Git, ein Versionskontrollsystem für verteilte und mobile Mitarbeit

Carsten Wiesbaum, esentri AG

In der heutigen Software-Entwicklung trifft man häufig verteilte Projekt-Teams und Projekt-Mitarbeiter mit erhöhter Reisetätigkeit. Dezentrale Versionsverwaltungs-Systeme (VCS) wie Git bieten in diesem Umfeld einige Vorteile. Dennoch werden ihre zentralen Verwandten häufig bevorzugt. Was sind die Vorteile dezentraler Systeme in einem solchen Umfeld? Was sind die Gründe für diese Ablehnung? Wie sehen mögliche Arbeitsabläufe mit dezentralen Systemen am Beispiel von Git aus?

Ein wichtiger Aspekt des täglichen Arbeitsablaufs innerhalb der Software-Entwicklung ist die Verwaltung des Quellcodes und seiner Versionen *[1]*. Für diese Aufgabe haben sich Versionsverwaltungssysteme bewährt und etabliert. Sie ermöglichen es, jeden Entwicklungsstand eines Projektes zu sichern und ihn den Projekt-Mitgliedern zur Verfügung zu stellen. Dabei gehören Funktionen wie die Verwaltung des Quellcodes, die Arbeit mit der Projekt-Historie sowie Branching und Tagging zu den Standardfunktionen dieser Werkzeuge.

Für lange Zeit waren zentrale Versionsverwaltungs-Systeme wie Concurrent Versions System (CVS) und die nach dessen Vorbild weiterentwickelte Subversion (SVN) sowohl in offenen Projekten als auch für Firmen die erste Wahl. Bei zentralen Systemen erfolgt die eigentliche Versionsverwaltung auf einem zentralen Server. Dieser enthält ein Quellcode-Repository, auf das Projektmitglieder zugreifen. Die Projektmitglieder erhalten vom Server eine Kopie der angeforderten Projektversion und können auf dieser lokal arbeiten. Änderungen werden später direkt zum Server übertragen und sind dann für alle Projektmitglieder verfügbar *(siehe Abbildung 1)*.

Seit einiger Zeit bekommen jedoch auch dezentrale Versionsverwaltungs-Systeme immer mehr Aufmerksamkeit. Hier besitzt jedes Projektmitglied eine komplette Kopie des Quellcode-Repository. Die einzelnen Projektmitglieder können die verschiedenen Versionen des Projekts untereinander austauschen und gemeinsam daran arbeiten. Dabei entsteht ein Netzwerk von Projektmitgliedern und eine Vielzahl möglicher Arbeitsabläufe ist möglich *(siehe Abbildung 2)*.

Gerade in der heutzutage immer häufiger verteilten und agil ablaufenden Software-Entwicklung können dezentrale Versionsverwaltungssysteme einige Vorteile bieten. Dennoch trifft man oft auf Ablehnung beim Einsatz dieser Systeme. Dabei spielen Angst vor Kontrollverlust (kein zentraler Server = Chaos) und die Befürchtung eines hohen Trainingsaufwands oft eine Rolle.

Dieser Artikel beschreibt zunächst die Vorteile dezentraler Versionsverwaltungs-Systeme in verteilten Projekt-Szenarien und zeigt das Prinzip von Local und Remote Branches. Dabei werden die wichtigsten Kommandos für die Interaktion mit Branches durch Beispiele erklärt sowie ein möglicher Ansatz für eine Projektstruktur mit verteilten Teams und einem zentralen Git-Server beschrieben.

Vorteile dezentraler Systeme

Es gibt häufig Projekt-Szenarien, in denen die Entwicklung durch mehrere agile

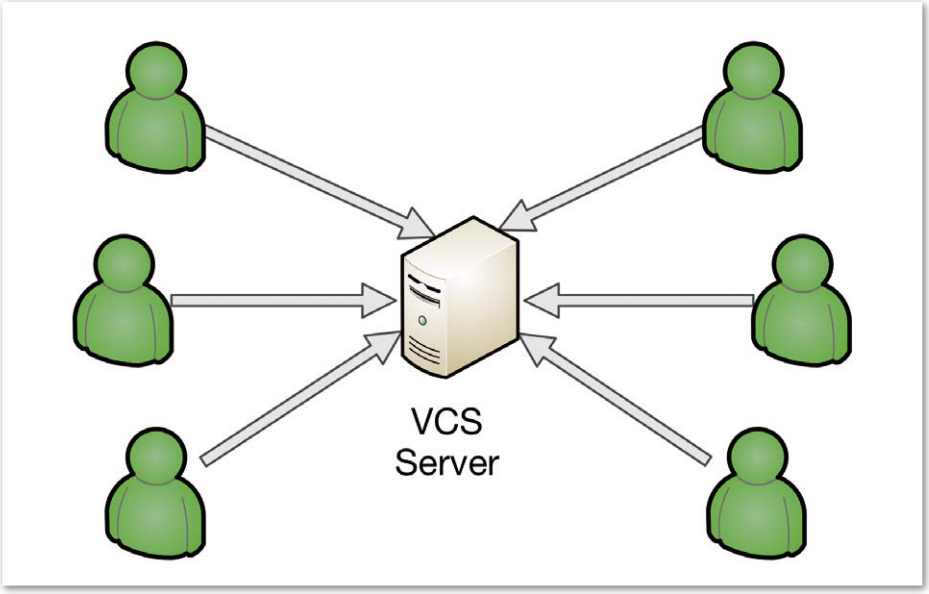

Abbildung 1: Zentrales VCS

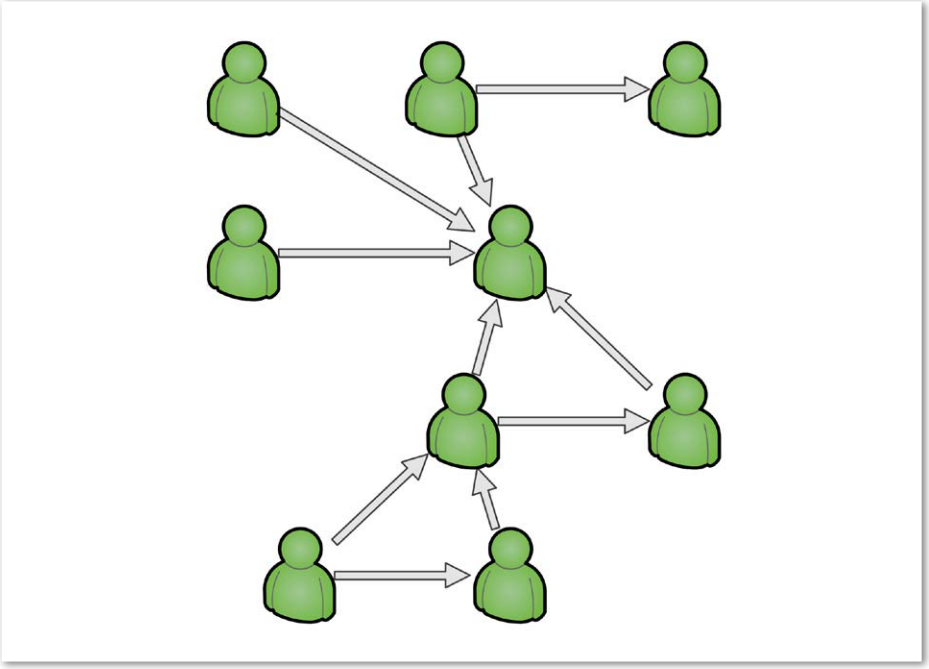

Abbildung 2: Netzwerk eines dezentralen VCS

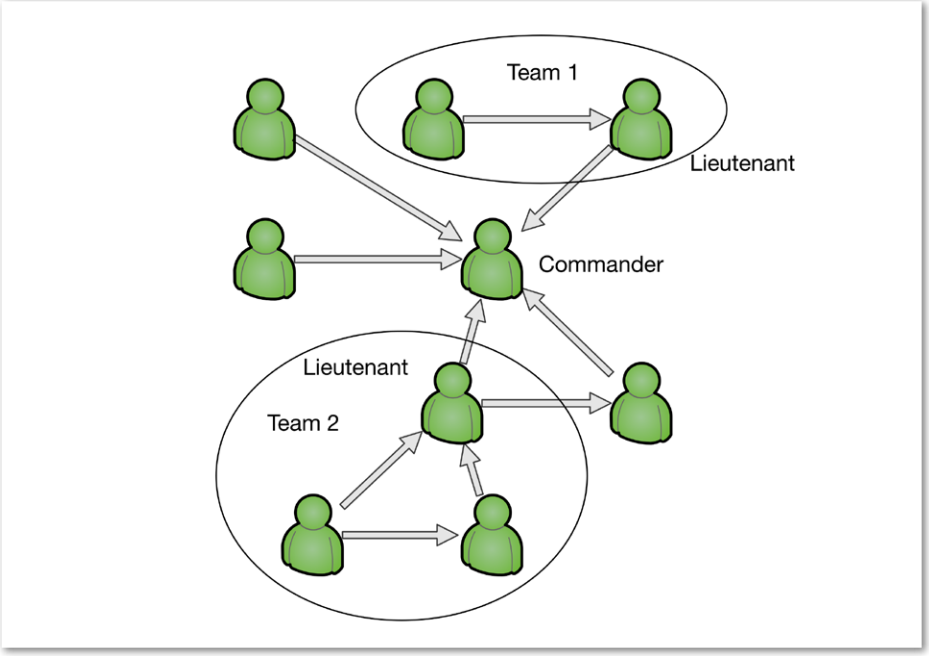

Abbildung 3: Commander/Lieutenant Model

Teams, Projektmitglieder und Dienstleister erfolgt. Dies können mehrere Projektteams sein, die organisatorisch und geologisch getrennt sind, Projektmitglieder, die flexible Arbeitsmodelle nutzen und zum Beispiel im Home-Office arbeiten, oder externe Ressourcen, die aus Gründen von fehlender Expertise oder plötzlichen Ressourcen-Engpässen hinzugezogen werden. Häufig ist auch eine erhöhte Reisetätigkeit vorzufinden, sei es von einzelnen Projektmitgliedern, die Termine an einem anderen Standort wahrnehmen, oder externe Ressourcen, die von ihrem eigenen Wohnsitz zum Kunden anreisen müssen.

Auch wenn zentrale Versionskontroll-Systeme in so einem Umfeld funktionieren können, bieten sich gerade dezentrale Systeme für solche Projektszenarien an. In diesem Artikel wird Git beispielhaft verwendet, um die Vorteile dezentraler Versionskontroll-Systeme in den beschriebenen Szenarien aufzuzeigen. Git wurde ursprünglich von Linus Torvalds als Ersatz für das damalige Versionskontroll-System des Linux-Kernels ins Leben gerufen *[2]*. Einige der Design-Aspekte dabei waren:

- Dezentrale Entwicklung
- Unterstützung für nicht-lineare Entwicklung
- Geschwindigkeit

Am Linux-Kernel arbeitet eine Vielzahl weltweit verteilter Entwickler in einer Community. Die involvierten Charaktere und deren Fähigkeiten sind sehr unterschiedlich, die Spanne reicht von engagierten Open-Source-Entwicklern bis zu Mitarbeitern von Firmen, die für ihren Beitrag zum Projekt bezahlt werden. Es kristallisieren sich einzelne Gruppen heraus, die sich auf spezielle Bereiche des Linux-Kernels fokussieren, zum Beispiel auf Dateisysteme oder Netzwerke. Die innere Struktur und Arbeitsweise ist von den Gruppen größtenteils selbst bestimmt. Sie sind über einzelne Personen miteinander vernetzt. Änderungen und Patches werden zunächst innerhalb der Gruppen verteilt und entwickelt.

Sobald ein stabiler Stand erreicht ist, werden die Änderungen von den entsprechend vernetzten Personen weiter publiziert, bis sie letztendlich in den offiziellen Kernel-Branch aufgenommen sind. Dabei hat sich innerhalb der Community ein „Web of Trust“ etabliert; der Kern der Community vertraut bestimmten Personen und akzeptiert ihre Änderungen. Diese Personen kennen und vertrauen wiederum anderen Personen in ihrem Netzwerk und so weiter. Dieses Entwicklungsmodell heißt auch „Commander/Lieutenant Model“ *(siehe Abbildung 3)*.

Betrachtet man das Linux-Kernel-Projekt als ein Beispiel für ein stark verteiltes Projekt mit heterogenen Projektmitgliedern, lassen sich einige vorteilhafte Aspekte von Git und dessen Einsatz im Entwicklungsprozess ableiten. Durch das dezentrale Prinzip besitzt jedes Projektmitglied eine komplette Kopie des Repository. Daher ist für die meisten alltäglichen Arbeiten keine Netzwerk-Verbindung erforderlich. Änderungen werden zunächst lokal durchgeführt und abgelegt. Wenn ein entsprechender Stand erreicht ist,

kann man diesen mit anderen teilen oder in ein übergeordnetes Repository übertragen. Diese Fähigkeit ist besonders hilfreich für Projektmitarbeiter, die nicht immer Zugriff auf den Versionsverwaltungs-Server haben, etwa bei der Arbeit im Home-Office oder während einer Reise.

Ein weiterer Vorteil des lokalen Repository ist die Möglichkeit, alle Funktionalitäten zur Interaktion mit der Projekthistorie auch ohne Netzwerk-Verbindungen durchzuführen. So kann in der Historie gestöbert oder eine ganze Projektversion wiederhergestellt werden, ohne dafür Zugriff auf einen zentralen Server zu benötigen. Auch für Experimentierfreudige bietet das dezentrale Prinzip Vorteile, so kann eine Projekt-/Produktidee von Einzelnen oder im Team „kontrolliert" entwickelt werden. Dazu ist lediglich ein neues lokales Repository zu erstellen *(siehe Listing 1)*. Anschließend wird das Repository wie gewohnt verwendet. Sollte sich die ursprüngliche Idee als praktikabel erweisen, kann das lokale Repository mit anderen geteilt und gemeinsam weiterentwickelt werden.

Ein weiterer interessanter Design-Aspekt von Git ist die nicht-lineare Entwicklung. Grundidee ist, dass die Verwendung von Branches ein alltägliches Werkzeug in der Software-Entwicklung darstellt. An dieser Stelle spielt auch der Design-Aspekt „Geschwindigkeit" eine große Rolle; die Erstellung von Branches und alle Operationen, die auf ihnen ausgeführt werden, sind generell sehr günstig. Daher fällt die Verwendung von Branches in Git leicht und fühlt sich intuitiv richtig an.

Hinsichtlich agiler und verteilter Teams ist dieser Aspekt interessant, weil jedes Team und Teammitglied seinen eigenen Workflow entwickeln kann. So lässt sich innerhalb der Teams ein eigenes Branching Model adaptieren, das zu den einzelnen Charakteren und Fähigkeiten der Teammitglieder passt. Ähnlich kann eine entsprechende Branching-Strategie von den Teams untereinander entwickelt und verfolgt werden. Natürlich sollten bestimmte Rahmenbedingungen vorgegeben und eingehalten werden, Git ermöglicht jedoch eine Anpassung an die eigenen Arbeitsweisen und kann damit zur Produktivitätssteigerung beitragen.

```
mkdir experiment
cd experiment
git init
```

Listing 1

Local und Remote Branches

Die genannten Punkte machen Git attraktiv für den Einsatz in verteilten Projekten. Oft scheitert die Einführung jedoch schon an ein paar simplen Fragen, die gerade beim ersten Versuch, ein dezentrales System einzusetzen, gestellt werden. Wie funktioniert der Austausch der einzelnen Entwicklungsstände? Kann es, durch die Freiheiten, die Git einem jeden einräumt, nicht in einem riesigen Chaos enden? Wie kann die Projektleitung ohne zentralen Server die Kontrolle über das Projekt behalten?

Berechtigte Fragen – um sie zu beantworten, muss zunächst das Branching-Prinzip von Git betrachtet werden *[1]*. Durch die Fokussierung auf die nicht-lineare Entwicklung als zentralem Design-Aspekt von Git ist Branching ein alltägliches Werkzeug. Jedes Repository enthält für gewöhnlich einen sogenannten „Master-Branch". Dort befindet sich der aktuelle Entwicklungsstand. Von diesem Branch werden weitere erzeugt. Git unterscheidet zwischen zwei Arten von Branches, „local" und „remote". Local Branches sind die eigenen Branches, die auf dem lokalen System existieren. Remote Branches hingegen befinden sich auf anderen Systemen, von ihnen können Änderungen gezogen („fetch" und „pull") und bei entsprechenden Berechtigungen auch weggeschrieben werden („push").

Wie gesagt, die Arbeit mit Branches ist in Git günstig. Aus diesem Grund können neue Branches häufig erstellt werden, etwa für jede neue Funktion oder für jeden Bugfix. Um einen neuen Branch für „Feature 042" zu erzeugen und auf diesen zu wechseln, reicht ein einziger Befehl in der Kommandozeile: „git branch –b feature-042". Auf diesem Branch kann das Feature dann isoliert entwickelt werden.

Sollte es eine wichtigere Aufgabe geben, etwa einen Hotfix, kann die bisherige Arbeit in dem Branch gesichert („stashing") und ein neuer Branch ohne die bisherigen Änderungen von „feature-042" erstellt werden. Ist der Hotfix fertig implementiert, kann die Arbeit am Feature-Branch fortgesetzt werden. Nachdem die Arbeit an einem Branch abgeschlossen wurde, werden die Änderungen für gewöhnlich wieder in den Master-Branch integriert und der Feature-Branch gelöscht *(siehe Abbildung 4)*.

Der Austausch von Entwicklungsständen erfolgt über das Prinzip der Remote Branches. Diese unterscheiden sich von lokalen Branches lediglich durch ihren Speicherort; sie liegen in anderen Repository-Kopien auf einem entfernten System. Für den Zugriff auf diese Branches

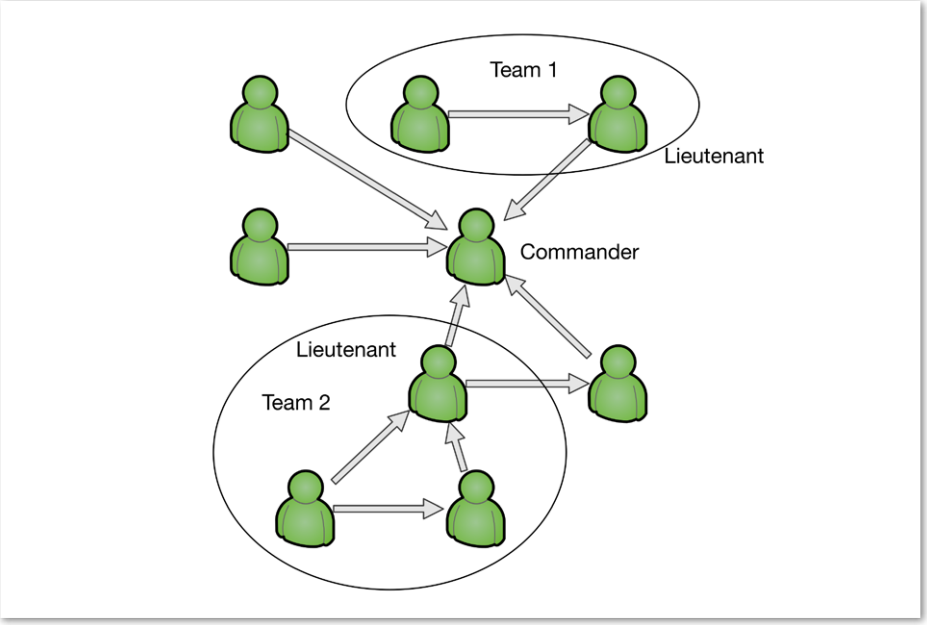

Abbildung 4: Simples Beispiel für Branching

```
example-project]$ git fetch
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://demo.gitlab.com/esentri/example-project.git
   b555a8d..3d8be2a  master -> origin/master
```

Listing 2

```
example-project]$ git merge remotes/origin/master
Updating b555a8d..3d8be2a
Fast-forward
README |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Listing 3

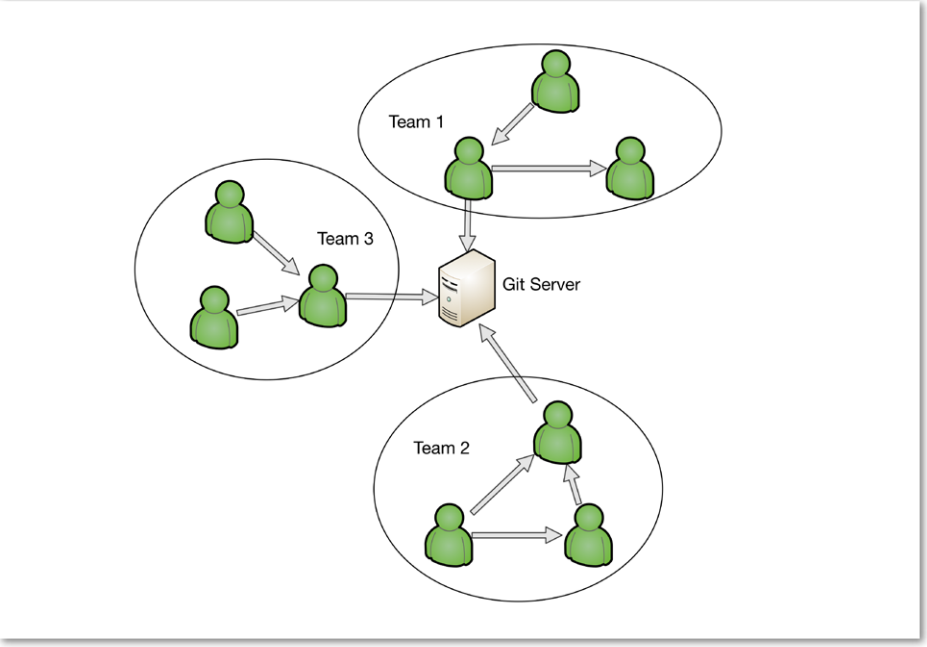

Abbildung 5: Git-Server-Szenario

gibt es die vier Kommandos „fetch“, „merge“, „pull“ und „push“. Mit „fetch“ werden zunächst alle Änderungen vom Remote Branch heruntergeladen *(siehe Listing 2)*.

Zu diesem Zeitpunkt wurden die Änderungen noch nicht auf einen lokalen Branch angewendet. Man könnte sie nun zunächst auf Korrektheit prüfen und erst anschließend in einen eigenen lokalen Branch mit dem Befehl „merge“ einpflegen *(siehe Listing 3)*.

Wenn man die Änderungen direkt anwenden möchte, kann auch das Kommando „pull“ genutzt werden, es führt „fetch“ und „merge“ in einem Arbeitsschritt aus.

Zentraler Server im dezentralen System

Hinsichtlich des Branching-Konzepts ist festzuhalten, dass ein Remote Branch nicht zwangsweise auf dem System eines anderen Entwicklers zur Verfügung gestellt werden muss. Auch Git ermöglicht die Verwendung eines zentralen Servers als Remote Branch. Normalerweise wird hierfür ein Bare Repository verwendet. Dieses enthält die gesamte Projekthistorie, jedoch keine Arbeitskopie, auf der Änderungen durchgeführt werden können. Um ein Bare Repository auf einem Server zu erstellen, muss die Option „bare“ bei der Repository-Erstellung angegeben sein *(siehe Listing 4)*.

```
mkdir experiment.git
cd experiment.git
git init --bare
```

Listing 4

Eine Kopie des Repository kann anschließend von Entwicklern über den Befehl „git clone“ erstellt werden. Bei diesem Prozess wird der Remote Branch des Servers automatisch als primäres Ziel für den Befehl „push“ konfiguriert: „git clone *https://myRepoHost.com/myproject/experiment.git*“.

Änderungen, die auf der lokalen Kopie eines Remote Branch durchgeführt wurden, können anschließend mit dem Befehl „push“ zurück auf das entfernte System geschrieben werden. Voraussetzung hierfür sind natürlich die entsprechenden Schreibrechte.

Abbildung 5 zeigt eine mögliche Adaption des Commander/Lieutenant-Modells für ein Projekt mit verteilten Teams, die gemeinsam mit einem zentralen Server arbeiten. Der Server nimmt in diesem Fall die Rolle des Commander ein. Er vertraut nur den Teamleitern der jeweiligen Teams; nur sie haben Schreibrechte auf das zentrale Repository und können Änderungen einpflegen. Der Teamleiter fungiert so als eine Qualitätssicherungs-Instanz. Darüber hinaus hat jedes Team die Möglichkeit, seinen internen Entwicklungsprozess selbst zu bestimmen.

Fazit

Versionsverwaltungs-Systeme sind ein unentbehrlicher Bestandteil der heutigen Software-Entwicklung. Es gibt sowohl ausgereifte zentrale als auch dezentrale Systeme. Gerade in verteilten Projektszenarien bieten dezentrale Systeme jedoch einige Vorteile. Durch die Unabhängigkeit von einer bestehenden Netzwerk-Verbindung können Projektmitarbeiter mit erhöhter Reisetätigkeit auch unterwegs effektiv arbeiten. Zudem ermöglicht das dezentrale Prinzip eine Vielzahl von Arbeitsabläufen, sowohl für den einzelnen Projektmitarbeiter als auch für einzelne Projektteams. Damit kann der Entwicklungsprozess auf die Fähigkeiten und Rahmenbedingungen abgestimmt und angepasst werden.

Gerade durch diese Flexibilität entstehen jedoch auch Risiken, so kann es ohne konkrete Vorgaben durchaus zu einer chaotischen Entwicklung kommen. Bevor die Entwicklung mit Git innerhalb eines Projekts startet, sollten daher einige klare Regeln für die Entwicklung mit dem System aufgestellt und eine grundlegende Integrationsstrategie festgelegt werden.

Durch die native Unterstützung von Git in JDeveloper 12*c* ist es gerade für Projekte im Umfeld der Oracle Fusion Middleware eine interessante Option. Die Entwicklungsumgebung bietet alle grundlegenden Funktionen für die Interaktion sowie die Erstellung entfernter und lokaler Repositories und sollte in jedem Fall als eine valide Option für die Quellcode-Verwaltung eines neuen Projekts in Betracht gezogen werden.

Weitere Informationen

[1] fournova Software GmbH, Learn Version Control With Git: http://www.git-tower.com/learn/ebook/command-line/introduction

[2] Scott Chacon, Pro Git: http://git-scm.com/book

Carsten Wiesbaum
carsten.wiesbaum@esentri.com

Apex und Phonegap? Das kann Apex doch mit HTML5

Daniel Horwedel, merlin.zwo InfoDesign GmbH & Co. KG

Mobile Anwendungen lassen sich sehr gut mit Apex entwickeln. Insbesondere das jQuery-mobile-Framework und das Responsive-Theme geben dem Entwickler einfach zu nutzende Werkzeuge an die Hand, um seine Apex-Web-Anwendung für Mobilgeräte zu optimieren. Sobald aber der Zugriff auf Hardware-Funktionen des Gerätes nötig ist, ist meist der Einsatz eines Frameworks wie Cordova oder PhoneGap erforderlich. Mit zunehmender Verbreitung von HTML5 lassen sich inzwischen aber die meisten Anforderungen auch ohne Verwendung eines zusätzlichen Frameworks mit JavaScript- und HTML5-Bordmitteln umsetzen.

Die Realisierung einer Web-Anwendung mit HTML5 und JavaScript ohne Verwendung eines Frameworks wie Cordova erfordert leider noch sehr intensives Testen auf den verschiedensten Zielplattformen, da die entsprechenden APIs zwar durch W3C standardisiert sind – aber leider durch die Browser-Hersteller unterschiedlich und nicht immer vollumfänglich umgesetzt werden. Dennoch bieten sich damit interessante, vielversprechende und leichtgewichtige Möglichkeiten zur Umsetzung von modernen, Feature-reichen Web-Anwendungen.

GUI: Responsive

Den einfachsten Teil bei der Erstellung einer modernen mobilen Browser-Anwendung mittels Apex stellt die Benutzeroberfläche dar. Hier bietet Apex dem Entwickler zwei verschiedene, je nach konkretem Anwendungszweck unterschiedlich gut geeignete GUI-Konzepte. Mit der jQuery-mobile-basierten Benutzeroberfläche lässt sich mit geringem Aufwand eine für mobile Endgeräte optimierte Benutzeroberfläche erstellen, die die Besonderheiten dieser Geräte (kleines, meist hochauflösendes Display, Touch-Bedienung) besonders berücksichtigt und versucht, ein Nutzererlebnis im Stil einer nativen App nachzubilden.

Alternativ dazu bietet Apex mit dem Theme 25 ein Responsive-GUI, bei dem eine optimale Bedienbarkeit auf verschiedensten Geräten und Bildschirmgrößen zu Lasten einer möglichst nativen Benutzerführung im Vordergrund steht. Eine weitere interessante Möglichkeit zur Umsetzung einer gut bedienbaren, universellen Benutzeroberfläche stellt Twitters Bootstrap-Framework dar – inzwischen sind auch fertige Apex Bootstrap-Themes *(siehe „http://apex-plugin.com/oracle-apex-plugins/themes/css-layout/bootstrap3-theme_396.html")* verfügbar.

Die Hardware

Moderne HTML5-fähige Browser ermöglichen den einfachen Zugriff auf die am häufigsten benötigten Hardware-Funktionen eines Smartphones: den GPS-Sensor, die Kamera und das Mikrofon. Durch die Kombination dieser beiden Funktionen lassen sich auch weiterführende Anwendungen realisieren, etwa ein QR-Code-Scanner innerhalb der Apex-Anwendung.