Belegarbeit GIT

Fabian Goerge
Sami Ede
Richard Peters
Jens Meise

9. Januar 2013

Inhaltsverzeichnis

1 Einführung und Ziel der Arbeit

1.1 Einführung

Git ist ein freies Versionsverwaltungssystem, dass von Linus Torvald entwickelt wurde. (Das zu der Zeit existierende Versionsverwaltungssystem BitKeeper war aus Lizenzgründen nicht mehr für das Linuxentwicklerteam zugänglich, und so entschied sich Torvald dafür, ein eigenes System zu schreiben.)
Ein Versionsverwaltungssystem verfolgt Änderungen in Dokumenten oder Dateien und speichert die durch die Änderungen entstehenden Versionen in einem Verzeichnis so ab, dass sie mit einem Zeitstempel und einer Benuzerkennung versehen werden und später wieder hergestellt werden können.
Git bietet weiterhin die Möglichkeit sich eigene Branches zu erstellen, so dass man unabhängig vom "Hauptstrang" arbeiten kann, ohne in Gefahr zu laufen, wichtige Teile des Codes unbeabsichtlich zu zerstören, um so zum Beispiel Portierungen einer Applikation auf andere Betriebssysteme zu entwickeln.
Git speichert außerdem alle Dateien lokal, so dass auch ohne Internetverbindung gearbeitet werden kann, die Zusammenführung (Merge) der Branches muss dann zu einem späteren Zeitpunkt stattfinden. Jeder Benutzer verfügt also über eine komplette Kopie des Repository.
Die Speicherung der Versionsgeschichte erfolgt so, dass der Name einer beliebigen Revision auf der vollständigen Geschichte basiert, die zu dieser Version geführt hat, so dass es nicht möglich ist, die Versionsgeschichte nachträglich zu ändern. Diese Funktion ermöglicht auch, bestimmte Versionen gesondert digital zu signieren, um z.B. den Zustand zum Zeitpunkt des Erscheinens einer neuen Version besonders zu kennzeichnen.
Ein weiterer Vorteil von Git ist zudem eine große Vielfalt von Möglichkeiten zum Austausch zwischen Repositories; Git unterstützt eine Vielzahl von Protokollen zum Dateiaustausch, darunter auch ein eigenes, sehr effizientes (git://).

1.2 Ziel

Das Ziel dieses Projektes, ist die Modellierung von Git, die Darstellung und Rekonstruktion einer Software über Diagramme, sowie die Auseinandersetzung mit den verschiedenen Modellierungsarten.
Dafür werden wir uns zunächst mit einem Use-Case-Modell beschäftigen und dieses anschließend in einigen Use-Case-Szenarien elaborieren.
Außerdem wird ein Domänenmodell dargestellt, sowie einige ausgewählte Zustandsdiagramme.
Abschließend erläutern wir einige Erfahrungen aus der Teamarbeit und eine Schlussbetrachtung.

2 Textuelle Beschreibung

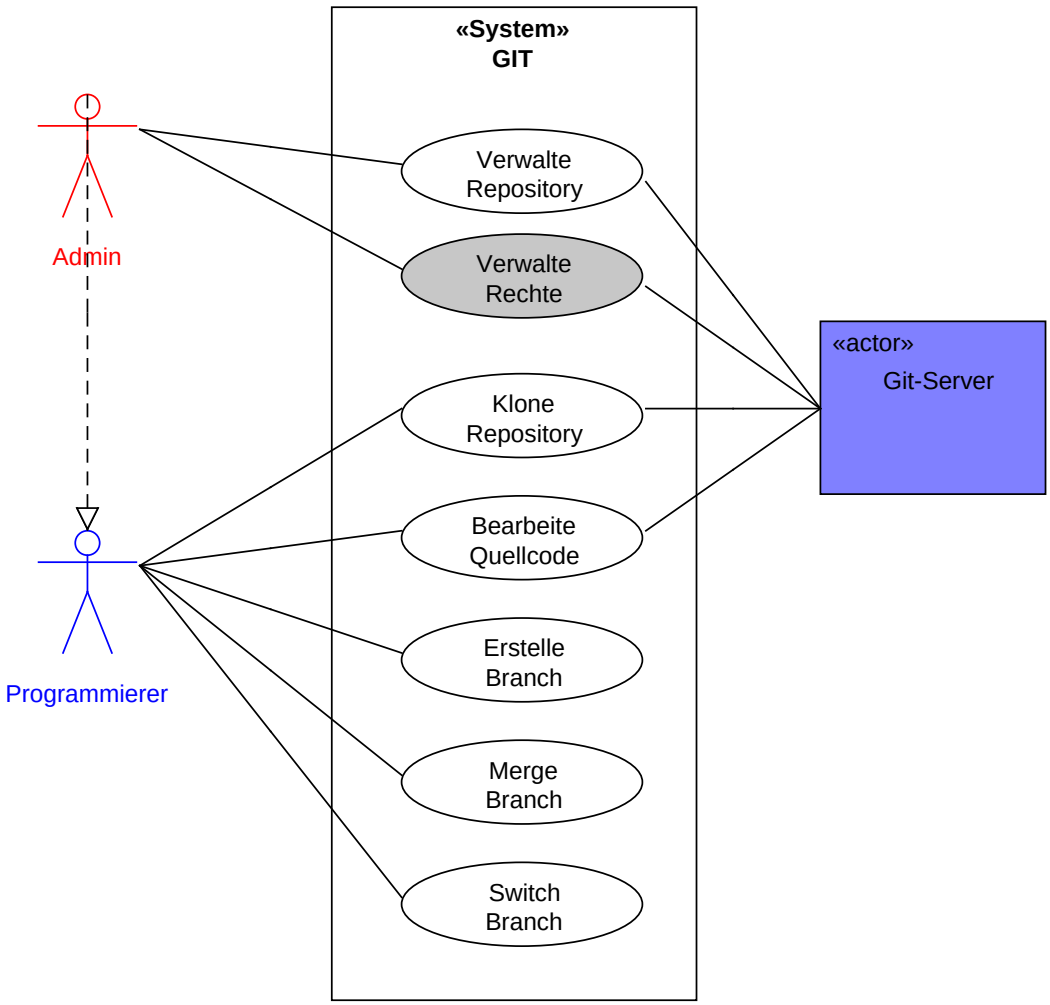
Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien. Git hilft, wenn mehrere Programmierer an einem Projekt arbeiten, indem es verschiedene Features vereint:

- Git ermöglicht eine nicht-lineare Entwicklung; das heißt, es ist möglich, an verschiedenen „Positionen“ des Quelltextes zu arbeiten, und diese im Nachhinein automatisch zu vereinen („mergen“).
- des Weiteren bietet Git die Möglichkeit eines lokalen Arbeitens. Jeder User besitzt eine lokale Kopie des kompletten Repositories (inklusive der Versionsgeschichte), so dass auch ohne Anbindung in ein Netzwerk gearbeitet werden kann. Dies ermöglicht sehr schnelles Arbeiten! Dies bedeutet auch gleichzeitig Multi-Backups! Jeder User speichert die komplette Versionsgeschichte!
- Mehrere Möglichkeiten, Repositories abzugleichen; Git besitzt ein eigenes Protokoll, und unterstützt unter anderem http, https, ftp und rsync Übertragungen.
- Versionsgeschichte wird kryptografisch gesichert, so dass es nicht möglich ist, Änderungen an der Versionsgeschichte vorzunehmen, ohne dass sich der Name der Revision ändert.

Außerdem bietet Git ein Webinterface.

3 Modellierung

3.1 Use-Case-Modell



Dieses Use-Cases repräsentieren die Interaktion zwischen dem Programmierer (*Aktor*) und dem Git-System (*Subjekt*)

Der Programmierer benutzt das Git-System um ein bestimmtes Ziel zu erreichen. Der Git-Server (*Sekundäre Aktor*) wird genutzt um mit den lokalen GIT Instanzen synchronisiert zu werden.

Der Programmierer repräsentiert eine Personengruppe. An einem Projekt können beliebig viele Programmierer beteiligt seien. Der Admin erbt alle Fähigkeiten von dem Programmierer. Demnach kann der Admin auch alle anderen Interaktionen ausüben.

Im Use-Case sind die Möglichkeiten, verwalte Repository, Clone Repository, bearbeite Quellcode, erstelle Branch, Merge Branch und Switch Branch aufgeführt.

Optional besteht die Möglichkeit für den Admin die Rechte mit Hilfe der Betriebssystemverwaltung über ein Skript in GIT zu integrieren.

Eine detaillierte Beschreibung der Abläufe folgt in den Use-Case-Szenarien.

3.1.1 Use-Case-Szenarien (Storys)

Use Case:	Clone Repository
Kurzbeschreibung:	Der Programmierer kopiert sämtliche Daten des Repositories auf seinen lokalen Rechner.
Vorbedingung:	Ein Repository muss auf dem Server existieren.
Primärer Aktor:	Programmierer
Nachbedingung:	Das Repository auf dem Rechner des Programmierers ist mit dem auf dem Server identisch.

Erfolgsszenario:

1. Der Programmierer gibt das Kommando zum Klonen eines Repositories.
2. Das System verbindet sich mit dem Server.
3. Das System überträgt das Repository vom Server auf den Rechner des Programmierers.
4. Das System informiert den Programmierer über den Erfolg der Aktion.
5. Das Use Case endet erfolgreich.

Erweiterungen:

(1.-4.)a.	Der Nutzer bricht den Vorgang ab.
(1.-4.)a.1.	Das Use-Case endet erfolglos.
2.b.	Die Verbindung zum Server kann nicht hergestellt werden.
2.b.1.	Das System informiert den Nutzer.
2.b.2.	Das Use Case endet erfolglos.
3.b.	Die Verbindung bricht während der Übertragung ab.
3.b.1.	Das System informiert den Nutzer.
3.b.2.	Das Use Case endet erfolglos.

Use Case:	Quellcode bearbeiten
Kurzbeschreibung:	Der Programmierer möchte den Quellcode ändern. Dazu muss er das Repository auf den neuesten Stand bringen (‚Pull'), den Quellcode bearbeiten, die Änderungen übernehmen (‚Commit') und die Änderungen an den Server übertragen (‚Push').
Vorbedingung:	Ein Repository muss sich auf dem Rechner und auf dem Server befinden.
Primärer Aktor:	Programmierer
Nachbedingung:	Die Änderungen des Programmierers werden registriert und in die History aufgenommen.

Erfolgsszenario:

1. Der Programmierer gibt das Kommando zum Holen von Änderungen (‚Pull').
2. Das System verbindet sich mit dem Server.
3. Das System vergleicht den lokalen Zustand des Branches mit dem auf dem Server.
4. Das System integriert (‚merge') die Änderungen der Dateien in die lokalen Dateien auf dem Rechner.
5. Das System informiert den Programmierer über den Erfolg der Aktion.
6. Der Programmierer ändert den Quellcode.
7. Das System verfolgt die Änderungen und zeigt sie an.
8. Der Programmierer gibt das Kommando zum Übernehmen (‚Commit') von Änderungen mit einer Kurzbeschreibung seiner Änderungen.
9. Das System übernimmt die Änderungen und trägt sie in die History des aktuellen Branches ein.
10. Das System informiert den Programmierer über den Erfolg der Aktion.
11. Der Programmierer gibt das Kommando zum Übertragen der Änderungen (‚Push') auf den Server.
12. Das System integriert den aktuellen Branch des Rechners in den entsprechenden Branch auf dem Server (‚merge').
13. Das System liefert dem Programmierer eine Erfolgsmeldung.
14. Das Use Case endet erfolgreich.

Erweiterungen:

(1.-13.)a.	Der Nutzer bricht den Vorgang ab.
(1.-13.)a.1.	Das Use-Case endet erfolglos.
2.b.	Die Verbindung zum Server kann nicht hergestellt werden.
2.b.1.	Das System informiert den Nutzer.
2.b.2.	Das Use Case endet erfolglos.
4.b.	Das System stellt Überschneidungen fest.
4.b.1.	Der Nutzer wird aufgefordert die zu übernehmenden Änderungen auszuwählen.
4.b.2.	Fortsetzung bei 5.
11.b.	Die Verbindung zum Server kann nicht hergestellt werden.
11.b.1.	Das System informiert den Nutzer.
11.b.2.	Das Use Case endet erfolglos.
12.b.	Das System stellt Überschneidungen fest.
12.b.1.	Der Nutzer wird aufgefordert die zu übernehmenden Änderungen auszuwählen.
12.b.2.	Fortsetzung bei 13.

Use Case:	Branch erstellen
Kurzbeschreibung:	Das System erstellt einen neuen Branch mit dem Zustand des Branches auf dem sich der Programmierer befindet.
Vorbedingung:	Es muss ein Repository existieren.
Primärer Aktor:	Programmierer
Nachbedingung:	Es gibt einen neuen Branch.
Erfolgsszenario:	

1. Der Programmierer gibt das Kommando zum Erstellen eines Branches mit dem Namen des neuen Branches.
2. Das System erzeugt einen neuen Branch.
3. Das Use Case endet erfolgreich.

Erweiterungen:

1.b.	Das System stellt fest, dass dieser Name schon vergeben ist.
1.b.1.	Das System fordert den Nutzer auf einen anderen Namen zu wählen.
1.b.2.	Der Nutzer gibt einen noch nicht vergebenen Namen ein.
1.b.3.	Fortsetzung bei 2.
(1.-2.)a.	Der Nutzer bricht den Vorgang ab.
(1.-2.)a.1.	Das Use-Case endet erfolglos.

Use Case:	Branch mergen
Kurzbeschreibung:	Ein Branch wird in einen anderen Branch integriert.
Vorbedingung:	Es müssen min. 2 Branches existieren.
Primärer Aktor:	Programmierer
Nachbedingung:	Es gibt einen neuen Branch.
Erfolgsszenario:	

1. Der Programmierer gibt das Kommando zum Mergen eines Branches mit dem Namen des zu integrierenden Branches.
2. Das System integriert den Branche in den aktiven Branche.
3. Das Use Case endet erfolgreich.

Erweiterungen:

1.b.	Das System stellt Überschneidungen fest.
1.b.1.	Der Nutzer wird aufgefordert die zu übernehmenden Änderungen auszuwählen.
1.b.2.	Fortsetzung bei 2.
(1.-2.)a.	Der Nutzer bricht den Vorgang ab.
(1.-2.)a.1.	Das Use-Case endet erfolglos.

Use Case:	Branch switchen
Kurzbeschreibung:	Der aktuelle Branch auf dem Rechner wird gewechselt (‚switch')
Vorbedingung:	Es müssen mindestens 2 Branches existieren.
Primärer Aktor:	Programmierer
Nachbedingung:	Der aktuelle Branch wurde gewechselt.
Erfolgsszenario:	

1. Der Programmierer gibt das Kommando zum Wechseln eines Branches (‚switch') mit dem Namen des Zielbranches.
2. Das System wechselt zum Zielbranch.
3. Das Use Case endet erfolgreich.

Erweiterungen:

(1.-2.)a.	Der Nutzer bricht den Vorgang ab.
(1.-2.)a.1.	Das Use-Case endet erfolglos.

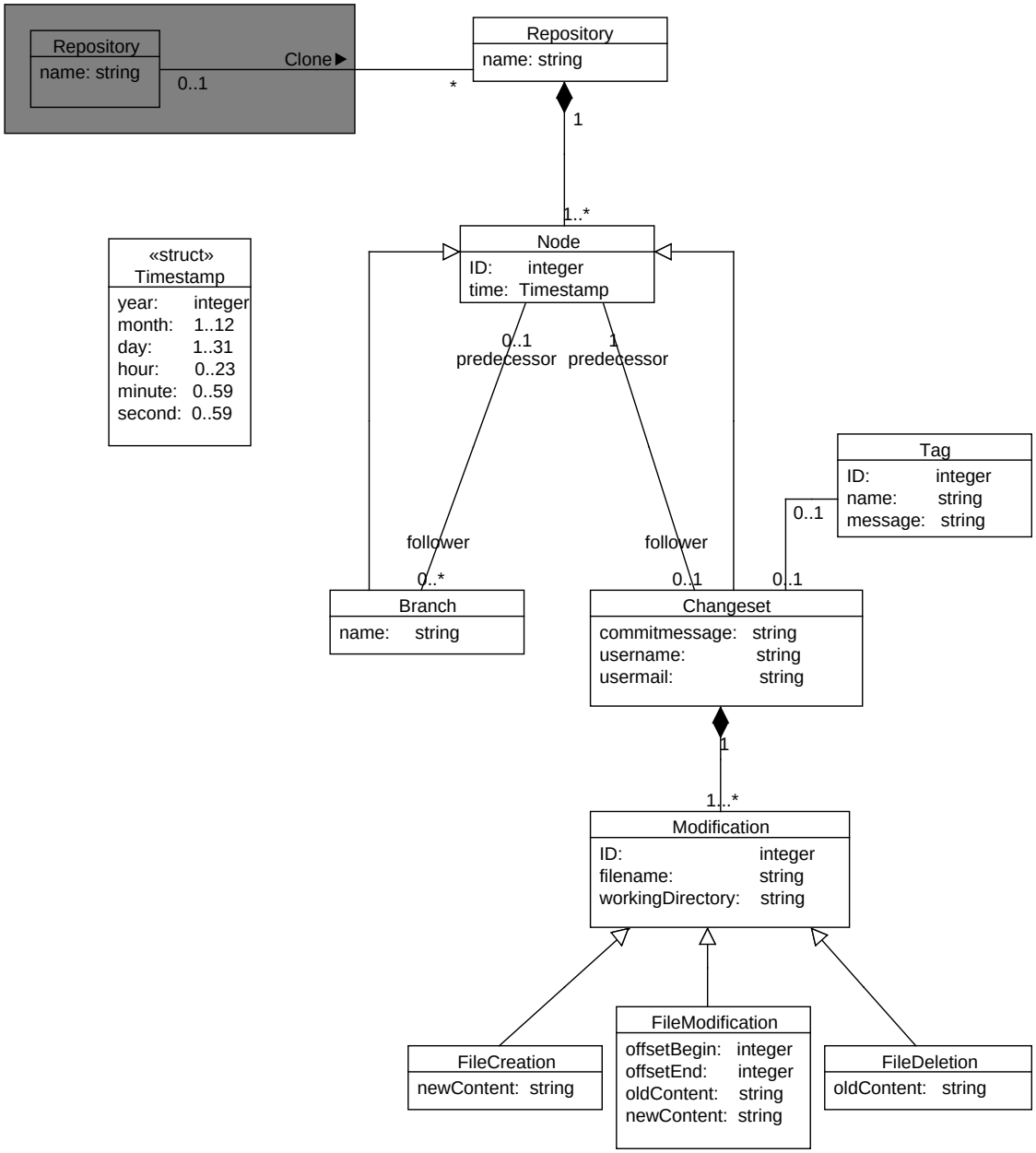
Use Case:	Repository verwalten
Kurzbeschreibung:	Ein Repository wird verwaltet.
Vorbedingung:	Keine – es kann auch erst bei der Verwaltung ein Repository angelegt werden.
Primärer Aktor:	Admin
Nachbedingung:	Repository verändert.
Erfolgsszenario:	

1. Der Admin spezifiziert eine Verwaltungsoperation (löschen, erstellen, umbenennen).
2. Das System führt die entsprechende Aktion aus und informiert den Admin über den Erfolg der Aktion.
3. Das Use Case endet erfolgreich.

Erweiterungen:

2.b.	Das System kann keine Verbindung zum Git-Server herstellen.
2.b.1.	Das Use-Case endet erfolglos
(1.-2.)a.	Der Nutzer bricht den Vorgang ab.
(1.-2.)a.1.	Das Use-Case endet erfolglos.

3.2 Domänenmodell



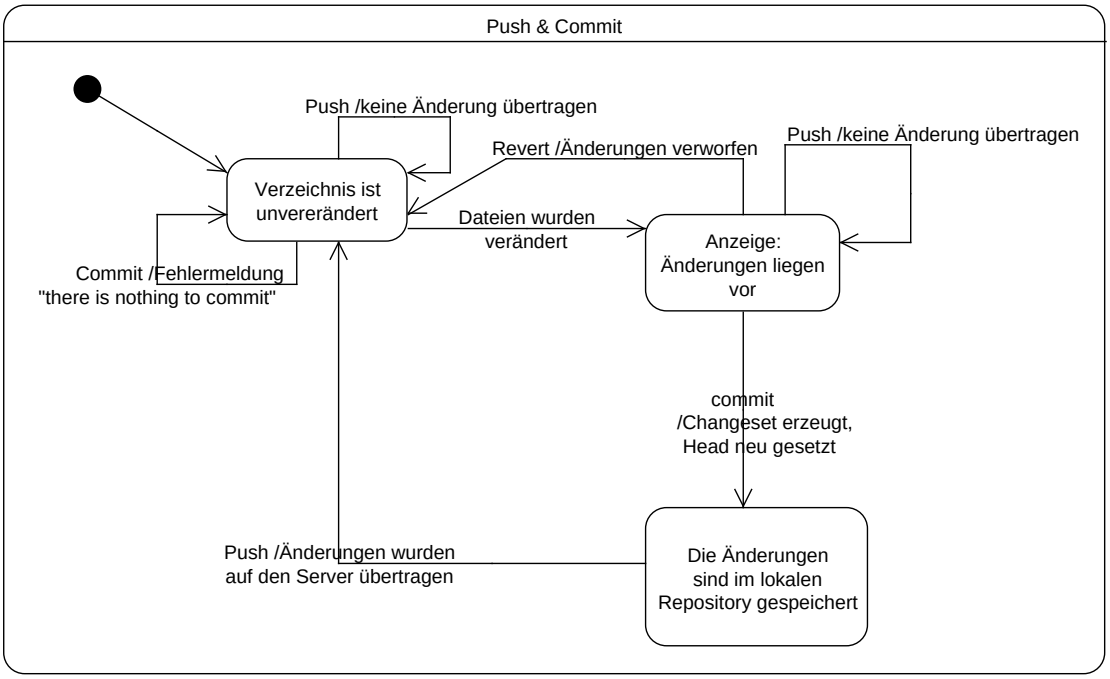
Die Abbildung stellt das Domänenmodell der Software GIT dar.

Das Repository ist die Hauptklasse unseres Modells. Es kann ein Repository auf einem Server existieren, von dem beliebig viele lokale Repositories geklont werden können (siehe Clone Repository - Zustandsdiagramme). Zu jedem Repository gehören mindestens ein oder mehrere Branches und beliebig viele Changesets. Branches und Changesets sind Knoten. Changesets benötigen immer einen Vorgängerknoten somit gilt, dass ein Branch immer zuerst existieren muss. Auf einen Knoten können beliebig viele Branches folgen. Branches verzweigen also die History. Jedes Changeset beinhaltet mindestens eine Modifikation. Es gibt mehrere Arten von Modifikationen. Eine FileModification beschreibt eine zusammenhängende Textpassage (durch die Offsets definiert) in einer Datei die geändert wurde. Außerdem können Dateien erzeugt oder gelöscht werden. Wenn ein File verschoben wird, entspricht das intern der Löschung und Neuerzeugung des Files.

3.3 Ausgewählte Zustandsdiagramme

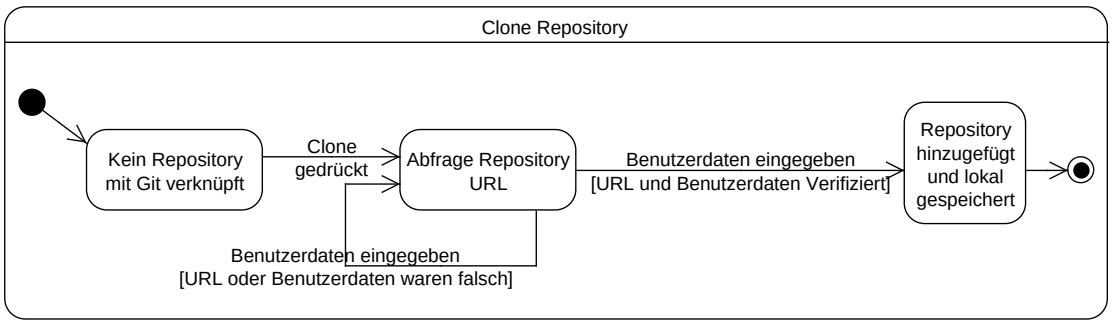
Um das Verhalten eines Programms im laufenden Betrieb zu untersuchen, bedient man sich sogenannter **Zustandsdiagramme**
Zustandsdiagramme bieten ein geeignetes Vokabular um das Ein- und Ausgabeverhalten eines Programms kompakt und präzise zu beschreiben. Da die Lebenszyklen der Objekte in Git komplex sind wurde hier ausgewählte Zustandsdiagramme dargestellt.

Push und Commit



Die Möglichkeit zu commiten gibt es nur, wenn Änderungen vorliegen. Ansonsten wird eine Fehlermeldung ausgegeben, die den Nutzer darauf aufmerksam macht, dass es nichts zu commiten gibt. Wurden Dateien oder Dokumente verändert, wird im SmartGit Fenster angezeigt, welche Dateien verändert wurden.
Der Nutzer hat in jedem Zustand die Möglichkeit, den Push-Button zu betätigen, allerdings mit unterschiedlichem Effekt:
Wurden die Änderungen noch nicht committed werden sie auch nicht auf den Server übertragen. Es gibt nun den Fall, dass Änderungen vorgenommen und committed wurden, ohne das sie gepusht wurden. Wenn der Nutzer nun weitere Änderung vornimmt, hat er die Möglichkeit die vorhergegangenen Änderungen auf den Server zu pushen, ohne dass die aktuellen Änderungen mit übernommen werden.

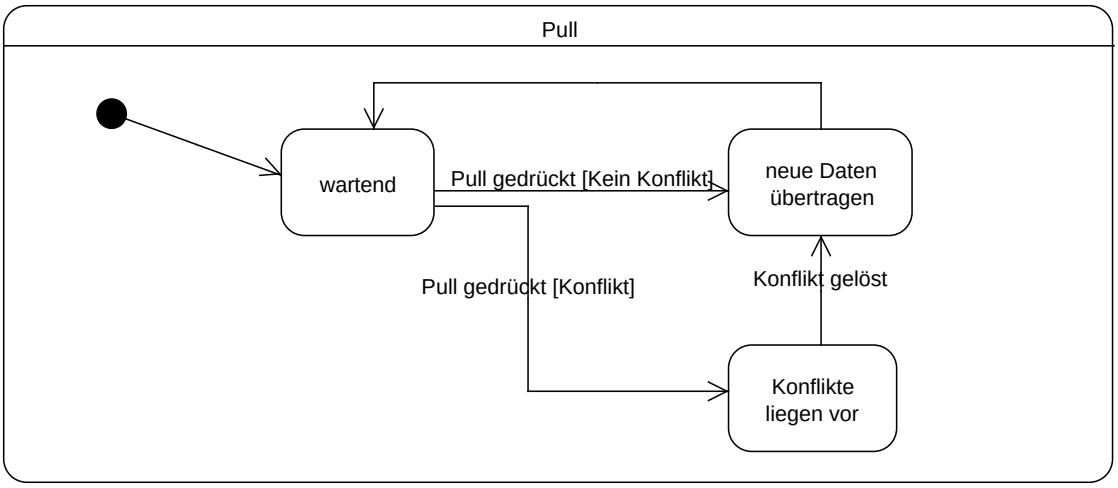
Clone Repository



Zu Beginn eines Projektes kann der Anwender ein existierendes Repository zu seinem Projekt hinzufügen. Dafür benötigt er die genaue Web-Adresse eines bereits existierenden Repositorys. Diese wird dann vom System validiert. Bei falscher Eingabe gibt es eine Fehlermeldung und der Nutzer gelangt zurück in das Abfragemenü. Bei erfolgreicher Anmeldung wird das Repository lokal auf dem Computer gespeichert.

Alternative kann auch ein lokales leeres Repository initialisiert werden.

Pull



Zu jedem vom Nutzer gewählten Zeitpunkt hat er die Möglichkeit, sein lokales Repository mit dem externen Repository abzugleichen.
Dafür wählt er die Option Pull, die alle seit dem letzten Abgleich veränderten Daten übernimmt. Ein Konflikt tritt auf, wenn Änderungen an der gleichen Stelle vorgenommen wurden. Der Nutzer muss dann den Konflikt lösen in dem er dem Programm sagt welche Änderung übernommen werden soll.

3.4 Glossar

Branch engl. Zweig - Ein Branch ist ein Entwicklungszweig in der Geschichte. 2

Changeset In einem Changeset werden mehrere Änderungen zu einer Einheit zusammengefasst. 9
Clone engl. kopieren - Ein entferntes Repository wird auf einen lokalen Rechner übertragen. 4
Commit engl. übergeben - Mit diesem Befehl werden lokale Änderungen im lokalen Repository gespeichert. 6

Merge engl. zusammenführen - Vereinigung von Branches. 2

Pull engl. ziehen - Pull lädt alle Änderungen vom externen Repository. 6
Push engl. anschieben - Dieser Befehl überträgt die lokalen Änderungen auf den Server. 6

Repository engl. Lager, Depot - Repositorium enthält die Geschichte eines Projekts. Jeder Nutzer hat ein eigenes Repository mit der gesamten Geschichte. 2

Switch engl. wechseln - Switch ermöglicht das Wechseln auf einen anderen Branch (arbeitet man z.B. auf einem seperaten Branch, könnte man mit Switch zurück zum Master Branch wechseln). 4

4 Erfahrung aus der Teamarbeit

Zu Beginn unseres Projektes haben wir uns dazu entschieden GIT für die Versionsverwaltung und den Austausch zu nutzen. Da die üblichen Textverwaltungsprogramme ihren Inhalt nicht als Klartext speichern, welches eine Voraussetzung für GIT ist, haben wir uns dazu entschieden LaTeX zu nutzen. Somit hat uns GIT beste Möglichkeiten geboten gemeinsam an dem Projekt zu arbeiten. Die Versionsverwaltung und das Austauschen der Daten ermöglichten uns auch in Zweiergruppen an den einzelnen Punkten zu arbeiten. Das gemeinschaftliche Zusammenfügen der Daten lief nicht immer problemlos, somit hatten wir die Möglichkeit uns intensiv mit der Software auseinander zu setzen. Wie bereits aus anderen Gruppenarbeiten zu erwarten lief die Teamarbeit problemlos ab. Die Teamarbeit lässt sich alles in allem als erfolgreich betrachten. Auch wenn es nicht immer einfach war ein gemeinsames Treffen zu koordinieren.

5 Schlussbetrachtung

Zusammenfassend lässt sich sagen, dass diese Projektarbeit eine gute Möglichkeit war sich mit GIT und mit den Werkzeugen des Software Engineerings auseinander zu setzen. Es ist allerdings zu bedenken, dass man eine gewisse Einbearbeitungszeit benötigt um mit GIT umgehen zu können. Des Weiteren haben wir dieses Projekt dazu genutzt uns mit LaTeX näher zu beschäftigen. Es hat uns fasziniert die Möglichkeiten von LaTeX zu entdecken, auch wenn es hier und da einige Probleme gab.