**Protokoll Übung 3**

Gruppe: **Schwabakla**

– Robert Dziuba, Tobias Klatt & Inga Schwarze

1. **Git-Basics**

Wir schreiben unser eigenes kleines Git-Handbuch.

*Was sind die Grundprinzipien der Versionskontrolle?*

Versionskontrollsysteme (VCS) protokollieren Änderungen an einer Datei oder einer Anzahl von Dateien über die Zeit hinweg, so dass man zu jedem Zeitpunkt auf Versionen und Änderungen zugreifen kann. In unserem Fall ist das sehr nützlich, da alle am selben Quellcode arbeiten können und Änderungen für alle gültig sind. Durch das Entstehen einer Dateihistorie kann immer auf die aktuellste, aber auch auf ältere Versionen zurückgegriffen werden.

*Versionskontrolle an einem Beispiel*:

Es arbeiten zwei Programmierer an einem Projekt mit Versionskontrolle. Verändert einer der Programmierer eine Datei so wird, nach einen Update, diese Änderung auch beim zweiten Programmierer sichtbar. Die Versionskontrolle erkennt dabei vollautomatisch alle Änderungen und versucht diese zusammenzuführen. Jede Änderung wird in einer sogenannten Dateihistorie gespeichert und ist dort Unveränderbar gespeichert (nur der Teil in der Dateihistorie, nicht die reale Datei!!!!).

Die Versionskontrolle ermöglicht es den beiden außerdem zu jeder Änderungversion des Programms „zurückzuspringen“ und sich den Code zu diesem Zeitpunkt anzuschauen und/ oder zu ändern.

*Wissenswert*: Im Prinzip ist eine Versionskontrolle mit allen Dateiformaten möglich. Selbst Dateien die ein Binär-Dateiformat haben z.B. Photoshop-Dateien und somit nicht direkt editierbar sind (Mit Editoren sind diese nicht lesbar).

**Merging:** Den gerade beschriebenen Prozess des Zusammenführens nennt man Merging. Hierbei versucht die Versionskontrolle Änderung vollautomatische in das Projekt zu integrieren. Sollte die Zusammenführung nicht konfliktfrei auszuführen sein, wird der Programmierer mit dem letzten Änderungsversuch benachrichtigt und muss seine Änderungen „per Hand“ hinzufügen. z.B Programmierer A und B ändern Datei A und versuchen diese Datei dann mit der Versionskontrolle zu mergen.

**Branching:** Zu Deutsch bedeutet Branch Zweig. In GIT wird mit einen Verästelungssystem gearbeitet. Wenn nichts weiter angegeben, haben wir nur einen Ast namens master. Auf diesem Ast liegen unsere Commits wie auf einer Schnur gefädelt.

Lass uns das Ganze an einem Beispiel durchgehen:

1. Arbeite an einer Webseite.
2. Erstell einen Branch für irgendeine neue Geschichte, an der Du arbeitest.
3. Arbeite in dem Branch.

In diesem Augenblick kommt ein Anruf, dass ein kritisches Problem aufgetreten ist und sofort gelöst werden muss. Du machst folgendes:

1. Schalte zurück zu Deinem „Produktiv“-Zweig.
2. Erstelle eine Branch für den Hotfix.
3. Nach dem Testen führst Du den Hotfix-Branch mit dem „Produktiv“-Branch zusammen.
4. Schalte wieder auf Deine alte Arbeit zurück und werkel weiter.

In GIT ist es also möglich mehrere „Äste“ zu erstellen und an diesen zu arbeiten. Jeder Ast ist dabei völlig unabhängig von den anderen Ästen im selben Projekt. Es können z.B. Fehler an Dateien behoben werden, ohne dass der Hauptstrang (Master) davon in Mitleidenschaft gezogen wird oder es können zusätzliche Features eingeführt werden. Bei Abschluss der Arbeit an einem Branch kann dieser wieder in den Hauptstrang (master) eingefügt werden. Führte die Arbeit in einem Branch zu keinem Ergebnis, kann er auch einfach unbeachtet bleiben.

**Tags:** Wie die meisten anderen VCS kann Git bestimmte Punkte in der Historie als besonders wichtig markieren, also taggen. Normalerweise verwendet man diese Funktionalität, um Release Versionen zu markieren (z.B. v1.0).

**Forking:** Zu Deutsch: Gabeln. Das Versionskontrollsystem spreizt sich. Beispiel: Ubuntu: Ubuntu basiert auf dem Betriebssystem Debian. Da Debian bis heute weiterentwickelt wird, spaltete sich Ubuntu irgendwann ab, daher ist Ubuntu ein Fork von Debian. Ubuntu hat den Stand von Debian übernommen und arbeitet eigenständig daran weiter. Es ist eine parellele Weiterentwicklung und Ubuntu kann jederzeit vom Debian-Projekt alle Änderungen übernehmen (siehe Branch).

**Das Git-Konzept:** Git bietet moderne Möglichkeiten der Versionskontrolle und ist kostenlos. Dezentralisiert: Änderungen können zunächst lokal versioniert werden und dann wird der Stand auf ein zentriertes System hochgeladen. Es ist möglich, von der Kommandozeile zu arbeiten, oder mit einem grafischen Tool: Wir nutzen dazu TortoiseGit.

**Repository anlegen:** Wenn man mit Git arbeiten möchte, ist es notwendig, ein Repository anzulegen. Man unterscheidet zwischen 2 verschiedenen Arten von Repositories.

Das *local Repository* ist dasRepository, dass immer erstellt wird wenn man mit GIT arbeitet.

Ein *remote Repository* ist dann notwendig, wenn mehrere Menschen an einem Projekt arbeiten und ihre *local Repository* miteinander teilen wollen. Hierzu kann auf eine viel zahl von *remote Repository* Server zugegriffen werden (z.B. GitHub, Bitbucket etc.). Diese bieten zum Teil kostenlose Dienste an und die Wahl, seine Projekte public oder private zu sharen.

Ein lokales Repository über die Kommandozeile anlegen:

* Navigieren in einen vorgesehenen Ordner (Bei Windows z.B. cd /d D: …)
* Befinden wir uns im vorgesehenen Ordner erfolgt die Eingabe *git init*

Ein Repository über TortoiseGit anlegen:

* Rechtsklick im gewünschten Ordner > *Git Create Repository here*

Es wird ein versteckter Ordner .git erstellt. Die Dateien in diesem Ordner sind unlesbar und gegen Missbrauch gesichert. Mit den Befehlen add und dann commit können Dateien hinzugefügt werden. Den Bereich zwischen Add und Commit nennt man **Staging**, er ist ein sog. Zwischenbereich/Zwischenspeicher.

In der Kommandozeile fügt der Befehl git add eine Datei zum Staging hinzu. Mit git add –A werden alle Dateien hinzugefügt. Bei TortoiseGit funktioniert dies über einen Rechtsklick > *Add*.

Nun kann man auch den commit-Befehl nutzen. In der Kommandozeile: git commit –m „Kommentar“. In TortoiseGit erfolgt dies über den Rechtsklick *Git Commit*. Der Befehl *git rm* löscht eine Datei, mit *git mv* kann eine Datei umbenannt werden.

Alternativ kann auch ein bereits vorhandenes remoteRepository geklont werden. In der Kommandozeile ist dafür der Befehl git clone URL vorgesehen. Die URL kann z.B. aus einem Projekt aus GitHub entnommen werden. In TortoiseGit erfolgt zum Clonen ein Rechtsklick im gewünschten Ordner > *Git Clone*, dann wird die entsprechende URL eingegeben.

Veranschaulichung: In einem Ordner (Working Ordner) dem wir zur GIT hinzufügen wollen legen wir mit *git init* eine neue *lokale* Repositoryan. Mit dem add-Befehl können Dateien dann ins Staging überführt werden. Erst aus diesem Bereich kann mit dem commit-Befehl eine neue Version im lokalen Repository abgelegt werden. Das lokale Repository ist nicht mit dem Working-Ordner gleichzusetzen. *Achtung*: Hierbei handelt es sich nur um lokale Versionskontrolle.

Das öffentliche Repository, das **remote Repository** wird auf einem Server gehostet.

Um Dateien mit anderen zu teilen sind die Befehle **Push** und **Pull** wichtig. Pushing steht für Bereitstellung. Der Versionsstand vom lokalen Repository wird auf das remote Repository übertragen. Ein konfliktfreies Pushing nennt man Fast-Forward-Push. Pulling steht für das Übernehmen der Änderungen von den anderen Teammitgliedern. Der aktuelle Stand wird also heruntergeladen.

Für das Hosting nutzen wir die Plattform GitHub. Wir können dort ein remote Repository einrichten.

Mit dem Konsolenbefehl *git remote add Name URL* teilen wir Git mit, wo sich unser remote Repository befindet. Remote –v zeigt alle Remotes an, rename/delete macht es möglich, ein Remote umzubenennen oder zu entfernen. In TortoiseGit erfolgt dies über einen Rechtsklick > TortoiseGit > Settings > Git > Remote. Nun können wir pushen und pullen. Änderungen sind nun auch im WebInterface (GitHub) sichtbar. Wissenswert: Push wird nur nach Nutzernamen- und Passworteingabe durchgeführt.

**Branching und Tagging:** Branches rmöglichen es, unterschiedliche Zweige unseres Projektes zu verwalten (der Hauptzweig ist dein Zweig namens master). Mit Tag können wir unsere Commits besonders kennzeichnen.

Die Erstellung eines Branches lokal: Mit *git branch* „name“ (in der Konsole) oder *Create Branch* in TortoiseGit. Benennen wir unsere neue Branch . Um auch in diesem Branch arbeiten zu können, muss ein **Checkout** durchgeführt werden. Dies erfolgt über den Konsolenbefehl *git checkout* oder in TortoiseGit über *Switch/Checkout*. Mit *git branch –d* (Konsole) oder Rechtsklick > TortoiseGit > Switch > Rechtsklick auf Branch > *Delete Branch* können Branches wieder gelöscht werden.

Die Erstellung eines Tags lassen sich analog zu Branches erstellen. Der Konsolenbefehl lautet *git tag „name“* oder über TortoiseGit > *Create tag* in der grafischen Oberfläche.

**Fetching:** Es ist möglich, alle remote-Branches herunterzuladen. Im Gegensatz zum Pulling werden die Daten allerdings nicht übernommen, sondern extra abgelegt. Dadurch werden unsere Daten nicht gefährdet. Der Konsolenbefehl lautet *git fetch RemoteName*, in TortoiseGit erfolgt dies über *Fetch*.

**Rebase und Merge** sind Konfliktauflösungsprozesse. Rebase bedeutet begradigen, hier werden im Prinzip die neuen Commit-Pfade an die alten dran gehängt. Merge bedeutet vereinheitlichen und fügt ein neues Merge-Commit an, bestehend aus allen Inhalten. *Wissensert*: Der Pull-Befehl kann konfiguriert werden. Empfehlenswert ist die Einstellung *fetch & rebase*.

Außerdem wissenswert: Lokale Repositories sind nonbare, Remote Repositories sind bare. Das bedeutet, dass in Remote Repositories kein Working Tree enthalten ist, weil nicht direkt darin gearbeitet wird. Im lokalen Repository schon.

*Anwendungsbeispiel:* Wir wollen nun eine Übungsaufgabe mithilfe von Git realisieren.

Ein Gruppenleiter erstellt hierzu bei GitHub ein neues Repository.



Wir können den markierten Link einfach herauskopieren und das Repository auf unseren Computern anlegen. Der Gruppenleiter muss die anderen Mitglieder in seinem GitHub dazu berechtigen, zu pushen. In den Einstellungen von GitHub kann unter *Collaborators* ein oder mehrere Benutzername(n) eingegeben werden.

Wir können nun das Remote Repository (RR) auf ein Lokales Repository (LR) klonen. Dazu suchen sich alle Gruppenmitglieder einen geeigneten Ort auf ihrem Rechner aus. Da wir mit Eclipse arbeiten, bietet es sich an, den Ordner im Workspace zu platzieren. Mit einem Rechtsklick wählen wir Git Clone und geben dann als URL den HTTPS-Link ein. Das LR muss nicht zwingend den gleichen Namen wie das RR haben. Nun sollte ein versteckter Ordner .git vorliegen (Einblenden unter Ansicht: Häkchen bei *ausgeblendete Elemente*).

In Eclipse können wir nun ein neues Java-Projekt anlegen, das den Namen des LR’s trägt. Zum Testen erstellen wir eine einfache txt-Datei mit willkürlichem Inhalt und speichern sie im nun existierenden src Ordner. Nun können wir mit dem Commit-Befehl arbeiten und noch einen Kommentar für alle Gruppenmitglieder hinzufügen. Wichtig ist, dass alle Dateien ausgewählt werden müssen, die geaddet und *commitet* werden sollen. Nur diese Dateien werden versioniert.

Um diesen Zustand nun in das RR zu laden, muss ein Push durchgeführt werden. Ein weiterer Rechtsklick im Projektordner und die Auswahl *Push* genügt. Alternativ kann auch direkt im Statusfenster des Commits *gepusht* werden. Für einen Push ist immer eine Passworteingabe bei https-Varianten erforderlich. Nach diesem Push befindet sich ein Abbild unseres Projektordners auf GitHub.

Wenn nun ein einzelnes Teammitglied ein eigenes Feature bearbeiten, kann dafür ein Branch erstellt werden. Im Menüfenster muss dazu ein Name für den Branch des Zweigs angegeben werden. Auf die gleiche Weise kann auch ein Tag erstellt werden. Wir führen dann einen Checkout durch, um auch im angelegten Branch arbeiten zu können.

Hat das Gruppenmitglied nun ein tolles Feature in seinem Branch fertiggestellt, müssen wir das nicht auf seinem Rechner ansehen. Der Rest des Teams führt ein Pull durch und wählen dazu den gepushten Branch aus. So sehen wir alle Dateien, die im gepushten Commit enthalten waren.

Es empfiehlt sich, mindestens vor jedem Push auch ein Pull durchzuführen, um einem möglichen Konflikt wegen verschiedener Versionen zwischen lokalem und remoten Repository aus dem Weg zu gehen. Push und Pull arbeiten immer mit dem aktuell ausgewählten Branch.

1. **Anzeige strukturierter Daten in JavaFX**

Views!

JavaFX stellt uns 3 verschiedene "Views" zur Verfügung um Daten horizontal oder vertikal darzustellen. Dabei kann der User Elemente auswählen oder mit ihnen interagieren. Eine View hat einen generischen Typen und kann somit den Typen der Daten repräsentieren und mit ihnen arbeiten.

ListView

Eine ListView kann selbst keine Daten aufnehmen und wir müssen unsere Kontakte über eine besondere ArrayList (observableArrayList) hinzufügen. Eine observableArrayList ermöglicht uns auch mit eventHandler zu arbeiten.... hier wird das nicht benutzt.

// wir definieren ein FXCollections.observableArrayList, welches die darzustellenden Daten enthält

ObservableList<ContactDetails> namen = FXCollections.*observableArrayList*()

// wir fügen jeden einzelnen Kontakt (Person) in unsere observableArrayList

**for**(ContactDetails person : personen){

namen.add(person);

}

WICHTIG: die Daten, die angezeigt werden sollen, werden nicht als Strings übergeben, sondern für eine höhere Flexibilität, als Objekte. Dies ermöglicht uns, direkt mit den Objekten zu arbeiten. Damit die ListView "weiß", welcher der Daten anzuzeigen sind, müssen wir dies mit Hilfe von setCellFactory in einer inneren Funktion angeben.

// wir editieren die Form, wie wir die Daten in unserer Liste anzeigen wollen

// diese Methode ist aus dem Oracle Handbuch für ListViews

liste.setCellFactory((listItem) -> {

// setCellFactory verlangt ein neues Objekt ListCell

**return** **new** ListCell<ContactDetails>() {

// wir überschreiben die Methode updateItem

@Override

**protected** **void** updateItem(ContactDetails item, **boolean** empty) {

**super**.updateItem(item, empty);

// Wenn unserer ContactDetails leer ist .....

**if** (item == **null** || empty) {

setText(**null**);

} **else** { // und wenn nicht geben wir Nachname und Vorname aus.

setText(item.getNachname() + ", " + item.getVorname());

}

}

};

});

TableView

Die TableView ist Verwandt mit dem ListView und gehört zu den etwas komplexeren Komponenten. Dank JavaFX werden uns Eigenschaften wie z.B. Sortierung der der Spalten, Layout etc. von Java abgenommen. Auch bei der TableView sollte direkt mit den Objekten gearbeitet werden. Dazu muss auch hier wieder über setCellValueFactory angegeben werden, welche Attribute unseres Objektes angezeigt werden sollen. Auch hier werden die Daten erst in einer observableArrayList gespeicht und dann der TableView übergeben.

// Deklaration des Tabellenkopfes mit dem Namen

TableColumn<ContactDetails, String> vorname = **new** TableColumn<ContactDetails, String>("Vorname");

// hier sagen wir der Tabelle, dass es das Attribut vorname aus unseren Objekt nehmen soll.

// Dabei wird intern auf die getter Methoden zurückgegriffen

vorname.setCellValueFactory(**new** PropertyValueFactory<ContactDetails, String>("vorname"));

Treeviews

Treeviews ermöglichen uns Daten in einer verschachtelte Baumstruktur anzuzeigen, wie wir es z.B. aus dem Dateimanager kennen. Dabei wird nicht mehr mit einer Liste (observableArrayList) gearbeitet sondern mit Knoten. Jedem Knoten kann, nach Benennung ein weiterer Kinds-Knoten übergeben werden. Somit kann eine Komplexe Baumstruktur erreicht werden. Der Wurzel-Knoten ist immer der oberste.

// wir erstellen unseren Wurzel-Knoten mit dem Namen "Adressbuch"

TreeItem<String> rootItem = **new** TreeItem<String> ("Adressbuch");

…

// jeder Kind-Knoten wird seinem Eltern-Knoten zugewiesen

item.getChildren().addAll(

nameItem,

nachnameItem,

telefonItem,

emailItem,

adressItem);

// und schlussendlich an unseren Wurzel-Knoten übergeben

rootItem.getChildren().add(item);

