Rechteverwaltung in Git innerhalb eines Repository

Abschlussprojekt zum Fachinformatiker für Systemintegration

Martin Heckel

Prüfungsperiode: Sommer 2019

Ausbildungsbetrieb: GK Software SE

Projektbetreuer: Michael Saalfrank

Inhaltsverzeichnis

1 Vorwort

Nach dem erfolgreichen Abschluss der Sekundarstufe 2 mit dem Abitur im Mai 2016 begann ich im August 2016 eine duale Ausbildung zum Fachinformatiker für Systemintegration bei der GK Software AG [1] (heute GK Software SE, im Folgenden kurz GK) in Schöneck. Im Rahmen des Programms „Hochschule Dual“ [2] der Hochschule Hof [3] absolvierte ich die schulische Ausbildung an der Staatlichen Berufsschule I [4] in Bayreuth. Seit dem zweiten Ausbildungsjahr studiere ich neben der Ausbildung Informatik an der Hochschule Hof. Bei GK habe ich im IT-Service in den Teams für Infrastruktur und Client-Management gearbeitet. Momentan arbeite ich im Infrastruktur-Team.

Die Server bei GK werden mit Hilfe von Puppet [5], einer Anwendung zur zentralen Administration von Rechnern, verwaltet. Die Konfiguration von Puppet erfolgt über Konfigurationsdateien, die definieren, wie die Rechner konfiguriert werden sollen. In diesen Dateien ist u.A. beschrieben, welche Pakete auf den Servern installiert werden, welche Konfigurationsdateien welche Inhalte haben sollen, welche Benutzer (bzw. die zugehörigen SSH-Schlüssel) sich per SSH [6] anmelden können, etc. Die Konfigurationsdateien für Puppet werden über ein Git-Repository [7] verwaltet. Das bietet neben den Vorteilen der Versionsverwaltung auch den Vorzug, dass der von GK eingesetzte Git-Server, GitLab [8], mit Hilfe von Pipelines [9] bei jedem Push auf das Repository automatisch Tests durchführt, die die Puppet-Konfiguration auf syntaktische Korrektheit prüfen. Sind diese Tests erfolgreich, wird die Konfiguration automatisch auf den produktiven Branch übertragen und mit Puppet auf die Server angewendet.

Einige Entwickler aus anderen Teams oder Abteilungen agieren als DevOps, d.h. sie schreiben Software, deren Betrieb sie auch selbst betreuen. Dafür ist es notwendig, dass diese Teams die Server auf denen die Dienste installiert werden entsprechend anpassen können. Um die Installation von Software und die Verwaltung der Konfigurationsdateien und Scripte auf diesen Servern dennoch über Puppet zu steuern und die damit verbundenen Vorteile nutzen zu können, schreiben die Entwickler für jede gewünschte Anpassung ein Ticket. Im Fall von Scripten oder Konfigurationsdateien werden die Inhalte dieser Dateien an das Ticket angehängt. Dieses Ticket wird dann von einem Mitarbeiter der Infrastruktur-Teams bearbeitet, der die angeforderten Veränderungen am Puppet-Repository durchführt.

Es ist sicherheitskritisch, den Entwicklern Zugriff auf das gesamte Repository zu geben, da in diesem Repository alle Konfigurationsdateien für Puppet

verwaltet werden. Teilweise enthalten diese Dateien Informationen, z.B. Passwörter, die Mitarbeiter von anderen Teams nicht lesen dürfen. Außerdem hätten die Mitarbeiter externer Teams damit die Möglichkeit, die Konfiguration der meisten Server bei GK zu verändern. Da die Konfiguration von Puppet nur getestet und angewendet werden kann, wenn sie vollständig ist, ist es auch nicht möglich, einzelne Repositories für die Entwickler einzurichten, die unabhängig von dem Puppet-Repository sind.

Der Prozess, für jeden Änderungswunsch ein Ticket zu schreiben ist aufwändig, langsam und fehleranfällig. Im Rahmen dieser Projektarbeit soll ein Weg gefunden und umgesetzt werden, den Zugriff auf ein GitLab-Repository auf Basis von Dateien regulieren zu können, um den Entwicklern die Möglichkeit zu geben, ihre Puppet-Konfigurationsdateien selbst zu verwalten. Dadurch könnten die Entwickler die Konfiguration ihrer Server anpassen ohne auf das Infrastruktur-Team warten zu müssen.

Befehle, die in die Kommandozeile eingegeben wurden, sind durch eine graue Box ohne Rand hervorgehoben. Aus Gründen der Übersichtlichkeit wurden an einigen Stellen Zeilenumbrüche eingefügt die in der ursprünglichen Datei oder dem ursprünglichen Befehl nicht vorhanden waren. Solche Zeilenumbrüche sind daran zu erkennen dass sie mit einem „\“ (Backslash) enden:

```
echo "Das ist ein Befehl, der auf der Kommandozeile ausgefuehrt wird."
echo "Das ist ein Befehl, der auf der Kommandozeile ausgefuehrt wird \
und so lang ist, dass er umgebrochen wurde."
```

Das gesamte Projekt wird mit dem Betriebssystem GNU/Linux umgesetzt.

2 Planung

2.1 Ist Analyse

Die Infrastruktur bei GK wird, soweit das möglich ist, über Puppet verwaltet. Das bietet die Möglichkeit, die gesamte Infrastruktur an einer Stelle zu definieren. Puppet kümmert sich dann darum, die Infrastruktur so anzupassen, dass sie der Konfiguration entspricht. Dieses Vorgehen spart viel Zeit und verringert die Wahrscheinlichkeit des Auftretens von Fehlern bei der Installation und Konfiguration der Server. Die Konfigurationsdateien für Puppet werden in einem Repository auf einem GitLab-Server verwaltet. Das bietet den Vorteil, dass die Konfiguration mit Hilfe von GitLab-Pipelines automatisch getestet und ausgerollt werden kann. Git ist eine Anwendung zur Versionsverwaltung. Dabei ist ein Git-Repository immer lokal, d.h. nur auf dem Rechner des Mitarbeiters. Es ist allerdings möglich, das Repository mit einem Git-Server zu synchronisieren. Aus diesem Grund ist es notwendig, eine Änderung zuerst zum lokalen Repository hinzuzufügen („commit“) und dieses dann auf den Server hochzuladen („push“). Außerdem kann der Stand vom Server in das lokale Repository geladen werden („pull“).

Die Pipelines von GitLab werden von separaten Servern, den GitLab-Runnern, ausgeführt. Diese Runner clonen dabei das Repository vom Server und führen die in der Datei .gitlab-ci.yml definierten Anweisungen aus. Wenn dabei das Repository verändert wird, müssen die Änderungen zurück zum Server gepusht werden. Diese Tatsache wird im Folgenden vereinfacht (es wird so dargestellt als ob die Pipelines direkt auf dem GitLab-Server laufen würden, das Pullen und Pushen wird also vernachlässigt). In der folgenden Abbildung ist der aktuelle Ablauf dargestellt:

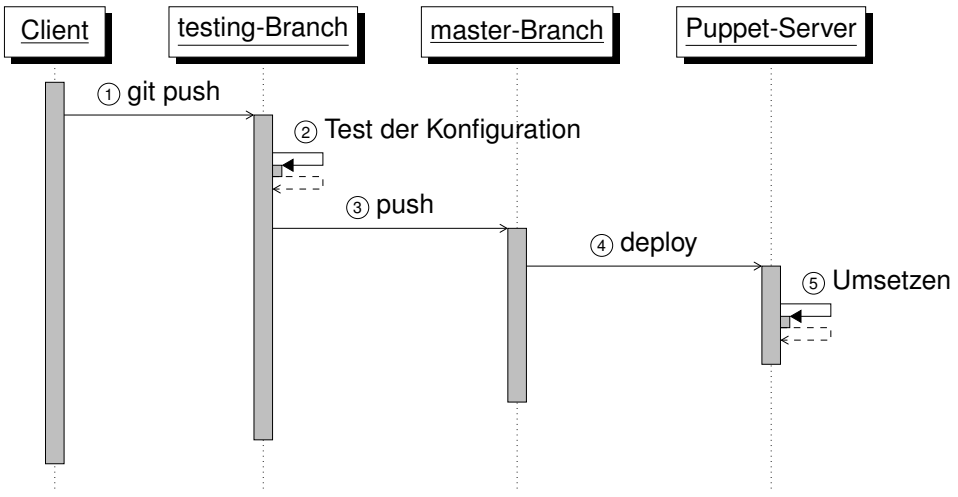

Abbildung 1: Workflow der Pipeline des Puppet-Repository

Pusht ein Mitarbeiter des Infrastruktur-Teams eine Änderung am Puppet-Repository auf dem GitLab-Server ①, wird die Konfiguration automatisch durch Puppet auf syntaktische Korrektheit getestet ②. Ist die Konfiguration gültig, wird sie automatisch auf den produktiven Branch gepusht ③, auf den Puppet-Server deployed ④ und von Puppet umgesetzt ⑤. Durch diesen Mechanismus ist es nicht möglich, dass Puppet eine syntaktisch fehlerhafte Konfiguration bekommt. Mitarbeiter des Infrastruktur-Teams müssen bei Anpassungen also nur einen Commit in das entsprechende Repository pushen, die weiteren Schritte erfolgen automatisiert. Durch diese Vorgehensweise werden Fehler vermieden und Zeit gespart. Zusätzlich kann der zeitliche Verlauf der Konfiguration durch die Versionierung nachvollzogen und ggf. eine bestimmte Version wiederhergestellt werden.

Da die DevOps bei GK nicht auf das Puppet-Repository zugreifen dürfen, benötigen sie die Unterstützung des Infrastruktur-Teams, um gewünschte Änderungen umzusetzen. Der aktuell verwendete Arbeitsablauf wird in der folgenden Abbildung dargestellt. Der in Abbildung 1 bereits beschriebene Teil des Ablaufs wurde aus Gründen der Übersichtlichkeit stark vereinfacht:

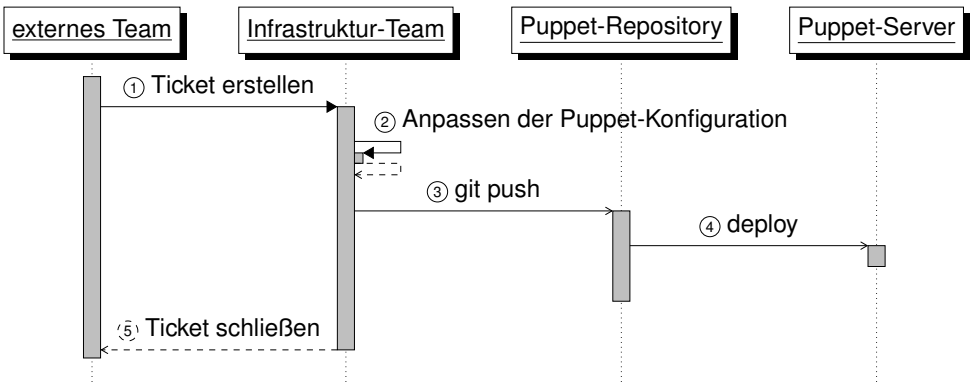

Abbildung 2: Prozess zum Anpassen der Konfiguration

Momentan schreiben die Mitarbeiter der externen Teams ein Ticket mit einer Installationsanfrage, einem Script oder einer Konfigurationsdatei ①. Der Mitarbeiter des Infrastruktur-Teams, der das Ticket bearbeitet, führt die gewünschte Änderung am Puppet-Repository aus ② und pusht die Änderung auf den GitLab-Server ③. Im Anschluss wird die Konfiguration automatisch angewendet ④ und das Ticket geschlossen ⑤. Diese Schritte kosten sowohl die Mitarbeiter der externen Teams als auch die Mitarbeiter des Infrastruktur-Teams Zeit. Unter Umständen werden externe Teams unnötig ausgebremst, da ihre Änderungen noch nicht zum Repository hinzugefügt wurde. Ist eine von dem externen Team geschickten Datei fehlerhaft, ist die Fehlerbehebung durch das verzögerte Feedback möglicherweise sehr zeitaufwändig. Außer-

dem kann das manuelle Kopieren der Dateien für Fehler sorgen, die durch automatisiertes Kopieren verhindert werden könnten.

2.2 Soll Analyse

Um die Vorteile der zentralen Verwaltung durch Puppet und der Versionsverwaltung und automatischen Tests durch GitLab nicht zu verlieren, den Prozess aber trotzdem zu vereinfachen, wird eine Lösung angestrebt, um den Mitarbeitern der externen Teams Zugriff auf Teile des Puppet-Repositories zu geben, sodass diese ihre eigenen Dienste selbst anpassen können. Dabei soll unbedingt verhindert werden, dass diese Mitarbeiter Dateien außerhalb ihres abgegrenzten Bereiches lesen oder ändern können.

Entsprechend sollen die Mitarbeiter der externen Teams in ihren eigenen Bereichen Konfigurationen hinzufügen und anpassen ohne Zugriff auf das Repository mit der globalen Konfiguration zu benötigen. Durch dieses Vorgehen müssen die externen Teams nicht auf das Infrastruktur-Team warten und können eigene Anpassungen sehr schnell umsetzen. Die Mitarbeiter des Infrastruktur-Teams müssen sich nicht mehr um das Anpassen der Konfigurationen externer Teams kümmern und können in der gesparten Zeit an anderen Aufgaben arbeiten. Das spart für alle beteiligten Teams Zeit und senkt die Wahrscheinlichkeit von Fehlern.

2.3 Vorüberlegungen

Prinzipiell gibt es zwei mögliche Ansätze um das Problem zu lösen: Man kann versuchen, die Zugriffskontrolle mit GitLab zu realisieren, wofür GitLab allerdings über entsprechende Funktionen verfügen muss. Alternativ kann man die Rechteverwaltung eines Git-Servers verwenden. Damit lassen sich Rechte allerdings nur auf der Ebene eines Repository konfigurieren. Es wäre also notwendig, mehrere Repositories anzulegen und diese dann zu synchronisieren.

Nach einiger Recherche war sicher, dass GitLab in der zum Zeitpunkt der Arbeit bei GK verwendeten Version 11.7.5 über keine Möglichkeiten zum Verwalten von Berechtigungen auf Dateiebene innerhalb eines Repositories verfügt. Entsprechend wurde es notwendig, eine Lösung für die Synchronisation mehrerer Repositories zu finden. Dafür gibt es wiederum zwei Ansätze: Submodule [10] und Subtrees [11].

Submodule verwalten andere Repositories in Ordnern innerhalb eines Re-

pository. Dabei „kennt“ das übergeordnete Repository nur den Ordner und „weiß“, dass der Inhalt dieses Ordners von einem anderen Repository verwaltet wird. Entsprechend wird auch nur der Ordner und nicht der Inhalt des Ordners versioniert. Das bringt den Nachteil mit sich, dass Mitarbeiter, die am Haupt-Repository arbeiten, immer darauf achten müssen, ob sie etwas am Haupt-Repository oder in einem Submodule verändert haben. Wenn die Mitarbeiter dabei so vorgehen wie sie es gewohnt sind, kommt es zu unerwarteten und unschönen Effekten. Aktualisiert z.B. ein Mitarbeiter eine Datei im Submodul und pusht die Änderung zum übergeordneten Repository, vergisst aber, die Änderung zum Subrepository zu pushen, können andere Mitarbeiter das Submodul nicht in der im übergeordneten Repository angegebenen Version pullen. Da die Lösung den Prozess vereinfachen soll, scheidet diese Möglichkeit also aus.

Subtrees verwalten andere Repositories in Ordnern innerhalb eines Repository. Im Gegensatz zu Submodules werden bei Subtrees allerdings alle Dateien des Ordners, in dem das andere Repository liegt, auch zum übergeordneten Repository hinzugefügt. Dadurch ist es möglich, auf dem übergeordneten Repository mit den gewohnten Git-Befehlen [12] zu arbeiten ohne zusätzliche Dinge beachten zu müssen. Das reduziert die Wahrscheinlichkeit für unschöne Effekte sehr stark. Mit Hilfe von Subtrees ist es außerdem möglich, vom untergeordneten Repository zu pullen oder Änderungen an dieses Repository zu pushen.

Um das Pushen und Pullen der Subtrees zu automatisieren, können die Pipelines in GitLab verwendet werden. Ein möglicher Workflow könnte wie Folgt aussehen:

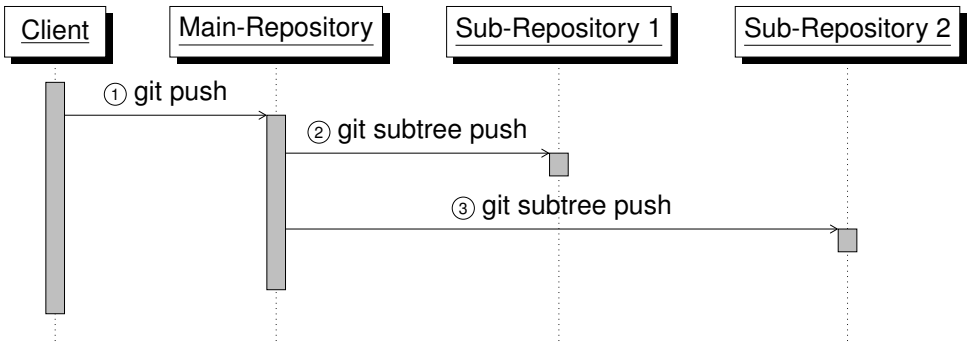

Abbildung 3: Pushen auf übergeordnetes Repository

Wenn von einem Client (z.B. dem Rechner eines Mitarbeiters des Infrastruktur-Teams) eine Änderung auf das übergeordnete Repository gepusht wird ①, wird von GitLab eine Pipeline ausgeführt, die alle Sub-Repositories aktua-

lisiert. Entsprechend wird **git subtree push** auf alle, in diesem Fall Sub-Repository 1 ② und Sub-Repository 2 ③, untergeordneten Repositories ausgeführt.

Ein Workflow für das Pushen auf ein untergeordnetes Repository könnte wie Folgt aussehen:

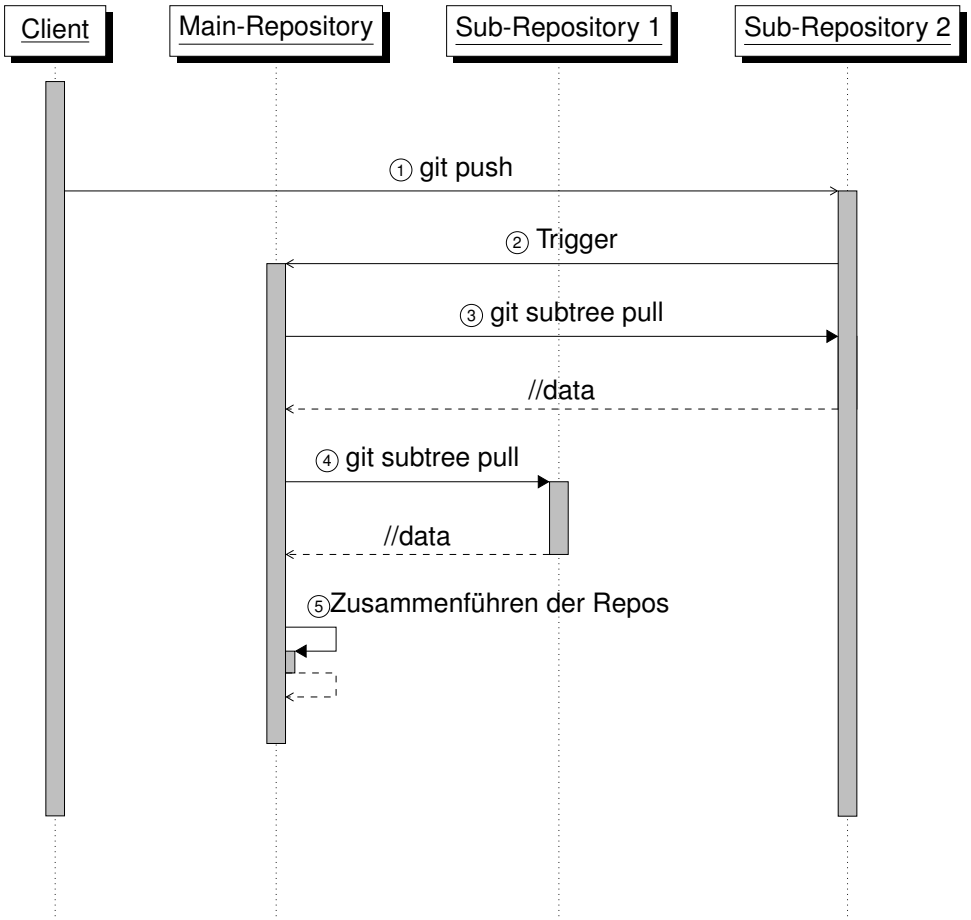

Abbildung 4: Pushen auf untergeordnetes Repository

Wenn von einem Client (z.B. dem Rechner eines Mitarbeiters eines externen Teams) eine Änderung auf ein untergeordnetes Repository gepusht wird ①, wird von GitLab eine Pipeline ausgeführt, die eine Pipeline im übergeordneten Repository triggert ②. Diese pullt von allen Subtrees ③, ④. Dabei kann das Problem auftreten, dass es zu einem von Git nicht behebbaren Merge-Konflikt kommt, wenn eine Datei sowohl im übergeordneten als auch im untergeordneten Repository verändert wird und die Änderungen (fast) gleichzeitig gepusht werden. Das Problem tritt bei „normalen“ Repositories nicht in dieser Form auf, da GitLab das Pushen serialisiert, d.h. erst den ersten und dann den zweiten Push durchführt. Der zweite Push würde in diesem Fall fehlschlagen und der Benutzer müsste die Konflikte vor dem Pushen manuell beheben. Da es sich hier allerdings um verschiedene Repo-

sitories handelt, haben beide Pushes auf die Repositories funktioniert, da es ja keinen Konflikt gab. Nun versucht das übergeordnete Repository, auf das untergeordnete Repository zu pushen, was aufgrund des Konfliktes nicht funktioniert. Das übergeordnete Repository kann auch nicht automatisch vom untergeordneten Repository pullen, da auch in diesem Fall ein Merge-Konflikt auftritt, der manuell behoben werden muss. Ein solcher Konflikt kann von dieser Lösung nicht automatisch behoben werden. Ein Mitarbeiter des Infrastruktur-Teams muss den Konflikt manuell beheben.

Puppet erfordert eine bestimmte Verzeichnisstruktur, in der die Konfigurationsdateien liegen müssen. Entsprechend ist es notwendig, die Subtrees in diese Verzeichnisstruktur zu integrieren. Da Subtrees erfordern, in einem separaten Verzeichnis zu liegen, ist es nicht möglich, die Dateien direkt an die notwendigen Stellen zu verschieben. Allerdings besteht die Möglichkeit, symbolische Links („Symlinks“) zu verwenden. Dabei wird im für Puppet relevanten Verzeichnis ein Link auf die entsprechende Datei im Unterverzeichnis des Subtrees hinzugefügt. Durch ein Script ist es möglich, die Ordnerstrukturen der Subtrees automatisch mit der Ordnerstruktur des übergeordneten Repository zu vereinigen ⑤. Das Script erkennt die Ordnerstrukturen der Subtrees, legt entsprechende Ordner (falls noch nicht vorhanden) im Hauptverzeichnis an und erstellt daraufhin die Symlinks.

Dieses Script kann zur Pipeline hinzugefügt werden, sodass zuerst alle Subtrees gepullt und daraufhin die Verzeichnisstrukturen vereinigt werden.

2.4 Testphase

Bevor der oben beschriebene Ablauf auf das produktive Repository angewendet werden kann, ist es notwendig, den Workflow zu testen. Aus diesem Grund wurde eine Testumgebung mit einem übergeordneten Repository „main“ und zwei untergeordneten Repositories „sub1“ und „sub2“ angelegt. Das Repository **main** hat folgende Verzeichnisstruktur:

```
./
 | Code
   |>
 | external
   | sub1
     |>
   | sub2
     |>
 | .gitlab-ci.yml
 | link.sh
 | pull_subtrees.sh
 | push_subtrees.sh
 | subtrees.conf
```

Die Verzeichnisstrukturen der untergeordneten Repositories sind identisch:

```
./
 | Code
   |>
 | .gitlab-ci.yml
 | Mails.conf
 | sendMails.sh
```

In dieser Phase wurden die an diese Projektdokumentation angehängten Scripte und Konfigurationsdateien geschrieben und am Ende dieser Phase an die Verzeichnisstruktur des Puppet-Repository angepasst. Im Fogenden werden die an der Testumgebung durchgeführten Tests beschrieben:

- Anlegen einer Datei in **sub1**

  Im Verzeichnis **Code** des Repository sub1 wird eine Datei mit dem Namen „Hello.c“ mit folgenden Inhalt angelegt:

  ```c
  #include<stdio.h>

  int main (int argc, const char *argv[]) {
      printf("Hello World!");
  }
  ```

  Anschließend wird diese Datei zum Repository hinzugefügt, die Änderung committet und das lokale Repository auf den GitLab-Server gepusht. Nach dem Ablauf der in 2.3 beschriebenen Schritte ist zu erwarten, dass die Datei **Hello.c** im Repository main unter external/-sub1/Code liegt, was auch der Fall ist. Außerdem gibt es in main einem Symlink von **Code/Hello.c** nach **../external/sub1/Code/hello.c**.

- Bearbeiten einer Datei in **main**, die in **sub1** erstellt wurde

  Im vorherigen Schritt wurde die Datei **Hello.c** im Verzeichnis **Code** angelegt. Nun wird diese Datei im main-Repository bearbeitet. Dazu wird die Datei wie Folgt angepasst:

  ```c
  #include<stdio.h>

  int main (int argc, const char *argv[]) {
      printf("Hello new World!");
  }
  ```

  Im Anschluss wird die Änderung committet und zum GitLab-Server gepusht. Nach dem Ablauf der Pipelines befindet sich auch in sub1 die aktualisierte Version der Datei **Hello.c**.

- Erzeugen eines Merge-Konflikts zwischen **main** und **sub1**

  Um den Konflikt zu erzeugen, wird die Datei **Hello.c** im Ordner **Code** auf beiden Repositories verschieden verändert:

  main:

```
#include<stdio.h>

int main (int argc, const char *argv[]) {
    printf("Hello my new World!");
}
```

  sub1:

```
#include<stdio.h>

int main (int argc, const char *argv[]) {
    printf("Hello your new World!");
}
```

  Anschließend werden die Änderungen in den jeweiligen Repositories committet. Nun wird direkt nacheinander auf main und sub1 gepusht (zeitliche Differenz ca. 1s). In GitLab schlagen beide Pipelines wie zu erwarten fehl. Nun muss der aufgetretene Konflikt behoben werden. Dazu wechselt man auf dem Client in das Main-Repository. Nun pullt man (falls noch nicht geschehen) den aktuellen Stand vom GitLab-Server:

```
git pull
```

  Anschließend pullt man den Subtree, bei dem der Merge fehlgeschlagen ist (in diesem Fall sub1):

```
git subtree pull --prefix external/sub1 <subrepo-clone-resource> \
master -m "Pulled Sub1"
```

  Dabei wird **<subrepo-clone-resource >** durch die Resource auf dem GitLab-Server ersetzt (z.B. git@gitlab.example.com:user/sub1). Dieser pull gibt nun die Meldung aus, dass der automatische Merge der Datei **Hello.c** fehlgeschlagen ist und der aufgetretene Konflikt manuell behoben werden muss.

  Nun wird die Datei mit einem Editor bearbeitet und der Konflikt manuell behoben. Im Anschluss werden die Änderungen committet und auf den GitLab-Server gepusht:

```
git commit -am "Solved conflicts"
git push
```

- Erzeugen eines Konflikts beim Zusammenführen der Verzeichnisse zwischen **main** und **sub1**

  Um diesen Konflikt zu erzeugen, wird die Datei **Test.c** im Verzeichnis **Code** des Main-Repository angelegt. Da der Inhalt der Datei für diesen Konflikt nicht relevant ist, reicht es aus, eine leere Datei anzulegen:

  ```
  touch Code/Test.c
  ```

  Anschließend wird die Datei zum Repository hinzugefügt. Die Änderungen werden committet und auf den GitLab-Server gepusht.

  Nun wird die Datei **Test.c** im Verzeichnis **Code** des Repository sub1 analog zum main-Repository angelegt. Darauf hin wird die Datei zum Repository hinzugefügt. Im Anschluss werden die Änderungen committet und zum GitLab-Server gepusht. Die durch den Push getriggerte Pipeline startet das Script zum Zusammenführen der Verzeichnisse. Da es einen Konflikt gibt (die Datei Code/Test.c existiert bereits im Main-Repository), schlägt die Pipeline fehl.

  Der Konflikt kann durch das Umbenennen der Datei in sub1 aufgelöst werden:

  ```
  mv Code/Test.c Code/Test1.c
  ```

  Nun werden die Änderungen wieder committet und auf den GitLab-Server gepusht. Im Anschluss laufen alle getriggerten Pipelines wieder erfolgreich. Der Konflikt ist behoben.

3 Durchführung

3.1 Voraussetzungen

Zur Umsetzung ist es notwendig, eine GitLab-Instanz und einen GitLab-Runner zum Ausführen der Pipelines zu haben. Außerdem sollte ein Client mit Git zur Verfügung stehen, um die Subtrees initial anzulegen. Das Hauptrepository wird bereits genutzt und muss aus diesem Grund nicht angelegt werden.

3.2 Synchronisation der Repositories

Am Anfang muss für jedes Team ein Repository in GitLab angelegt werden. Dafür klickt man auf dem GitLab-Dashboard auf „New Project“. Im Anschluss

wird ein Name für das Projekt vergeben. Im Rahmen dieser Projektarbeit wurden folgende Repositories angelegt:

- puppet-am-team
- puppet-mca-devops
- puppet-orc-devops

Diese Namen wurden vom Betreuer des Projekts vorgegeben.

Die im Folgenden benannten Dateien befinden sich im Anhang dieser Projektdokumentation [13]. Anschließend klont man diese drei Repositories und das Haupt-Repository auf einen lokalen Rechner. Nun kopiert man die **.gitlab-ci.yml** eines untergeordneten Repository sowie die **Mails.conf** und **sendMails.sh** in die jeweiligen untergeordneten Repositories. Außerdem werden in jedem Subrepository die Ordner **manifests** und **modules** angelegt. In den Ordner **manifests** wird die Datei **example.pp** kopiert. Im Ordner **modules** wird das Verzeichnis **example** mit den Unterverzeichnissen **files**, **manifests** und **templates** angelegt. Daraufhin wird die Datei **exampleScript.sh** in **files**, die Datei **init.pp** in **manifests** und die Datei **exampleTemplate.erb** in **templates** kopiert.

In der **.gitlab-ci.yml** sind die Pipelines definiert. Die Pipelines der untergeordneten Repositories verfügen über jeweils zwei Jobs: einen Job, der beim Pushen auf das Verzeichnis aufgerufen wird und dann wiederrum eine Pipeline im Haupt-Repository triggert. Der andere Job wird im Fall eines Mergekonflikts von einer Pipeline des Haupt-Repository getriggert. Dies ist notwendig, da die Aufrufe der Trigger über die API durch einen einfachen HTTPS-Aufruf realisiert werden und entsprechend Fire-and-Forget sind, da der API-Aufruf nicht wartet bis die getriggerte Pipeline beendet ist. Es gibt also aus Sicht des Subrepository keine Möglichkeit, den Ausgang der Pipeline im Hauptrepository auszuwerten und die Benutzer ggf. auf Fehler hinzuweisen. Da es allerdings notwendig ist, die Benutzer über Fehler zu informieren (es sollte kein Fehler auftreten, über den der Benutzer nicht informiert wird), gibt es den zweiten Bereich, der im Fehlerfall vom übergeordneten Repository getriggert wird. Dadurch bekommt der Benutzer eine aussagekräftige Fehlermeldung.

Durch das Script **sendMails.sh** wird eine Mail mit der Fehlermeldung an alle Mailadressen aus der Datei **Mail.conf** geschickt. Dadurch können die Teams selbst verwalten wer im Fall eines Fehlers informiert wird. Die Ordner **manifests** und **modules** werden für die Verwaltung der Puppet-Konfigurationsdateien benötigt.

Anschließend werden bei GitLab im Main-Repository unter **Settings** → **CI/CD** → **Pipeline Triggers** drei Trigger für die jeweiligen Sub-Repositories angelegt. Nun wird in jedem untergeordneten Repository ein Pipeline-Trigger für das übergeordnete Repository angelegt.

Daraufhin werden die Dateien **push_subtrees.sh**, **pull_subtrees.sh**, **link.sh** und **subtrees.conf** im übergeordneten Repository in den Unterordner **scripts** kopiert. Zusätzlich wird die Datei **.gitlab-ci.yml** des übergeordneten Repository in den Root-Ordner des Haupt-Repository kopiert.

Nun wird im Haupt-Repository der Ordner **subtrees** angelegt, in dem ein Subtree wie Folgt angelegt werden kann:

```
git subtree add --prefix subtrees/<subtree-name> <git-resource> <branch> \
-m "<comment>"
```

Dieser Schritt wird für alle oben aufgelisteten Sub-Repositories ausgeführt. Die Teile in spitzen Klammern sind Platzhalter und werden wie Folgt ersetzt:

- **subtree-name** Name des Sub-Repository
- **git-resource** URL von der das untergeordnete Repository gecloned werden kann (z.B. https://gitlab.example.com/user/project.git)
- **branch** Branch des untergeordneten Repository der gecloned werden soll
- **comment** Kommentar (für die History)

Jetzt können die eben erstellten Subtrees zur Datei **scripts/subtrees.conf** hinzugefügt werden. Dabei wird folgendes Schema verwendet:

```
subtrees/<subtree-name>/;<subrepo-clone-resource>;<subrepo-branch>;\
<subrepo-id>;<subrepo-trigger-token>
```

Die subrepo-id und der subrepo-trigger-token kann aus GitLab unter **Settings** → **CI/CD** → **Pipeline Triggers** kopiert werden. Die Datei **scripts/subtrees.conf** enhält jetzt einen Eintrag für jeden Subtree.

Nun werden alle Dateien des übergeordneten Repository zu Git hinzugefügt und gepusht. Anschließend werden alle Dateien der untergeordneten Repositories zu Git hinzugefügt und gepusht.

3.3 Dynamische Anpassung der Verzeichnisstruktur

Die Funktionsweise der Synchronisation der Repositories wurde bereits im Abschnitt 2 „Planung“ ausführlich beschrieben. Aus diesem Grund wurden in diesem Abschnitt nur die Schritte zur Installation beschrieben. Da das

Script zum dynamischen Linken der Ordnerstrukturen bisher nur oberflächlich beschrieben wurde, wird dies in diesem Abschnitt nachgeholt.

Diese Aufgabe wird von dem Script **link.sh** übernommen, welches sich wie die anderen Scripts auch im Anhang befindet. Prinzipiell werden die Verzeichnisse mit Hilfe von Symlinks zusammengeführt. Dies bietet gegenüber dem Kopieren der Dateien einige Vorteile:

- Die Quelldatei ist immer bekannt, da diese verlinkt ist. Dadurch lässt sich die Quelldatei auch nach einer Veränderung noch eindeutig identifizieren. Wurde eine Datei kopiert und eine Datei verändert, ist dies nicht möglich.
- Wenn die verlinkte Datei bearbeitet wird, wird der Link automatisch aufgelöst und somit direkt an der Quelldatei gearbeitet. Dadurch werden Änderungen immer direkt an der Datei durchgeführt, die geändert werden muss (sonst müsste die veränderte Datei wieder in das Verzeichnis des Subtree kopiert werden). Für einen Mitarbeiter des Infrastruktur-Teams fühlt sich die Bearbeitung also an als würde er direkt mit den Dateien arbeiten. Auch beim Hinzufügen aller Änderungen zum Git-Repository fällt die Verwendung von Symlinks abgesehen von Tools wie **git status** nicht auf. Der Mitarbeiter kann wie gewohnt arbeiten.
- Wenn eine Quelldatei entfernt wird, ist der Symlink „broken“, da die Datei, auf die er verweist, nicht mehr existiert. Broken Symlinks lassen sich mit dem Programm **find** einfach finden und entfernen. Dadurch wird der Prozess des Zusammenführens vereinfacht, da nicht für jede vorhandene Datei geloggt werden muss, woher sie kam und geprüft werden muss, ob die Quelldatei noch vorhanden ist

Wenn das Script gestartet wird, wechselt es in das Verzeichnis **code/environments/production/** und entfernt rekursiv alle toten Links. Daraufhin werden rekursiv alle leeren Ordner entfernt. Anschließend iteriert das Script über alle Subtrees und über die Ordner in diesen Subtrees. Für jede Datei wird versucht, an einer äquivalenten Stelle unter **code/environments/production/** ein Symlink zu erstellen. Schlägt dies Fehl, da bereits eine Datei mit gleichen Namen vorhanden ist, wird geprüft, ob der Symlink (falls es sich um einen Symlink handelt) auf die Datei zeigt, die verlinkt werden soll. Ist dies nicht der Fall, handelt es sich um einen Konflikt, da die anzulegende Datei bereits von einer anderen Quelle (anderer Subtree oder Haupt-Repository) erstellt wurde.

Um die oben beschriebene „Fehler-Pipeline“ des entsprechenden Sub-

Repository zu triggern, werden für den API-Aufruf die Projekt-ID und der Token für die Pipeline benötigt. Diese werden mit Hilfe der Funktion **getProjectDetails** aus der Konfigurationsdatei der Subtrees (**scripts/subtrees.conf**) ausgelesen. Im Anschluss wird die Pipeline des entsprechenden Sub-Repository getriggert. Danach gibt das Script die Fehlermeldung aus und beendet sich mit Fehlercode 1, was zum Fehlschlagen der Pipeline führt.

Durch diesen Mechanismus wissen sowohl die Mitarbeiter des Infrastruktur-Teams als auch die Mitarbeiter der externen Teams, dass ein Fehler aufgetreten ist. Die Mitarbeiter der externen Teams können ihre Datei umbenennen, um den Fehler zu beheben. Wenn das nicht möglich ist, ist es notwendig, dass sich das entsprechende Team mit dem Infrastruktur-Team abspricht, um das Problem zu beheben.

Das Script ignoriert dabei die Datei **example.pp** im Ordner manifests und das Verzeichnis **example** im Ordner modules. Dadurch kann eine Beispielkonfiguration in den Repositories der externen Teams abgelegt werden ohne in die produktive Ordnerstruktur des übergeordneten Repository hinzugefügt zu werden. Mitarbeiter externer Teams können sich beim Schreiben ihrer Konfigurationsdatei an dieser Beispielkonfiguration orientieren.

3.4 Testphase

Vor der Präsentation der Ergebnisse vor Mitarbeitern der anderen Teams wurde testweise ein Modul im Repository eines externen Teams angelegt. Anschließend wurden Tests analog zu den in Abschnitt 2.4 beschriebenen Testfällen durchgeführt.

Im Anschluss daran wurde die Lösung den Mitarbeitern der externen Teams vorgestellt und den externen Teams Zugriff auf die Repositories gegeben. Daraufhin wurde die Lösung von den Mitarbeitern der entsprechenden Teams getestet.

4 Auswertung

4.1 Aufgetretene Probleme

Im Gegensatz zu einem Repository auf einem Server besteht bei verteilten Repositories das Problem, dass es zum gleichzeitigen Pushen auf beide Repositories kommen kann. Dadurch schlagen beide Synchronisations-Zyklen

fehl, wenn die Pushs Änderungen beinhalten, die Git nicht automatisch mergen kann. Dies lässt sich wie folgt begründen:

Wenn das übergeordnete Repository den Subtree pullt, versucht git, die auftretenden Konflikte automatisch zu mergen. Wenn das nicht möglich ist, muss manuell gemerged werden, was durch die Pipeline nicht realisierbar ist.

Versucht das übergeordnete Repository, den Subtree zu pushen, ist dies nicht möglich, da es auf dem untergeordneten Repository bereits einen Commit gibt, der dem übergeordneten Repository nicht bekannt ist. Dieses Problem muss durch einen Pull behoben werden, der aus dem oben beschriebenen Grund fehlschlägt.

Um dieses Problem zu lösen, muss das übergeordnete Repository gepullt werden. Danach müssen der Subtree gepullt und die auftretenden Probleme beim Merge manuell behoben werden. Nun kann der aktuelle Arbeitsstand commited und gepusht werden. Darauf hin wird das entsprechende untergeordnete Repository automatisch durch die Pipeline aktualisiert und das Problem ist behoben.

```
git pull
git subtree pull --prefix subtrees/<subtree-name> <git-resource> <branch> \
-m "<comment>"
# Resolve the merge conflicts
git push
```

Die Teile in spitzen Klammern sind Platzhalter mit gleicher Bedeutung wie in Abschnitt 3.2.

4.2 Nutzen für das Unternehmen

Der vor der Umsetzung dieses Projekts bei GK eingesetzte Prozess zum Anpassen von Serverkonfigurationen durch externe Teams war langsam und verhältnismäßig fehleranfällig. Durch diesen Workflow wurden sowohl durch Mitarbeiter des Infrastruktur-Teams als auch durch Mitarbeitern externer Teams unnötig viel Zeit benötigt. Durch die vollständige Automatisierung dieses Workflows haben die Mitarbeiter der externen Teams nun die Möglichkeit, ihrer Scripte und Konfigurationsdateien selbst zu deployen. Dadurch können Fehler in den Scripten direkt bemerkt und behoben werden, ohne jeweils auf den Deploy warten zu müssen. Entsprechend geht das Beheben von Fehlern an Scripten und Konfigurationsdateien, die über Puppet ausgerollt werden, deutlich schneller. Da der eigentliche Deploy-Vorgang für die Mitarbeiter der externen Teams vereinfacht wurde, können diese ihre Arbeitszeit besser

nutzen. Außerdem fällt die Belastung des Infrastruktur-Teams mit den bisher erstellten Tickets weg, wodurch die betroffenen Mitarbeiter ihrer Arbeitszeit auch effektiver nutzen können.

Im Gegensatz zum bisherigen Prozess können die externen Teams nun selbstständig an der Konfiguration ihrer Server arbeiten. Dadurch wird die Verantwortung vom Infrastruktur-Team auf die externen Teams verlagert. Aus diesem Grund ist das bisher durch den Prozess erzwungene 4-Augen-Prinzip nun optional und muss von den Teams eigentverantwortlich angewandt werden.

Insgesamt wird also von allen beteiligten Teams Zeit gespart und das Ausrollen neuer oder angepasster Dateien geht schneller, wodurch die Dienste schneller auf der aktuellen Version laufen.

5 Schlusswort

Die im Rahmen dieser Projektarbeit gestellte Aufgabe war sehr interessant. Es gab eine Recherchephase mit der Möglichkeit, verschiedene Lösungsansätze zu finden und deren Vor- und Nachteile gegeneinander abzuwägen. Am Ende dieser Phase konnte in Absprache mit dem Auftraggeber, dem Infrastruktur-Team, einer dieser Ansätze zur Umsetzung ausgewählt werden. Dabei war es möglich, die gefundenen Ansätze in einer Testumgebung zu testen und dem Auftraggeber zu demonstrieren. Im Anschluss konnte die gefundene Lösung eigenständig an der Produktivumgebung umgesetzt werden.

Zu Beginn des Projekts beschränkten sich die Git-Erfahrungen des Autors auf grundlegende Kenntnisse (Klonen, Pushen, Pullen, Mergen, Branches). Durch das Projekt wurden die Grundlagen von GitLab CI/CD erlernt und das Wissen zu git um Subtrees erweitert.

Bei diesem Projekt bestand die Möglichkeit, eigenständig zu arbeiten und auch — in Absprache mit dem Auftraggeber — selbst Entscheidungen zu treffen.

6 Referenzen

[1] https://www.gk-software.com

[2] https://www.hof-university.de/studieninteressierte/duales-studienangebot/

[3] https://www.hof-university.de/

[4] https://bs1-bt.de/

[5] https://puppet.com/

[6] https://datatracker.ietf.org/wg/secsh/documents/

[7] https://git-scm.com/

[8] https://about.gitlab.com/

[9] https://docs.gitlab.com/ee/ci/pipelines.html

[10] https://git-scm.com/book/en/v2/Git-Tools-Submodules

[11] https://git-scm.com/book/de/v1/Git-Tools-Subtree-Merging

[12] git clone, git pull, git checkout, git add, git commit, etc.

[13] manpages von bash, curl, find, git, git subtree, mail, rmdir, sed, tr

7 Abbildungsverzeichnis

8 Anlagenverzeichnis

- **.gitlab-ci.yml des übergeordneten Repository**

  Konfiguration der Pipeline des übergeordneten Repository, Wenn die Variable PULL gesetzt ist (dies ist beim Aufruf des Triggers der Fall), werden alle Subtrees gepullt. Sonst (dies ist beim automatischen Aufruf nach dem Pushen auf dieses Repository der Fall) wird auf alle Subtrees gepusht. Im Anhang ist nur der für diese Dokumentation relevante Teil der Datei aufgeführt. Nicht relevante Teile wurden aus Gründen der Übersichtlichkeit weg gelassen.

- **.gitlab-ci.yml des untergeordneten Repository**

  Konfiguration der Pipeline eines untergeordneten Repository, wenn auf dieses Repository gepusht wird, wird die Pipeline des übergeordneten Repository getriggert und die Variable PULL gesetzt.

- **link.sh**

  Dieses Script dient zum automatischen Erzeugen der Symlinks.

- **notifyError.sh**

  Dieses Script dient zum Benachrichtigen eines Sub-Repository über eine Pipeline, wenn die Synchronisation im übergeordneten Repository fehlgeschlagen ist.

- **pull_subtrees.sh**

  Dieses Script sorgt dafür, dass von allen in der Datei subtrees.conf definierten Subtrees gepullt wird. Die tatsächlich verwendete Git-Resource wurde durch **gitlab.example.com:examples/puppet.git** ersetzt.

- **push_subtrees.sh**

  Dieses Script sorgt dafür, dass auf alle in der Datei subtrees.conf definierten Subtrees gepusht wird.

- **sendMails.sh**

  Dieses Script sorgt dafür, dass eine Mail mit der übergebenen Fehlermeldung an alle in Mails.conf definierten Mailadressen geschickt wird.

- **subtrees.conf**

  CSV-Datei, die für jeden Subtree den Pfad (Präfix), die Remote Resource und den Branch spezifiziert. Die tatsächlich verwendeten Git-

Resourcen wurden analog zu pull_subtrees.sh ersetzt.

- **Mails.conf**

  CSV-Datei, die für jede Mailadresse, an die Fehlermeldungen geschickt werden sollen, einen Eintrag enthält.

- **manifests/examples.pp**

  Haupt-Manifest für das Puppet-Beispielmodul

- **modules/example/files/exampleScript.sh**

  Beispiel-Script das durch das Puppet-Beispielmodul auf den Zielrechner kopiert werden würde

- **modules/example/manifests/init.pp**

  Manifest für das Puppet-Beispielmodul

- **modules/example/templates/exampleTemplate.erb**

  Beispiel-Template das durch das Puppet-Beispielmodul auf den Zielrechner angewendet würde

- **Verzeichnisstruktur des Puppet-Repository**
- **Verzeichnisstruktur des untergeordneten Repository**

9 Glossar

- **Puppet**

  Puppet ist eine Anwendung zur zentralen Administration von Servern über ein Netzwerk. Mit Hilfe von Puppet lassen sich z.B. Anwendungen installieren, Dateien synchronisieren oder Anwendungen auf dem Zielsystem ausführen.

- **DevOps**

  DevOps ist ein Kunstwort aus den Begriffen „Development“ (engl. Entwicklung) und „Operation“ (engl. Betrieb). DevOps sind Entwickler, die die von ihnen entwickelte Software auch selbst betreiben. Dabei wird die eigentliche Infrastruktur nach wie vor von Administratoren verwaltet. DevOps installieren z.B. die Abhängigkeiten ihrer Software, konfigurieren Webserver und Datenbanken, etc.

- **Git**

Git ist eine Anwendung zur Versionsverwaltung. Dabei ist ein Git-Repository immer lokal, d.h. nur auf dem Rechner des Mitarbeiters. Es ist allerdings möglich, das Repository mit einem Git-Server zu synchronisieren. Aus diesem Grund ist es notwendig, eine Änderung zuerst zum lokalen Repository hinzuzufügen („commit“) und dieses dann auf den Server hochzuladen („push“).

- **GitLab**

  GitLab ist ein webbasierter Git-Server. Neben den üblichen Funktionalitäten eines Git-Servers bietet GitLab noch einige weitere Funktionen (z.B. CI/CD Pipelines, etc.)

- **GitLab-Pipeline**

  Eine GitLab-Pipeline enthält eine Folge von Jobs, die beim Starten der Pipeline ausgeführt werden. Jeder Job besteht aus Anweisungen, z.B. Kommandozeilen-Befehlen. Zusätzlich besteht die Möglichkeit, bestimmte Jobs nur auszuführen, wenn entsprechende Bedingungen erfüllt sind. Außerdem können Jobs einer Stage (dt. Stufe) zugeordnet werden. Eine Stage wird nur ausgeführt, wenn die vorherige Stage erfolgreich beendet wurde.

- **git submodule**

  Git submodule ist eine Möglichkeit, ein Repository in einem Ordner eines anderen Repository zu verwalten. Da das Submodule allerdings nur als Versionsnummer des Ordners vom übergeordneten Repository verwaltet wird, ist es an vielen Stellen notwendig, dass sich Anwender entsprechend verhalten, da es sonst zu unschönen Effekten kommen kann.

- **git subtree**

  Git subtree ist eine alternative Möglichkeit zu git submodules, um ein Repository in einem Ordner eines anderen Repository zu verwalten. Dabei werden alle Dateien des subtree auch im übergeordneten Repository verwaltet, wodurch die Anwender die üblichen Git-Befehle verwenden können. Allerdings ist es zur Synchronisation mit dem untergeordneten Repository notwendig, auf dieses zu pushen oder von diesem zu pullen.

- **ssh**

  SSH (secure shell) ist eine Anwendung zur verschlüsselten Remote-Verwaltung von Rechnern. Ähnlich wie bei Telnet bekommt der Benutzer

nach Anmeldung eine Shell auf dem remote System. Im Gegensatz zu SSH ist Telnet allerdings nicht verschlüsselt.

- **Symlink**

  Ein Symlink (symbolischer Link) ist eine Datei vom Typ „Link“, die einen Verweis auf eine andere Datei enthält. Wird dieser Link (als Datei) bearbeitet, wird der Link automatisch aufgelöst, wodurch praktisch die verlinkte Datei und nicht der Link bearbeitet wird.

10 Eidesstattliche Erklärung

Ich versichere durch meine Unterschrift, dass ich diese Projektarbeit mit dem Thema „Rechteverwaltung in Git innerhalb eines Repository“ selbst-ständig, ohne fremde Hilfe angefertigt, alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen, als solche kenntlich gemacht und mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Projektarbeit hat in dieser oder ähnlicher Form weder der Industrie- und Handelskammer Chemnitz noch einer anderen Prüfungsinstitution vorgelegen. Mir ist bekannt, dass gemäß der Prüfungsordnung für die Durchführung von Abschlussprüfungen der Industrie- und Handelskammer Chemnitz Täuschungshandlungen zum Ausschluss von der Prüfung führen können und die Prüfung als nicht bestanden erklärt werden kann.

Ort, Datum	Unterschrift