Fakultät
Informatik und Mathematik

Bachelorarbeit

über das Thema

Visualisierung von Git-Repositories

Autor: Verena Bader
baderver@fim.uni-passau.de

Betreuer: Prof. Dr.-Ing. Sven Apel
Olaf Leßenich

Abgabedatum: 30.09.2016

I Kurzfassung

Das verteilte Versionsverwaltungssystem Git[1] ist weit verbreitet, da es für kleine bis sehr große Projekte geeignet ist. Um dem Benutzer den Überblick über und die Arbeit mit seinem Projekt in Git zu erleichtern, existieren schon verschiedene graphische Darstellungen, der in Git verwendeten Projektstruktur. Aufgrund von Größe und Komplexität dieser Strukturen, ist es schwierig eine geeignete Darstellung zu finden, die die Wünsche des Benutzers erfüllen kann und gleichzeitig überschaubar ist.

In dieser Arbeit wurde versucht eine Umsetzung einer Visualisierung zu finden, die höhere Interaktivität bietet und Funktionalität bereitstellt, die sich aus Gesprächen mit Git-Nutzern als potentiell hilfreich in der Praxis herauskristallisierte. Die entwickelte Visualisierung wurde in einer Webanwendung umgesetzt. Nachdem die Umsetzung der Visualisierung abgeschlossen war, wurden Interviews mit verschieden erfahrenen Git-Nutzern durchgeführt, um zu überprüfen wie erfolgreich die entwickelte Visualisierung verwendbar ist. Außerdem wurde betrachtet, welche zusätzliche Funktionen für weiterführende Entwicklungen der Anwendung interessant sein könnten.

[1] Offizielle Webseite: `https://git-scm.com/` (letzter Zugriff 17.09.2016)

II Inhaltsverzeichnis

III Abbildungsverzeichnis

1 Einleitung

Große Softwareprojekte erfordern die Verwendung von Versionsverwaltungssystemen, wie beispielsweise Git. Für den einzelnen Benutzer kann es jedoch schnell schwierig werden, den Überblick über die Entwicklung des Projekts zu behalten. Aus diesem Grund versucht die im Zuge dieser Arbeit entstandene Anwendung eine benutzerfreundlichere Darstellung von Projekten, die mit Git verwaltet werden, zu finden.

1.1 Vorstellung von Git

Git ist eine freie Software zur verteilten Versionsverwaltung. Im Jahr 2005 fehlte der Linux Entwickler Community ein Tool zur Verwaltung ihrer Softwareänderungen, da es zu Lizenzproblemen mit dem bis dahin verwendeten Versionsverwaltungssystem kam. Aus diesem Grund entwickelten sie ihr eigenes Tool, Git, dessen Stärken in der Geschwindigkeit und Effektivität liegt (siehe [Cha09], Kap. 1.2). Im Folgenden wird ein kurzer Überblick über die Grundlagen von Git und einige zentrale Begrifflichkeiten gegeben.

Ein Projekt, das von Git verwaltet wird, ist in einem Repository abgelegt. Dabei werden die einzelnen eingecheckten Projektzustände in Form von Commits gespeichert. Der Unterschied zu anderen Versionsverwaltungssystemen liegt darin, dass Git pro Commit eine Momentaufnahme (Snapshot) jeder Datei kennt, im Gegensatz zu einer Verwaltung von Dateideltas (Diffs) (siehe [Cha09], Kap. 1.3).

Git speichert eine Historie aller existierender Commits eines Repositorys. Die Commits haben eine Referenz zu ihrem Vorgänger, wodurch sie miteinander verbunden sind und in einer festen Reihenfolge stehen. Zusätzlich zu ihrem Vorgänger, hat ein Commit Attribute, wie z.B. einen eindeutigen (SHA) Hash, einen Autor, eine Nachricht, die der Autor des Commits frei schreiben kann und ein Datum.

Darüber hinaus bietet Git sogenannte Tags an, mit welcher einzelne Commits des Repositorys markiert werden können. Ein Tag fungiert wie eine Art Namensschild, wobei er einen Namen und optional eine Beschreibung beinhaltet (siehe [Cha09], Kap. 2.6). Meist werden mit Tags spezielle Events in der Historie markiert, wie z.B. ein Release oder das Finden eines Bugs.

Abgesehen von der Hauptentwicklungslinie können weitere Entwicklungslinien entstehen, indem sie von der Hauptlinie abzweigen. Dies wird als Branching bezeichnet bzw. die diversen Entwicklungslinien werden Branches genannt. Ein Branch wird durch einen Verweis, der auf den neusten Commit des Branches zeigt, markiert. Diese Verweise werden

im weiteren Verlauf als Heads bezeichnet (siehe [Cha09], Kap. 3, 3.1).

Um Branches oder Entwicklungen von verschiedenen Benutzern wieder zusammen zu führen, können Commits gemergt werden. Git versucht mit Hilfe einer Merge-Strategie die Commits zusammenzuführen und legt einen Mergecommit an. Falls die Strategie auf Probleme stößt, kommt es zu einem Konflikt, der vom Benutzer manuell gelöst werden muss (siehe [Cha09], Kap. 3.2).

1.2 Vorstellung vorhandener Visualisierungen

Es gibt einige Tools, die eine graphische Darstellung eines Repositorys liefern. Der Artikel [Sch14] beschreibt eine Auswahl solcher Tools. Git selbst beinhaltet den Repository Browser gitk, der den Verlauf der Branches mit ihren einzelnen Commits in einfacher Form darstellt. Alle diese Tools legen jedoch nicht ihren Fokus auf die Überschaubarkeit der Visualisierung oder der interaktiven Bewegung innerhalb der Darstellung. Die meisten dieser Anwendungen sind darauf ausgelegt, Git Befehle über eine graphische Oberfläche auszuführen. Die Darstellung des Repositorys ist meist sehr minimalistisch gehalten. So wird es vor allem bei vielen Branches und Entwicklern schnell sehr unübersichtlich für den Betrachter.

Stefan Elsen versucht in [Els13] eine für den Menschen übersichtlichere und besser erkennbare Visualisierung von Git Repositorys zu entwickeln. Er stellt ein Repository als gerichteten Graph dar, senkt allerdings den Detailgrad in seinen Darstellungen, indem er Commits, die eindeutig einem Branch zugeordnet werden können und Schnittpunkte mehrerer Branches zu jeweils einem Knoten zusammen fasst. Des Weiteren betrachtet er über den Zeitverlauf der Entwicklung des Repositorys, das enthaltene Dateisystem und dessen Veränderungen mit Hilfe von Sunburst Diagrammen.

1.3 Ziel der Arbeit

Die im Rahmen dieser Arbeit entstandene Anwendung ist kein Tool, das ein Repository verwalten oder andere Git Befehle steuern können soll.

Stattdessen setzt die Arbeit ihren Fokus auf die Visualisierung der Struktur des Repositorys und zusätzlicher Informationen, die ein Repository beinhaltet. Dabei soll die Visualisierung helfen, die Fülle an Informationen für den Menschen besser einsehbar zu machen. Zusätzlich soll der Benutzer mit der Darstellung interagieren können, wie beispielsweise Elemente selektieren oder Zoom Interaktionen durchführen. Um den tatsächlichen Nutzen

für den Benutzer herauszufinden, wurde nach Abschluss der Umsetzung der Anwendung eine kleine Evaluierung durchgeführt.

2 Planung der Visualisierung - Vorstellung der entwickelten Use Cases

Bevor eine Visualisierung erarbeitet werden kann, muss geplant werden, welchen Anwendungsfällen die Darstellung genügen soll. Dazu werden im weiteren Verlauf Use Cases aus der Sicht verschiedener Rollen entwickelt, welche mit der Anwendung interagieren. Die Use Cases beschreiben Fälle, die in der Praxis des Öfteren auftreten bzw. interessant sein können. Sie entstanden aus Gesprächen mit Git-Nutzern, bei denen sie gefragt wurden, welche Funktionen für ihre täglichen Aufgaben mit Git hilfreich sein könnten.

2.1 Use Cases aus der Sicht eines neuen Entwicklers

Die Use Cases, die im Folgenden beschrieben sind, betrachten die Rolle eines Entwicklers, der neu zu einem bestehenden Softwareprojekt hinzu gekommen ist und folglich dieses noch nicht (gut) kennt.

Use Case A Der erste Use Case, wie in Abbildung 1 zu sehen, beschreibt einen neuen Entwickler im Team, der sich einen Überblick über die bestehende Struktur des Repositorys verschaffen möchte. Dabei interessieren ihn weniger einzelne Commits, als die Gesamtstruktur und der Verlauf der existierenden Branches.

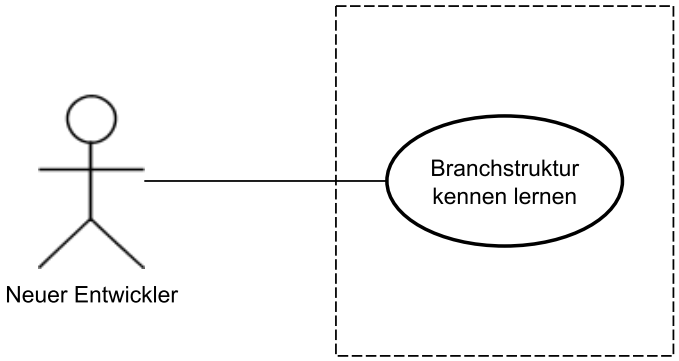

Abbildung 1: Ein neuer Entwickler möchte die Branchstruktur kennen lernen.

Use Case B Abbildung 2 zeigt als weiteren Fall einen neuen Entwickler, der Fragen zu den Inhalten eines Feature-Branches hat. Aus diesem Grund möchte er jemanden kontaktieren, der in diesem Branch entwickelt. Der Kontakt sollte über ausreichendes Wissen über den Branch verfügen. Demzufolge ist ein Entwickler wünschenswert, welcher maßgeblich zu den Änderungen in dem Branch beigetragen hat.

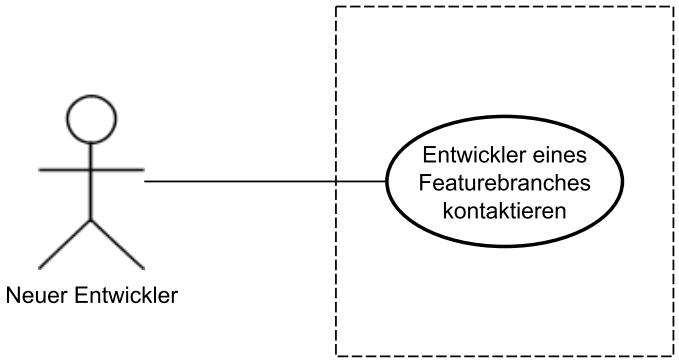

Abbildung 2: Ein neuer Entwickler möchte speziellen Branch-Entwickler kontaktieren.

2.2 Use Cases aus der Sicht eines mit dem Projekt vertrauten Entwicklers

In der Rolle befindet sich jetzt ein Entwickler, dem das betrachtete Projekt vertraut ist bzw. der schon eine längere Zeit an der Entwicklung des Projektes beteiligt ist. Auf speziellere Rolleneigenschaften wird individuell pro Use Case eingegangen.

Use Case C Der in Abbildung 3 gezeigte Entwickler hat in einer Datei einen Fehler gefunden. Er möchte nun herausfinden wann das Problem entstanden ist und wie weit es sich schon in andere Branches verbreitet hat. Dazu interessieren ihn alle Commits, die die gesuchte Datei verändert bzw. neu hinzu bekommen haben.

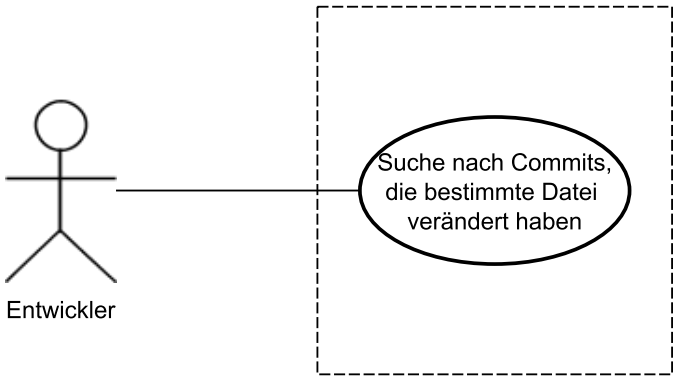

Abbildung 3: Ein Entwickler möchte die Commits nach einer Datei filtern.

Zusätzlich zu der Möglichkeit Commits nach Dateien filtern zu können, sollen auch die Fälle, in denen nach Ordnern bzw. nach Commit-Autoren gefiltert werden soll, abgedeckt werden.

Use Case D Ein ähnliches Szenario zeigt Abbildung 4. Der Entwickler ist auf einen Fehler gestoßen, der nach seiner Vermutung, bei der Durchführung eines Merges entstanden sein muss. Aus diesem Grund möchte er die bisher entstandenen Mergecommits betrachten, um den Fehlerursprung zu finden.

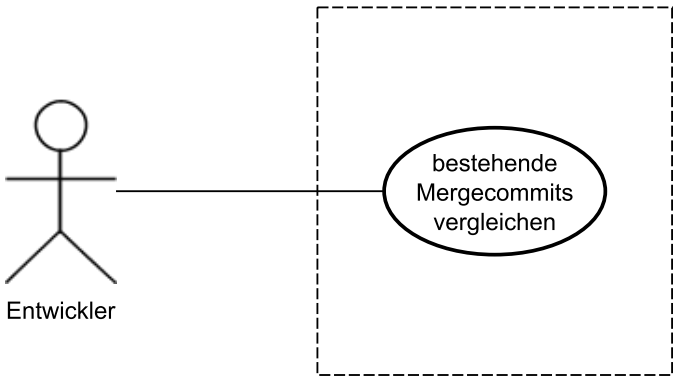

Abbildung 4: Ein Entwickler möchte entstandene Merges vergleichen.

Use Case E Der in Abbildung 5 betrachtete Use Case, legt seinen Fokus auf getaggte Commits. Insbesondere sucht der gezeigte Entwickler nach getaggten Releasecommits, da er in einem Release einen Fehler gefunden hat, den er jetzt zurückverfolgen möchte. Dabei interessiert er sich zu einem für diesen speziellen Release und zum anderen für vergangene und darauf folgende Releases, um zu sehen wo der Fehler enthalten ist.

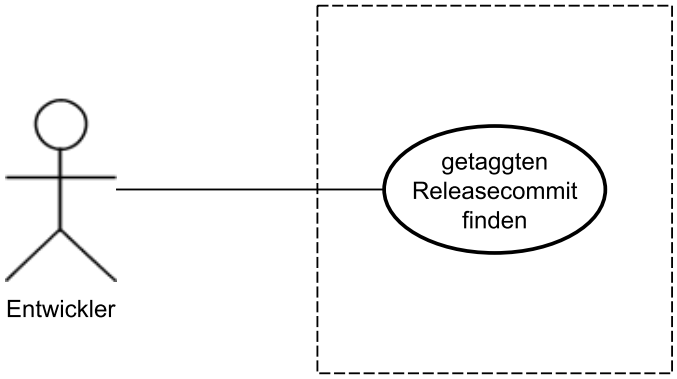

Abbildung 5: Ein Entwickler sucht nach Commits mit Release-Tag.

Use Case F In Abbildung 6 möchte der Entwickler entscheiden, welche Branches sich noch kurzfristig in den bevor stehenden Release mergen lassen. Er sucht nach Branches, die ein relativ kleines Delta zu dem Branch aufweisen, mit dem sie gemergt werden sollen. Damit erhofft er sich einen möglichst kleinen Aufwand beim Mergen bzw. möglichst wenig Konflikte. Außerdem soll der aktuellste Commit der Branches noch nicht zu weit in der

Vergangenheit liegen.

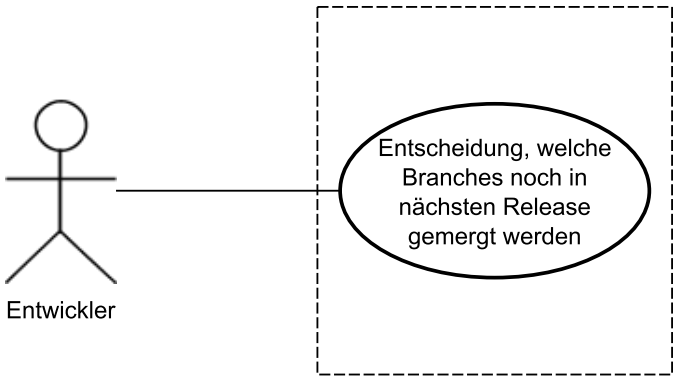

Abbildung 6: Ein Entwickler sucht nach einfach in den Release mergebaren Branches.

Use Case G Bei dem Entwickler in Abbildung 7 handelt es sich um jemanden, der an einem neuen Feature arbeitet. Er arbeitet auf Grund dessen an einem stark vom Hauptbranch abweichenden Branch. Bei dem Branch handelt es sich um einen aktiven Branch, dessen letzter Commit sehr aktuell ist, allerdings ist wie schon erwähnt sein Delta zum Hauptbranch sehr groß.

Der Entwickler sucht nach einer Art Wendepunkt, also nach dem Commit in seinem Branch der noch „gut“ mergebar ist. „Gut“ kann z.B. konfliktfrei heißen. Der Entwickler möchte ein paar Commits zurück gehen, da es zu aufwändig wäre den aktuellsten Commit zu mergen oder der Inhalt das neuesten Commits nicht integrierbar ist.

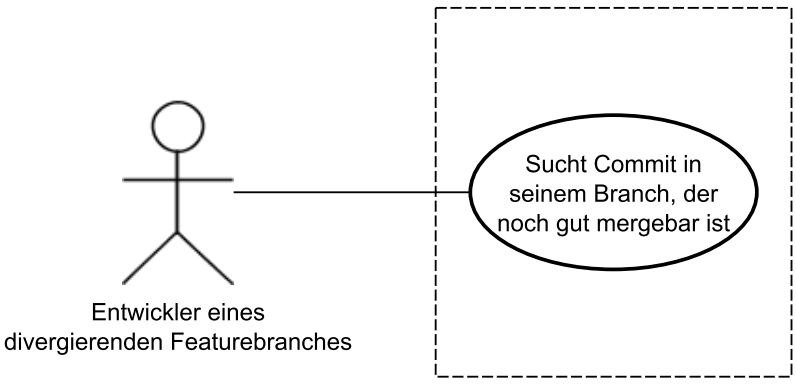

Abbildung 7: Ein Entwickler sucht nach einem Commit, der noch einfach mergebar ist.

2.3 Use Cases aus der Sicht eines Projektmanagers

Schließlich wird die Rolle des Projektmanagers eingenommen, der sich weniger für einzelne Commits als für einen Branch-übergreifenden Überblick interessiert.

Use Case H In Abbildung 8 interessiert sich der Projektmanager für divergierende Branches, wie sie in Abbildung 7 auch betrachtet werden. Er sucht nach Branches, in denen der letzte Commit vor kurzer Zeit entstanden ist, welche allerdings ein großes Delta zum Hauptbranch aufweisen. Meist handelt es sich hierbei um Branches, die neue Features in das Projekt bringen, da das mit viel neuem Quellcode verbunden ist. Der Projektmanager möchte einschätzen wie aufwändig es werden könnte, diese Branches zurück in den Hauptbranch zu mergen.

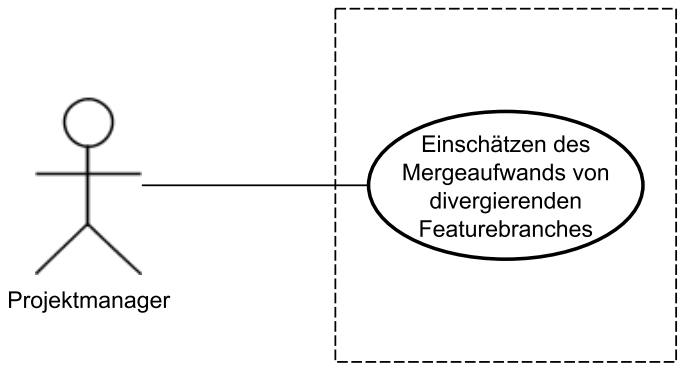

Abbildung 8: Ein Projektmanager sucht nach stark divergierenden Branches.

Use Case I Der Projektmanager in Abbildung 9 möchte die im Repository existierenden Branches reduzieren, weshalb er nach veralteten Branches sucht. Diese Branches zeichnen sich dadurch aus, dass ihr letzter Commit weiter in der Vergangenheit zurück liegt.

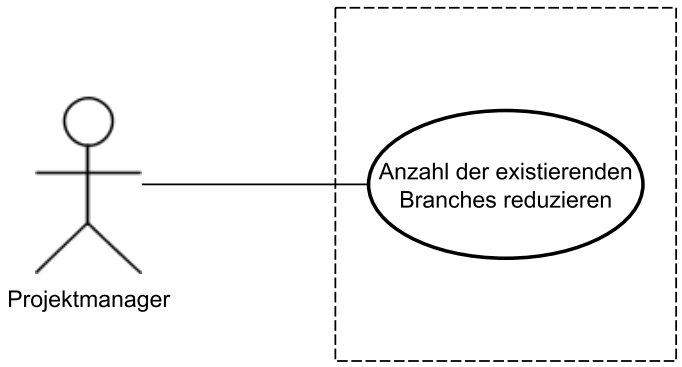

Abbildung 9: Ein Projektmanager sucht nach veralteten Branches.

Nachdem alle Use Cases, die umgesetzt werden sollen, beschrieben wurden, kann die Umsetzung der Visualisierung folgen. Im weiteren Verlauf werden die in diesem Kapitel beschriebenen Use Cases wieder aufgegriffen und mit ihrer jeweiligen umgesetzten Ansicht in Verbindung gebracht.

3 Umsetzung der Anwendung

Bei der Umsetzung handelt es sich um eine Webanwendung, die in den meisten modernen Browsern verwendbar ist. Über die Anwendung lässt sich bequem das Repository laden, welches visualisiert werden soll. Im Folgenden wird genauer auf die interne und externe Gestaltung der Anwendung eingegangen und vor allem das zentrale Element, die Visualisierung des Repositorys in der Anwendung betrachtet.

Als Beispielprojekt wird ein kleines öffentliches Github Projekt verwendet. In den folgenden Betrachtungen wird immer das Projekt choo[2] von Yoshua Wuyts in der Version vom 28.06.2016 als Beispiel verwendet. Das Projekt besitzt alle Eigenschaften, um die Use Cases abdecken zu können. Es enthält vier Branches, viele Mergecommits, einige Tags und mehrere beteiligte Personen, auch wenn diese abgesehen vom Hauptentwickler nur mit wenig Commits über Pull-Requests beteiligt sind. Das Projekt wurde gewählt, da es mit ca. 200 Commits eine gute Größe für die hier enthaltenen Abbildungen hat. Außerdem ist das Projekt, aufgrund seiner überschaubaren Größe, in der späteren Evaluierung für die Fälle geeignet, die laut Beschreibung von jemanden betrachtet werden müssten, der das Projekt kennt.

3.1 Interne Struktur der Anwendung

Zuerst zum internen Aufbau der Anwendung. Optimal wäre eine Struktur, wie sie bei Abbildung 10 gezeigt wird. Es existiert ein Web-Service, der das Anwendungsskript, das auf dem Client läuft, mit Daten aus dem aktuell geladenem Repository beliefert. So können Repositorys lokal beim Web-Service liegen bzw. dorthin geklont werden. Die Aufgabe des Web-Services besteht darin, die Daten aus dem Repository in ein passendes Format zu bringen und je nach Anfrage vom Clientskript, Datenblöcke an den Client zu senden. So kann das Skript auf dem Client einfach Daten nachladen oder aktualisieren, ohne das gesamte Repository selbst bereitstellen zu müssen. Das Skript ist verantwortlich für das Weiterverarbeiten der Daten, wie Layouten und Rendern.

[2]Github Projekt choo: `https://github.com/yoshuawuyts/choo/` (letzter Zugriff 04.09.2016)

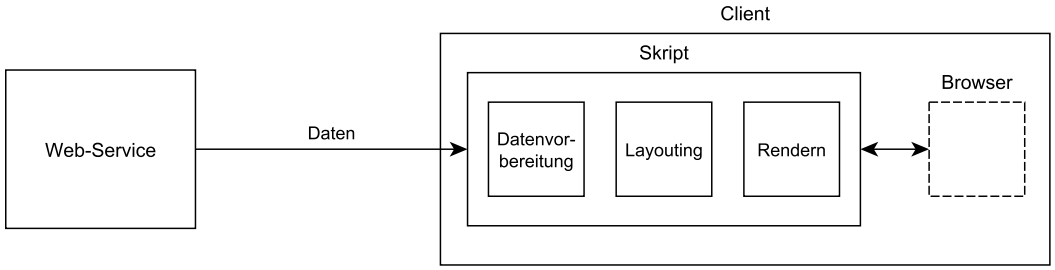

Abbildung 10: Angestrebte Server-Client Struktur

Da die Entwicklung eines Servers bzw. Web-Services, der die beschriebenen Aufgaben übernimmt, über den Umfang dieser Arbeit hinaus geht, wurde der Web-Service simuliert.

Abbildung 11 zeigt, wie in der umgesetzten Version der Web-Service durch eine externe Komponente ersetzt wurde. Das hat den Nachteil, dass das Clientskript so den gesamten Datensatz selbst halten muss und nicht dynamisch Daten nachladen kann. Die externe Komponente wird durch ein Java-Programm gestellt, welches ein Repository mit Hilfe von JGit[3] einliest und alle wichtigen Daten, wie z.B. alle Commits, in eine JSON-Datei schreibt. Das Format der Datei passt dabei zu dem Format, dass das Skript auf dem Client erwartet und wäre für weitere Entwicklungen auch als Austauschformat mit einen Webservice geeignet.

Das Skript ist in Javascript umgesetzt und verwendet die D3[4] Bibliothek, um das Repository zu visualisieren. Die über die Datei geladenen Daten, werden vom Skript vorbereitet, um sie layouten und rendern zu können.

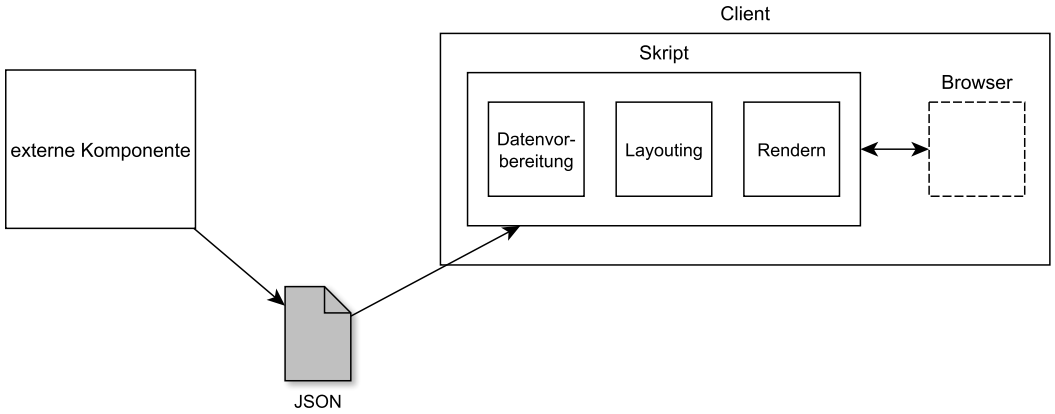

Abbildung 11: Umgesetzte interne Struktur

[3]Offizielle JGit Webseite: `https://eclipse.org/jgit/` (letzter Zugriff 30.08.2016)
[4]Offizielle D3 Webseite: `https://d3js.org/` (letzter Zugriff 30.08.2016)

3.2 Gestaltung der Anwendungsoberfläche

Ziel der Gestaltung der Anwendungsoberfläche ist, der Darstellung des Repositorys möglichst viel Platz zu gewähren. Aus diesem Grund versucht die Anwendung stets die volle Größe des Browserfensters zu nutzen.

3.2.1 Überblick über Oberflächenelemente

Wie in Abbildung 12 zu sehen, wird der Hauptanteil der Anwendungsfläche für die Darstellung des Repositorys genutzt. Der Bereich am linken Rand (1) wird von den möglichen Einstellungsoptionen eingenommen, welche die angezeigte Darstellung je nach Bedarf anpasst. Diese können nach Belieben ausgeblendet werden, um der Visualisierung den nutzbaren Platz zu vergrößern. Der Bereich rechts oben (2) zeigt eine Schaltfläche zum Laden von Repositorys und eine zum Anzeigen der Entwickler des dargestellten Projekts.

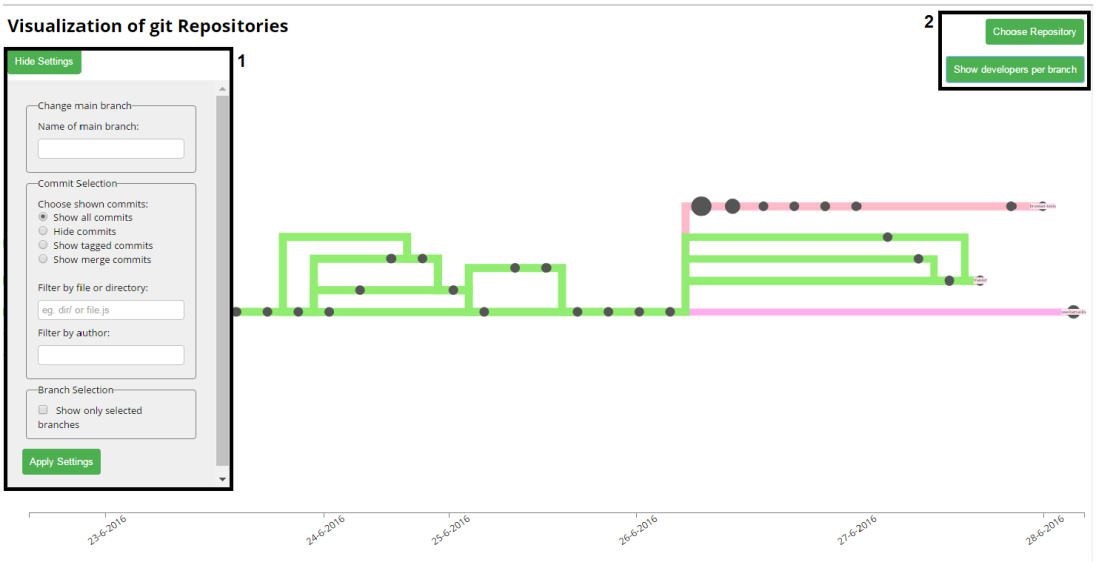

Abbildung 12: Wichtige Elemente der Anwendungsoberfläche

3.2.2 Visualisierung des Repositorys als Graph

Für die Visualisierung des Repositorys wurde die Darstellung als Graph gewählt, da dies zu der von Git vorgegebenen Struktur passt. Beim Zeichnen des Graphen sollen die Zeitinformationen der Commits berücksichtigt werden. Der Graph soll horizontal von links nach rechts verlaufen, um zum einen die Bildschirmbreite auszunutzen und um zum anderen dem Zeitverlauf vom ältesten zum neusten Commit folgen zu können. Dabei sollen

die Commits passend zu ihren Zeitinformationen entlang der Horizontalen verteilt werden. Das bedeutet, dass ein Commit mit älterem Datum weiter links zu finden ist, als einer mit aktuellerem Datum. Zusätzlich zu der spezifischen Anordnung der Commits sind grundsätzlich die bekannten Ästhetikkriterien für Graphen erwünscht [BETT94].

Um das Layout des Graphen wie gerade beschrieben zu gestalten, wurde der Layouter KLay Layered genutzt, der in [Sch] vorgestellt wird. Er wurde an der Universität Kiel basierend auf den Ideen von Sugiyama et al. [STT81] entwickelt. Der ursprüngliche Algorithmus, wie er in dem Paper vorgestellt wird, liefert eine hierarchische Graphdarstellung, indem die Knoten in Ebenen eingeteilt sind. Als Ausgangsgraph kann ein allgemeiner Graph verwendet werden. Der Algorithmus durchläuft fünf Phasen, die nun kurz vorgestellt werden. In Phase 1 werden, falls vorhanden, Zyklen aus dem Graph entfernt. Phase 2 teilt die Knoten in Ebenen ein, so dass jede Kante von einer höheren in eine tiefere Ebene führt. Hierbei werden für Linien, die mehrere Ebenen überspannen, Dummy-Knoten eingesetzt. Weiter werden in Phase 3 die Knoten so in ihrer Ebene umgeordnet, dass die Anzahl der Linienkreuzungen minimiert wird. Phase 4 weist den Knoten ihre Koordinaten zu und achtet dabei auf die Anzahl der geknickten Linien.

Um den Layouter in der Webanwendung zusammen mit D3 nutzen zu können, wurde die nach Javascript portierte Version in Form eines D3-Adapters[5] verwendet. Dieser ersetzt den D3 eigenen Layouter. Das Layout kann über einige Einstellungsparameter in begrenztem Rahmen beeinflusst werden. Da es sich hierbei nicht um einen an Git angepassten Layouter handelt, bräuchte das Layout noch einige Verbesserungen, wie zum Beispiel, dass der ausgewählte Hauptbranch sich im Zentrum befindet. Des Weiteren werden die Knoten nicht immer günstig im Raum platziert, weil wie vorhin beschrieben, der Algorithmus die Knoten in Ebenen einteilt. Um es für den Benutzer nachvollziehbarer zu machen, sollen die Knoten allerdings anhand ihres Datums eingeteilt werden. Aus diesem Grund muss nach Durchlaufen des Layouters etwas nachjustiert werden, um diese Eigenschaft zu erreichen. Dies bewirkt eine nicht immer günstige Lage der Linien.

Um die Orientierung beim Betrachten des Graphen zu unterstützen, existiert eine an den Graph angepasste Zeitleiste. Die Zeitleiste startet mit dem Datum des ältesten Commits. Alle Commits, die in einem Bereich zwischen zwei Tagen liegen, wurden am ersten der beiden Tage commitet.

Damit dem Betrachter ein noch besserer Überblick über das Repository gegeben wird, werden Unterschiede zwischen Commits bzw. zwischen Branches hervorgehoben. So unterscheiden sich Commits anhand ihres Durchmessers, welcher sich aus der Anzahl der in

[5] Github Repository des D3 Adapters KLayJS-D3: `https://github.com/OpenKieler/klayjs-d3` (letzter Zugriff 02.09.2016)

dem Commit veränderten Zeilen ergibt. Die verschiedenen Branches sind über die Dicke ihrer Linien zu unterscheiden. Diese wird aus der Anzahl der Commits im Branch berechnet. Die hier beschriebenen Metriken zum Berechnen der Commitgröße bzw. Branchdicke sind nur eine von vielen Möglichkeiten. Die optischen Unterschiede sollen dem Betrachter helfen sich zum einen schneller innerhalb der Visualisierung orientieren zu können und zum anderen intuitiver die Unterschiede der Commits bzw. Branches einzuschätzen.

Eine weitere Möglichkeit Unterschiede zu betonen, ist die Zuordnung von Farben. Aus diesem Grund werden die verschiedenen Branches, mit Hilfe der Bibliothek randomColor[6], die unterscheidbare Farben liefert, eingefärbt. Als erstes wird dem ausgewählten Hauptbranch eine Farbe zugeteilt, anschließend werden alle anderen Branches, bis zum ersten Commit, den sie mit dem Hauptbranch teilen, eingefärbt. Der jeweilige Head eines Branches wird durch ein Label markiert.

3.2.3 Interaktion mit dem Graph

Eine Interaktion mit der Visualisierung erfolgt bei allen Aktionen mit der Maus. Der gesamte Graph kann mit der Maus in alle Richtungen bewegt werden. Außerdem kann in die Darstellung hinein bzw. aus der Darstellung heraus gezoomt werden. Bei beiden Aktionen passt sich die Zeitleiste dem sichtbaren Ausschnitt der Visualisierung an.

Eine weitere Interaktion ist das Bewegen der Maus über einen dargestellten Commit. Dabei wird eine neben dem Commit schwebende Informationsbox sichtbar, die die wichtigsten Informationen des Commits, wie Datum, Nachricht und Autor, beinhaltet. Zusätzlich kann ein Commit angeklickt werden, um ihn zu markieren. Seine Informationen werden in einer fixierten Box am rechten Rand der Anwendung angezeigt.

Mit den visualisierten Heads kann ebenfalls interagiert werden. Durch Anklicken des Heads, wird der gesamte Branch farblich hervorgehoben.

3.3 Umsetzung der Use Cases

Im Folgenden werden für die in Kapitel 2 vorgestellten Use Cases Lösungen bzw. passende Ansichten, die die Anwendung bietet, vorgestellt. Die Erklärungen werden mit Ausschnitten aus dem visualisierten Beispielprojekt unterstützt.

Um wie in Use Case A gefordert, dem Betrachter einen guten Überblick über das Repository zu gewähren, werden alle Commits ausgeblendet. So wird der Fokus von einzelnen

[6] Offizielle Webseite randomColor: `https://randomcolor.llllll.li/` (letzter Zugriff 03.09.2016)

Commits auf die Gesamtstruktur des Repositorys verlagert. Mit Hilfe der differenzierten Linienfärbung, lässt sich ein Eindruck über den Verlauf der Branches gewinnen. Zusätzlich kann durch zoomen auch die gröbere Struktur des Repositorys überblickt werden. Abbildung 13 zeigt, dass das Beispielprojekt vier Branches umfasst, wobei einer im Vergleich zu den anderen etwas älter ist.

Abbildung 13: Überblick über Branches des Repositorys

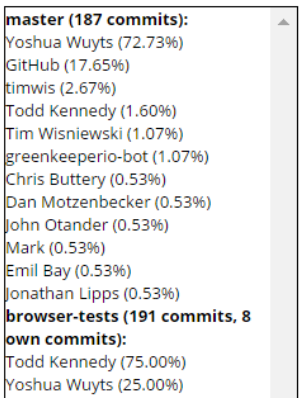

Abbildung 14: Entwickler pro Branch

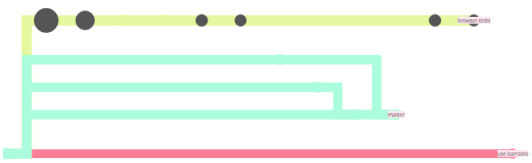

Abbildung 15: Aktiver Autorenfilter

Use Case B interessiert sich für die Entwickler des Projekts bzw. der einzelnen Branches. Um diese zu sehen, wird über die Schaltfläche „Show developers per branch“ oben rechts eine Übersicht eingeblendet. Die Anzeige ist in die einzelnen Branches unterteilt und beginnt mit dem eingestellten Hauptbranch, für den alle Entwickler mit ihren prozentualen Entwicklungsanteil aufgelistet sind. Abbildung 14 zeigt einen Ausschnitt aus der Entwickleranzeige des Beispielprojekts. Für jeden Branch wird die Anzahl der Commits angezeigt. Bei allen Branches, außer dem Hauptbranch, wird zusätzlich die Anzahl der Commits, die nicht im Hauptbranch enthalten sind, genannt. Um Commits mit ihren Entwicklern nicht mehrfach zu zählen, werden für diese Branches nur die Entwickler gelistet, die für die Commits außerhalb des Hauptbranches verantwortlich sind. Die Anzeige kann wieder ausgeblendet werden.

Um Use Case C zu lösen, ist es möglich Commits nach einer Datei bzw. einem Ordner oder einem Autor zu filtern. Grundsätzlich können die Filter kombiniert werden,

allerdings kann dabei ein leeres Ergebnis entstehen. Die Filter werden durch eine einfache Autovervollständigung unterstützt, die mit Hilfe von Awesomplete[7] umgesetzt ist. Je nach gesetzten Filtern werden Commits ausgeblendet, die die Filter nicht erfüllen. Abbildung 15 zeigt einen Ausschnitt aus dem Graph, in dem nur Commits angezeigt werden, deren Autor Todd Kennedy ist.

Use Case D fordert eine ähnliche Lösung. Hier werden nun nur die Commits angezeigt, bei denen es sich um Mergecommits handelt. Durch die Größendifferenzen der dargestellten Commits, kann der Umfang der Merges etwas abgeschätzt werden und somit eine potenzielle Fehlerquelle entdeckt werden.

Abbildung 16: Commit mit Tag-Information

Use Case E wird dadurch behandelt, dass alle, außer getaggte Commits, ausgeblendet werden. So entsteht zum Beispiel eine Übersicht über die Regelmäßigkeit von Releases. Die Tag Information, wie Name und Nachricht, wird zusätzlich zur Commit Information im Tooltip bzw. in der fixierten Informationsbox angezeigt. Abbildung 16 zeigt einen getaggten Commit, welcher einen Release mit der Versionsnummer 2.3.1 darstellt.

Use Case F wird durch die Angabe eines Datums, dass das maximale Alter des aktuellsten Commits der Branches angibt, und einem maximalen Delta, angegeben in Anzahl an Codezeilen, dass die Branches im Vergleich zum Hauptbranch annehmen dürfen, erreicht. Branches, die die eingestellten Bedingungen nicht erfüllen werden ausgeblendet. Nur der Hauptbranch bleibt immer bestehen, um einen beständigen Bezugspunkt zu behalten. Abbildung 17 zeigt wie der Graph mit den Einstellungen Datum = „21-06-2016“ und dem Delta = „300 Codezeilen“ aussieht. Im Vergleich zu Abbildung 13, welche alle Branches anzeigt, wurde der Branch oben rechts ausgeblendet obwohl sein aktuellster Commit am 27-06-2016 entstand. Der Branch weist offenbar ein höheres Delta als 300 auf.

Ähnlich wie bei Use Case F wird für Use Case H ein Datum und ein Delta angegeben. Allerdings markiert dieses Delta nun die Untergrenze, die angibt um wie viele Codezeilen

[7] Offizielle Webseite Awesomplete: `https://leaverou.github.io/awesomplete/` (letzter Zugriff 04.09.2016)

Abbildung 17: Aktuelle Branches mit kleinem Delta

sich ein Branch mindestens zum Hauptbranch unterscheiden muss. Diejenigen Branches, die die Bedingungen nicht erfüllen, werden ausgeblendet.

Use Case G ist eine Art Zusatzoption zu Use Case H . Zusätzlich zu den Bedingungen von Use Case H wird für jeden Branch der Commit gesucht, der noch konfliktfrei in den Hauptbranch gemergt werden kann. Dabei wird beim Head des Branches gestartet und bis zum ersten gemeinsamen Commit mit dem Hauptbranch auf konfliktfreies Mergen getestet. Gesucht ist der erste Commit, der konfliktfrei mergebar ist. Dieser wird in der Visualisierung entsprechend farblich hervorgehoben.

Der letzte Fall, Use Case I , fordert eine Angabe eines Datums, das angibt bis zu welchem Zeitpunkt ein Branch als veraltet gilt. Demzufolge werden die Branches ausgeblendet, deren aktuellster Commit jünger ist als das angegebene Datum.

Die verfügbaren Einstellungsoptionen sind größtenteils kombinierbar, was zu weiteren möglichen Ansichten führt. So können beispielsweise Mergecommits nach einem bestimmten Autor gefiltert werden.

4 Evaluierung der Anwendung

Nach Fertigstellung der Anwendung soll nun eine kleine Evaluierung, in Form von Einzelinterviews, durchgeführt werden. Das Hauptziel ist dabei, das Einholen eines Meinungsbildes, wie passend die Ansichten für die Use Cases sind und welche Funktionen noch für weiterführende Arbeiten interessant bzw. hilfreich sein könnten.

4.1 Evaluierungsplan

Da die Evaluierung in Form von Interviews abgelaufen ist, wird im Folgenden deren Struktur und Inhalte vorgestellt. Zielgruppe des Interviews sind Personen, die zumindest eine grundlegende Erfahrung mit Git haben. So wird sichergestellt das Git und die Begrifflichkeiten, die mit Git verbunden sind, bekannt sind. Für die Interviews wurde, wie schon in Abschnitt 3, das Projekt choo, in der gleichen Version, als Anwendungsfall verwendet.

Das Interview ist in drei Abschnitte aufgeteilt. Gestartet wird mit einer Befragung der interviewten Person zu den bisherigen Erfahrungen mit Git.

- „Wie lange arbeiten Sie schon mit Git?“
- „Haben Sie schon an größeren Projekten mit Git gearbeitet?“
  Dabei gilt als groß, ein Projekt mit mehreren beteiligten Entwicklern (mind. 2) und einer Commitanzahl von mehr als 500 Commits.
- „Wie regelmäßig verwenden Sie Git?“
  Optionen sind mehrmals am Tag/in der Woche/im Monat.

Interessant dabei ist, ob Personen mit weniger Erfahrung mit Git, auf andere Dinge in der Anwendung achten bzw. andere Verbesserungsvorschläge haben, als Personen mit viel Erfahrung.

Im Weiteren wird der Fokus auf die umgesetzten Use Cases gelegt. Der Teilnehmer bekommt nacheinander die Use Cases vorgestellt und soll sich nun in der Anwendung zurecht finden, um die Use Cases mit Hilfe der Visualisierung zu lösen. Dabei wird versucht möglichst wenig Hilfestellung zu geben, um einen Eindruck über die Übersichtlichkeit und Benutzerfreundlichkeit der Anwendung zu gewinnen. In diesem Hauptabschnitt des Interviews soll folgendes festgehalten werden:

- „Wie interessant ist der Use Case in der Praxis?“
- „Wie gut wurde der Use Case in der Visualisierung umgesetzt?“

- „Wie hilfreich ist die Visualisierung beim Lösen des Use Cases?“

Der Abschluss des Interviews gestaltet sich durch einige Fragen über als fehlend wahrgenommene Funktionalität, Gesamteindruck der Anwendung und mögliche Funktionen, die in der Zukunft vorstellbar oder interessant wären.

4.2 Auswertung der Evaluierung

Im Folgenden werden die Ergebnisse der durchgeführten Interviews vorgestellt. Es wurden drei Interviews abgehalten, deren Teilnehmer als erstes in anonymisierter Form vorgestellt werden. Weiter werden die Ergebnisse, die beim Durchgehen der Use Cases entstanden sind, beschrieben. Als letztes folgt der Gesamteindruck der Teilnehmer und interessante Funktionen, die sich die Teilnehmer für spätere Entwicklungen vorstellen können. Bei der hier beschriebenen Auswertung handelt es sich um eine Zusammenfassung der protokollierten Interviews.

4.2.1 Vorstellung der Teilnehmer

Bei Teilnehmer 1 handelt es sich um jemanden, der schon über 10 Jahre Erfahrung mit Git hat und das Tool täglich sehr umfangreich verwendet. Außerdem arbeitet er an zwei sehr großen Projekten (LLVM[8] und gcc) mit, bei denen Git im Einsatz ist, und hat auch schon viele kleinere Projekte mit Git umgesetzt.

Teilnehmer 2 arbeitet seit 4-5 Jahren mit Git und hat ebenfalls schon Erfahrung mit sehr großen Projekten gesammelt, da er unter anderem an dem LLVM Projekt mitarbeitet. Git wird täglich von ihm genutzt.

Teilnehmer 3 hat ca. 4 Jahre Erfahrung mit Git und war an kleineren bis mittelgroßen Projekten beteiligt, wie z.B. JDime[9]. Er verwendet Git mehrmals die Woche.

4.2.2 Ergebnisse der Use Case Betrachtung

Um die Ergebnisse des Hauptteils der Interviews darzulegen, werden im Folgenden die einzelnen Use Cases durchlaufen, wie sie auch in den Interviews angesprochen wurden.

[8] Offizielle Webseite LLVM: `http://llvm.org/` (letzter Zugriff 17.09.2016)
[9] Offizielle Webseite JDime: `http://www.infosun.fim.uni-passau.de/se/JDime/` (letzter Zugriff 17.09.2016)

Für alle Use Cases hat sich herausgestellt, dass die drei Teilnehmer die jeweiligen Einstellungsoptionen sehr intuitiv finden und bedienen konnten. Teilnehmer 2 erwähnt auch, dass das Einstellungsmenü klar strukturiert ist.

Da Use Case A gleichzeitig auch der erste Kontakt mit der Anwendung ist, werden hier von den Teilnehmer Themen angesprochen, die nicht zwingend exakt in den Bereich dieses Use Cases fallen. Die Teilnehmer finden sich gut in die Anwendung ein, da sie schnell einen Überblick über die möglichen Interaktionen bekommen, wie z.B. Markieren der Branches über ihre Heads oder Markieren von Commits. Für diese Funktionen wäre es noch benutzerfreundlicher, wenn z.B. das Selektieren und Deselektieren über eine Liste der Branchnamen geregelt werden könnte. Zusätzlich wird die Bedeutung von Farbe und Dicke der Linien richtig interpretiert und auch als hilfreich gesehen. Die einzige Unklarheit lag hierbei bei der gewählten Metrik, die die Liniendicke bestimmt.
Um den Use Case zu lösen, haben die Teilnehmer die vier existierenden Branches erkannt und konnten dabei die wichtigeren aktuellen Branches von dem kleinen etwas älteren Branch unterscheiden. Nur Teilnehmer 1 hatte anfänglich etwas Probleme die Struktur zu überblicken, da ihn die durch den Layouter gewählte Linienführung und die existierenden Pull Requests etwas verwirrt haben. Aus diesem Grund sollte der momentan verwendete Layouter durch einen speziell für Git entwickelten ersetzt werden. Außerdem hat Teilnehmer 1 vorgeschlagen Pull Requests speziell zu markieren, um sie von Merges von zwei Branches unterscheiden zu können.

Für Use Case B können alle Teilnehmer die angegebene Übersicht, mit ihren Zahlwerten zu Commitanzahl und -rate, korrekt interpretieren. Der Hauptentwickler des Projekts wird sofort erkannt. Die Funktion wird als wichtige Informationsquelle gesehen, allerdings könnte sie noch um die E-Mail Adressen der Entwickler erweitert werden, um jemanden sofort kontaktieren zu können. Eine weitere interessante Idee zur Erweiterung der Übersicht beschreibt das Hervorheben der Personen, die anteilig die meisten Merges produzieren.

Die Umsetzung von Use Case C hat vor allem bei Teilnehmer 1 großen Anklang gefunden. Er bewertet die Funktion, vor allem in Kombination mit ausgewählten Mergecommits, als sehr nützlich. Alle Teilnehmer filtern erfolgreich nach einem Autor bzw. einer Datei bzw. einem Ordner, wobei hier die Autovervollständigung stark unterstützt. Eine mögliche Verbesserung der Funktion würde das Vorschlagen von beispielsweise Top-Autoren liefern.

Bei der Lösung von Use Case D fallen den Teilnehmer nun verstärkt die Größenunterschiede der Commits auf, welche zwei der drei Teilnehmer sofort richtig interpretieren. Die Anzeige der Mergecommits trägt laut Teilnehmer 1 stark an dem Verständnis der Struktur des Repositorys bei. Allerdings bräuchten Mergecommits noch einige zusätzliche Informa-

tionen im Gegensatz zu normalen Commits. Dazu gehört z.B. die Angabe der Parents, eine kurze Statistik wie viel sich verändert hat oder die Anzahl der Konflikte. Speziell beim Mergen wäre auch das Einbinden von Codeausschnitten in die Anwendung interessant. Auch wäre es in diesem Fall nützlich, die Metrik, die für die Commitgröße verantwortlich ist, einstellen zu können.

Use Case E wird von den Teilnehmer als praktische Funktion empfunden. Sie können die Releases gut überblicken und finden so auch Releases mit größeren Änderungen. Eine Kombination mit der Anzeige von Mergecommits wäre noch eine sinnvolle Ergänzung. Als zusätzliche Information könnte eine kurze Statistik über die Änderungen zum vorangegangenen Tag im Branch angegeben werden. Teilnehmer 2 hat den Vorschlag Signed Tags mit einzubinden.

Use Case F wird von zwei der drei Teilnehmer als wichtig in der Praxis gesehen, da so der Überblick bei sehr vielen Branches als besser eingeschätzt wird. Alle Teilnehmer können die Branches korrekt einschätzen und finden heraus, dass der Branch der in Abbildung 13 oben rechts angezeigt wird, im Vergleich zu den anderen Branches, ein großes Delta haben muss. Um die Einstellungsoptionen weiter zu verbessern, sollte die gewählte Metrik für das einzugebende Delta angegeben werden. Teilnehmer 1 würde anstatt die ausgeblendeten Branches komplett zu entfernen, die Branches nur ausgrauen. Die anderen beiden Teilnehmer hingegen finden die Lösung gut so wie sie ist, da die ausgeblendeten Branches gerade nicht relevant sind und somit auch nicht zu sehen sein müssen.

Use Case H wird ähnlich wie Use Case F bewertet. Die Umsetzung dieses Falles ist laut Teilnehmer 1 auch eine gut Methode, um Teilnehmer für einen Rebase zu identifizieren. Sobald ein Branch über ein bestimmtes Delta hinaus wächst, kann ein entsprechender Rebase angefragt werden.

Da Use Case G mit Bezug auf Use Case H zu betrachten ist, werden sie in den Interviews auch in dieser Reihenfolge angesprochen. Der Use Case ist hilfreich, um Zeit z.B. beim Mergen zu sparen. Die dahinter liegende Metrik wird von allen Teilnehmer richtig interpretiert, trotzdem sollte eine Farbe zum Markieren gewählt werden, die ausdrückt, dass der markierte Commit mit einer positiven Eigenschaft belegt wurde. Da sich dieser Fall auf zukünftige Aktionen bezieht, in diesem Beispiel das mögliche Mergen von Branches, kommen von den Teilnehmer Vorschläge für weitere vorausschauende Berechnungen. So könnten zum Beispiel potenziell auftretende Konflikte bei einem angestrebten Rebase angezeigt werden.

Für das Lösen von Use Case I erkennen die Teilnehmer nach Datumsangabe schnell den, im Vergleich zu den anderen Branches, weiter zurück liegenden Branch. Eine darauf fol-

gende Aktion könnte sein, den Branch-Besitzer heraus zu finden und einen Rebase oder ein Löschen des Branches anzufragen. Auch hier fällt etwa auf, das das Selektieren von Branches noch etwas angenehmer gestaltet werden sollte. Eine Möglichkeit bei der Anzeige der Branches wäre, den Teil der Hauptlinie den alle Branches gemeinsam haben auszublenden, da dieser Teil in den Branch spezifischen Betrachtungen nicht unbedingt benötigt wird.

4.2.3 Gesamteindruck und Erweiterungsideen

Nun zum letzten Teil des Interviews, der den Fokus noch einmal auf den Gesamteindruck der Anwendung und weitere zukünftige interessante Anwendungsfälle legt.

Teilnehmer 1 hat während des Interviews schon anklingen lassen, dass er die vorhandenen Funktionen auch in der Praxis hilfreich findet und sich gut vorstellen kann sie zu verwenden. Da Teilnehmer 1 vor allem bei dem LLVM Projekt oft einen Rebase durchführen muss, sucht er noch nach mehr Unterstützung in diese Richtung, die er bis jetzt noch bei keinem Tool gefunden hat. Hier wäre z.B., wie vorhin schon beschrieben, eine Funktion zum Voraussagen von möglicherweise auftretenden Konflikten bei einem Rebase vorstellbar. Das heißt es wird nach einem Indikator gesucht, wie aufwendig bzw. schwierig ein Commit innerhalb eines Rebases sich entwickelt.

Teilnehmer 2 ist der Meinung, dass die Umsetzung für Einsteiger in ein Projekt oder Personen, wie z.B. Projektleiter, die nicht sofort an dem dahinter liegenden Code interessiert sind, sehr gut geeignet und gut gelöst ist. Um es für die Verwendung von Entwicklern geeignet zu machen, müsste der dahinter liegende Code noch mit einbezogen werden. So könnte man z.B. den Diff von zwei ausgewählten Commits einblenden oder für Mergecommits die gepatchten Codestellen anzeigen. Als weitere Funktion könnte sich Teilnehmer 2 sehr gut vorstellen, die Informationen einer Testsuite in die Umsetzung mit einzubinden. Beispielsweise könnten Commits je nach Ergebnis des Testdurchlaufes verschieden eingefärbt oder mit einer Symbolik versehen werden.

Teilnehmer 3 hat zwar noch nicht mit allen Fällen Erfahrungen, kann sich allerdings vorstellen, dass die Visualisierung vor allem bei vielen Branches sehr helfen kann. Andere Tools bieten viele der umgesetzten Funktionen nicht an, bzw. sind konzeptuell nicht darauf ausgelegt, was die Umsetzung als etwas Neues hervorhebt. Da die Basis zur Visualisierung schon da ist, kam Teilnehmer 3 auf die Idee, einen Bisect graphisch darzustellen. Außerdem könnte man dem Benutzer die Möglichkeit geben, die Metriken je nach seinen Bedürfnissen dynamisch einstellen zu können.

5 Ausblick

Diese Arbeit stellt den ersten Versuch dar, eine Visualisierung aufzusetzen, die dem Nutzer neue Möglichkeiten gibt, aus einem Repository Informationen zu gewinnen. Dies ist so weit gelungen, dass die umgesetzten Use Cases für die Anwendung in der Praxis geeignet und interessant sind. Außerdem hat sich die Visualisierung als hilfreich für Einsteiger in ein Projekt und Personen, die nicht sofort an konkretem Code interessiert sind, herausgestellt. Für Entwickler müsste noch mehr Bezug zu dem dahinter liegenden Projektcode hergestellt werden.

Als erster weiterer Schritt ist die Erweiterung um einen Server wichtig, mit dem dynamisch Repositorys geladen werden können, ohne vorher eine Datei erstellen zu müssen. Das Vorhandensein eines Servers, bringt auch andere Vorzüge. So hat man die Möglichkeit aufwendigerer Berechnungen auszulagern, wie das im Interview vorgeschlagene Berechnen eines Indikators, wie aufwendig ein Rebase für einen Commit werden kann.

Zusätzlich wäre für zukünftige Entwicklungen ein speziell an Git angepasster Layouter von großer Bedeutung, um gezielter auf die Anordnung der Commits und die Positionierung von Branches einwirken zu können.

Als weitere potenziell für den Betrachter hilfreiche Erweiterungen, sind die Ideen, die während der Interviews entstanden sind, definitiv vorstellbar. So kann man eine angebundene Testsuite in die Visualisierung mit einbinden, indem man die Commits je nach Ergebnis markiert. Eine weitere Anmerkung war, Code stärker mit einzubinden. Vorgeschlagen wurde die Option, zwei Commits manuell, z.B. mit der Maus, auszuwählen und den Diff der beiden einzublenden. Eine etwas weiterführende Idee, wäre z.B. einen Bisect graphisch darzustellen. Dabei wird der Betrachter mit Hilfe der Darstellung durch den Vorgang des Bisects geführt. Ebenso interessant ist ein Vorschlag, der über die Grenzen des zweidimensionalen hinausgeht. Dabei geht es um eine drei- oder mehrdimensionale Darstellung eines Repositorys mit dem Einbezug der existierenden Forks.

Ein anderer Ansatzpunkt für Verbesserungen, ist der Einsatz anderer bzw. weiterer Metriken für Größenverhältnisse und Farbzuordnung. Mit diesen und weiteren Ideen wäre es möglich, die Anwendung weiter an die Bedürfnisse des Endnutzers anzupassen und sie für den Alltag einsatzfähig zu machen.

6 Quellenverzeichnis

[BETT94] Battista, Giuseppe D. ; Eades, Peter ; Tamassia, Roberto ; Tollis, Ioannis G.: Algorithms for Drawing Graphs: an Annotated Bibliography. In: *Comput. Geom.* 4 (1994), S. 235–282

[Cha09] Chacon, Scott: *Pro Git*. 1st. Berkely, CA, USA : Apress, 2009. – ISBN 1430218339, 9781430218333

[Els13] Elsen, Stefan: VisGi: Visualizing Git branches. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*, 2013, 1–4

[Sch] Schulze, Christoph D.: *KLay Layered*. KIELER Project. `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered`. – letzter Zugriff 02.09.2016

[Sch14] Schürmann, Tim: *Fünf Git-GUIs im Test*. Linux-Magazin. `http://www.linux-magazin.de/Ausgaben/2014/03/Bitparade`. Version: März 2014. – letzter Zugriff 17.09.2016

[STT81] Sugiyama, Kozo ; Tagawa, Shojiro ; Toda, Mitsuhiko: Methods for Visual Understanding of Hierarchical System Structures. In: *IEEE Trans. Systems, Man, and Cybernetics* 11 (1981), Nr. 2, S. 109–125

Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen als Hilfsmittel benutzt habe.

....

Ort, Datum Unterschrift