Duale Hochschule Baden-Württemberg
Mannheim

Projektarbeit 1

Optimierung des Release-Prozesses bei SAP TwoGo mithilfe von Continuous Delivery

Studiengang Wirtschaftsinformatik Software Engineering

Bearbeitungszeitraum: 08.08.2013 bis 08.11.2013

Verfasser	Johannes Haaß
Matrikelnummer	4101368
Kurs	WISE12B
Studiengangsleiter	Prof. Dr. Thomas Holey
Ausbildungsfirma	SAP AG Dietmar-Hopp-Allee 16 Walldorf
Firmenbetreuer	Dirk Lehmann dirk.lehmann@sap.com
Wissenschaftlicher Betreuer	Tobias Kißmer tobias.kissmer@schaeffler.com

Abstract

Mitfahrgelegenheiten erleben momentan eine starke Nachfrage. Besonders kleine Unternehmen und Start-ups haben diesen Trend erkannt und entsprechende Cloud-Lösungen auf den Markt gebracht. Durch schlanke Strukturen und flache Hierarchien können diese sehr schnell neue Funktionen an den Kunden bringen. Große Unternehmen, wie beispielsweise SAP, können durch ihre Größe dieses Tempo nicht mitgehen und sind damit im Bereich der Cloud-Lösungen im Nachteil.

Im Rahmen dieser Projektarbeit werden Optimierungen durchgeführt, um diesen Nachteil auszugleichen. Dazu wird der Release-Prozess der SAP Mitfahrlösung TwoGo analysiert und den zentralen Merkmalen von Continuous Delivery gegenübergestellt. Aus dieser Gegenüberstellung werden potentielle Optimierungsmöglichkeiten für den Release-Prozess von SAP TwoGo abgeleitet. Diese betreffen sowohl technische als auch organisatorische Aspekte.

Eine konkrete Optimierungsmöglichkeit ist der Deployment-Test. Dieser überprüft den Deployment-Vorgang und erkennt dabei Fehler, Warnungen und Erfolge. Zur Umsetzung dieses Deployment-Tests wird der Deployment-Vorgang analysiert, ein Entwurf erstellt und dieser implementiert und getestet.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CD	Continuous Delivery
Bash	Bourne-Again-Shell
CI	Continuous Integration
VCS	Version Control System
WTS	Windows Terminal Server
DMZ	demilitarisierte Zone
SSH	Secure Shell
KISS	Keep it simple and stupid
WAR	Web Application Archive
API	Application-Programming-Interface
JSON	JavaScript Object Notation
PAP	Programmablaufplan
RPC	Remote Procedure Call

Abbildungsverzeichnis

1 Einleitung

1.1 Problemstellung und -abgrenzung

Mit zunehmenden Kraftstoffkosten und immer überfüllteren Straßen steigt die Nachfrage bezüglich Mitfahrgelegenheiten kontinuierlich an. Diese steigende Nachfrage wurde von einigen, wie beispielsweise den Gründer von *www.mitfahrgelegenheit.de* , erkannt, die in der Folge Start-ups gegründet haben.[1] Auch SAP erkannte diesen Trend und bietet seinen Kunden mit TwoGo eine Fahrgemeinschaftslösung, die speziell an Unternehmen gerichtet ist.
Durch die geringe Mitarbeiteranzahl und die flache Hierarchie können Start-ups dem Kunden sehr schnell neue Funktionen einer Software zur Verfügung stellen.[2] Dies ist bei SAP durch die Größe des Unternehmens, die Unternehmensstruktur, interne Richtlinien und durch die Historie von SAP weitaus zeitaufwendiger. In der Vergangenheit war SAP hauptsächlich mit *on Premise* Lösungen, wie zum Beispiel R3 oder der Business Suite, erfolgreich. Zwar lässt sich dem Geschäftsbericht von 2012[3] entnehmen, dass mittlerweile auch vermehrt Cloud-Lösungen, zu denen auch TwoGo zählt, verkauft werden, dennoch führen die oben genannten Faktoren dazu, dass der Release einer Software bei SAP momentan nicht die Schnelligkeit erfüllt, welche bei Cloud-Produkten benötigt wird. Dementsprechend befindet sich SAP in diesem Bereich gegenüber Start-ups im Nachteil. Diese Projektarbeit soll zu der Aufhebung dieses Nachteils beitragen.
Dazu beschäftigt sich diese Projektarbeit mit der Einführung von Continuous Delivery in Verbindung mit Continuous Integration zur Optimierung des Release-Prozesses von SAP TwoGo.

1 (Hicks, 2012)
2 Vgl. (Swartout, 2012, Seite 7ff)
3 (SAP AG, 2013, Seite 100f)

1.2 Ziel der Arbeit und Vorgehensweise

Im Rahmen dieser Projektarbeit soll der Release-Prozess für SAP TwoGo optimiert werden. Dazu sollen potentielle Optimierungsmöglichkeiten im aktuellen Release-Prozess ermittelt werden. Eine dieser Möglichkeiten wird ein automatischer Deployment-Test sein, welcher entwickelt, getestet und produktiv genutzt werden soll. Die Aufgabe des Deployment-Tests liegt darin zu überprüfen, ob die Binärdateien nach dem Kompilieren korrekt auf den Server transportiert wurden und ob dieser nach dem Transport noch fehlerfrei arbeitet.

Um Optimierungsmöglichkeiten zu finden, werden zunächst der aktuelle Release-Prozess sowie die TwoGo Infrastruktur näher beleuchtet. Im folgenden Kapitel werden die Prinzipien von CD dargestellt und erläutert, woraus sich dessen Vor- und Nachteile ermitteln lassen. Nach der Darstellung dieser Themen wird es durch einen Vergleich des aktuellen Release-Prozesses mit den Prinzipien von CD möglich sein, Optimierungsmöglichkeiten festzustellen.
Bei der Umsetzung des Deployment-Tests im praktischen Teil dieser Projektarbeit wird zunächst analysiert, welche Merkmale getestet werden können. Daraus wird ein Lösungsansatz generiert und implementiert. Um die korrekte Funktion zu gewährleisten, wird ein Testvorgang durchgeführt.

2 Grundlagen / Methodischer Ansatz

2.1 SAP TwoGo

In zahlreichen Gesellschaftsbereichen spielen Nachhaltigkeit und Umweltschutz aktuell eine bedeutende Rolle. Auch Unternehmen haben dies erkannt und möchten verstärkt umweltschonend handeln. Eine ganz konkrete Thematik ist hierbei unter anderem der Arbeitsweg der Mitarbeiter, da ein Großteil oft weite Strecken alleine zurücklegt, um zur Arbeit zu gelangen.[1] Laut Schindler pendeln in Deutschland jeden Tag 32 Millionen Arbeitnehmer zur Arbeit. Daraus ergeben sich Kosten in Höhe von 700 Millionen Euro pro Tag.[2] Außerdem geht damit einher, dass das Staurisiko steigt und die Kraftstoffkosten für den Einzelnen sehr hoch sind.
An dieser Stelle setzt TwoGo by SAP an.[3] TwoGo stellt den Mitarbeitern eine Plattform zur Bildung von Fahrgemeinschaften zur Verfügung. Dadurch lassen sich zum einen die schädlichen Treibhausemissionen senken und zum anderen können die Treibstoffkosten reduziert werden. Diese Vorteile sind sowohl für den Arbeitnehmer positiv einzustufen, da sein Geldbeutel geschont wird, als auch für den Arbeitgeber, da seine Umweltbilanz verbessert wird, die Kosten für die Wartung seiner Firmenwagenflotte verringert werden und der Wiederverkaufswert der Firmenwagen steigt.
Aber nicht nur aus wirtschaftlicher Sicht bringt TwoGo Vorteile, auch der Mensch profitiert, da er bei gemeinsamen Fahrten zur Arbeit neue Kollegen kennenlernen kann. Zudem kann auch die Beziehung zwischen dem Unternehmen und dem einzelnen Mitarbeiter unter anderem in Form von zusätzlichen Leistungen für TwoGo-Nutzer gestärkt werden. Diese Leistungen könnten zum Beispiel besonders gut gelegene Parkplätze sein, die nur von Fahrgemeinschaften genutzt werden dürfen. Schon diese Argumente zeigen deutlich, welche Vorteile eine

[1] Vgl. (SAP, 2013)
[2] (Schindler, 2013)
[3] Vgl. (SAP, 2013)

solche Plattform mit sich bringen kann.
Damit diese Plattform optimal eingesetzt werden kann, muss eine ortsunabhängige und unternehmensübergreifende Nutzung, am besten im Form einer Cloud, gewährleistet werden. Um diese zwei wichtigen Punkte sicherzustellen, wird TwoGo in SAP Rechenzentren gehostet. Dies gewährleistet außerdem ein hohes Maß an Sicherheit und eine durchgehende Erreichbarkeit von TwoGo. Damit auch der Endanwender von der Cloud profitiert und auch jederzeit auf TwoGo zugreifen kann, gibt es insgesamt zwei Wege, TwoGo zu nutzen: entweder über einen PC/Laptop oder über ein Smartphone. Am PC lässt sich auf TwoGo über den Internet Browser oder Microsoft Outlook zugreifen und am Smartphone über die mobile Internetseite oder die native App. Somit ist der Zugriff immer gewährleistet und der Anwender kann zu jeder Zeit eine Mitfahrgelegenheit suchen oder anlegen.

2.2 Entwicklungslandschaft

Um TwoGo regelmäßig und relativ zügig mit neuen Funktionen und Bugfixes zu versorgen[1], wird bei der Entwicklung Scrum genutzt. Dies ermöglicht es, Softwareentwicklung nach den Grundsätzen der agilen Softwareentwicklung durchzuführen. Zur Unterstützung des Scrum Prozesses wird als Project Management Tool *Trac* eingesetzt. Es ermöglicht das zentrale Speichern von Dokumentation, Stories, Fehlern und Aufgaben.
Die Entwickler programmieren das TwoGo Backend mithilfe der Entwicklungsumgebung *Eclipse* und nutzen Java als Programmiersprache. Für die Oberfläche wird JavaScript und HTML eingesetzt. Zur Datenhaltung wird eine *SAP MaxDB* Datenbank genutzt. Damit die einzelnen Veränderungen in den Quelldateien jederzeit nachverfolgt und gegebenenfalls zurückgenommen werden können, wird ein Version Control System genutzt. Momentan werden hierfür *Perforce* und *git* eingesetzt. Aus den Quelldateien werden mithilfe von *Maven* Binärdateien erzeugt, welche danach in einer WAR-Datei[2] gepackt und auf einem zentralen Server, dem sogenannten *Nexus*, abgelegt werden.
Um kontinuierliches Kompilieren von Quelldateien, automatisiertes Testen und

[1] Vgl. (TwoGo, 2013)
[2] Vgl. (Danny und Yutaka, 2003, Seite 329)

Ausrollen von Binärdaten zu ermöglichen, wird *Jenkins* als Continuous Integration System verwendet. Dabei nutzt *Jenkins* die bereits erwähnten Komponenten *VCS*, *Maven* und den *Nexus*. Das Zusammenwirken dieser wird mithilfe Abbildung 2.1 dargestellt. Die Funktion der Komponenten und deren Zweck werden ausführlich in Kapitel 3 erläutert.

2.3 Aktueller Deploymentprozess

In der Entwicklung von SAP TwoGo gibt es drei Landschaften (Abbildung 2.1): eine *Dev-Landschaft* (Entwicklungslandschaft), eine *Patch-Landschaft* (Testlandschaft) und eine *Cons-Landschaft* (Produktivlandschaft).[1] Diese Landschaften haben jeweils drei Server: einen Datenbankserver und zwei Backend-Server (Matcher, Matchfinder). Durch die Verwendung mehrerer Landschaften wird gewährleistet, dass Entwickler, Tester und Anwender ein eigenständiges System haben.

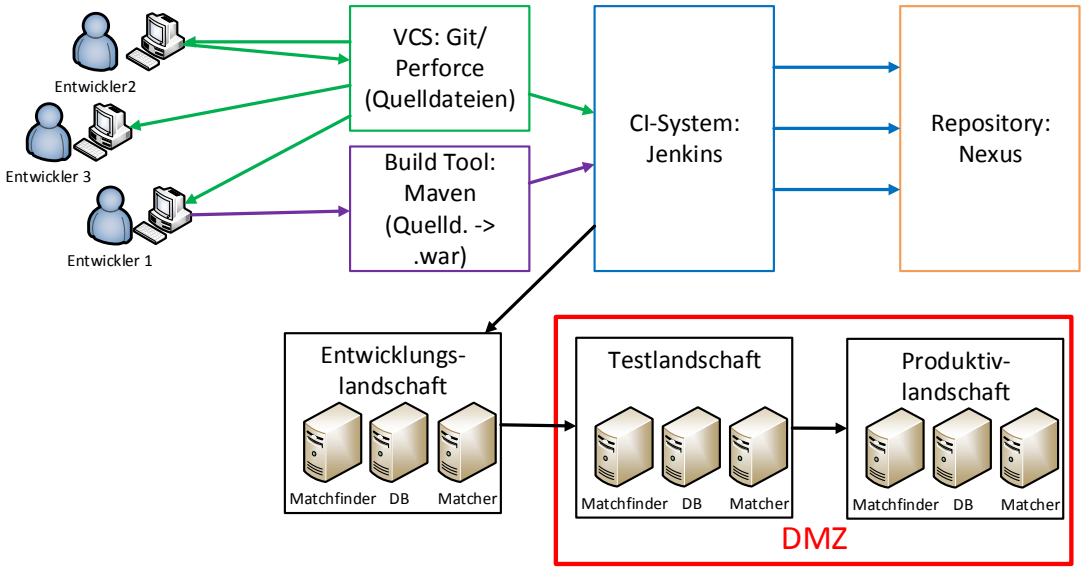


Abbildung 2.1: TwoGo Server

Soll nun eine bestimmte Version von TwoGo von der Entwicklungslandschaft auf die Produktivlandschaft oder Testlandschaft transportiert werden, spricht man von Deployment. Diesen Deploymentprozss kann man bei SAP TwoGo aus zwei

[1] Vgl. (TwoGo, 2013)

Perspektiven betrachten, aus der technischen sowie der organisatorischen Sicht.
Organisatorisch ist der Deploymentprozess stark an die unternehmensinterne Struktur und an interne Regelungen von SAP gebunden. Zunächst wird ein *Quality Gate* Meeting durchgeführt. Dabei wir anhand von festgelegten Qualitätskriterien beschlossen, ob ein neuer Release stattfindet und wenn ja, welche Features ausgerollt werden sollen. Daraufhin müssen weitere Abteilungen durchlaufen werden. Eine dieser Abteilungen prüft beispielsweise die verwendeten open-source Bibliotheken, während eine andere den gesamten Quellcode auf potentielle Sicherheitslücken überprüft. Dies führt dazu, dass der Deploymentprozess sehr zeitaufwendig ist, wodurch ein neuer Release momentan nur alle drei Wochen möglich ist.
Auf der technischen Seite müssen zwei unterschiedliche Fälle unterschieden werden. Möchte der Entwickler etwas auf die Entwicklungslandschaft ausbringen, so kann er dies mithilfe von Jenkins (siehe Kapitel 2.3 und 3.2) über ein Deployment Skript vollautomatisch ablaufen lassen.
Bei den anderen beiden Landschaften (Test und Produktiv) ist dies komplizierter. Bereits das Zugreifen auf die Server ist nicht einfach, da sich diese innerhalb einer Sicherheitszone, der sogenannten DMZ (demilitarisierte Zone), befinden. Zugriff auf diese Landschaften ist nur über WTS und unter Einhaltung strenger Sicherheitsrichtlinien manuell möglich. Hat man diese Barriere passiert, müssen die frisch kompilierten Binärdateien mithilfe eines Deployment Skriptes deployed werden. Dabei kommt es allerdings manchmal zu Komplikationen, wenn Dateien in einer falschen Version vorliegen, woraufhin diese manuell geändert und kopiert werden müssen. Da die Binärdateien bei jedem Deployment und für jede Landschaft neu kompiliert werden, entstehen überflüssige Redundanzen. Ein weiteres Problem sind die verwendeten open-source Bibliotheken, welche zentral auf dem Nexus gespeichert werden, weil diese gelegentlich in einer falschen Version vorliegen. Dadurch können Folgefehler beim Kompilieren entstehen. All diese Probleme führen dazu, dass sich das Deployment auch aus technischer Sicht in die Länge zieht.
Insgesamt verdeutlichen diese beiden Perspektiven die Zeitbeanspruchung und die Fehleranfälligkeit, welche ein schnelleres Deployment verhindern.

2.4 Unix

Die Server in der TwoGo Entwicklungslandschaft nutzen sowohl Unix-basierte Betriebssysteme als auch Microsoft Windows. Betrachtet man zudem die heutige Verteilung von Betriebssystemen[1], so erkennt man, dass nur diese zwei Arten von Betriebssystemen auf dem Markt verbreitet sind.
Die bekanntesten Vertreter von Unix-Betriebssystemen sind Mac OS X von Apple, Linux, FreeBSD sowie Android von Google. Schon an diesen vier Beispielen lässt sich das Einsatzspektrum von Unix-Betriebssystemen gut abschätzen, welches beim Server beginnt und beim Smartphone endet. Ihre Gemeinsamkeit besteht darin, dass alle die grundlegenden Konzepte von Unix nutzen. Zu den wichtigen Konzepten von Unix gehören zum einen die Schnittstellen und zum anderen das Dateisystem.
Unix lässt sich in fünf Schichten unterteilen[2], wodurch ein modularer Aufbau gewährleistet wird, der es auch ermöglicht, einzelne Schichten auszutauschen. Abbildung 2.2 zeigt diese Schichten. Sie folgen einem logischen Aufbau, der mit der Hardware beginnt und mit dem Benutzer endet. Elementare Funktionen, wie die Prozessverwaltung, sind vom Benutzer entkoppelt. Auf die einzelnen Funktionen der Schichten kann über Schnittstellen, wie beispielsweise die Systemaufrufschnittstelle, zugegriffen werden. Somit fungieren die Schnittstellen als Verbindung zwischen den einzelnen Schichten.

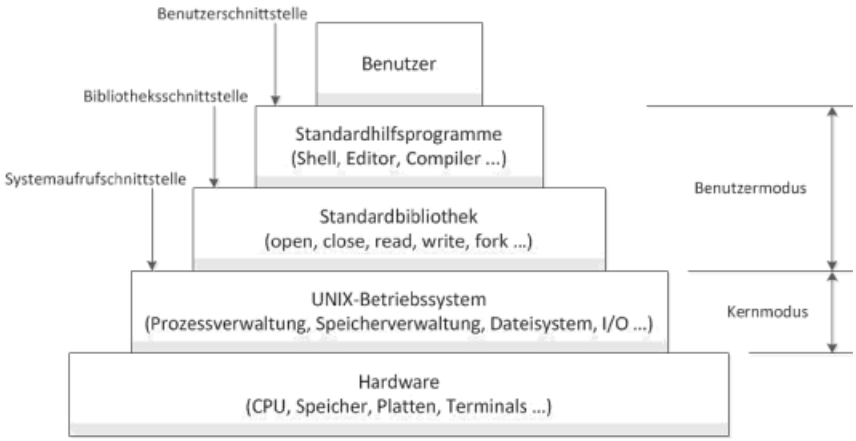


Abbildung 2.2: Unix Schichtenmodell
(Ehses et al., 2011, Seite 12)

1 (Fittkau und Maaß, 2011)
2 Vgl. (Ehses, Köhler, Riemer, Stenzel und Victor, 2011, Seite 11ff)

Die zweite Besonderheit von Unix-Betriebssystemen liegt, wie oben genannt, in der Dateistruktur.[1] Diese ist baumartig aufgebaut und gruppiert Dateien nach ihrer Art und Zugehörigkeit. Hierdurch ist die Struktur unabhängig von der Hardware, das heißt, es gibt keine Laufwerke wie man es von Windows kennt. Die grundlegende Dateistruktur ist in der folgenden Grafik 2.3 dargestellt:

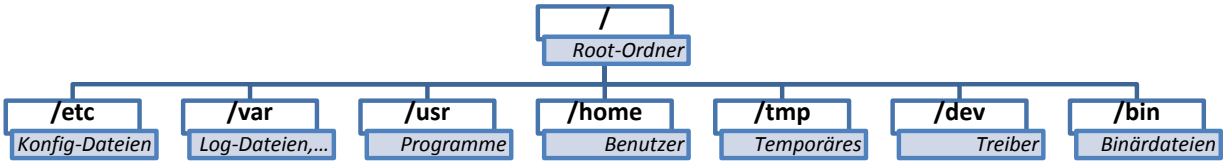


Abbildung 2.3: Unix Dateistruktur

2.5 Shell-Programmierung mit Bash

Die meisten Betriebssysteme nutzen grafische Oberflächen, um dem Anwender ein einfaches Arbeiten zu ermöglichen, es gibt jedoch Situationen, in denen grafische Oberflächen ineffektiv werden. Wenn man zum Beispiel den Dateinamen aller Dateien in einem Verzeichnis mit einer fortlaufender Nummer versehen will, ist dies über die grafische Oberfläche sehr umständlich. In solchen Situationen ist es effizienter, die Kommandozeile zu benutzen. Mit dieser lassen sich bestimmte Systembefehle aufrufen. Im eben genannten Beispiel benennt der Befehl *rename* Dateien oder Verzeichnisse um.
Aber auch die Kommandozeile hat den Nachteil, dass der Anwender immer den Befehl manuell eingeben muss. Bei einmaligen Tätigkeiten, wie dem Umbenennen von Dateien stellt dies noch kein Problem dar. Möchte man jedoch zum Beispiel bei Systemstart automatisch die neuesten Dateien von einem Server herunterladen, ist selbst die Kommandozeile nicht mehr effektiv. Nun hat man zwei Möglichkeiten. Entweder man schreibt in einer beliebigen Programmiersprache ein entsprechendes Programm oder man *automatisiert* die Kommandozeile. Letzteres nennt man *scripting* oder *Shell-Programmierung*. Man erstellt dabei ein sogenanntes *Shell Script*. Dies ist im Wesentlichen eine einfache Textdatei, welche mit Kommandozeilen-Befehlen gefüllt ist.

[1] Vgl. (Burtch, 2004, Seite 7ff)

Unter Unix gibt es verschiedene Shell-Arten[1], die sich hauptsächlich in ihrer Syntax und der Befehlsvielfalt unterscheiden. Trotz einer Vielzahl von Shell-Arten sind viele von ihnen sehr ähnlich. Die wohl am meisten genutzte Shell ist Bash. Diese funktioniert sowohl unter Linux als auch unter OS X und ist für jeden frei zugänglich (open-source). In der Fachliteratur wird Bash oft auch als *Quasi-Standard* bezeichnet.
Um ein Shell-Skript zu schreiben[2], benötigt man nur einen Texteditor, welcher in der Regel bei jedem Betriebssystem bereits vorinstalliert ist. Damit das System die Datei richtig interpretieren kann, muss in der ersten Zeile des Skriptes lediglich das *She-Bang (#!)* in Verbindung mit der verwendeten Shell-Art stehen. Bei Bash sieht dies folgendermaßen aus: *#!/bin/bash* . Damit sind die Grundlagen für ein Shell-Skript gegeben und man kann somit auch die weiteren Vorteile von Shell-Skripten nutzen. So bietet Bash die Möglichkeit, Variablen, Bedingungen, Schleifen und Unterprogramme zu verwenden.
Es folgt nun ein Skript, das automatisch Daten von einem Server herunterlädt und in den entsprechenden Ordner verschiebt. An diesem Beispiel kann man gut erkennen, wie ein Bash-Skript aufgebaut ist. In diesem Beispiel sind Befehle fett markiert und Kommentare werden mit einem # eingeleitet.

```
1  #!/bin/bash
2  #Laedt die neusten *.war Dateien herunter
3  ########################################
4  #Variablen deklarieren
5  declare  TEMP_LOC=/tmp/warfiles   #Tmp Verzeichnis
6  declare  TGT_LOC=/home/deploy   #Zielort
7  #Tmp Ordner erstellen
8  mkdir -p $TEMP_LOC
9  cd $TEMP_LOC            # Verzeichnis wechseln
10 #Download
11 wget -q "http://server:8010/attachment/kraftwerk.war"
12 wget -q "http://server:0380/attachment/web.war"
13
14 mv -u kraftwerk.war $TGT_LOC    #Datei 1 verschieben
15 mv -u web.war $TGT_LOC          #Datei 2 verschieben
16 rm -fr $TEMP_LOC                #Tmp Verzeichnis loeschen
17 kill 0                            #Skript beenden
```

[1] Vgl. (Burtch, 2004, Seite 4)
[2] Vgl. (Wolf, 2010, Seite 23)

3 Continuous Delivery

3.1 Einleitung

Dieses Kapitel beschäftigt sich mit den Grundlagen von Continuous Delivery. Zum Ende dieses Kapitels werden Argumente für und gegen CD dargestellt und ein Vergleich mit der momentanen Situation bei TwoGo durchgeführt.
Softwareentwicklung ist ein komplexer und mehrschichtiger Prozess (Abbildung 3.1). In der Regel beginnt alles mit einer guten Idee, gefolgt von der Entwicklung, dem Testen der Software und schließlich dem Liefern (eng.: deliver) der Software an den Kunden.

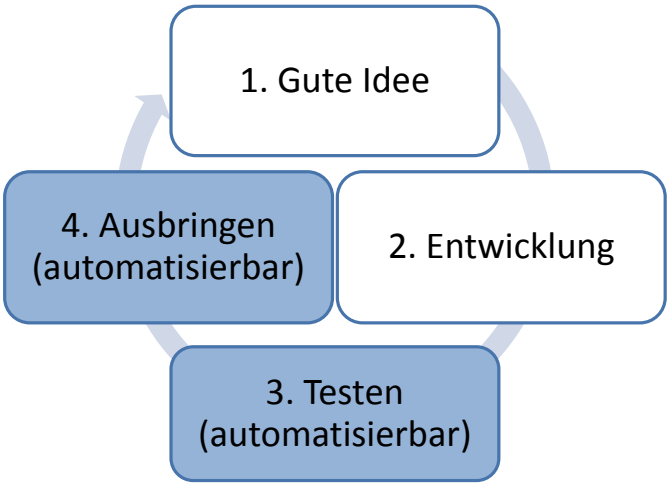


Abbildung 3.1: Entwicklungsprozess
(Swartout, 2012, Seite 9 (angelehnt))

Für ein Softwareunternehmen ist der letzte Teil der wichtigste, denn nur wenn die Software beim Kunden produktiv eingesetzt werden kann, wird Gewinn generiert. Daraus lässt sich schließen, dass bei diesem Prozess hohe Effektivität und Geschwindigkeit eine entscheidende Rolle spielen. Dieses Ziel lässt sich durch den Einsatz von CD erreichen.[1]

[1] Vgl. (Swartout, 2012, Kapitel 1)

CD beschreibt die Optimierung dieses Prozesses und geht dabei sowohl auf technische als auch auf organisatorische Aspekte ein. Ein zentrales Merkmal von CD ist es, Teile des Prozesses zu automatisieren und sich auf wichtige Elemente, wie das Programmieren, zu fokussieren. Der Grund für diese Überlegung ist einfach: Wieso sollte man Aufgaben, die sehr zeitaufwendig und komplex sind, manuell ausüben, wenn man diese auch automatisieren kann?
In vielen Teams, wie dem SAP TwoGo-Team[1], wird beispielsweise das Produkt händisch ausgebracht (eng.: deploy). Dies ist ein komplexer und aufwendiger Prozess, bei dem es leicht zu menschlichen Fehlern kommen kann. Durch die Automatisierung können diese Fehler reduziert werden und die überschüssige Arbeitskraft in wichtigeren Bereichen genutzt werden. Es wird hiermit erkennbar, wo CD ansetzt und welche Vorteile es bietet.[2]
Durch die vollständige Umsetzung von CD soll es letztendlich möglich sein, eine beliebige Version der Software mit dem Klicken eines Buttons zu einem beliebigen Zeitpunkt auszuliefern. Dabei soll auch mehrmals täglich ausgeliefert werden können, weshalb Schnelligkeit und Effektivität entscheidende Merkmale sind.[3]
Dieser Prozess wird in der Fachliteratur auch als *deployment pipeline* beschrieben (zum Beispiel (Humble und Farley, 2011)).

3.2 Deployment Pipeline

Die Deployment Pipeline, dargestellt in Abbildung 3.2, beschreibt einen automatisierten Prozess, welcher mit dem Freischalten einer Änderung (eng.: *commit*) eines Entwicklers in das VCS beginnt und mit dem Release der Software endet. Dazwischen erfolgen noch viele weitere Schritte, wie das Kompilieren und das Testen. Diese sind notwendig um sicherzustellen, dass der Quellcode fehlerfrei ist und produktiv eingesetzt werden kann.[4]

1 siehe Kapitel 2.3
2 Vgl. (Humble und Farley, 2011, Seite 3ff)
3 Vgl. (Humble und Farley, 2011, Seite 5)
4 Vgl. (Humble und Farley, 2011, Kapitel 5)

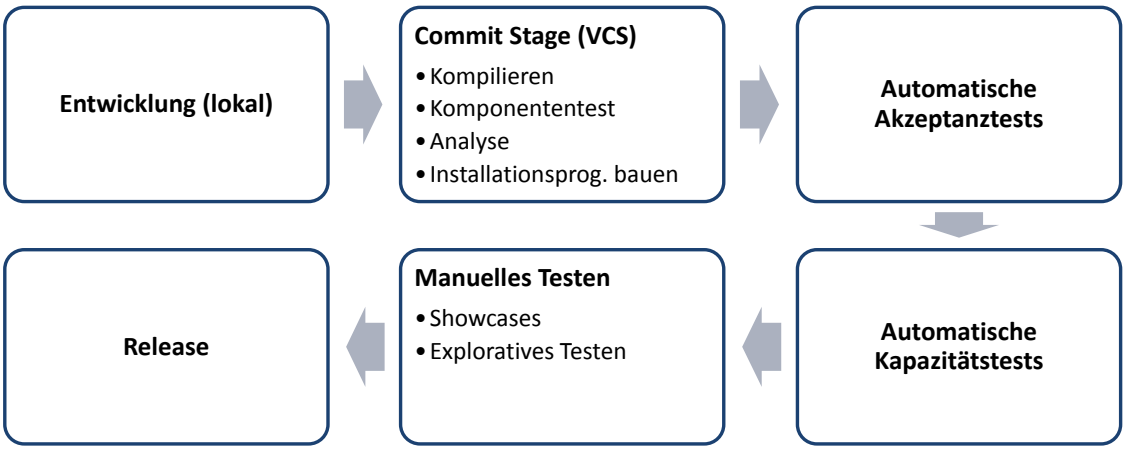


Abbildung 3.2: Deployment Pipeline
(Humble und Farley, 2011, Seite 4 (angepasst))

Ein zentrales Element der Deployment Pipeline ist der Auslösezeitpunkt. Nach jedem Commit im VCS wird diese Pipeline von neuem ausgelöst. Dadurch wird es möglich, dass jede Änderung in den Quelldateien potentiell ausgebracht werden kann. Um einen guten Überblick zu gewährleisten, stellt jeder Durchgang der Deployment Pipeline eine neue Version der Software dar. Um die Sicherheit zu erhöhen, bleibt sowohl von den Quelldateien als auch den Binärdateien jede Version im VCS bzw. dem zentralen Repository gespeichert. Die dadurch entstehende Historie gewährleistet, dass zu jedem Zeitpunkt eine Wiederherstellung einer älteren Version durchgeführt werden kann.
Für die Realisierung der Deployment Pipeline werden bestimmte Tools, wie das bereits mehrfach genannte VCS, Skripte, Tests und organisatorische Umstrukturierungen benötigt. Diese werden in den Unterpunkten dieses Kapitels genauer erläutert.

3.2.1 Continuous Integration

CI (deutsch: Kontinuierliche Integration) ist ein essentieller Bestandteil der Deployment Pipeline, weil es einen großen Teil der Aufgaben übernimmt. Nach (Fowler, 2006) wird CI folgendermaßen definiert (übersetzt aus dem englischen Original[1]):

[1] Vgl. (Wiest, 2011, Seite 13f)

> »*Die Kontinuierliche Integration ist eine Softwareentwicklungspraktik, bei der Teammitglieder ihre Arbeit häufig integrieren. Üblicherweise integriert jede Person im Team mindestens einmal täglich was zu mehreren Integrationen am Tag führt. Jede Integration wird durch einen vollautomatischen Build (und Test) geprüft, um Fehler so schnell wie möglich aufzudecken.*«

Damit diese Definition eines CI-Systems gilt, müssen folgende Anforderungen erfüllt werden, wofür wiederum bestimmte Tools bzw. Systeme erforderlich sind. Die Anforderungen sind[1]:

- *Gemeinsame Codebasis*
  Der gesamte Quellcode und alle weiteren Dateien, die Bestandteil der Software sind, werden an einem zentralen Ort gespeichert. Dazu wird in der Regel ein VCS benutzt. In einem solchen System erhält jede Datei eine Versionsnummer und mit jeder Änderung wird diese erhöht. Die Besonderheit an einem VCS ist dabei, dass man zu jeder beliebigen Version zurückspringen kann. Dadurch kann Datenverlust verhindert werden und jeder Mitarbeiter kann immer auf den aktuellen Quellcode zurückgreifen. Bekannte VCS Tools sind *Git* oder *Perforce*.

- *Vollautomatischer Build*
  Das CI soll aus den Quelldateien, welche es über das VCS bekommt, vollautomatisch und bei jeder Änderung neue Binärdateien erzeugen. Dazu ruft es ein System/Tool auf, welches aus den Quelldateien Binärdateien kompiliert. Dazu können zum Beispiel *Maven* oder *Ant* eingesetzt werden. Die erzeugten Binärdateien werden danach auf einem zentralen Repository gespeichert. Dieses Repository kann Bestandteil des CI Systems sein oder befindet sich auf einem externen Server wie einem *Nexus*. Für die weiteren Schritte ist es dadurch nicht mehr notwendig, Quelldateien neu zu kompilieren, da diese Zugriff auf das Repository haben.

- *Automatische Tests*
  Laut Definition sollen mithilfe von CI Fehler so schnell wie möglich gefunden werden. Deshalb ist es notwendig, dass das CI System auch automatische und produktnahe Tests durchführen kann. Um möglichst viele Fehler zu finden, werden unterschiedliche Arten von Tests durchgeführt, dazu gehören

[1] Vgl. (Fowler, 2006) und Vgl. (Wiest, 2011, Seite 15f)

Quellcodetests oder Akzeptanztests. Die Testarten werden im nächsten Kapitel näher vorgestellt. Auch bei den Tests nutzt das CI System verschiedene Tools wie beispielsweise *Selenium*.

- *Feedback*
  Durch das kontinuierliche Bauen und Testen bekommt der Entwickler sehr schnell Feedback über seine Veränderungen am Quellcode. Um auf Probleme beim Bauen hinzuweisen, werden oft Ampeln oder das Abspielen von Geräuschen verwendet. Da die Veränderungen aufgrund des regelmäßigen Integrierens der Entwickler nur kleine Änderungen darstellen, kann er den Fehler schneller verstehen und beheben.

Vergleicht man diese Punkte mit den Prozessschritten der Deployment Pipeline, so erkennt man, dass bereits viele Punkte, wie das automatische Kompilieren und Testen, durch ein CI System erfüllt werden. Mit vielen CI Systemen, wie beispielsweise *Jenkins*, ist es zudem möglich durch die korrekte Konfiguration fast alle Schritte der Deployment Pipeline abzudecken.

3.2.2 Testarten

Ein weiterer wichtiger Bestandteil der Deployment Pipeline liegt in der automatischen oder manuellen Durchführung von Tests. Um dabei möglichst alle Fehler zu finden, ist es notwendig, mehrere Testarten einzusetzen. In der agilen Softwareentwicklung, welche bei CD sehr nützlich ist (siehe Kapitel 3.4.2), werden die Testarten vier unterschiedlichen Kategorien zugeordnet. Diese sind in der Matrix in Abbildung 3.3 dargestellt und werden nun näher erläutert.[1]

[1] Vgl. (Humble und Farley, 2011, Kapitel 4)

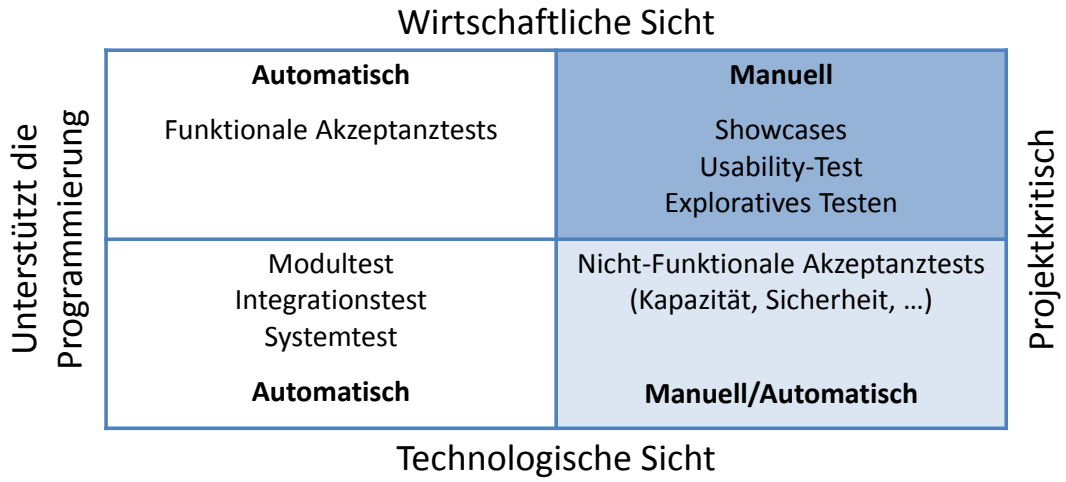


Abbildung 3.3: Testarten
(Humble und Farley, 2011, Grafik 4.1 (übersetzt))

Mit dieser Darstellung wird sichergestellt, dass möglichst alle Aspekte eines Produktes getestet werden. Generell lassen sich Tests, die Programmierer unterstützen, automatisieren und solche, die das Produkt an sich betreffen, meistens nicht automatisieren.
Aus wirtschaftlicher Sicht ist es essentiell, dass alle Anwendungsfälle (eng.: Use Cases) erfüllt werden. Als User möchte ich beispielsweise, dass sich Fenster y öffnet, wenn ich auf den Knopf x drücke. Dies kann man mit sogenannten automatischen Akzeptanztests testen. Verläuft dieser Test fehlerfrei, weiß der Entwickler, ob sein Code diesen Use Case erfüllt. Es gibt jedoch Aspekte eines Produktes, die man nicht automatisch testen kann, wie beispielsweise das Design oder der Bedienkomfort (eng.: Usability), da diese sehr subjektiv und vom Endbenutzer abhängig sind. In diesem Bereich ist somit manuelles Testen unvermeidbar.
Bei der technischen Sicht geht es darum, dass der Code generell funktioniert und spezielle Richtlinien erfüllt. Dazu gibt es automatische Modul- und Integrationstests. Diese überprüfen einzelne bzw. mehrere Codeabschnitte, wie beispielsweise eine bzw. mehrere Java Klassen und melden so dem Entwickler eventuelle Fehler. Während die Software deployed und installiert wird, werden automatische Systemtests, zum Beispiel ein Deployment-Test, ausgeführt. Diese gewährleisten, dass das Produkt richtig installiert und konfiguriert ist und überprüfen außerdem, ob die Systemumgebung korrekt funktioniert. Auch aus technischer Sicht gibt es Aspekte, die nicht oder nur teilweise automatisch getestet werden können. Dazu gehören beispielsweise Kapazität, Verfügbarkeit und Sicherheit.

All diese Tests stellen sicher, dass der Entwickler schnell Feedback erhält und dass der Großteil des Quellcodes durch automatische Tests überprüft wird. Dies ist notwendig, denn jede Änderung kann nach CD auf dem Produktivsystem deployed werden, in welchem keine Fehler auftreten sollen.

3.2.3 Automatisches Deployment

Der letzte Schritt in der Deployment Pipeline ist das automatische Deployment. Dieses wird vom Entwickler manuell über einen Knopfdruck gestartet, nachdem er eine Funktion vollständig programmiert und getestet hat. Die Aufgabe des automatischen Deployments ist es, die Binärdateien vom zentralen Repository auf die entsprechenden Server zu kopieren, die Software zu installieren und je nach Serverkonfiguration entsprechend zu konfigurieren. Zunächst einmal ist dazu das Wissen notwendig, auf welchen Systemen die Software deployed wird. Bei kleinen Projekten kann dies der Server sein, auf dem auch das CI System läuft. Bei größeren Projekten sind dies oft mehrere Server, die auf verschiedenen Landschaften, wie beispielsweise Entwicklung, Test und Produktion, verteilt sind. Um Fehleranfälligkeit und Rechenzeit zu vermeiden ist es empfehlenswert, immer dieselben Binärdateien, unabhängig von ihrem Zielserver, zu deployen. Dies kann erreicht werden, indem diese Server in ihrer Struktur und Konfiguration gleich sind oder die Binärdateien unabhängig von diesen Merkmalen kompiliert werden. Für letzteres ist es notwendig, auf den einzelnen Servern entsprechende Konfigurationsdateien zu pflegen.[1]
Um automatisches Deployment umzusetzen, gibt es unterschiedliche Wege. Eine Möglichkeit besteht darin, selbst programmierte Skripte zu nutzen. Mit diesen ist es beispielsweise möglich, über Netzwerkprotokolle, wie SSH, auf die Zielserver zuzugreifen. Diese Skripte lassen sich entweder manuell aufrufen oder werden im CI System aufgerufen. Letzteres hat den Vorteil, dass die Deployment Pipeline komplett in einem System abgedeckt werden kann. Einen weiteren Weg stellen Deployment Tools wie *ControlTier* oder *BMC BladeLogic* da, welche in Verbindung mit Infrastruktur Management Tools wie *Puppet* oder *CfEngine* eingesetzt werden.[2]
Auf welche Weise man das automatische Deployment realisiert, hängt von vielen Faktoren ab. Wichtig dabei ist, dass ein schnelles Deployment erreicht wird,

[1] Vgl. (Humble und Farley, 2011, Seite 134f) und vgl. (Swartout, 2012, Seite 61)
[2] Vgl. (Humble und Farley, 2011, Seite 160ff)

um mehrmaliges deployen am Tag zu ermöglichen. Nachdem das Deployment ausgeführt wurde, sollte noch ein Deployment-Test durchgeführt werden, um sicherzustellen, dass der gesamte Vorgang korrekt und fehlerfrei abgelaufen ist.[1]

3.2.4 Organisatorische Umstrukturierungen

Nachdem in den vorherigen Unterpunkten primär die technischen Aspekte beleuchtet wurden, sind bei der Deployment Pipeline auch organisatorische Aspekte sehr wichtig. Denn nicht nur die technischen Anforderungen müssen erfüllt sein, sondern auch die Mitarbeiter müssen ihr Arbeitsverhalten anpassen. Grundsätzlich muss das Team in der Lage sein, sehr schnell neue Features zu entwickeln, damit die Deployment Pipeline sehr oft durchlaufen werden kann. Um dies zu erreichen, bietet sich die Agile Softwareentwicklung an, da hier der Fokus auf wichtige Elemente, wie das Programmieren, gelegt wird und nach dem KISS (Keep it simple and stupid) Prinzip gehandelt wird.[2]
In der Fachliteratur wird dabei vor allem *Kanban* als Agiler Prozess bei CD verwendet (zum Beispiel (Humble und Farley, 2011) und (Swartout, 2012)).
Kanban ähnelt stark dem Scrum Prozess, welcher mittlerweile bei vielen IT Unternehmen, wie auch SAP, Standard ist. Bei beiden Prozessen werden die einzelnen Funktionen einer Software in viele kleine Aufgaben aufgeteilt, damit diese schneller entwickelt werden können. Bei Scrum gibt es feste Sprint-Zyklen, welche dazu führen, dass die einzelnen Funktionen erst am Ende des Sprints fertig sind. Daraus ergibt sich, dass ein Deployment immer nur am Ende eines Sprints möglich ist. Bei Kanban gibt es diese Einschränkung nicht, das heißt eine neue Funktion kann unmittelbar nach Fertigstellung deployed werden[3]. Damit ermöglicht Kanban, dass die Deployment Pipeline bei jedem abgeschlossenen neuen Teil von neuem angestoßen werden kann.

1 Vgl. (Humble und Farley, 2011, Seite 163)
2 Vgl. (Swartout, 2012, Kapitel 2)
3 Vgl. (Kniberg und Skarin, 2010, Seite 50)

3.3 Bewertung von Continuous Delivery

Kapitel 3.2 hat verdeutlicht, dass viele technische Eingriffe und organisatorische Umstrukturierungen notwendig sind, um ein erfolgreiches CD umzusetzen. Dementsprechend hat CD neben vielen Vorteilen auch einige Nachteile, die im Folgenden dargestellt werden.
Zu den Nachteilen gehört primär der hohe Aufwand CD umzusetzen. Zum einen müssen viele Programme bzw. Systeme, wie das CI System oder das VCS Tool, installiert und eingerichtet werden. Bis diese alle Spezifikationen erfüllen und fehlerfrei laufen, können mehrere Wochen vergehen. Zudem sind für diese Programme/Systeme eventuell auch Softwarelizenzen oder Hardware, wie zum Beispiel Server, erforderlich. Es entsteht somit neben einem hohen zeitlichen Aufwand auch ein finanzieller Aufwand. Auch die manuellen und automatischen Tests müssen erstellt, beziehungsweise modifiziert werden und die Skripte für das automatische Deployment müssen geschrieben werden, damit alle Schritte der Deployment Pipeline abgedeckt sind. All dies zeigt, dass viele technische Veränderungen notwendig sind. Dabei muss gewährleistet sein, dass alle Elemente fehlerfrei laufen, damit die eigentliche Entwicklung der Software reibungslos ablaufen kann. Auch aus Team-Sicht entstehen Nachteile. Das Team muss unter Umständen auf einen völlig neuen Entwicklungsprozess umsteigen. Solche radikalen Veränderungen führen am Anfang dazu, dass vieles langsamer und ineffektiver abläuft. Im schlimmsten Fall verzögern einige Teammitglieder diese Veränderung, weil sie gegen eine Umsetzung von CD sind oder bei gewohnten Vorgehensweisen bleiben möchten.
Es gibt jedoch auch viele positive Aspekte. Nachdem CD vollständig umgesetzt ist, können neue Funktionen innerhalb von wenigen Minuten deployed werden. Dies führt dazu, dass der Anwender sehr schnell von diesen profitieren kann. Damit verbunden ist auch, dass finanzielle Forderungen früher beglichen werden. Zugleich kann es auch ein Vorteil gegenüber der Konkurrenz sein, da man Kundenwünsche in kurzer Zeit umsetzen kann. Auch während der Entwicklung entstehen Vorteile, weil diese gegenüber klassischen Entwicklungsmodellen durch das Wegfallen von Wartezeiten viel effektiver ist. Neue Entwicklungen können damit unmittelbar nach Fertigstellung deployed werden. Zudem profitieren die Entwickler von CD, da sie sehr schnell Feedback über ihre Veränderungen im Code bekommen. Es ist für sie dann leichter, den Fehler nachzuvollziehen und

zu beheben, da die Veränderungen immer recht gering sind. Durch das Trennen der Binärdaten von den Konfigurationsdaten wird zudem sichergestellt, dass der Code auf jedem weiteren System läuft. Sollte es bei diesen schnellen Deployments trotz diverser Tests zu Problemen kommen oder ein Kunde wünscht sich eine ältere Version, ist es durch das zentrale Repository möglich, in kürzester Zeit eine ältere Version zu deployen.
Insgesamt betrachtet überwiegen für mich die Vorteile, da die Nachteile keine Rolle mehr spielen, sobald CD komplett umgesetzt ist. Natürlich muss man, gerade während der Umsetzung, auch die Nachteile beachten und muss mit einer gewissen Einführungszeit rechnen, bis alles korrekt funktioniert. Nach dieser Phase hat man aber die Möglichkeit, neue Funktionen sehr schnell zu deployen und überflüssige Wartezeiten werden vermieden.

3.4 Vergleich

In den Kapiteln 3.1 bis 3.3 wurden die Merkmale von CD ausführlich dargestellt, erläutert und bewertet. Kapitel 2.2 stellte die verwendeten Systeme von TwoGo vor und Kapitel 2.3 beleuchtete den aktuelle Deployment Prozess von TwoGo. Dieses Kapitel vergleicht diese Themen nun, um zu ermitteln, welche Schritte bei TwoGo noch notwendig sind, um CD vollständig umzusetzen.
Wie man Kapitel 2.2 entnehmen kann, nutzt TwoGo bereits viele Programme, die für CD notwendig sind. Das Team nutzt ein zentrales VCS System, ein Tool zum Kompilieren der Quelldateien und ein zentrales Repository, auf welchem die Binärdaten gespeichert werden. Um diese Tools zu verbinden, wird ein CI System eingesetzt. In diesem sind ebenfalls automatische Akzeptanz-, Modul- und Integrationstests vorhanden, welche bei jedem Build-Vorgang aufgerufen werden. Das Deployment auf die Entwicklungs-, Test- bzw. Produktivlandschaft wird mithilfe eines Skripts bewerkstelligt. Dabei ist die Konfiguration der Test- und Produktivlandschaft identisch. Für das Feedback an die Entwickler werden bei Fehlschlag oder Erfolg Sounddateien abgespielt.
Anhand dieser Punkte zeigt sich, dass alle notwendigen Systeme für ein funktionierendes CD vorhanden sind und auch schon vieles für ein erfolgreiches CD getan wurde.
Andererseits fehlen immer noch einige Elemente und es gibt unternehmensbeding-

te Hindernisse. Zu letzteren gehören alle in Kapitel 2.3 beschriebenen internen Regelungen, die den Deploymentprozess verzögern und somit CD momentan grundsätzlich verhindern. Zudem ist auch die organisatorische Struktur innerhalb des Teams nicht gänzlich auf CD ausgerichtet. Um dies zu ändern, müsste der Softwareentwicklungsprozess von Scrum auf Kanban geändert werden. Da beides sehr ähnlich ist, sollte eine Umstellung kein größeres Problem darstellen. Mit dieser Veränderung dürfen die manuellen Tests nicht mehr, wie bisher, vom gesamten Team ausgeführt werden, sondern sollten von den einzelnen Entwicklern nach Fertigstellung ihrer Funktion selbst durchgeführt werden.
Auf der technischen Seite sind ebenfalls Probleme vorhanden. So dürfen die Binärdateien nicht mehrmals gebaut werden. Dieses Problem kann gelöst werden, indem die Konfigurationsdateien, welche Bestandteil der Binärdateien sind, auf die Server ausgelagert werden. Damit die verwendeten open-source Bibliotheken immer in der richtigen Version verfügbar sind, müssen diese korrekt verwaltet werden, wie beispielsweise über den zentralen SAP Nexus.
Das vollautomatische Deployment auf den Entwicklungsserver ist bereits möglich, aber auf die Server innerhalb der DMZ ist dies aufgrund der Sicherheitsrichtlinien nicht möglich. Der Zugriff müsste entweder automatisiert ablaufen oder die Sicherheitsrichtlinien müssten reduziert werden, um den Zugriff zu vereinfachen. Zudem fehlt dem Deploymentskript momentan ein zentrales Element: der Deployment-Test. Durch dessen Fehlen ist es nur durch manuelles Überprüfen möglich festzustellen, ob das Deployment erfolgreich war.

4 Praktischer Teil: Deployment-Test

4.1 Analyse und Anforderungen

Der Deployment-Test wurde bereits in Kapitel 3 mehrfach kurz erwähnt. Dieser Test soll unmittelbar nach dem Deployment ablaufen. Die primäre Aufgabe des Deployment-Tests ist es, Fehler während des Deployment Vorganges zu finden und sicherzustellen, dass der Server danach weiterhin fehlerfrei läuft. Da bei der Entwicklung von SAP TwoGo ein Deployment Skript eingesetzt wird, ist es notwendig zu analysieren, welche Schritte dieses Skript abarbeitet. Aus dieser Analyse lassen sich daraufhin die Anforderungen an den Deployment-Test festlegen. Der Programmablaufplan in Abbildung 4.1 zeigt den groben Ablauf des Deployment Skripts.

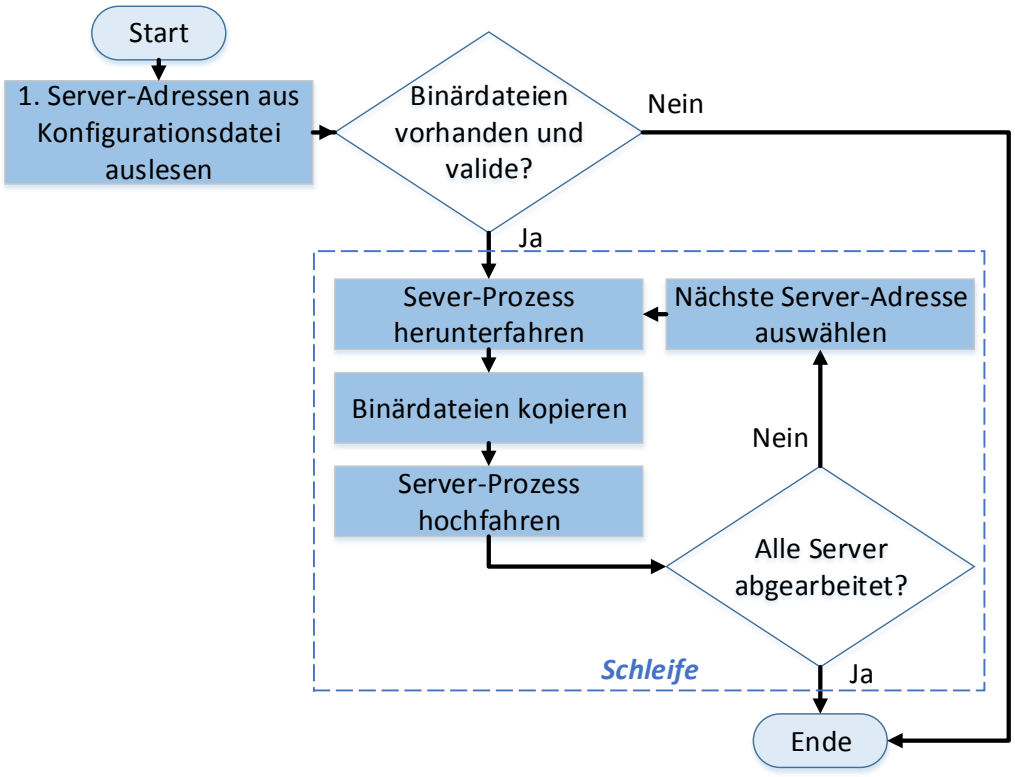


Abbildung 4.1: Ablauf des Deployment Skriptes

Zunächst liest das Skript bei Start eine *.landscape* Konfigurationsdatei. Diese enthält alle Serveradressen einer einzelnen Landschaft, wie zum Beispiel die Adressen des Datenbankservers, Backend-Servers und Matchfinder-Servers der Produktivlandschaft. Im nächsten Schritt wird überprüft, ob die Binärdateien vorhanden sind. Danach werden diese temporär entpackt. Damit wird sichergestellt, dass keine Dateifehler vorhanden sind und das Entpacken später fehlerfrei abläuft. Nach diesen Sicherungsschritten beginnt das eigentliche Deployment. Mithilfe einer Schleife werden die einzelnen Server der gewählten Landschaft nacheinander abgearbeitet. In der Schleife wird zunächst der Server-Prozess heruntergefahren, damit im Anschluss daran die Binärdateien kopiert werden können. Im letzten Schritt der Schleife wird der Server-Prozess wieder hochgefahren. Während des Hochfahrens wird eine Log-Datei erstellt. In dieser ist unter anderem gespeichert, ob alle Teilsysteme von TwoGo, wie die Oberfläche oder das Administrationstool, korrekt gestartet worden sind und welchen Inhalt bestimmte Umgebungsvariablen haben. Abbildung 4.2 zeigt einen Ausschnitt der Log-Datei.

```
584 + KEY: user.timezone                                    VALUE: Europe/Berlin
585 + KEY: velocimacro.arguments.strict                              VALUE: true
586 + KEY: virusscanner.enabled                                 VALUE: true
587
588 2013-09-09 07:27:44,699 INFO  [Start Level Event Dispatcher]: +------------------------------------------------------------------------
589 2013-09-09 07:27:44,721 INFO  [Start Level Event Dispatcher]: starting subsystem [com.sap.twogo.boot.impl.SSLBypassBootstrapEntry]
590 2013-09-09 07:27:44,748 INFO  [Start Level Event Dispatcher]: started subsystem [com.sap.twogo.boot.impl.SSLBypassBootstrapEntry]
591 2013-09-09 07:27:44,766 INFO  [Start Level Event Dispatcher]: starting subsystem [com.sap.twogo.boot.impl.RideMatcherBootstrapEntry]
592 2013-09-09 07:27:44,957 INFO  [Start Level Event Dispatcher]: started subsystem [com.sap.twogo.boot.impl.RideMatcherBootstrapEntry]
593 2013-09-09 07:27:44,958 INFO  [Start Level Event Dispatcher]: starting subsystem [com.sap.twogo.scheduler.SchedulerService]
594 2013-09-09 07:28:06,805 ERROR [Start Level Event Dispatcher]: Error while reading contact whitelist file. Proceeding without whitelist
```

Abbildung 4.2: Ausschnitt aus Log-Datei

Nach dem Hochfahren wird überprüft, ob alle Server der Konfigurationsdatei abgearbeitet worden sind. Falls dies noch nicht zutrifft, werden die gleichen Schritte für den nächsten Server durchgeführt. Ansonsten wird das Deployment Skript beendet.
Anhand dieser Analyse lässt sich erkennen, an welchen Stellen Fehler entstehen könnten, die nicht überprüft und gemeldet werden. Zum einen kann das Problem auftreten, dass die Server-Prozesse nicht richtig hochgefahren werden. Diese Fehler könnten durch unterschiedliche Parameter verursacht werden. Man müsste somit überprüfen, ob die Server-Prozesse nach dem Hochfahren korrekt laufen. Weiterhin könnten in den Log-Dateien Fehler oder Warnungen protokolliert worden sein. Diese sollten gefunden werden. Beim Hochfahren der Server-Prozesse könnte es ebenfalls dazu kommen, dass die Konnektivität der Server-Prozesse

nicht funktioniert oder eingeschränkt ist. Diese sollte ebenfalls überprüft werden. Daraus lassen sich folgende Anforderungen an den Deployment-Test festlegen:

- Läuft der Server-Prozess?
- Enthält die Log-Datei Fehler?
- Enthält die Log-Datei Warnungen?
- Ist der Server erreichbar?

4.2 Lösungsansatz

Durch die Analyse konnte ein Lösungsansatz abgleitet werden. Zunächst stellte sich die Frage, zu welchem Zeitpunkt der Deployment-Test aufgerufen werden soll. Da dieser unmittelbar nach dem Deploymentskript ausgeführt werden soll (vgl. Kapitel 3.2.3), ist es logisch, den Test am Ende des Skriptes automatisch aufzurufen. Der Erfolg des Deployment-Tests, und damit verbunden des gesamten Deployments, kann über eine neue Log-Datei ermittelt werden, welche das Resultat jeder einzelnen Überprüfung enthält. Diese Überprüfungen entsprechen den Anforderungen aus Kapitel 4.1. Ihre Abarbeitung wird in dem PAP in Abbildung 4.3 dargestellt.

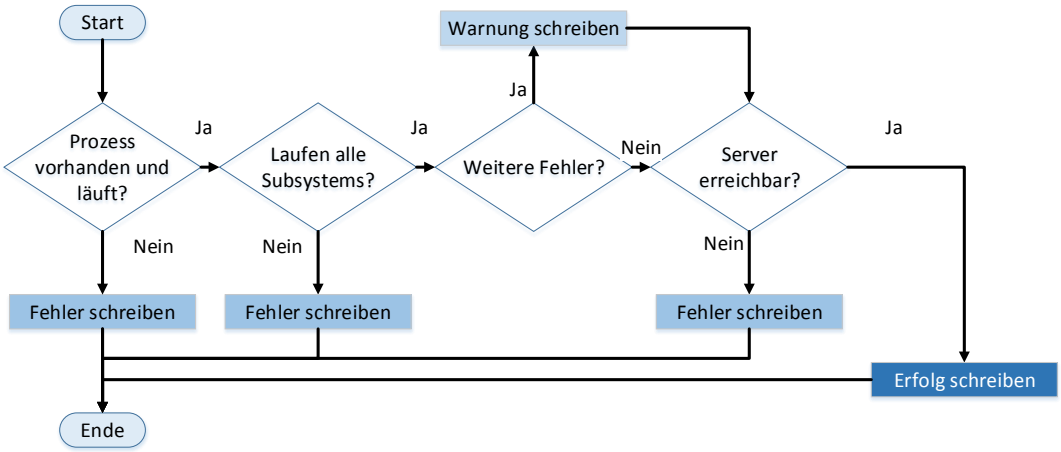


Abbildung 4.3: Ablauf des Deployment Tests

1. *Läuft der Server-Prozess?* Da der Server-Prozess eine notwendige Voraussetzung für den Betrieb von TwoGo ist, wird im ersten Schritt geprüft, ob

dieser Prozess läuft. Sollte dies nicht der Fall sein ist es nicht mehr nötig, die folgenden Schritte auszuführen und der Test kann beendet werden.

2. *Enthält die Log-Datei Fehler?* Um Fehler innerhalb der Log-Datei zu entdecken, muss diese zunächst einmal existieren. Trifft dies zu, muss der Deployment-Test sechs mal die Zeile *started subsystem* finden. Dieser Wert ergibt sich aus der Anzahl der Komponenten von TwoGo. Sollten nicht alle Systeme laufen, kann der Test ebenfalls abgebrochen werden, da TwoGo sonst nicht fehlerfrei funktionieren kann.

3. *Enthält die Log-Datei Warnungen?* Zwischen den Meldungen *started subsystem* könnten weitere Fehler auftreten, die Punkt 2 nicht entdeckt. Dazu wird überprüft, ob alle Meldungen innerhalb von 14 Zeilen vorhanden sind. Dieser Wert wurde durch Versuche ermittelt, deren Ergebnis war, dass alle Fehlermeldungen wesentlich länger als 14 Zeilen sind. Da diese Fehler nicht zwangsläufig den Betrieb von TwoGo stören, wird nur eine Warnung gespeichert und darauf hingewiesen, dass die Log-Datei des Prozesses manuell überprüft werden sollte.

4. *Ist der Server erreichbar?* Die letzte Überprüfung soll gewährleisten, dass der Server auch von außerhalb erreichbar ist. Um dies zuverlässig sicherzustellen, wird ein JSON Objekt an den Server gesendet. Dieser JSON-RPC ist Bestandteil der TwoGo API. Sollte das JSON Objekt nicht ankommen oder die falsche Antwort vom Server zurückkommen wird eine Fehlermeldung generiert.

Nach dem Durchlauf des Deployment-Tests wird dem Nutzer, der das Deployment Skript gestartet hat, die aus diesen Überprüfungen entstandene Log-Datei angezeigt.

4.3 Implementierung

Damit der Deployment-Test einfach und effektiv im Deployment Skript aufgerufen werden kann, sollte der Test in der selben Sprache geschrieben werden wie das Deployment Skript. Dieses ist ein Shell Skript und ist in Bash geschrieben.

Somit ist die Frage, mit welcher Technologie das Skript realisiert wird, bereits beantwortet. Durch die Verwendung von Bash wird außerdem ein einfacher Zugriff auf Betriebssystem-Befehle und Dateiverwaltung gewährleistet.
Um den PAP in Abbildung 4.3 am einfachsten zu implementieren, wurden die vier Überprüfungen mithilfe von Funktionen realisiert. Durch diesen Schritt wird das Skript sehr übersichtlich und es entstehen drei Bereiche. Im ersten Teil werden Variablen deklariert, welche beispielsweise Dateipfade enthalten. Die Funktionen sind im zweiten Teil gespeichert und im dritten Teil ist das eigentliche Hauptprogramm, welches die Funktionen aufruft und die Log-Datei schreibt. Die Reihenfolge dieser Teile sind bei Bash notwendig, um die Funktionalität zu gewährleisten. Dies beruht darauf, dass die Funktionen die Variablen nutzen und vom Hauptprogramm aufgerufen werden. Diese Teilung ermöglicht es außerdem, spätere Änderungen leicht vorzunehmen. Daraus ergibt sich folgendes Hauptprogramm:

```
1  check_PID $#
2    if [ $? -eq 1 ]; then
3    echo "11||Process is running" >> $dLog_file
4    else
5    echo "00||Process is NOT running" >> $dLog_file
6    exit 0
7    fi
8  check_log $*
9    if [ $? -eq 1 ]; then
10   echo "11||Logfile is ok" >> $dLog_file
11   else
12   echo "00||Not all subsystems have been started" >> $dLog_file
13   exit 0
14   fi
15 check_warning $*
16   if [ $? -eq 1 ]; then
17   echo "11||No warnings" >> $dLog_file
18   else
19   echo "10||Please check the log manually" >> $dLog_file
20   fi
21 check_JSON $*
22   if [ $? -eq 1 ]; then
23   echo "11||Connected to server" >> $dLog_file
24   else
25   echo "00||Problem with server connection" >> $dLog_file
```

```
26  fi
```

Der Aufruf des Deployment-Tests wird, wie bereits erwähnt, vom Deployment Skript ausgeführt. Um zu erreichen, dass der Test auch auf den jeweiligen Servern läuft, wird er über den Befehl *ssh* aufgerufen. Um zu gewährleisten, dass der Anwender, der das Skript aufruft, auch direkt die Fehlermeldungen aus dem Log des Deployment-Test erhält, werden diese im Deployment Skript ausgelesen. Dazu wird eine Schleife benutzt, welche jede Zeile der Log-Datei liest, herausfindet um welche Art von Fehler es sich handelt und die entsprechende Nachricht schreibt.

```
1  while read line
2   do
3        if [[ $line == *00* ]] ; then     # 00 == Error
4           text=${line#*||}
5           writeError "$text"
6        elif [[ $line == *11* ]]; then    # 11 == Info
7           text=${line#*||}
8           writeInfo "$text"
9        else
10          text=${line#*||}
11          writeInfo "$text"        # else == Warning
12       fi
13 done < deployCheck.log
```

Durch die Integration in das Deployment Skript ist die Implementation der Deployment-Tests abgeschlossen.

4.4 Test

Um die korrekte Funktion des Deployment-Tests sicherzustellen, muss dieser getestet werden. Um dabei nicht auf einen Produktivserver zuzugreifen und somit den Produktivbetrieb zu beeinflussen, wurden alle notwendigen Programme und Konfigurationen auf einer virtuellen Maschine mit Linux als Betriebssystem installiert. Um einen realitätsnahen Test durchzuführen, wurden die aktuellen Binärdateien des Nexus verwendet. Danach wurden diese auf der virtuellen Maschine deployed.
Das erste Resultat dieses Tests ergab, dass der Deployment Test bereits gestartet

wird, wenn der Server noch hochfährt. Dies führt dazu, dass der Test fehlschlägt. Es war somit notwendig, eine Veränderung vorzunehmen, welche sicherstellt, dass der Deployment-Test erst nach dem Hochfahren ausgeführt wird. Als die einfachste Lösung für dieses Problem hat sich eine Pause herausgestellt. Die Dauer dieser Pause wurde auf Basis mehrerer Testläufe auf 45 Sekunden festgelegt.
Ein weiteres Problem ergab sich beim Überprüfen der Konnektivität. Hier antwortete der Server nicht mit dem erwarteten JSON Objekt, sondern mit dem folgenden Objekt:


```
{
  "id": 1,
  "result": null,
  "error": {
    "messageId": "SERVICE_ARGUMENT_COUNT_MISMATCH",
    "text": "Unknown method signature echo(0) on service \"\/Echo\"",
    "jsonDetail": null
  }
```


Abbildung 4.4: Falsches JSON Objekt

Der Grund für dieses Verhalten ist, dass bei der Implementation einige Sicherheitsaspekte nicht beachtet wurden. Um ein für den Server gültiges JSON Objekt zu senden, muss dieses einen Security-Token und einen Cookie enthalten. Diese können durch das Senden eines anderen JSON Objektes angefordert werden. Danach kann man den Token und das Cookie dem eigentlichen JSON Objekt übergeben. Mit dieser Veränderung in der Implementierung des Deplyoment-Tests funktioniert diese Funktion korrekt und das richtige Objekt wird zurückgeliefert:


```
{
  "id": 1,
  "result": "server_test",
  "error": null
}
```


Abbildung 4.5: Korrektes JSON Objekt

Neben diesen zwei Fehlern gab es noch weitere kleine Fehler. Diese beruhten jedoch auf Syntaxfehlern und konnten einfach behoben werden. Mit der Überprüfung des Deployment-Tests und der damit verbunden Behebung der Fehler kann sichergestellt werden, dass der Deployment-Test die Anforderungen aus Kapitel 4.1 erfüllt und produktiv eingesetzt werden kann. Damit ist der Deployment-Test vollständig.

5 Zusammenfassung und Ausblick

Ziel dieser Projektarbeit war es, den Release-Prozess von SAP TwoGo zu optimieren. Primär sollte dabei der Faktor Zeit durch das Aufzeigen von Optimierungsmöglichkeiten und die Entwicklung eines Deployment-Tests optimiert werden.
Um dieses Ziel zu erreichen, wurde zunächst die Infrastruktur und der aktuelle Deployment Prozess von SAP TwoGo betrachtet. Dabei wurden sowohl die technischen als auch die organisatorischen Aspekte erörtert. Um die Entwicklung des Deployment-Tests zu ermöglichen, wurde Basiswissen über Unix-Systeme und Shell-Programmierung dargestellt. Zudem wurden die zentralen Merkmale von CD in Verbindung mit CI erläutert. Mit diesem Hintergrund konnten die Vor- und Nachteile von CD herausgearbeitet werden.
Aus diesen Themen konnte danach ein Vergleich zwischen CD und dem aktuellen Deploymentprozess durchgeführt werden. Das Resultat dieses Vergleiches war, dass bereits viele Aspekte von CD umgesetzt wurden, wie zum Beispiel die Verwendung eines CI Systems. Es haben sich jedoch auch noch fehlende Aspekte gezeigt, wie beispielsweise der Deployment-Test. Dessen Entwicklung wurde ausführlich in Kapitel 4 beschrieben.
Zusammenfassend lässt sich sagen, dass viele Optimierungsmöglichkeiten gefunden wurden und eine dieser Möglichkeiten konkret umgesetzt wurde. Damit bleiben jedoch noch viele weitere Optimierungsmöglichkeiten offen, wie beispielsweise dem einmaligen Kompilieren der Quelldateien. Daraus ergibt sich, dass momentan ein schnelles Deployment nach dem Prinzip von CD nicht möglich ist. Die Gründe hierfür sind primär organisatorischer Herkunft, da die internen Prozesse und Strukturen innerhalb der SAP ein schnelleres Deployment verhindern. Für die Zukunft und die damit verbundene vollständige Umsetzung von CD ist es notwendig, Veränderungen an diesen Prozessen und Strukturen vorzunehmen. Zudem müssen auch noch einige technische Veränderungen durchgeführt werden, wie dem einmaligen Bauen von Binärdateien.
In der Summe sind bereits viele Teile des CD Puzzles am richtigen Platz, fehlende Teile konnten identifiziert werden und eines dieser fehlenden Teile wurde mit dem Deployment-Test eingefügt.

Literaturverzeichnis

Burtch, K. O. (2004), *Linux shell scripting with Bash: a comprehensive guide and reference für Linux users and administrators*, Sams, Indianapolis, Ind.

Danny, C. und Yutaka, Y. (2003), 'Java™ servlet specification version 2.4', `http://download.oracle.com/otn-pub/jcp/servlet-2.4-fr-spec-oth-JSpec/servlet-2_4-fr-spec.pdf?AuthParam=1382621969_6c50eea3ae4a5471c72853feb0de8311`. Abgerufen am 24.10.2023.

Ehses, E., Köhler, L., Riemer, P., Stenzel, H. und Victor, F. (2011), *Systemprogrammierung in UNIX / Linux: Grundlegende Betriebssystemkonzepte und Praxisorientierte Anwendungen*, Springer DE.

Fittkau und Maaß (2011), 'Welches betriebssystem haben sie auf ihrem computer installiert?'. Umfrage in Anhang Abschnitt A.

Fowler, M. (2006), 'Continuous integration', `http://www.martinfowler.com/articles/continuousIntegration.html`. Abgerufen am 18.10.2013.

Hicks, J. (2012), 'Germany's carpooling.com proves rideshare works', `http://www.forbes.com/sites/jenniferhicks/2012/06/08/germanys-carpooling-com-proves-rideshare-works/`. Abgerufen am 18.10.2013.

Humble, J. und Farley, D. (2011), *Continuous delivery: reliable software releases through build, test, and deployment automation*, Addison-Wesley, Upper Saddle River, NJ [u.a.].

Kniberg, H. und Skarin, M. (2010), *Kanban and Scrum - Making the Most of Both*, Lulu Enterprises Incorporated.

SAP (2013), 'TwoGo by SAP fördert aktiv Nutzung von Fahrgemeinschaften und schafft neue Kommmunikationswege im Unternehmen', `http://www.sap.com/corporate-de/news.epx?pressid=20799`. Abgerufen am 18.10.2013.

SAP AG (2013), 'SAP Geschäftsbericht 2012', `http://www.sap.com/corporate-de/investors/pdf/SAP-2012-Geschaeftsbericht.pdf`. Abgerufen am 18.10.2013.

Schindler, M. (2013), 'SAP erleichtert Fahrgemeinschaften für Unternehmen', `http://www.silicon.de/41583318/sap-erleichtert-fahrgemeinschaften-fur-unternehmen/`. Abgerufen am 18.10.2013.

Swartout, P. (2012), *Continuous Delivery and DevOps a Quickstart guide.*, Packt Pub., Birmingham.

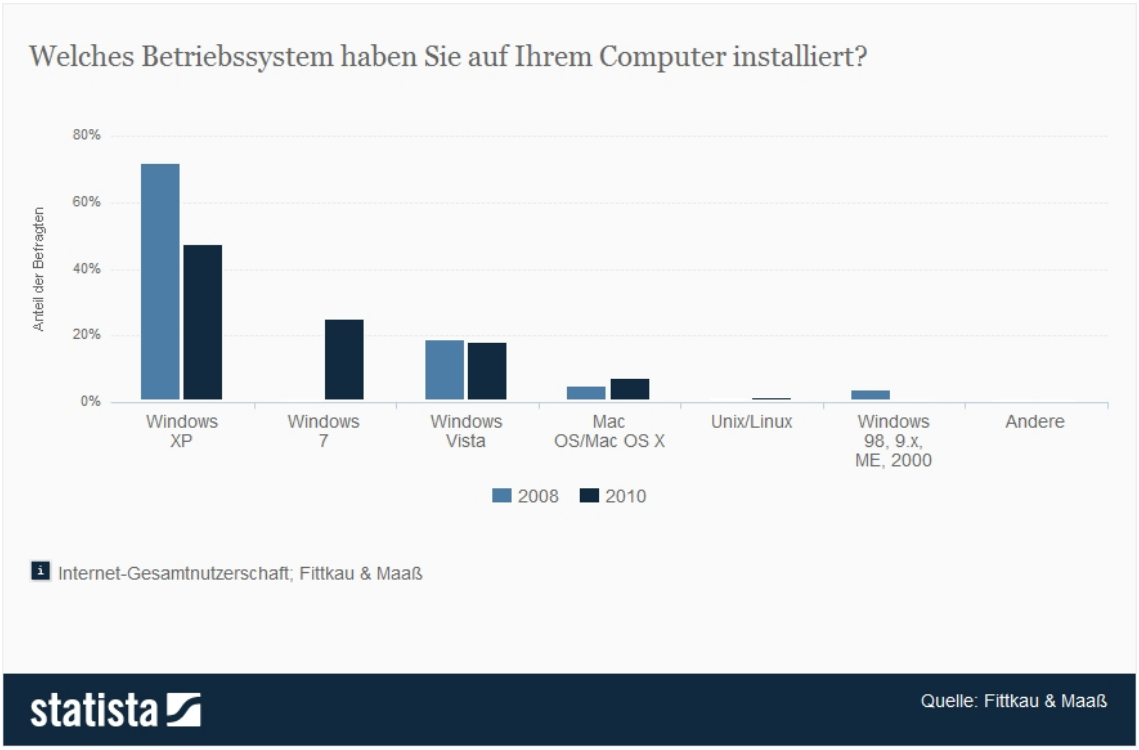
TwoGo (2013), 'Internes TwoGo Wikipedia'. Befindet sich in Anhang Abschnitt B.

Wiest, S. G. (2011), *Continuous Integration mit Hudson: Grundlagen und Praxiswissen für Einsteiger und Umsteiger*, dpunkt-Verl., Heidelberg.

Wolf, J. (2010), *Shell-Programmierung: [das umfassende Handbuch ; CD-ROM mit Openbooks zu Linux und C]*, Galileo Press, Bonn.

Anhang

Abschnitt A Umfrage:



Abschnitt B Ausschnitt aus internem Wiki:

Project Information

TwoGo is a project about ride sharing and is intended for SAP employees only in the first phase. Later it shall be opened to everybody. It's sponsored by the sustainability group at SAP and focuses on our CO2 reduction goals.

The project hosts an outside-facing wiki, but relevant technical or team-internal information is stored in this team-facing wiki. Tickets are managed here as well. We used to use JIRA, switched now to Trac for its better cross-integration with source code repositories, build server and more.

Table of Contents

Development Infrastructure

The project uses Trac as project management tool with its Wiki for internal documentation, its Issue Tracker for stories, defects features and tasks, Perforce and git as VCS, JDK of version 6, Maven as build tool, Nexus to host your build artifacts and the build dependencies you require, Jenkins for continuous build and test, and Eclipse as IDE (see CodingStandard) for JAVA development, respectively WebStorm for JavaScript development. See the above links for detailed installation instructions. Developers just need to install a JDK, Perforce, Maven, and Eclipse with plugins plus the local server environment (identical with our central build and dev system). It includes the application and database server. Developers will need to install it so they can execute tests, run a full Maven build (which in turn executes the tests), run the application from Eclipse, or control the application server or inspect the database content.
In order to work our Android client or our new UI/landing page (a.k.a ScriptStorm) please see additional here

Another alternative is to use a Linux VMWare Image which is prepared for the TwoGo developer. The VMWare Image contains configured Perforce Client, Eclipse IDE, Lean Java Server and MaxDB. Please see VMImage for details.

Testing is treated as integral part of the development process and infrastructure. The TwoGo team runs unit, integration, scenario and specific DSL-based tests for all developed components. In addition Findbugs and Checkstyle are used for static code analysis (limited test scope to highest priority, but for them all issues must be cleared) and Cobertura for code coverage (50 % or higher must be reached for a stable Jenkins build and 80 % for a good "sunny" build). To ensure a reasonable quality and maintainability of the code please adhere to the Code Guidelines.

A demoable version can be accessed here https://spwdfvml0898.wdf.sap.corp:8443.

The latest JavaDoc (Backend only) can be accessed on our Jenkins build

Maven project information (Backend only) can be found here

Architecture and Concepts

There is a FeatureSpec available that explains some of the remaining features in more detail.

TwoGo was shifted away from River to a mainstream technology-mix platform with higher productivity and reliability. See here for more information.

TwoGo will facilitate the services of Nokia Maps (former Navteq). Details can be found here.

TwoGos web client is implemented in SAP UI5.

Release Management

TwoGo Codelines

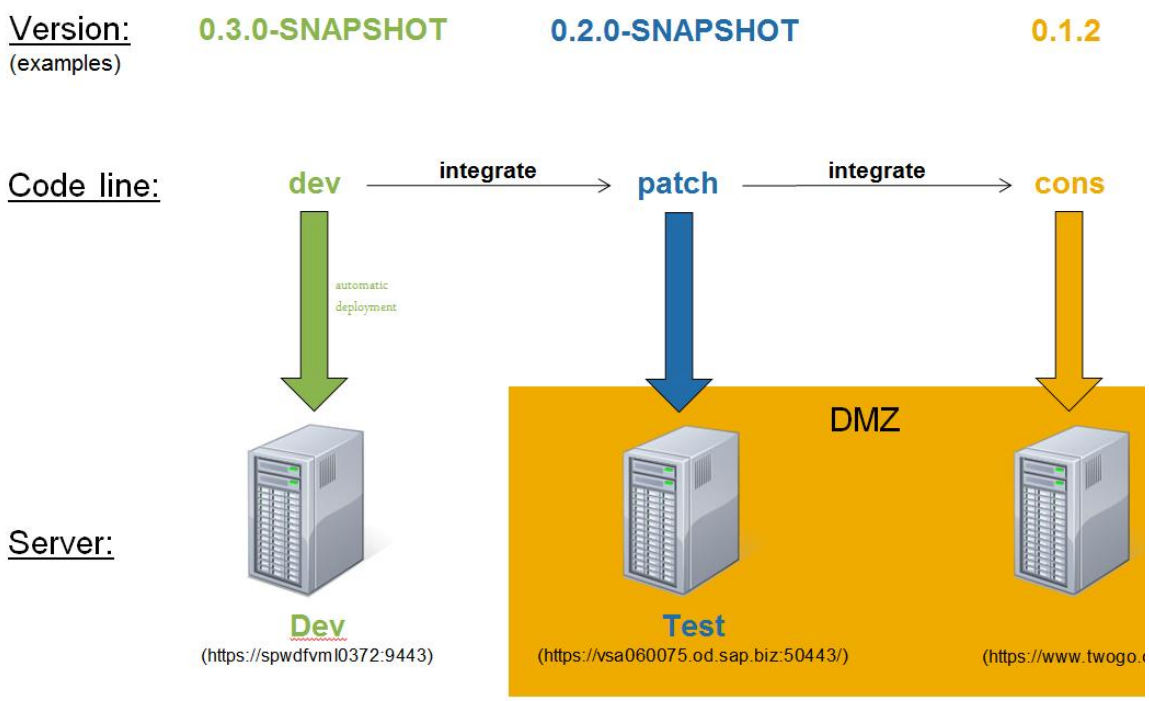
TwoGo source code is located in Perforce Port **perforce3406:3406**, depot **ptg.** The depot contains 4 (standard SAP) code lines: **dev, export, patch** and **cons**, whereas TwoGo only uses 3 of them: **dev, patch** and **cons** (and to be precise, TwoGo uses them not in their intended SAP functions).

- **dev** code line is owned by all TwoGo developers. No restriction applies to any check-ins, except the code must be buildable on CI Server (Jenkins: http://spwdfvm1246.wdf.sap.corp:8080/jenkins/). After each check-in an automatic deployment (incl. new change list) is triggered from CI Server to central Dev machine.
- **patch** code line is owned by TwoGo Quality Management (QM) Team. This code line is created by integrating dev code line to patch code line at a certain point in time (currently: every second week). TwoGo developers have full access (read/write) to this code line, however they're only allowed to change file(s) in patch code line on QM Team request. Deployments from patch code line will take place on request to central Test server.
- **cons** code line is owned by TwoGo Administrators. This code line is created by integrating patch code line to cons code line at a certain point in time (currently: every second week before dev gets integrated to patch). TwoGo developers have full access (read/write) to this code line, however they're only allowed to change file(s) in cons code line on Product Owner request (Hot Fix). Deployments from cons code line will take place on request to central TwoGo productive server (https://www.twogo.com).

TwoGo Versioning

TwoGo version numbers are formatted as proposed by Apache Maven project
`<major version>.<minor version>.<incremental version>-<qualifier>`

- TwoGo increases major version numbers to indicate incompatible API changes and/or new/reworked major features (incl. bug fixes)
- TwoGo increases minor version numbers to indicate compatible API changes and/or new/reworked features (incl. bug fixes)
- TwoGo increases incremental version number to indicate bug fix provisioning
- TwoGo uses Apache Maven qualifier `SNAPSHOT` to indicate instable (frequently changed) versions, which have not been officially released



1. After integrating patch code line to cons code line, the qualifier `SNAPSHOT` is removed from version number (in cons code line) to release (via TwoGo Nexus) the version.
2. In case a severe bug fix (a.k.a. HotFix) must be applied to productive server, the fix is checked in to cons code line, build on CI server and deployed to productive server. To identify this fix, TwoGo increments the incremental version number (e.g. `0.1.1` → `0.1.2`)
3. After integrating dev code line to patch code line, the minor version number gets incremented (in dev code line) (e.g. `0.2.0-SNAPSHOT` → `0.3.0-SNAPSHOT`)

In case TwoGo plans to release a new major version, the major version number gets incremented (e.g. `0.4.0-SNAPSHOT` → `1.0.0-SNAPTSHOT`). It is considered a best practice to reset all subsequent version numbers to 0, when incrementing a preceding version number. (e.g. `1.2.0-SNAPSHOT` → `2.0.0-SNAPHOT`) The qualifier `SNAPSHOT` remains, until patch code line gets integrated to cons code line.

Integrating code lines and incrementing TwoGo versions

Code line integration

Integrating code lines is done via SAP Perforce Management System.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich:

1. dass ich meine Projektarbeit 1 mit dem Thema *Optimierung des Release-Prozesses bei SAP TwoGo mithilfe von Continuous Delivery* ohne fremde Hilfe angefertigt habe;
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;
3. dass ich meine Projektarbeit 1 bei keiner anderen Prüfung vorgelegt habe;
4. dass die eingereichte elektronische Fassung exakt mit der eingereichten schriftlichen Fassung übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, 11. November 2013

Johannes Haaß