt beschrieben werden.

Tabelle 5.1.: Die mit R1 bis R6-7 bezeichneten Anforderungen an das Testsystem. Diese Liste wurde mit Hilfe der Entwickler durch Gespräche, Interviews und durch Feedback erstellt und weiterentwickelt.

Req#	Name	Beschreibung
	Infrastruktur	
R1	*Continuous Delivery*-Pipeline	Das System muss die *Continuous Delivery*-Pipeline abbilden → Referenz [HF10]
R2	ROS-Unterstützung	Die Umgebung von ROS soll als zu testendes System unterstützt werden.
R3	Jenkins	Als *Continuous Integration*-Server soll Jenkins zum Einsatz kommen.
R4	Legacy-Logging	Logging soll Möglichkeiten haben, den alten Logging-Mechanismus zu migrieren.
	Das Testsystem	
R5	Akzeptanztest	Das zu entwickelnde System muss automatisierte Akzeptanztests durchführen: Testfall-Definition → Testfallausführung → Testfallevaluierung → Publishing der Ergebnisse.
R5-1	Testfälle	Das Testsystem unterstützt 1 … n Testfälle. Jeder Testfall unterstützt 1 … m Metriken.
R5-2	Erstellung von Testfällen	Entwickler müssen Testfälle ohne viel Aufwand (< 5 Minuten) erstellen können.
R5-3	Start des SUT	Das Testsystem soll eine flexible Möglichkeit bieten, das zu testende System zu starten. Das zu testende System kann aus mehreren – getrennt zu startenden – Prozessen bestehen. Dabei sollen Starter wie Roslaunch und konfigurierbare Pausen nach jedem Prozessstart unterstützt werden.
	Testvalidierung	
R6	Testanalyse	Testergebnisse werden analysiert und ausgewertet.
R6-1	Eingabe der Validierung	Eingabe der Validierung ist eine RosBag, welche aus dem Testsystem kommt.
R6-2	Validierungstechnologien	Die Validierung erfolgt durch Python- oder/und Matlab-Skripte variabler Anzahl.
R6-3	Ausgabe der Validierung	Die Ausgabe der Validierungsergebnisse wird in Testreports abgespeichert. Diese werden im YAML- und XML-Format abgespeichert.
R6-4	Testreport	Mehrere Testergebnisse können in einem Testreport zusammengefasst werden.
R6-5	Visualisierung	Graphische Darstellung der Ergebnisse (z. B.: Generierung einer HTML-Seite).
R6-6	Speicherung	Testreports und RosBags werden persistent abgespeichert.
R6-7	Darstellung	Darstellung der Ergebnisse über die Zeit an einem Beispiel demonstriert.

Tabelle 5.2.: Beschreibung der Szenarien, welche durch das Testsystem getestet werden sollen.

Name	Beschreibung
Szenario 0	
Dummy	Einfaches Programm, welches den Testprozess nutzt.
Szenario 1	
Pfad abfahren	Der Roboter folgt einem Pfad.
Szenario 2	
Pfadplanung	Die Software des Roboters plant einen Pfad.
Szenario 3	
Kartenaufbau	Im Speicher des Roboters wird eine Karte der Umgebung aufgebaut.
Szenario 4	
Volle Gartenabdeckung	Der Roboter mäht den gesamten Garten in parallelen Bahnen

Anforderungsliste

Die grundlegenden Anforderungen wurden in einem Anforderungsworkshop mit drei Mitarbeitern des Teams, das den Rasenmäher entwickelt, festgelegt. Im Verlauf der Arbeit wurde diese Liste mittels Interviews und Feedback, sowie der in Kapitel 4 erwähnten Entwicklungsteams weiter verfeinert. Tabelle 5.1 enthält die Anforderungen, welche als Ziele festgelegt wurden. Die Hauptanforderungen bilden hierbei die Punkte Infrastruktur, Testsystem und Testvalidierung.

Infrastruktur R1-R4 Die Infrastruktur beschreibt das Umfeld, in dem die Akzeptanztests durchgeführt werden sollen. Dies betrifft hauptsächlich eingesetzte Werkzeuge und den *Continuous Delivery*-Prozess, in den alles eingebettet werden soll.

Testsystem R5 Das Testsystem stellt die allgemeinen Anforderungen an das Testsystem. Dazu gehört die Testfalldefinition und die Einbettung des SUT in das Testsystem.

Testvalidierung R6 Die Testvalidierung befasst sich mit der Feststellung, ob eine Ausführung und somit der Test erfolgreich war, sowie mit der Darstellung der Ergebnisse.

Szenarien

Zur besseren Abgrenzung der Anforderungen an das Testsystem, wurden in einem Workshop vier Szenarien definiert, welche implementiert werden sollen. Sie dienten als Grundlage für die Überlegungen zum Testsystem und gleichzeitig als „Proof-of-Concept“. Die Beschreibung der Szenarien befindet sich in Tabelle 5.2. Hierbei wurde der Fokus auf mögliche Metriken gelegt, welche aus dem System entnehmbar sein sollen. Die dabei entstandenen Metriken, sollen auch die Entscheidung, welche Validierungsmethoden benötigt werden, unterstützen.

Die Umsetzung dieser Anforderungen wird im nachfolgenden Absatz beschrieben. Hierbei wird zuerst das generelle Vorgehen dieser Arbeit erklärt und im Anschluss die Implementierung beschrieben.

5.2. *Continuous Delivery*-Prozess

Der in dieser Arbeit verfolgte Ansatz für *Continuous Delivery* folgt lose dem von Humble und Farley [HF10] beschriebenen Prozess. Dabei wird im Besonderen das Konzept des kontinuierlichen Testens übernommen. Abbildung 5.1 beschreibt den Prozess vom Entwickeln einer neuen Funktion bis hin zum Testen im Akzeptanztest. Der Prozess wird dabei in zwei Teile gegliedert. Zum einen gibt es den „lokalen Teil“, welcher die Aufgaben beschreibt, die ein Entwickler im Entwicklungsprozess lokal durchführen muss. Dieser beschreibt Aufgaben, welche der Entwickler lokal beim Entwickeln durchführen muss. Dabei wird vor allem das Versionsverwaltungstool `Git` eingesetzt. Zum anderen gibt es den „zentralen Teil“, in dem der Quellcode gebaut und getestet wird. Der „lokale Teil“ beinhaltet folgende Aufgaben:

Issue erstellen Der erste Schritt ist das Erstellen eines Issues. Dieser kann vom Entwickler selbst oder von anderen am Projekt Beteiligten erstellt werden. Der Issue wird einem konkreten Entwickler zugewiesen und von diesem bearbeitet. Hierfür wird, wie bereits im vorherigen Kapitel beschrieben, das „Track and Release“-Tool Jira verwendet.

Feature-*Branch* erstellen Damit sich verschiedene – oft auch instabile – Entwicklungsstände nicht gegenseitig behindern, wird für jeden Issue ein sogenannter „Feature“-*Branch* erstellt.

Feature implementieren Die Implementierung des Features erfolgt auf dessen *Branch*. Wenn währenddessen auf dem Haupt*branch* wichtige Änderungen vorgenommen werden, können diese mittels eines Rebase ebenfalls übernommen werden. Um zu gewährleisten, dass Änderungen nachvollziehbar und umkehrbar sind, sollten Commits klein gehalten werden.

Lokal testen Bevor Änderungen der Allgemeinheit zugänglich gemacht werden, müssen diese lokal auf einem Teil der Testplattform getestet werden. Treten bei diesen Tests Fehler auf, müssen im Prozess so lange Schritte zurück gegangen und die Implementierung überprüft werden, bis der lokale Test erfolgreich ist.

Pull-Request erstellen Wenn das neue Feature erfolgreich implementiert und getestet wurde, wird der Feature-*Branch* mittels eines sogenannten *Pull-Request*s in den Haupt*branch* gemergt und somit für alle Entwickler freigegeben.

Mit der Bestätigung eines *Pull-Request*s, wird der zum Bauen und Testen benutzte *Continuous Integration*-Server Jenkins getriggert. Auf diesem werden dann die folgenden Schritte durchgeführt: SUT und Testsystem bauen, beide Systeme testen und im Anschluss den Akzeptanztest durchführen. Das Bauen eines neuen Softwarestandes geschieht in der Regel automatisiert durch einen „Webhook“. Dies kann entweder durch eine URL erreicht werden, auf welche der Jenkins-Server hört (Listing 5.1) oder aber mit Hilfe der Versionsverwaltungsoberfläche – in diesem Projekt `Stash` – erreicht werden. Der Build des Testsystems und des SUT erfolgt durch das ROS-Tool `catkin_make`, welches sich um die gesamte Konfiguration des Builds kümmert. Anschließende Tests werden durch Unittests realisiert. Wenn der Test des SUT erfolgreich abgeschlossen wurde, kann der Akzeptanztest – mit neuen Testfällen oder neuen Features – ausgeführt werden. Der Akzeptanztest wird jedes Mal in einer frisch konfigurierten Umgebung ausgeführt. Hierfür werden die benötigten Softwarestände neu ausgecheckt und kompiliert. Zusätzlich muss der Jenkins-Server dafür sorgen, dass Build-Artefakte vor der Ausführung entfernt und nach der Ausführung abgespeichert werden. Dies kann in Jenkins

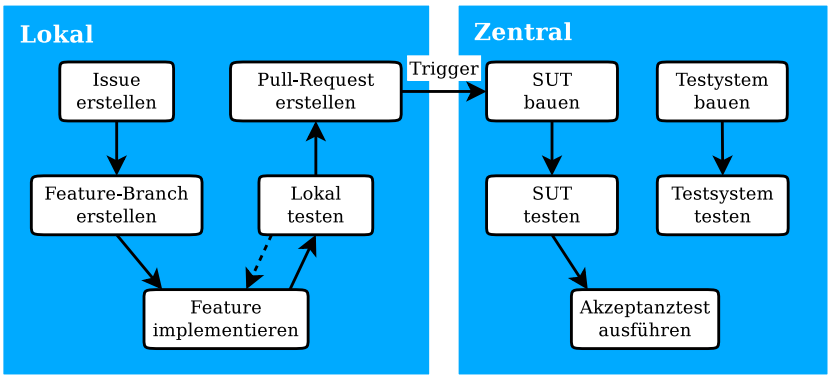


Abbildung 5.1.: *Continuous Delivery*-Prozess besteht aus den Teilen „Lokal“ und „Zentral“. „Lokal“ enthält die Prozessteile, die durch den Entwickler lokal ausgeführt werden. „Zentral“ beschreibt den Ablauf, der im zentralen Jenkins abläuft.

Listing 5.1 Die Jenkins Webhook-URL ermöglicht es, einen Build mittels URL zu triggern und somit bei Änderungen im Versionsverwaltungstool einen neuen Softwarestand zu bekommen.

```
${JENKINS_URL}/job/${JOB_NAME}/build?token=${SECURITY_TOKEN_NAME}
```

mit Hilfe von Bash-Skripten erreicht werden. Nach der Ausführung des Testsystems werden die Ergebnisse der Testfälle in der Jenkins-Oberfläche dargestellt. Dies geschieht zum einen mittels eines einfachen JUnit[1]-Diagramms (Abbildung 5.2b), welches die Anzahl der erfolgreichen Tests über die Zeit aufträgt und zum anderen durch textbasierte Testreports (Abbildung 5.2a).

Neben den bereits vorhandenen Anwendungen wie dem *Continuous Integration*-Server Jenkins oder der Versionsverwaltung Stash, wird ein System zum Durchführen von Akzeptanztests benötigt. Dessen Architektur und wichtige Entwicklungsentscheidungen finden ihre Dokumentation im nächsten Abschnitt.

5.3. Testsystem

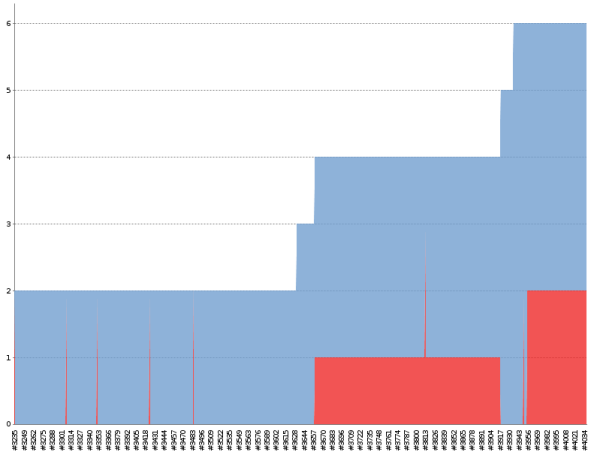
Dieser Abschnitt beginnt mit der Vorstellung des Testsystems und dessen Architektur, gefolgt von einer detaillierteren Dokumentation einzelner Komponenten und bei der Implementierung aufgetretener Probleme.

Das in Abbildung 5.3 dargestellte Testsystemkonzept sieht zur Auführung der Testfälle zwei Tätigkeiten vor: Die manuelle Erstellung der Testfälle – welche Szenarien genannt werden – durch den Entwickler und die automatisierte Ausführung der Szenarien durch das Testsystem. Im manuellen Schritt definiert der Entwickler Szenarien, die getestet werden sollen. Szenarien bestehen aus einer Aufgabe, einem Kontext und beliebig vielen Metriken. Nach der Definition führt das Testsystem die

[1] JUnit-Webseite: http://junit.org

```
results:
  - time: 2015-07-07T11:27:41
    bag: bag_2015-07-07_11-27-41.bag
    metrics:
      - name: End pose accuracy
        description: End pose accuracy
        expected: 10
        measured: 459.702423198
        operator: <
        state:
          test: FAILED
          result: Test was not successful.
          message: "Failed: 459.702 mm"
  - time: 2015-07-07T11:28:08
    bag: bag_2015-07-07_11-28-08.bag
    metrics:
      - name: path length
        description: Tests length of ground truth
        expected: 51.3590558751
        measured: 51.3247844626
        difference: 0.0342714125687
        tolerance: 5
        limit: 2.56795279376
        state:
          test: PASSED
          result: Test was successful.
          message: The measured paths deviation is 0.034 meter
```

(a) Der vollständige Testreport



(b) Diagramm der fehlgeschlagenen Tests

Abbildung 5.2.: Die Ausgabe der Testergebnisse im Jenkins. **(a)** Ausführlicher Testbericht im Text-Testreport. **(b)** Übersicht über die fehlgeschlagenen Tests im JUnit-Format.

Szenarien automatisiert aus. Die Ausführung der Szenarien beginnt mit der Erhebung der Testdaten. Dabei werden Laufzeitinformationen des SUT abgespeichert. Aus diesen Laufzeitinformationen werden Informationen zu den, im Szenario definierten, Metriken abgeleitet und deren Bedeutung interpretiert. Anhand dieser Metriken wird anschließend der Testreport erstellt, welcher Informationen über den Zustand des Systems enthält. Im Folgenden wird beschrieben, wie dieses Testkonzept im Testsystem implementiert wurde.

Das Testsystem ist eine sowohl vom SUT, als auch vom *Continuous Delivery*-Prozess unabhängige Anwendung und kann somit auch separat eingesetzt werden. Gestartet wird sie mittels einer ROS-Launch-Datei und generiert als Ausgabe verschiedene textbasierte Report-Dateien. Die Einbettung des Testsystems in den *Continuous Delivery*-Prozess wird in Abschnitt 5.2 beschrieben. Das Ziel ist es, den Entwickler beim Testen früh zu unterstützen, damit er auf Fehler bereits in einem frühen Stadium aufmerksam wird und diese früh behoben werden können.

Wie in Tabelle 5.1 beschrieben, soll das Testsystem in ROS eingebettete Systeme testen. Daher fiel die Entscheidung darauf, auch ROS-Vorzüge wie die ROS-Knoten und das ROS-Nachrichtensystem zu nutzen. Abbildung 5.4 stellt die Architektur des Testsystem und dessen Interaktion mit umgebenden Komponenten graphisch dar.
Das Testsystem besteht aus zwei ROS-Knoten. Der testausführende Knoten – `Testsystem` – ist der Hauptknoten, organisiert die generelle Ausführung der Szenarien und bietet auch die Schnittstelle zum expliziten SUT. Dieser Knoten ist in C++ implementiert. Der zweite Knoten – `Validation` – enthält die Validierung der Szenarien. Die Validierungsskripte sind in Python geschrieben. Wie im *Continuous*

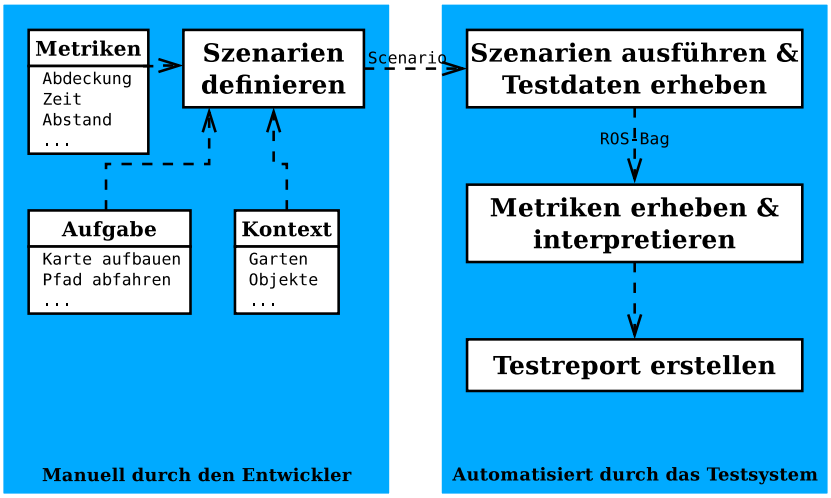


Abbildung 5.3.: Zur Ausführung des Testprozesses sind zwei Tätigkeiten notwendig: Zuerst muss der Entwickler manuell Szenarien definieren. Anschließend wird der Testfall automatisiert ausgeführt, indem Laufzeitinformation zum aktuellen Testlauf erhoben und ausgewertet werden.

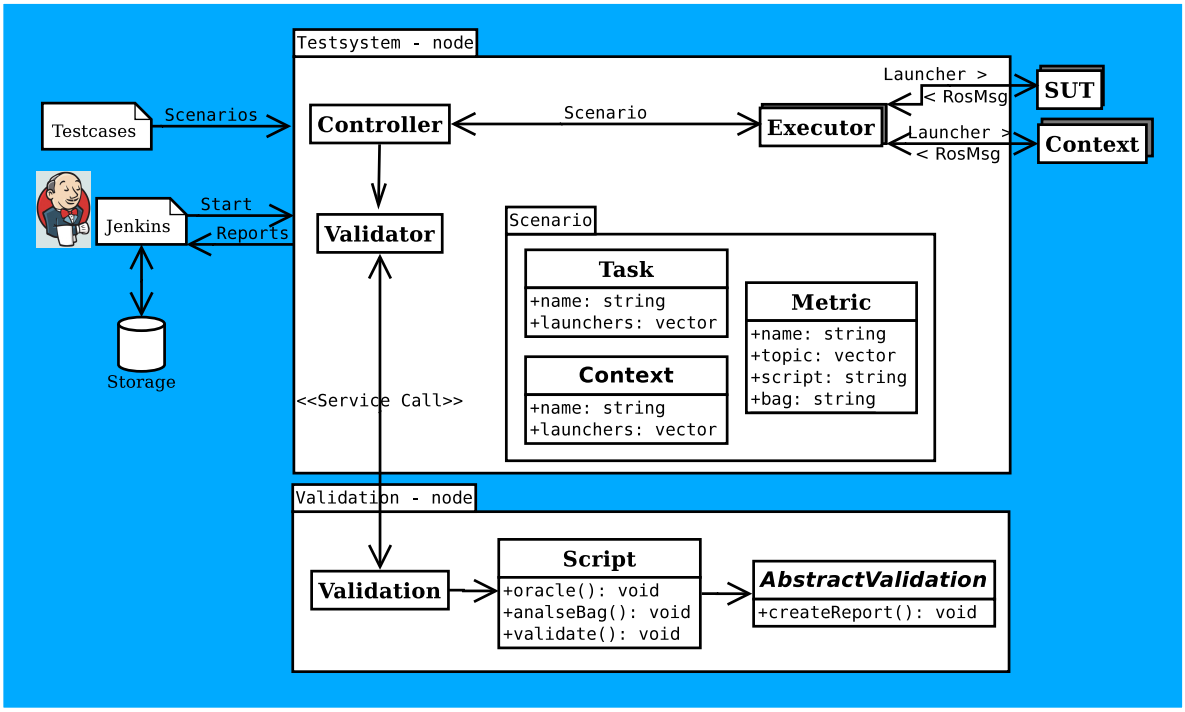


Abbildung 5.4.: Die Architektur des Testsystems. Sie beschreibt die beiden Knoten „Testsystem“ – zuständig für die Testausführung – und „Validation“ – zuständig für die Testvalidierung – sowie das Zusammenspiel mit Jenkins und dem SUT.

Delivery-Prozess wird auch im Testsystem das Szenarien-Modell (Abbildung 5.4) verwendet. Das bedeutet, dass ein Testfall mit einem Szenario gleichzusetzen ist. Dieses Szenario enthält wiederum eine Aufgabe (Task), einen optionalen Kontext (Context) und eine oder mehrere Metriken (Metrics). Szenarien werden in Form von YAML[2]-Dateien an den Testsystem-Knoten übergeben und dann einzeln, in der `Executor`-Klasse, zur Ausführung gebracht. Während der Ausführung sammelt der `Executor` die vom SUT veröffentlichten Nachrichten ein und speichert diese ab. Nach der expliziten Beendigung des Testfalls durch das SUT oder den Abbruch durch das Testsystem wird im `Controller` die Validierungskomponente – `Validator` – aufgerufen, welche für die Ausführung der Validierung und die Erstellung des Testreports zuständig ist.

Im Folgenden werden Teile des Testsystems genauer beschrieben. Dabei liegt der Fokus auf Problemen und Designentscheidungen. Zuerst wird das *Eingabeformat*, mit dessen Hilfe die Szenarien übergeben werden, behandelt. Anschließend geht es um die *Ausführung und Überwachung* des SUT. Abgeschlossen wird mit der Beschreibung der *Validierung*skomponente.

Eingabeformat

Das Eingabeformat basiert auf Szenarien. Eine `Task` enthält dabei Name und Beschreibung des Testfalls, sowie dazugehörige Launchers, wobei „Launcher“ stellvertretend für die Ausführung eines RosLaunchs, RosRun das Abspielen eines RosBags oder das Senden einer RosMessage steht. Der optionale `Context` stellt die Umgebung oder die Rahmenbedingungen, unter denen das SUT ausgeführt werden soll, dar. Er enthält dieselben Attribute wie `Task` und hat dabei im Endeffekt nur eine andere semantische Bedeutung. Die `Metric` gibt an, welche Laufzeitinformationen im SUT gemessen und ausgewertet werden sollen. Hierfür müssen neben dem Namen auch die *Topics* spezifiziert werden, auf welchen die Laufzeitinformationen veröffentlicht werden, sowie das Validierungsskript, welches das Resultat des Szenarios bestimmt. Die Speicherung der Laufzeitdaten des SUT wird einmal pro Szenario abgelegt.
Die Szenarien werden über eine YAML-Datei spezifiziert. Ein Beispiel hierfür findet sich in Listing 5.2. Um Aufgaben, Kontexte und Metriken wiederverwendbar zu machen, können diese in einem `Scenario-Store` abgelegt und in der YAML-Konfigurationsdatei referenziert werden.

Ausführung und Überwachung

Wenn die Szenarien definiert und eingelesen sind, werden sie dem `Executor` übergeben. Diese Komponente führt das SUT aus und überwacht es während seiner Laufzeit. Das Überwachen hat mehrere Aspekte:

[2] YAML-Webseite: http://yaml.org

Listing 5.2 Kommentierte Konfigurationsdatei für das Testsystem im YAML-Format. Die Datei kann mehrere Szenarien enthalten.

```yaml
testcases:                                  # Wurzel-Element
  - testcase:                               # Liste von 'Scenarios'
      task:                                 # Task-Objekt:
        name: Task title                    #   Name
        description: description of the task #   Beschreibung
        launchers:                          #   Liste der 'Launchers'
          - type: RosLaunch                 #     Launcher-Typ
            name: mypackage task.launch     #     Paket and Launch-Datei
            pause: 5                        #     Pause vor Launch (sec)
      context:                              # 'Context'-Objekt (wie Task)
        name: A Garden
        launchers:
          - type: RosRun                    #     Ein anderer Launcher-Typ
            name: mypackage context         #     Paket und Executable
            pause: 0.0
      metrics:                              # Liste von Metrics
        - topics:                           #   Metrics' Topics
            - /topic1
            - /topic2
          name: path length
          description: Metric description
          script: validation.pathlength.py  # Python-Skript (paket.modul.py)
      timeout: 300.0                        # SUT Timeout
  - testcase:                               # Weiteres Scenario
      task:
        id: tsk_parallellines
      context: ....
      ...
```

Start des SUT Hierbei müssen verschiedene Launcher ausgeführt werden.

1. Mittels C++ gibt es zum einen die Möglichkeit, mit den Befehlen `fork()` und `exec()` den aktuellen Prozess zu kopieren und dadurch für alle Objekte einen neuen Speicherbereich und somit einen vollständig unabhängigen Prozess zu erhalten. Diese beiden Befehle können mit der Funktion `system()` zusammengefasst werden. Listing 5.3 zeigt ein einfaches Beispiel für die `fork()-exec()`-Kombination.
   → Die Funktion `fork` liefert den Prozess-Identifikator (PID) des neuen Prozesses und lässt somit einen späteren Zugriff zu.

2. Eine weitere C++-Variante ist die Nutzung von Threads, welche zwar einen eigenen Container für das SUT bieten, jedoch denselben Speicherbereich nutzen.
   → Auch mittels Thread kann man später auf Prozesse zugreifen.

Listing 5.3 Kopieren eines Prozesses mittels `fork()`, Ausführung eines Befehls mittels `execvp()`. Dabei wird ein Prozess erstellt, der auf denselben Speicherbelegungen wie der originale Prozess aufbaut, dessen Speicher jedoch an einem anderen Speicherort liegt.

```
pid_t pid;
switch ( pid = fork() )
{
case -1:
   ROS_WARN("Error occurred");
   break;
case 0:
{
   execvp(exec_name, exec_args);
   exit(0);
   break;
}
default:
   // Der Prozess der Standardausführung.
   // pid kann hier gespeichert werden, um später damit zu arbeiten.
}
```

3. Die dritte Möglichkeit ist die Nutzung des ROS-internen RosLaunch-Konzeptes.
   → Die ROS-interne Lösung ist in Python programmiert und somit nur über Umwege mit C++ nutzbar. Außerdem werden andere Launcher als die von ROS bereitgestellten nicht unterstützt.

In dieser Arbeit wurde entschieden, dass die C++-Variante mit `fork` und `exec` genutzt wird. Sie unterstützt jegliche Launchertypen, die auch in einem Terminal ausgeführt werden können. Dabei ist es trotzdem möglich, mittels `kill` das SUT zu beenden.

Lebenszyklus Im Lebenszyklus des SUT kann es geschehen, dass sich das Programm beendet und das Testsystem davon nicht informiert wird. Es muss verhindert werden, dass das Testsystem weiter läuft, wenn das SUT bereits – egal ob abgestürzt oder unter normalen Bedingungen – beendet wurde. Hierfür bieten die geforkten Prozesse die Möglichkeit, mittels der zurückgegebenen PID zu testen, ob das Programm aktiv ist.
Ein anderer Fall ist, dass sich das SUT oder Teile davon nicht richtig beenden und noch immer aktiv sind. Auf Teilprogramme oder Abhängigkeiten, die für das SUT benötigt werden, kann nicht direkt zugegriffen werden. So zum Beispiel die Simulationsumgebung Gazebo[3], welche fehlerhaft ist und sich nicht zuverlässig schließt. Diese Prozesse müssen manuell oder im Quellcode beendet werden.

Timeout Ein weiteres Fehlverhalten, welches durch das Testsystem überwacht werden muss, ist die fehlende Reaktion des SUT während der Ausführung. Hierfür wird in der Konfigurationsdatei ein Timeout angegeben. Auch in diesem Fall ist es möglich, dass über die PID des SUT-Prozesses und der Funktion `kill` das SUT beendet wird.

[3] Gazebo-Webseite: http://gazebosim.org

Während der Ausführung des SUT sollen Laufzeitdaten gesammelt werden. Das in dieser Arbeit betrachtete SUT bietet keine Schnittstellen, welche den Zugriff auf Laufzeitinformationen bieten. Auch ist es bei einem System dieser Komplexität nur begrenzt möglich, lediglich einzelne Komponenten des Systems zu testen. Zudem liegt der Fokus des Testsystems auch auf dem Erstellen eines Akzeptanztests, welcher das Verhalten des Gesamtsystems im Blick hat. Aus diesem Grund ist es nötig, dass das SUT Informationen bereitstellt, welche über den aktuellen Zustand des Systems berichten. Zusätzlich zu aktuellen, das SUT betreffenden Daten, muss ein klares „Testende"-Kriterium festgelegt werden. Für die Implementierung dieses Sammelns der Laufzeitdaten gibt es verschiedene Möglichkeiten:

Eigenimplementierung Eine Möglichkeit besteht darin, ein eigenes Loggingverfahren zu implementieren, welches beispielsweise auf die Boost[4]-Bibliothek zurückgreift. Hierfür ist es nötig, eine eigene Infrastruktur aufzusetzen, um beispielsweise die Zuordnung der Nachrichten zu bestimmten Messungen zu gewährleisten. Auch ist die Typsicherheit der geloggten Daten nur mit einem gewissen Aufwand zu erreichen.

ROS-Message Eine weitere Möglichkeit bieten ROS-Messages. Sie sind ein fester Bestandteil der ROS-Umgebung und werden daher vollständig unterstützt. Sie erlauben komplexe Nachrichtenkonstrukte, welche typsicher und wiederverwendbar sind. Die Struktur der Nachricht wird in der Message-Definition beschrieben und kann von anderen Programmpaketen einfach übernommen werden. Listing 5.4 zeigt, wie eine ROS-Message im Testsystem Logging aussehen muss. Neben dem `Message`-Teil, welcher die eigentliche Nachricht enthält, gehört zu jeder Definition der *Header*, der einen Zeitstempel und die Version der Message-Definition enthält. Auch das „Testende"-Kriterium lässt sich in ROS-Messages abbilden und kann somit äquivalent zu anderen Laufzeitinformationsnachrichten behandelt werden.

Publish Zum Veröffentlichen einer Nachricht müssen im SUT kleine Änderungen vorgenommen werden. Zur Veröffentlichung wird zum einen ein `Publisher` benötigt, welcher auf eine Message-Definition und auf ein *Topic* festgelegt wird. Im Anschluss können auf diesem *Publisher* mehrere Nachrichten des festgelegten Typs veröffentlicht werden. Listing 5.5 beschreibt das Veröffentlichen einer „Testende"-Nachricht.

Subscribe Im Gegensatz zum Publish kann das Abonnieren, der für den Testfall benötigten Nachrichten automatisiert funktionieren. Hierfür sollen bei der Metrik-Definition alle benötigten *Topics* angegeben (Listing 5.2) und durch das Testsystem automatisiert aufgezeichnet werden.

Eigenimplementierung Eine Eigenimplementierung bietet den Vorteil, dass diese genau die Funktionen enthält, die benötigt werden. Hierfür muss ein generischer Subscriber implementiert werden, der auf mehrere Nachrichtentypen und *Topics* hört. Dafür muss für jedes *Topic* ein eigenes Subscriber-Objekt angelegt werden. Die Callback-Funktion, welche eingehende Nachrichten entgegennimmt, muss diese generisch verarbeiten.

[4]Boost-Webseite: http://www.boost.org

Listing 5.4 Definition einer ROS-Message zur Veröffentlichung von Laufzeitinformationen. Eine Nachricht muss den „Header“, welcher den Zeitstempel der Nachricht enthält sowie die Versionsnummer der Message-Definition beinhalten.

```
# Header
Header header
int32 version       # Version der Message-Definition

# Message
int32 iteration     # Typ Integer
int32 nCall
float64 pathLength  # Typ Float
failureCode error   # Andere Msg-Definition als Typ
```

ROS-Recorder ROS bietet viele kleine Werkzeuge, welche das Entwickeln unter ROS vereinfachen. Ein solches ist der *Recorder*. Er bietet als Kommandozeilen-Werkzeug die Möglichkeit, bestimmte *Topics* zu abonnieren und schreibt diese in den Container RosBag. Der Quellcode des ROS-Frameworks ist frei verfügbar. So auch das Paket `ros_comm`[5], welches den *Recorder* enthält. Der *Recorder* wurde als eigenständige Anwendung entwickelt, welche nach einer Ausführung vollständig beendet und dann von neuem gestartet wird. Daher müssen zwei Änderungen am Quellcode des *Recorder*s vorgenommen werden.

Aufgrund seiner Reife und der bereits vorhandenen umfangreichen Funktionalität wird zum Sammeln der Laufzeitinformationen der *Recorder* von ROS verwendet.
Der *Recorder* muss nach der Ausführung des SUT und der Beendigung des Tests kontrolliert beendet werden. Das heißt, dass alle Nachrichten, die gerade noch abgearbeitet, auch wirklich abgespeichert werden. Standardmäßig wird dies bei ROS durch das Kommandozeilen-Tastenkürzel `Strg-c` realisiert. Im Testsystem muss hierfür der *Recorder-Thread* zu einem kontrollierten Beenden gebracht werden. Dies wurde implementiert, indem das „Hören“ auf das Abbruchkommando `Strg-c`, durch ein „Retire“-Konstrukt ausgetauscht wurde. Im „Retire“-Konstrukt wird in C++ der Datentyp `std::atomic_bool`, welcher atomare Zugriffe auf eine Boolean-Variable bietet, verwendet. Die Variable `retired` vom Typ `std::atomic_bool` wird mit `false` initialisiert und auf `true` gesetzt, wenn der Recorder-Thread beendet werden soll.
Bei der Ausführung von mehreren Szenarien nacheinander wird auch der *Recorder* im selben Adressbereich mehr als einmal ausgeführt. Da der Recorder in aller Regel nur einmal ausgeführt wird, hat dies zur Folge, dass nicht getestete Nebeneffekte auftreten können. So werden abonnierte *Topics* bei Beendigung des Roboters nicht wieder abbestellt. Für das Abbestellen der *Topics* muss eine Liste angelegt werden, welche die abonnierten *Topics* enthält und diese am Ende wieder abbestellt werden.

[5] ROS-Paket des *Recorders*: http://wiki.ros.org/ros_comm

Listing 5.5 Publish einer ROS-Message am Beispiel der „End-Message“. Hierfür muss ein Publisher-Objekt erstellt werden, welches immer einem Message-Typ und einem *Topic* zugeordnet ist. Auf diesem Publisher-Objekt kann anschließend eine neue Nachricht veröffentlicht werden.

```
// Publisher stellt auf Topic '/end_msg' eine Nachricht vom Typ std_msgs::String bereit
ros::Publisher ts_end_pub_ = nodehandle.advertise<std_msgs::String>("/end_msg", 1);

// Publish message
std_msgs::String msg;
msg.data = std::string("%%FINISHED%%");
ts_end_pub_.publish(msg);
```

Validierung

Im Anschluss an die Ausführung der Testfälle müssen diese auch validiert werden.

Validierungsskripte

Zur Validierung dienen die Validierungsskripte. Bei einer Validierung muss auf statistische Methodiken zurückgegriffen werden. Hierfür können Sprachen wie Matlab oder Python genutzt werden. Für die Einbindung dieser Sprachen bieten sich zwei Ansätze an:

Direkter Aufruf Eine generische Möglichkeit ist das Starten der Validierungsskripte mittels eines Systemaufrufs. Hierbei können in C++ Funktionen wie `system()` oder `popen()` genutzt werden. Dabei stellt sich die Frage, wie Ergebnisse wieder zurückgegeben werden können. Die Methode `popen()` bietet die Möglichkeit, die Ergebnisse in eine *Pipe* zu schreiben und anschließend darauf zuzugreifen. `System()` gibt ebenfalls die Ausgabe des aufgerufenen Programms zurück.

Einbettung der Validierungssprache Für Sprachen wie Python und Matlab bietet C++ eine Schnittstelle, mit welcher direkt auf Komponenten dieser Sprachen zugegriffen werden kann. Dadurch kann direkt auf Funktionen, Parameter und Rückgabewerte zugegriffen werden.

ROS-Knoten Die dritte Möglichkeit bietet ROS selbst. Durch seine Knoten-Architektur ist es möglich, für neue logische Komponenten, aber auch als „Sprachbrücke“ einzelne ROS-Knoten zu verwenden. Kommuniziert werden, kann entweder durch ROS-Messages und ROS-Services, oder über den Parameter-Server (Abschnitt 2.4).

Im Fall des Testsystems bieten sich die beiden Sprachen `Matlab` und `Python` an. Python ist eine sehr vielseitige Sprache, welche zu den beiden Haupt-ROS-Sprachen gehört und daher auch sehr gut unterstützt wird. Matlab wurde im Rasenmäherprojekt bereits zur Auswertung geloggter Daten genutzt und könnte daher synergetische Effekte bei der Erstellung von Validierungsskripten bringen.

Matlab Matlab[6] wird vom Unternehmen „The MathWorks, Inc“ entwickelt. Es ist kommerziell erhältlich und dient zur Lösung von mathematischen Problemen. Matlab zählt nicht zu den von ROS offiziell unterstützten Sprachen. Allerdings arbeitet „The MathWorks“ an einer ROS-Integration, welche es ermöglicht, ROS-Knoten in Matlab zu starten und ROS-Funktionen zu nutzen. Matlab ermöglicht auch eine Einbettung in C++. Durch das Öffnen einer Matlab-Engine (Listing 5.6) kann im C++-Quellcode direkt auf Matlab-Befehle zugegriffen werden.
Für die in dieser Arbeit benötigten Validierungsskripte reicht es aus, wenn im Bezug auf ROS das Erstellen eines Knotens und das Lesen von *ROS-Bag*s unterstützt wird. Dies kann, zumindest zum Teil, durch eine *OpenSource*-Implementierung des *ROS-Bag*s[7] für Matlab realisiert werden. Es ist dabei nicht möglich, einen eigenen Knoten zu erstellen, jedoch können Knoten in C++ erstellt werden und auf diesen dann über die Matlab-Engine die Validierungen gestartet werden.
Eine nahezu vollständige ROS-Unterstützung bietet die Matlab-Toolbox „Robotics System Toolbox“[8]. Sie muss als Toolbox separat installiert werden. Listing 5.7 stellt dar, wie ein ROS-Knoten erstellt wird.

Python Python[9] gehört zu den ROS-nativen Sprachen. Es wird von der „Python Software Foundation“ entwickelt. Das Paket `NumPy`[10] bildet komplexe mathematische Funktionen ab, welche zur Validierung der Szenarien hilfreich sind. Es gibt zwei Möglichkeiten, nach welchen bei der Implementierung vorgegangen werden kann.
Zum einen ist es möglich, Python in C++ einzubetten. Mittels des Headers `Python.h` kann direkt auf Python-Module zugegriffen und deren Funktion, inklusive Parameter und Rückgabewerte, angesprochen werden. Mit Hilfe der `Boost`-Bibliothek[11] wird dieser Zugriff vereinfacht und sicherer. Listing 5.8 beschreibt, wie in C++ Python-Objekte erstellt und Funktionen darauf ausgeführt werden können.
Aufgrund der nativen Unterstützung durch das ROS-Framework ist es möglich, ROS in Pythonskripten zu starten und mittels Message- beziehungsweise Service-Aufrufen, sowie über den Parameter-Server auf Anfragen zu reagieren. Listing 5.9 beschreibt das Erstellen eines ROS-Knotens in Python.

Bei direkten Aufrufen kann nur eine vollständige Anwendung beziehungsweise ein Skript aufgerufen werden. Zudem müssen Parameter als String über den Programmaufruf übergeben werden. Die Einbettung der Sprache erlaubt dies in einem gewissen Umfang, allerdings sind die Konstrukte, welche die Einbettung ermöglichen, komplex und schwer verständlich, was eine gewisse Fehleranfälligkeit mit sich bringt. Die Implementierung eines eigenen ROS-Knotens ist eine zum ROS-Framework passende Lösung. Sie ist übersichtlich, da eine klare Trennung existiert und Parameter komfortabel und typsicher über den Parameter-Server übergeben werden können.

[6] Matlab-Webseite: http://de.mathworks.com
[7] Matlab-*ROS-Bag* Projektseite: http://github.com/bcharrow/matlab_rosbag
[8] ROS-Toolbox-Webseite : http://de.mathworks.com/hardware-support/robot-operating-system.html
[9] Python-Webseite: http://www.python.org
[10] NumPy-Webseiten: http://www.numpy.org
[11] Boost-Python-Webseite: www.boost.org/libs/python

Listing 5.6 Einbetten von Matlab-Skripten in C++. Die Matlab-*Engine* ermöglicht es, einzelne Befehle oder ganze Skripte zu nutzen.

```
#include "engine.h"
Engine* ep;
ep = engOpen("");
// hide window
engSetVisible(ep, 0);
if (!(ep)) {
   ROS_INFO_STREAM(stderr << " Can't start MATLAB engine\n");
   return *"";
}
string addPath = ;
char* string;
engOutputBuffer(ep, string, 1024);
engEvalString(ep, "addpath('/skript/pfad')");
int ptr = engEvalString(ep, "funktion");
return str(*string);
```

Listing 5.7 Erstellen eines ROS-Knotens und Lesen eines *ROS-Bag*s mit der Matlab-Toolbox (Robotics System Toolbox).

```
rosinit()
bag = rosbag(filepath)
bag.AvailableTopics
bagselect = select(bag, 'Topic', '/my_msg')
msgs = readMessages(bagselect);
rosshutdown
```

Zu Beginn der Arbeit sollten sowohl der Matlab- als auch der Python-Ansatz implementiert werden. Bei der Implementierung der Matlab-Lösung ergaben sich Probleme, beispielsweise beim Zugriff auf Matlab-Skripte in anderen Ordnern und generell bei der Installation der Toolbox. Aus Zeitgründen wurde daher die Implementierung der Matlab-Lösung abgebrochen. Für eine zukünftige Implementierung bietet es sich jedoch an, einen separaten ROS-Knoten zu erstellen, welcher über die Matlab-Engine Validierungsskripte ausführen kann. Dies kann unter Nutzung des OpenSource *ROS-Bag*s ohne Extra-Lizenz der sehr mächtigen, aber problembehafteten Toolbox geschehen, da nur noch die *ROS-Bag*-Unterstützung benötigt wird.
Da bei der Nutzung der Python-ROS-Pakete keine Probleme auftraten, bietet es sich an, ROS in einem eigenen Knoten zu betreiben.

Die Python-Validierung besteht aus zwei Modulen. Es gibt das `Validierung.py`-Skript, welches auf den Service-Trigger, welcher vom Testsystem bei einer ausstehenden Evaluierung durchgeführt wird, reagiert. Aus diesem Modul heraus wird das zweite Modul, das eigentliche Validierungsskript, aufgerufen. Es leitet die Klasse `AbstractValidation` ab, welche Basisfunktionen des Validierungsvorgangs enthält.

Listing 5.8 Einbetten von Python-Skripten in C++ unter Verwendung der Boost-Bibliothek.

```
#include <Python.h>
#include <boost/python.hpp>
#include <boost/python/object.hpp>
// Initialize calling of python scripts
using namespace boost::python;
Py_Initialize();
try
{
   // Import python module and call main-function on its class.
   // Try to retrieve xml report as string
   char* progName[] = { };
   PySys_SetArgv(0, progName);
   object vModule = import("package.module");
   object vClass = vModule.attr("classname")();
   object vObject = vClass.attr("function")(parameter1, parameter2, ...);
   xmlReport = extract<string>(vObject);
}
catch (const error\_already\_set& e)
{
   PyErr_Print();
}

return xmlReport;
```

Listing 5.9 Erstellen eines ROS-Knotens und Lesen eines *ROS-Bag*s mit Python.

```
import rosbag
rospy.init_node('pyvalidation', anonymous = True) 1 import rosbag
bag = rosbag.Bag('test.bag')
for topic, msg, t in bag.read_messages(topics=['chatter', 'numbers']):
   print msg
bag.close()
```

5.4. Zusammenfassung

In diesem Kapitel wurde ein *Continuous Delivery*-Prozess vorgestellt. Dieser Prozess ist dabei sehr stark auf den Akzeptanztest ausgerichtet, welcher das *Continuous Delivery* abschließt.
Hierfür wurden im Abschnitt 5.1 die Anforderungen an den gesamten Prozess abgesteckt. Diese Anforderungen zielen darauf ab, dass die Möglichkeit besteht, den aktuellen Zustand des SUT von Entwicklungsbeginn an im Blick zu haben. Um diese Anforderungen zu implementieren wurde ein Testsystem entworfen, welches vom SUT als ROS-Messages modellierte Laufzeitinformationen aufzeichnet und diese im Anschluss validiert. Gesteuert werden diese Testausführungen durch den *Continuous Integration*-Server Jenkins, welcher auch für die Darstellung der Resultate zuständig ist.

6. Evaluierung

Dieses Kapitel beschreibt die Evaluation des Testsystems. Bei der Evaluation soll ermittelt werden, wie gut das Testsystem im praktischen Einsatz funktioniert. Hierfür fand eine Befragung unter Mitarbeitern statt. Abschnitt 6.1 beschreibt das Vorgehen und den Aufbau der Evaluierung. Im zweiten Teil (Abschnitt 6.2) werden die Ergebnisse der Befragung zusammengefasst. Abschnitt 6.3 beschreibt mögliche Einflüsse, die die Ergebnisse der Befragung beeinflusst haben könnten. Abgeschlossen wird dieses Kapitel mit der Diskussion (Abschnitt 6.4) der Ergebnisse.

6.1. Befragung

In diesem Abschnitt wird das Vorgehen der Befragung und der dafür genutzte Fragebogen beschrieben.

Die Evaluation des Testsystems erfolgte durch eine Befragung relevanter Entwickler. Die befragten Entwickler waren Softwareentwickler, welche im Rahmen von autonomen Robotern Software schreiben und daher einschätzen können, auf welche Aspekte beim Testen geachtet werden muss. Für die Befragung konnten fünf Mitarbeiter der Abteilung, in der der autonome Rasenmäher entwickelt wird sowie thematisch benachbarter Abteilungen, gewonnen werden. Sie sind somit am entwickelten Testsystem oder an dessen Entwicklungsansätze interessiert.

Die Befragung fand in drei Phasen statt.

Workshop Im Workshop wurde anhand von Folien das Konzept und die Architektur des Testsystems vorgestellt. Beispielhafte Konfigurationen verdeutlichten, wie das System praktisch genutzt werden kann. Diese Auftaktveranstaltung sollte dazu dienen, die interessierten Entwickler mit dem neuen System vertraut zu machen.

Test In der zweiten Phase erfolgte das praktische Ausprobieren des Testsystems. Hierfür wurde jedem Entwickler eine Instanz des Testsystems sowie eine Anleitung für das Einrichten eines lokalen Jenkins-Servers und das Erstellen eines ersten Testfalls bereitgestellt. Die praktische Phase erstreckte sich über zwei Wochen, in denen die Entwickler selbständig das Testsystem besser kennenlernen und Testfälle erstellen konnten. Dabei gab es die Möglichkeit, Rückfragen zu stellen oder konkrete Hilfe bei Problemen zu bekommen.

Feedback Am Ende der zwei Wochen wurde jeder Entwickler gebeten, einen Fragebogen auszufüllen, der die Qualität des Testsystems bemisst. Der Fragebogen befindet sich im Anhang A.2.

Beschreibung des Fragebogens

Der Fragebogen besteht aus drei Teilen. Im ersten wird die Definition der Testfälle evaluiert. In diesem Abschnitt werden Fragen zum YAML-Eingabeformat, zur Erstellung der Validierungskripte, zur Erstellung des Reports und zur allgemeinen Bewertung der Testfalldefinition gestellt. Im zweiten Teil geht es um die Qualität der Ergebnisse, welche durch das Testsystem bereitgestellt werden. Diese beziehen sich auf die Ergebnisse, welche im Jenkins-Server angezeigt werden. Der letzte Teil enthält allgemeine Fragen zum Testsystem, die die Praxistauglichkeit des Testsystems erfassen sollen.

6.2. Ergebnisse

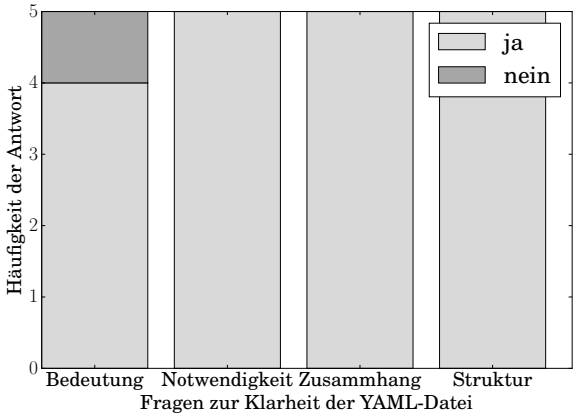
Dieser Abschnitt enthält die Ergebnisse der Befragung der Entwickler. Aufgrund der geringen Anzahl befragter Entwickler wurde die Auswertung der Ergebnisse manuell durchgeführt.

Erstellung eines Testfalls

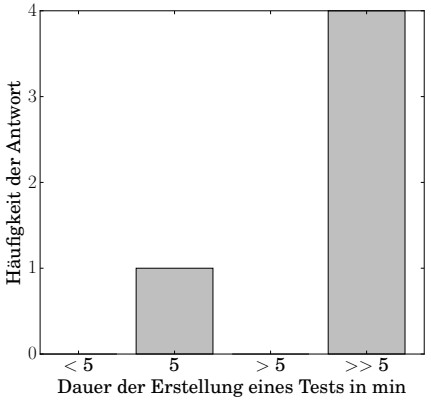
Der erste Teil des Fragebogens ergab folgende Ergebnisse: Bei den Fragen, welche sich damit beschäftigen, ob das Vorgehen bei der Eingabe über das YAML-Format klar war, wurde nur die Frage nach der Bedeutung der Attribute einmal mit *nein* beantwortet. Alle anderen Fragen wurden einstimmig mit *ja* beantwortet. Abbildung 6.1a zeigt die gesamte Bewertung zum Verständnis der Attribute der YAML-Datei, welche bei drei Fragen eine Zustimmung von 100 % und bei einer Frage von 80 % bekam. Für die Erstellung der Validierungsskripte muss Python verwendet werden. Die Befragung ergab, dass diese Sprache wenig bekannt ist. Generell ergaben sich hierdurch einige Probleme beim Implementieren der Validierungsskripte. Probleme traten außerdem beim Aktivieren von Python für die Arbeitspakete der Befragten auf. Zudem war das Verständnis für Attribute im Validierungsskript für 40 % der Befragten nicht durchgängig gegeben.
Das Erstellen eines Testfalls sollte im Regelfall fünf Minuten dauern. 80 % der Befragten gaben an, dass sie deutlich länger als diese Zeitspanne für die Erstellung gebraucht hätten. Dieselben 80 % gaben aber auch an, dass diese Dauer durch mehr Routine vermutlich verkürzt werden könne. Zwischen der Angabe der Dauer und der Einschätzung der Verbesserungsmöglichkeiten besteht eine Korrelation von 100 %, welche in Abbildung 6.1b dargestellt ist.
Insgesamt betrachtet gaben 60 % der Befragten an, dass die Erstellung der Testfälle für sie anstrengend war. Die Korrelation zwischen der gefühlten Anstrengung bei der Testfallerstellung und den fehlenden Kenntnissen in Python ergab einen Koeffizienten von 0.14 und ist damit nicht gegeben. Abbildung 6.2a zeigt aber, dass alle Befragten ihr Empfinden bei der Erstellung eines Tests mit *gut* oder *sehr gut* bewerteten.

Anzeige der Resultate

Der zweite wichtige Faktor, der betrachtet werden muss, ist die Darstellung der Testergebnisse. 100 % der Befragten bestätigten, dass die Details zu den Ergebnissen *gut* seien. Jedoch hatten 60 % der Befragten Vorschläge für weitere Informationen oder Artefakte, die angezeigt und archiviert werden sollten. Zu diesen gehören zum Beispiel Grafiken, welche den zurückgelegten Weg eines Roboters

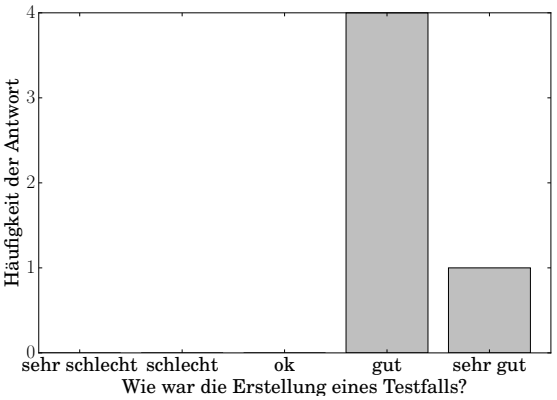


(a) Verständnis der YAML-Datei

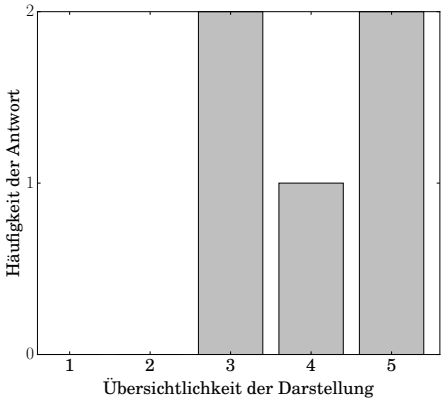


(b) Dauer der Erstellung eines Testfalls

Abbildung 6.1.: **(a)** zeigt die Antworten der Befragung zum Verständnis der YAML-Datei. Dabei wurden die Fragen gestellt, ob die Bedeutung der Attribute, die Notwendigkeit einzelner Attribute, der Zusammenhang der Attribute und die Gesamtstruktur der Datei verständlich waren. **(b)** zeigt die Häufigkeit der Dauer, die zur Erstellung eines Testfalls benötigt wurde. Ein Teilnehmer benötigte fünf Minuten, vier benötigten deutlich länger.



(a) Empfinden bei der Testfallerstellung



(b) Übersichtlichkeit der Darstellung

Abbildung 6.2.: **(a)** beschreibt die Empfindung der Testperson beim Erstellen des Testfalls **(b)** beschreibt die Häufigkeit von Antworten zur Frage zur Übersichtlichkeit der Ergebnisdarstellung. „1“ bedeutet „sehr unübersichtlich“, „5“ bedeutet „sehr übersichtlich“.

darstellen. Die Übersichtlichkeit der Ergebnisse wurde von 60 % der Befragten mit *übersichtlich* oder *sehr übersichtlich* bewertet (Abbildung 6.2b). 40 % bewerteten diese als *ok*. Als Grund für fehlende Übersichtlichkeit wurde angegeben, dass der Testreport nicht direkt angezeigt wird und dafür teilweise nicht sofort erkennbar sei, was die Ergebnisse sind.

Allgemeine Ergebnisse

Hinsichtlich der meisten Aspekte wurde das Testsystem positiv bewertet. Kritikpunkte waren die fehlende Unterstützung beim Ablegen von großen Dateien und die fehlende Ansteuerung einer Datenbank. Auch wurde angemerkt, dass Testfälle immer noch gut gepflegt werden müssten. Zudem sollte das Testsystem mehr Automatisierung bieten, um dem Entwickler die Arbeit noch mehr zu erleichtern. Bei der Erstellung sollten Entwickler noch besser unterstützt werden.

6.3. Threats to Validity

Dieser Abschnitt beschreibt Umstände, die die Ergebnisse der Evaluierung beeinflusst haben könnten. Im Allgemeinen sind dies Einflüsse, die durch die Konzeption oder durch Fehler bei der Durchführung der Evaluation auftraten.
Die Evaluierung wurde in einem Kreis von fünf Entwicklern durchgeführt. Dies ist eine zu geringe Anzahl um ein repräsentatives und statistisch auswertbares Ergebnis zu erhalten. Um die Brauchbarkeit des Systems feststellen zu können, war es notwendig , dass alle Befragten sich im Umfeld der Entwicklung von Robotersoftware auskennen und Erfahrungen haben, um einschätzen zu können, ob das System im täglichen Gebrauch funktioniert. Mitarbeiter müssen für diese Aufgabe speziell freigestellt werden, damit ausreichend Aufwand investiert werden kann.
Die Befragten konnten jedoch nicht vollständig von ihren alltäglichen Aufgaben entbunden werden, weswegen es trotzdem möglich war, dass die Durchführung der Evaluation eher einfach gehalten wurde und beispielsweise auf simple und leicht implementierbare Szenarien zurückgegriffen wurde.
Zwei der Befragten waren direkt an der Betreuung der Arbeit beteiligt und haben daher bereits von Beginn an Feedback zum Testsystem gegeben. Daher war es unwahrscheinlich, dass sie dem Ansatz des Testsystems komplett widersprechen würden.
Die Befragung hat ergeben, dass die Sprache Python nicht allen Befragten bekannt war. Dies kann einen negativen Einfluss auf das Ergebnis gehabt haben, da die Befragten neben dem Testsystem zusätzlich eine neue Sprache erlernen mussten.
Beim Testen des Testsystems wurden die Befragten nicht allein gelassen. Wenn Probleme auftraten wurden Tipps und Ideen weitergegeben. Dies hatte Einfluss auf die Nutzung des Systems und somit auf das Ergebnis der Evaluation.
Die Evaluation fand nicht anonymisiert statt. Dies kann zur Folge haben, dass die Befragten nicht immer zu 100 % ehrlich geantwortet und somit das Ergebnis verfälscht haben.

6.4. Diskussion

Die Teilnehmer der Befragung waren mit dem Testsystem zufrieden und bewerteten sowohl die Eingabe der Testfälle als auch die Ausgabe der Resultate als gut. Bei der Definition der Testfälle fiel auf, dass das Erstellen eines Testfalls deutlich länger dauerte, als in den Anforderungen angegeben. Allerdings ist zu erwarten, dass dies mit mehr Routine und dem Vorhandensein der Testgrundlagen in jedem Paket schneller geht. Bei einer Korrelation zeigte sich ein direkter Zusammenhang zwischen der Dauer und dem vermuteten Verbesserungspotential. Bei solch einer geringen Teilnehmerzahl kann dieser Schluss zwar nicht statistisch gezogen werden, jedoch ist eine Tendenz erkennbar. So ist davon auszugehen, dass durch Routine ein menschlicher Lerneffekt eintritt. Trotz der guten Bewertung der Eingabemethode fühlte sich deren Erstellung für die Testperson anstrengend an. Ein möglicher Grund hierfür sind die fehlenden Python-Kenntnisse. Auch hier könnte mit einer höheren Teilnehmerzahl unter Umständen eine Korrelation entdeckt werden.
Die Validierungsskripte könnten verbessert werden, indem sie automatisierter auffassen und weiterleiten, und generell mehr Informationen und Artefakte bereitstellen. Eine erhöhte Automatisierung ist zu einem gewissen Grad möglich. Jedoch werden dadurch Spezialfälle ausgegrenzt, da nur sehr allgemeine Informationen weitergegeben werden können. Weitere gewünschte Artefakte wie Grafiken, welche einen abgefahrenen Pfad darstellen, können mittels der Bibliothek *Numpy* bereits im aktuellen System erstellt werden.

7. Fazit

Dieses Kapitel bildet den Abschluss der vorliegenden Arbeit und enthält eine Zusammenfassung des Vorgestellten sowie einen Ausblick, welcher noch offene oder erweiterte Fragestellungen betrachtet.

7.1. Zusammenfassung

Autonome Roboter werden mit Hilfe vieler verschiedener Sensoren gesteuert. Das Zusammenspiel dieser Sensoren muss durch eine komplexe Software bereitgestellt werden. Damit Roboter sich weitgehend ohne die Kontrolle von außen bewegen können, muss die Entwicklung dieser Software in einem Prozess, welcher insbesondere die Funktion der Software validiert, geschehen. In der vorliegenden Arbeit wurde ein System zur kontinuierlichen Entwicklung von Software für autonome Roboter, auf Basis von *Continuous Delivery* nach Humble und Farley [HF10] entwickelt. Die Herausforderung lag dabei, dass ein flexibles Testsystem entworfen werden musste, welches aufgrund der hohen Komplexität und Schwankung der Eingabesensorik nicht-triviale Validierungstechniken benötigt.
Zu Beginn des Entwicklungsprozesses wurden drei Abteilungen der Robert Bosch GmbH befragt. Zwei der drei Abteilungen nutzten bereits einen der *Continuous Integration* ähnlichen Prozess. Alle Abteilungen wollten zukünftig mehr in Qualität – beispielsweise durch Tests – ihrer Software investieren. Auf Basis der Befragungsergebnisse wurde ein Entwicklungsprozess mit Fokus auf den kontinuierlichen und automatisierten Akzeptanztest, welcher die Funktion der Software testen soll, entwickelt. Als Grundlage für das Testsystem wurde die Werkzeugumgebung ROS verwendet, welche auch als Basis für die in den Abteilungen entwickelte Robotersoftware eingesetzt wird. Testfälle werden mit Hilfe von Szenarien beschrieben. Ein Szenario besteht aus einem Task – der Aufgabe, die der Roboter erledigen soll –, einem Context – dem Umfeld, in dem sich der Roboter befindet – und Metriken, welche Laufzeitinformationen der Robotersoftware auswerten. Die Laufzeitinformationen werden mittels ROS-Messages im SUT veröffentlicht und können so durch das Testsystem gelesen und abgespeichert werden. Die Auswertung der Metriken findet über sogenannte Validierungsskripte statt, welche mit *Python* und *Numpy* auch auf statistische Analyse-Methoden zurückgreifen kann. Die Anzeige der Ergebnisse findet mittels textbasierter Ergebnisreports in einem *Jenkins*-Integrationsserver statt.
Nach der Implementierung wurde das Testsystem durch fünf Entwickler evaluiert. In der Evaluierung konnten die Entwickler das Testsystem selbst ausprobieren und wurden anschließend um Feedback gebeten. Das generelle Feedback war positiv, wobei Verbesserungsvorschläge für die Erstellung und Anzeige der Testreports, sowie die Handhabung von großen Ein- und Ausgabedateien gemacht wurden.

7.2. Ausblick

Die vorliegende Arbeit bietet Möglichkeiten der Erweiterung und Vertiefung verschiedener Aspekte. Zu Beginn der Arbeit war das Ziel, eine Validierung mittels *Python* und *Matlab* zu ermöglichen und deren Eigenschaften gegeneinander abzuwiegen. Im Laufe der Arbeit kam es allerdings vermehrt zu Problemen mit *Matlab*, weswegen es nicht möglich war, diesen Ansatz ausgiebig zu testen. Deshalb könnte in einem weiteren Schritt die Integration von *Matlab* vorgenommen werden.
In der Evalierung wurde festgestellt, dass das Erstellen des Testreports immer noch manuelle Schritte erfordert, welche im Folgenden weiter reduziert werden könnten. Hierfür müsste ein generisches Konzept gefunden werden, welches ohne viele Sonderfälle und trotzdem mit ausreichen Informationen Testreports erstellen kann. Ein weiterer Punkt ist die Darstellung der Ergebnisse in *Jenkins*. Zum aktuellen Zeitpunkt beinhaltet dieser lediglich einen textbasierten Testreport, eine Grafik über die Anzahl der fehlgeschlagenen Tests, sowie manuell generierte Grafiken.
Eine zusätzliche Erweiterung der Anzeige wäre die Möglichkeit, einzelne Metriken über ihren zeitlichen Verlauf grafisch darzustellen. Beim Ausführen der Testfälle kommt es vor, dass große Datensätze ins Testsystem eingespeist werden müssen oder ausgegeben werden. Es wird ein geeignetes Konzept benötigt, um mit diesen Dateien umzugehen. Für diesen Zweck könnte ein Bosch internes Werkzeug verwendet werden, welches das Ablegen und den Zugriff großer Dateien ermöglicht.
Eine praktische Erweiterung wäre außerdem die Möglichkeit, das SUT während der Tests zu überwachen. Beispielweise wäre dies durch eine Echtzeitdarstellung von aktuellen Laufzeitinformationen oder durch Hinzufügen von *Debugger*-Funktionen möglich.
Das Ergebnis der vorliegenden Arbeit testet ausschließlich die Software eines Roboters. Die Funktion der Sensoren und anderer Hardwarekomponenten und das korrekte Zusammenspiel zwischen Hard- und Software muss ebenfalls sichergestellt werden. Um dies zu ermöglichen könnte das Testsystem zu einem ganzheitlichen Test erweitert werden. Damit mit Hilfe dieses Testaufbaus ein SUT validiert werden kann ist es jedoch notwendig, dass das SUT extern überwacht wird. Die Automatisierung der Testausführung und -validierung ist daher nur unter großem Aufwand möglich.

A. Fragenbogen

A.1. Ist-Analyse

Kevin Wenz (CR/AEG)
Continuous Delivery

0.1 Welche Vorstellung habt ihr von Continuous Integration, Delivery und Deployment?

1 Genereller Prozess

Die folgenden Fragen beziehen sich auf den generellen Prozess bei der Entwicklung von Software für Roboter.

1.1 Wie viele Personen sind an der Entwicklung der Software beteiligt?

1.2 Für welche Aufgaben in der Softwareerstellung gibt es spezielle Verantwortlichkeiten?

1.3 Wie sieht der Softwareentwicklungsprozess aus? Welche zentralen Technologien werden verwendet? Dabei sollte neben dem generellen Ablauf auch auf den Umgang mit Softwareversionierung und Automatisierung eingegangen werden.

1.4 Was muss gemacht werden, wenn Software ausgeführt werden soll?

1.5 Was muss gemacht werden um Software wirklich auszuliefern?

1.6 Wie häufig wird Software integriert?

1.7 Wie häufig wird Software delivert?

1.8 Wie häufig wird Software deployt?

1.9 Bei welcher Tätigkeit werden die meisten Fehler gefunden?

1.10 Wie schnell werden gefundene Fehler gefixt?

2 Testen

2.1 Werden in irgendeiner Form Tests gemacht?

2.2 Wie viele Personen kümmern sich um das Testen der Software?

2.3 Gibt es explizit definierte Testfälle? Wenn nein, wie wird getestet?

2.4 Woraus werden Testfälle abgeleitet?

Kevin Wenz (CR/AEG)
Continuous Delivery

2.5 Welche Ebenen von Tests gibt es? (Unit, Integration, System, Komponenten, ...)

2.6 Welche Arten von Tests gibt es? (Funktion, Stress, Performanz, ...)

2.7 Werden Testfälle auf einer Simulationsumgebung ausgeführt? (nach Testart/-ebene)

2.8 Wird auf der konkreter Hardware getestet? (nach Testart/-ebene)

2.9 Zu welchen Zeitpunkten wird getestet?

☐ nach jedem Commit ☐ nachts ☐ beim Mergen
☐ beim Integrieren ☐ wenn es gerade passt ☐ _____

2.10 Wie hoch ist die Testüberdeckung (in Prozent)?

2.11 Wird das gesamte System gleichmäßig getestet? Gibt es Komponenten die näher betrachtet werden?

2.12 Gibt es konkret formulierte Akzeptanzkriterien? Wenn ja, wie sieht solch eine Formulierung aus?

2.13 Werden diese Akzeptanzkriterien getestet?

2.14 Was muss gemacht werden, um die Tests der Software auszuführen?

3 Vorgehen Funktionstests

3.1 Wie ist das Grundkonzept der Funktionstests?

3.2 Wie erfolgt die Definition der Funktionstestfälle? (Vorgehen/Arbeitsschritte und Verantwortlichkeiten)

3.3 Wie wird auf Testergebnisse zugegriffen?

- ☐ Logging-Daten werden gesammelt
- ☐ Es werden Variablen/Zustände aus dem System abgegriffen
- ☐ Das Verhalten des Roboters wird überwacht (visuell/Sensoren)
- ☐ _____

3.4 Wie werden Ergebnisse validiert?

- ☐ Werte werden exakt verglichen (<>=)
- ☐ Wertebereiche werden überprüft
- ☐ Log-Messages werden nach Warnings/Fehlern geparsed
- ☐ Das Verhalten des Roboters wird überwacht (visuell/Sensoren)
- ☐ _____

3.5 Automatisierung

- ☐ Funktionstests sind (teil)automatisiert. Grad der Automatisierung: _____
- ☐ Man kann ohne viel Aufwand alles automatisieren
- ☐ Es gibt Teile die manuell bleiben müssen. Gründe: _____
- ☐ _____

Kevin Wenz (CR/AEG)
Continuous Delivery

3.6 Welche Testmaße werden berücksichtigt? (Code-Coverage, Anforderungsüberdeckung, ...)

4 Zufriedenheit

4.1 Bilden die vorhandenen Tests den Zustand des Systems gut ab? (Das heißt, wenn Tests nicht fehlschlagen, ist das System fehlerfrei)

☐ Ja ☐ Nein

4.2 Welche Schritte sind mühsam oder anstrengend?

4.3 Was sollte noch automatisiert werden?

4.4 Gibt es typische Fehlerquellen?

4.5 Weiteres Verbesserungspotential beim Testen?

4.6 Weiteres Verbesserungspotential im Entwicklungsprozess?

A.2. Evaluation

1 TESTDEFINITION

Eingabeformat

1.1 War die Bedeutung der Attribute in der YAML-Datei klar?

☐ Ja ☐ Nein

1.2 War klar, welche Attribute in der YAML-Datei angelegt werden müssen?

☐ Ja ☐ Nein

1.3 War der Zusammenhang zwischen den einzelnen Attributen klar?

☐ Ja ☐ Nein

1.4 Ergibt die gesamte Struktur der YAML-Datei in Ihren Augen Sinn?

☐ Ja ☐ Nein

1.5 Wurde das Attribut 'parameter' in der YAML-Datei verwendet?

☐ Ja ☐ Nein

1.6 Welche Probleme gab es beim Erlernen der YAML-Datei?

Klicken Sie hier, um Text einzugeben.

1.7 War das Dateiformat 'YAML' bereits bekannt?

☐ Ja ☐ Nein

Validierungsskript

1.8 Wie gut kennen Sie Python?

überhaupt nicht ☐ ☐ ☐ ☐ ☐ sehr gut

1.9 Welche Probleme gab es bei der Implmentierung der 'Oracle'-Funktion?

Klicken Sie hier, um Text einzugeben.

1.10 Welche Probleme gab es bei der Implementierung der 'BagAnalyse'-Funktion?

Klicken Sie hier, um Text einzugeben.

1.11 Welche Ergebnistypen wurden gewählt?

Kevin Wenz (CR/AEG)

fixed-term.kevin.wenz@de.bosch.com

☐ Compare ☐ Operator (<,>,=) ☐ Tolerance

1.12 An welcher Stelle wünschen Sie sich mehr Unterstützung bei der Erstellung der Validierungsskripte?

Klicken Sie hier, um Text einzugeben.

Report

1.13 An welcher Stelle gab es Probleme bei der Erstellung des Testreports?

Klicken Sie hier, um Text einzugeben.

1.14 Welche Teile des Reports sollten freier gestaltbar sein und was sollte noch exportiert werden?

Klicken Sie hier, um Text einzugeben.

Allgemein

1.15 Wie lange hat das Festlegen eines Testfalls gedauert?

☐ weniger als 5 min ☐ ungefähr 5 min

☐ mehr als 5 min ☐ deutlich mehr als 5 min

1.16 Würde diese Dauer durch mehr Routine verbessert werden?

☐ Ja ☐ Nein

1.17 War das Erstellen des Testfalls anstrengend?

☐ Ja ☐ Nein

1.18 Wie empfanden Sie den Prozess des Erstellens eines Testszenarios?

sehr schlecht ☐ ☐ ☐ ☐ ☐ sehr gut

2 DARSTELLUNG DER ERGEBNISSE

Die Fragen zur Darstellung beziehen sich auf die Darstellung in Jenkins

2.1 Wie übersichtlich ist die Darstellung der Ergebnisse?

Kevin Wenz (CR/AEG)

fixed-term.kevin.wenz@de.bosch.com

sehr unübersichtlich ☐ ☐ ☐ ☐ ☐ sehr übersichtlich

2.2 Sind ausreichend Informationen zu fehlgeschlagenen Tests vorhanden?

☐ Ja ☐ Nein

2.3 Sind zusätzliche Informationen angemessen erreichbar?

☐ Ja ☐ Nein

2.4 Welche Informationen fehlen oder sollten besser aufbereitet werden?

Klicken Sie hier, um Text einzugeben.

2.5 Welche Probleme gab es bei der Darstellung der Ergebnisse?

Klicken Sie hier, um Text einzugeben.

3 ALLGEMEIN

3.1 Welche Testfälle können nicht abgebildet werden? Was ist der Grund?

Klicken Sie hier, um Text einzugeben.

3.2 Welche Probleme sehen Sie beim produktiven Einsatz des Systems?

Klicken Sie hier, um Text einzugeben.

Kevin Wenz (CR/AEG)

fixed-term.kevin.wenz@de.bosch.com

B. Glossar und Abkürzungsverzeichnis

Glossar

***Big-Bang*-Integration** Das gleichzeitige Integrieren aller Komponenten in einem Schritt.

Pull-Request Durch einen Pull-Request wird beantragt, dass Änderungen eines Feature-*Branches* in den Haupt-*Branch* gemergt werden.

Aufgabe Aufgabe, die bei einem Testfall erledigt werden soll.

Kontext Umgebung, in der sich der Roboter bewegt.

Metrik Messpunkt, an welchem eine Eigenschaft des SUT gemessen werden kann.

Szenario Aufgabe, Kontext und Metriken ergeben ein Szenario. Ein Szenario entspricht einem Testfall.

Abkürzungsverzeichnis

BSD	Berkeley Software Distribution.
OCL	Object Constraint Language.
PID	Prozess-Identifikator.
ROS	Robot Operating System.
SUT	System-Under-Test.
UML	Unified Modeling Language.
XML	Extensible Markup Language.

Literaturverzeichnis

[BNM08] B. N. Biswal, P. Nanda, D. P. Mohapatra. A Novel Approach for Scenario-Based Test Case Generation. In *2008 International Conference on Information Technology. ICIT 2008, Bhubaneswar, India, December 17-20, 2008*, S. 244–247. 2008. (Zitiert auf den Seiten 28 und 30)

[Boe84] B. W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, 1984. (Zitiert auf Seite 20)

[Bos15] Bosch in Zahlen, 2015. URL http://www.bosch.de/de/de/our_company_1/facts_and_figures_1/facts-and-figures.php. (Zitiert auf Seite 31)

[Cau13] C. Caum. Continuous Delivery vs. Continuous Deployment: What's the Diff?, 2013. URL https://puppetlabs.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff. (Zitiert auf Seite 15)

[CB11] K. C. Castillos, J. Botella. Scenario Based Test Generation Using Test Designer. In *Fourth IEEE International Conference on Software Testing, Verification and Validation. ICST 2012, Berlin, Germany, March 21-25, 2011, Workshop Proceedings*, S. 79–88. 2011. (Zitiert auf den Seiten 29 und 30)

[CH07] Y. K. Chung, S.-M. Hwang. Software testing for intelligent robots. In *International Conference on Control, Automation and Systems, 2007. ICCAS 2007, Portsmouth, UK, September 18-25, 2007*, S. 2344–2349. IEEE, 2007. (Zitiert auf Seite 30)

[Chu13] R. Chuck. The Facebook Release Process, 2013. URL http://www.infoq.com/presentations/Facebook-Release-Process. (Zitiert auf Seite 13)

[DCT12] F. Dadeau, K. C. Castillos, R. Tissot. Scenario-based testing using symbolic animation of B models. *Softw. Test., Verif. Reliab.*, 22(6):407–434, 2012. (Zitiert auf den Seiten 29 und 30)

[DMG07] P. M. Duvall, S. Matyas, A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Pearson Education, 2007. (Zitiert auf Seite 13)

[DT09] F. Dadeau, R. Tissot. jSynoPSys - A Scenario-Based Testing Tool based on the Symbolic Animation of B Machines. *Electr. Notes Theor. Comput. Sci.*, 253(2):117–132, 2009. (Zitiert auf Seite 29)

[FF06] M. Fowler, M. Foemmel. Continuous Integration. *Thought-Works*, 2006. (Zitiert auf Seite 16)

[FFB13] D. G. Feitelson, E. Frachtenberg, K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013. (Zitiert auf den Seiten 13, 28 und 30)

[Fit09] T. Fitz. Continuous Deployment at IMVU: Doing the impossible fifty times a day. Online, 09. URL http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/. (Zitiert auf den Seiten 28 und 30)

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. (Zitiert auf den Seiten 10, 13, 15, 16, 22, 36, 38 und 57)

[HVFR04] J. Hartmann, M. Vieira, H. Foster, A. Ruder. UML-based test generation and execution. *Präsentation auf der TAV21 in Berlin*, 2004. (Zitiert auf den Seiten 28 und 30)

[HVFR05] J. Hartmann, M. Vieira, H. Foster, A. Ruder. A UML-based approach to system testing. *ISSE*, 1(1):12–24, 2005. (Zitiert auf den Seiten 28 und 30)

[IEE87] IEEE Standards Board. IEEE Standard for Software Unit Testing. *IEEE Std. 1008-1987*, 1987. (Zitiert auf den Seiten 19 und 20)

[Jen11] J. Jenkins. Velocity Culture. Video auf YouTube, 2011. URL https://www.youtube.com/watch?v=dxk8b9rSKOo. (Zitiert auf Seite 13)

[Jen15] Jenkins-Wiki, 2015. URL https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins. (Zitiert auf Seite 17)

[KFJ+07] E. Krotkov, S. Fish, L. Jackel, B. McBride, M. Perschbacher, J. Pippine. The DARPA PerceptOR evaluation experiments. *Autonomous Robots*, 22(1):19–35, 2007. (Zitiert auf Seite 29)

[L+96] M. R. Lyu, et al. *Handbook of software reliability engineering*, Band 222. IEEE computer society press CA, 1996. (Zitiert auf Seite 22)

[LFB13] J. Laval, L. Fabresse, N. Bouraqadi. A methodology for testing mobile autonomous robots. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, S. 1842–1847. 2013. (Zitiert auf Seite 29)

[Lig09] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, 2009. (Zitiert auf den Seiten 18 und 19)

[Mau08] A. Maurya. Case Study: Continuous deployment makes releases non-events. Online, 2008. URL http://www.startuplessonslearned.com/2010/01/case-study-continuous-deployment-makes.html. (Zitiert auf Seite 27)

[Mil08] A. Miller. A Hundred Days of Continuous Integration. In *Agile Development Conference. AGILE 2008, Toronto, Canda, August 4-8, 2008*, S. 289–293. 2008. (Zitiert auf den Seiten 27 und 30)

[MSOM12] Z. Micskei, Z. Szatmári, J. Oláh, I. Majzik. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *Agent and Multi-Agent Systems. Technologies and Applications*, S. 504–513. Springer, 2012. (Zitiert auf den Seiten 28 und 30)

[NS13] S. Neely, S. Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Development Conference, (AGILE) 2013, Nashville, TN, USA, August 5-9 2013*, S. 121–128. IEEE, 2013. (Zitiert auf den Seiten 13, 16, 27 und 30)

[OAB12] H. H. Olsson, H. Alahyari, J. Bosch. Climbing theSStairway to Heaven-A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *Conference on Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO. Cesme, Izmir, Turkey, September 5-8, 2012*, S. 392–399. IEEE, 2012. (Zitiert auf Seite 27)

[PK12] H. S. Park, J. S. Kang. *SITAF: Simulation-Based Interface Testing Automation Framework for Robot Software Component.* INTECH Open Access Publisher, 2012. (Zitiert auf den Seiten 28 und 30)

[Pul13] V. Pulkkinen. Continuous Deployment of Software. In *Cloud-based Software Engineering – Proceedings from the Seminar No. 58312107, Helsinki, Finnland, August, 2013. University of Helsinki*, S. 46–51. 2013. (Zitiert auf Seite 14)

[Rat14] R. Ratnayake. The Long Road to Continuous Delivery. Online, 2014. URL https://blog.inf.ed.ac.uk/sapm/2014/02/14/the-long-road-to-continuous-delivery/. (Zitiert auf den Seiten 27 und 30)

[ROS14] ROS-Wiki, 2014. URL http://wiki.ros.org/ROS/Introduction. (Zitiert auf den Seiten 23 und 25)

[SH05] E. F. Sorton, S. Hammaker. Simulated flight testing of an autonmous unmanned aerial vehicel using flightgear. *American Institute of Aeronautics and Austronautics, AIAA 2005*, 7083, 2005. (Zitiert auf Seite 30)

[Sho06] J. Shore. Continuous Integration on a Dollar a Day. Online, 2006. URL http://www.jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html. (Zitiert auf Seite 28)

[SKPK11] J.-R. Son, T.-Y. Kuc, J.-K. Park, H.-S. Kim. Simulation based functional and performance evaluation of robot components and modules. In *International Conference on Information Science and Applications, (ICISA) 2011, Jeju Island, South Korea, April 26-29, 2011*, S. 1–7. IEEE, 2011. (Zitiert auf den Seiten 29 und 30)

[SL12] A. Spillner, T. Linz. *Basiswissen Softwaretest.* dpunkt.Verlag GmbH, 5. Auflage, 2012. (Zitiert auf Seite 21)

[SM07] M. Sarma, R. Mall. Automatic Test Case Generation from UML Models. In *10th International Conference on Information Technology, ICIT 2007, Roukela, India, 17-20 December 2007*, S. 196–201. 2007. (Zitiert auf den Seiten 29 und 30)

[STC13] Y. Sun, G. Tao, G. X. H. Chen. The Fuzzy-AHP evaluation method for unmanned ground vehicles. *Applied Mathematics & Information Sciences*, 7(2):653–658, 2013. (Zitiert auf Seite 30)

[SYT10] M. Shimizu, S. Yotsukura, T. Takahashi. Proposal of a simulation platform to evaluate rescue robots in active disaster environment. In *Proceedings of SICE Annual Conference 2010, SICE 2010, Taipei, Taiwan, August 18-21, 2010*, S. 863–866. IEEE, 2010. (Zitiert auf Seite 29)

[TBPY01] W.-T. Tsai, X. Bai, R. Paul, L. Yu. Scenario-based functional regression testing. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. Chicago, Illinois USA, October 8-12, 2001 25th Annual International*, S. 496–501. IEEE, 2001. (Zitiert auf Seite 29)

[TNP+02] W. Tsai, Y. Na, R. A. Paul, F. Lu, A. Saimi. Adaptive Scenario-Based Object-Oriented Test Frameworks for Testing Embedded Systems. In *26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, Oxford, England, August 26-29, 2002, Proceedings*, S. 321–326. 2002. (Zitiert auf Seite 29)

[TYL+03] W. Tsai, L. Yu, X. Liu, A. Saimi, Y. Xiao. Scenario-based test case generation for state-based embedded systems. In *Conference Proceedings of the 2003 IEEE International Performance, Computing, and Communications Conference, IPCCC 2003. Phoenix, Arizona USA, April 9-11, 2003*, S. 335–342. IEEE, 2003. (Zitiert auf Seite 29)

[WM01] J. Wittevrongel, F. Maurer. SCENTOR: Scenario-Based Testing of E-Business Applications. In *10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2001), 20-22 June 2001, Cambridge, MA, USA*, S. 41–48. 2001. (Zitiert auf den Seiten 28 und 30)

[ZLZS14] C. Zhang, Y. Liu, D. Zhao, Y. Su. RoadView: A traffic scene simulator for autonomous vehicle simulation testing. In *Intelligent Transportation Systems, ITSC 2014, Qingdao, China, October 8-11, 2014 IEEE 17th International Conference on*, S. 1160–1165. IEEE, 2014. (Zitiert auf den Seiten 29 und 30)

Alle URLs wurden zuletzt am 13. 07. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift