Git mit Github.com

Git ist ein verteiltes Versionskontrollsystem. Somit erfüllt es hauptsächlich zwei Aufgaben:

- Die Versionierung und Sicherung von alten Zuständen (man kann immer wieder auf alte Versionen eines Projektes zugreifen)
- Die Koordination des verteilten Arbeitens (dazu gleich mehr)

Bei github.com handelt es sich um eine Plattform, die es ermöglicht anderen Code zur Verfügung zu stellen.

Was wir wissen

Am Anfang des SWP habe ich euch schon einige Befehle erklärt. Hier eine kleine Übersicht:

- `git clone git@github.com:niklasfi/mu` erstellt im aktuellen Verzeichnis einen Ordner „mu", in dem sich dann unser Projekt befindet
- `git status` gibt Informationen über den aktuellen Zustand des Repositorys aus
- `git add <Dateiname>` fügt *<Dateiname>* zum nächsten Commit hinzu
- `git commit -m "<Nachricht>"` erstellt einen Commit mit der angegebenen Nachricht an
- `git reset --hard HEAD` setzt alle Dateien im Dateisystem auf den Zustand des letzten Commmits zurück
- `git pull` ruft die letzten Änderungen aus dem Repository bei github.com ab
- `git push` pflegt neue Commits bei github.com ein

Branches

Bis jetzt haben wir die Möglichkeiten, die uns Git liefert noch lange nicht ausgeschöpft. Deshalb möchte ich euch jetzt Branches vorstellen. Bis jetzt haben wir immer alle unsere Änderungen auf einen „Haufen" geworfen. Immer wenn jemand eine Kleinigkeit am Code verändert hat, wurde diese Kleinigkeit gleich allen zur Verfügung gestellt. Das hat den Nachteil, dass man nur dann Änderungen am Code committen kann/sollte, wenn die Funktionalität an der man zur Zeit arbeitet vollständig und funktionstüchtig ist, da sonst eventuell unfertiger oder fehlerhafter Code verbreitet wird. Man möchte aber auch Zwischenstände sichern. Dies kann zum Beispiel dann nützlich sein, wenn man einmal eine andere Herangehensweise an ein Problem erproben möchte (und sich die Möglichkeit des Zurückkehrens zu einem späteren Zeitpunkt offen lassen will), oder man einfach den Computer wechseln möchte, ohne zuerst eine Kopie des mu Ordners anzulegen.

Die Lösung für oben beschriebenes Problem sind *Branches*. Ein Branch ist eine Art Kopie des aktuellen Verzeichnisses, die von git verwaltet wird. Im allgemeinen wird für eine neue Funktionalität erst ein neuer Branch angelegt, dieser dann mit Code gefüllt, und nach Fertigstellung des neuen Features wird das neue Feature aus dem Branch zum Hauptzweig des Repositories hinzugefügt. Diesen Vorgang nennt man mergen.

Gehen wir diesen Vorgang mal Schritt für Schritt durch:

- `git branch <Zweigname>` legt einen neuen Branch mit dem angegeben Namen an
- `git checkout <Zweigname>` wechselt in den Zweig
- *arbeiten*
- `git checkout master`: wir wechseln wieder auf den Hauptzweig
- `git merge <Zweigname>` fügt die Änderungen aus dem neuen Zweig dem Hauptzweig hinzu. Dies ist analog zu *git pull*, nur das statt ganzer Repositories jetzt Zweige zusammengefügt werden.

Wir können bei *git push* und *git pull* den gewollten Branch auswählen:

- `git push origin <Zweigname>` sendet lokale Änderungen im Zweig an github, und
- `git pull origin <Zweigname>` empfängt diese

Natürlich kann man auch zwischendurch immer mal wieder Änderungen vom master-Zweig auf einen Feature-Zweig überführen, falls der master-Branch zum Beispiel Änderungen enthält die auf dem Feature-Branch benötigt werden. Genauso ist es möglich zwischen Branches zu mergen, denn *git merge* bezieht sich immer auf den aktuellen Zweig.

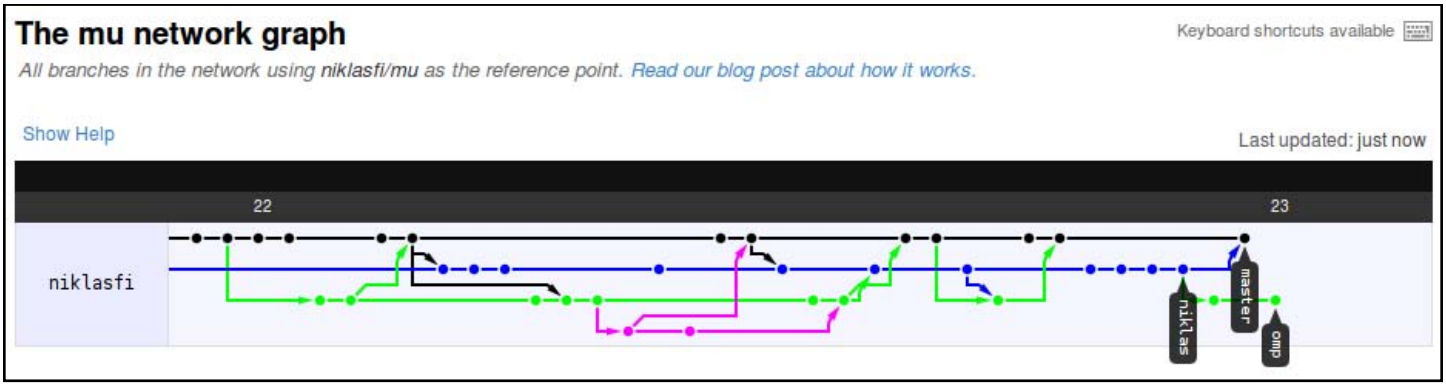

Abbildung 1: Ein Beispiel für einen Netzwerkgraphen (Abgerufen von https://github.com/niklasfi/mu/network am 23.05.2011)

Bis jetzt war der Netzwerkgraph unseres Projektes immer relativ linear. „Abzweigungen" wurden nur durch *git pull* und *git push* erzeugt, wenn zwei Mitarbeiter gleichzeitig etwas verändert haben. Abbildung 1 zeigt den Netzwerkgraphen des maschinelle Übersetzung Projektes mit 3 Branches. Die verschiedenen Ebenen stehen für Repositories, die an der Erstellung beteiligt waren und die Pfeilspitzen deuten merges an. Hier sieht man zum Beispiel, wie Änderungen vom Zweig „niklas" regelmäßig übernommen werden und dann für die Entwicklung genutzt werden.

Organisation mit Github

Github.com bietet außer des reinen Hostings von Repositories noch viele weitere interessante Möglichkeiten um Fehler im Progamm schnell zu beheben und Besprechung und Bewertung von eingechecktem Code zu ermöglichen.

In der Sektion *Issues* können Bugs beziehungsweise noch zu erledingende Aufgaben gesammelt werden. Für jede dieser wird eine Issue erstellt, welche einen *Assignee*, also einen zuständigen, haben kann. Außerdem können wichtige Informationen zum Fortschritt einer Aufgabe in den Kommentaren zu dieser gesammelt werden. Ist eine Aufgabe erledigt, so wird die Issue als *closed* markiert und verschwindet von der Issues-Hauptseite. Mehrere Issues können in einem Meilenstein gesammelt werden, der dann auch ein *due date*, als einen Abgabetermin, haben kann. Wir werden dieses Feature so verwenden, dass jedes Arbeitsblatt seinen eigenen *Milestone* bekommt.

Öffnet man eine Quelldatei in github, so bekommt man eine Texteditor ähnliche Ansicht. Für uns sind hier zwei Funktionen von github interessant: *blame* und *history*. Der blame Knopf befindet sich, genauso wie der History Knopf oben Rechts über dem eigentlichen Text der Datei. Wie der Name schon vermuten lässt, liegt die Hauptfunktionalität von blame darin den „schuldigen" für eine bestimmte Zeile Code zu ermitteln. Der jeweilige Author wird mit dem dazugehörigen Commit links in der Spalte genannt. Hier macht es sich bezahlt, seinen Namen in der Git Konfiguration einzutragen. In der History der Datei können alle Commits die Änderungen zu dieser Datei enthalten eingesehen werden. Öffnet man einen dieser Commits, werden Modifikationen in rot beziehungsweise grün hervorgehoben. Des weiteren bietet sich sowohl für jede einzelne Zeile, als auch für den ganzen Commit die Möglichkeit einen Kommentar zu verfassen. Diese Funktion kann sich als sehr hilfreich erweisen, wenn spezielle Fragen zu Teilen des Codes auftreten.