Änderungen zusammenführen

In der Regel geht es darum, den Entwicklungsstand von zwei verschiedenen Branches zusammenzuführen. Hierfür gibt es in Git zwei Optionen: `git merge` und `git rebase`. Daneben besteht die Möglichkeit, einzelne Commits über einen Cherry-Pick in einen anderen Branch zu übertragen.

Git Merge

Mit dem Befehl `git merge` lassen sich grundsätzlich mehrere Branches mergen. In der Praxis wird jedoch fast ausschließlich der Stand eines Branchs in einen anderen übertragen. Ein sogenannter *Octopus-Merge* führt mehrere Branches zusammen. Das ist komplex, macht keinen Spaß und ist nicht empfehlenswert. Schauen wir uns also den Standardfall an. Es gibt beispielsweise einen Feature-Branch, dessen Änderungen in den Hauptzweig `main` übertragen werden sollen.

Ein Merge wird immer vom Ziel-Branch aus angestoßen. Möchtest du deinen Feature-Branch in den Hauptzweig `main` mergen, solltest du also zunächst sicherstellen, dass du dich gerade im Branch `main` befindest und den aktuellen Stand heruntergeladen hast. Außerdem sollte dein lokales Arbeitsverzeichnis keine Änderungen enthalten, die noch nicht von Git getrackt werden. Diese müssen vorher committet, verworfen oder als Stash zwischengespeichert werden. Dann kann der Merge ausgeführt werden:

```
$ git merge feature-branch
```

Im einfachsten und glücklichsten Fall hat sich im Haupt-Branch in der Zwischenzeit nichts getan, denn dann fügt Git die Commits aus dem Feature-Branch einfach hinzu. Diese Variante nennt sich *Fast-Forward-Merge*, und hier kann Git einfach die Referenz auf den letzten Commit im `main`-Branch – diese Referenz wird auch `HEAD` genannt – auf den letzten Commit des Feature-Branchs setzen.

Ein Fast-Forward-Merge ist nicht möglich, wenn es zwischenzeitlich im Ziel-Branch zu Änderungen gekommen ist. Das ist in größeren Teams, in denen mehrere Personen an der Codebasis arbeiten, häufig der Fall. Dann versucht Git, die beiden Versionen zusammenzubringen. Dabei ist Git ziemlich smart und schafft es häufig, die Änderungen ohne menschliches Zutun sinnvoll zusammenzufügen. Git wendet in diesem Fall die *Recursive-Strategie* an und führt einen *3-Way-Merge* durch, bei dem ein Merge-Commit erstellt wird. Dieser Commit enthält die neu hinzugefügten Änderungen.

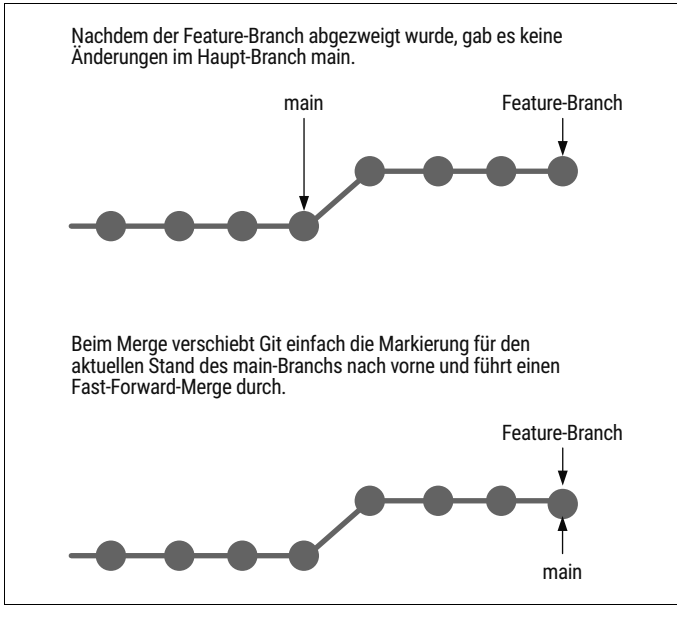

Abbildung 3-9: Ein Fast-Forward-Merge in Git

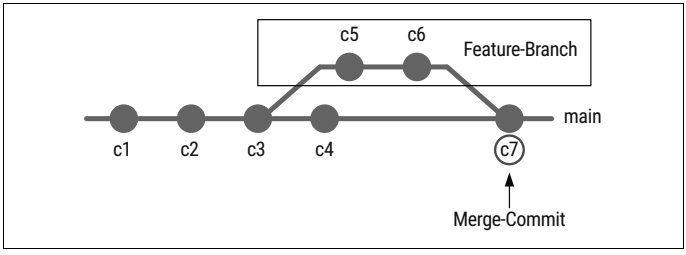

Abbildung 3-10: Gab es Änderungen im main-Branch, führt Git einen 3-Way-Merge durch und erstellt einen Merge-Commit.

Merge-Konflikte lösen

Nicht immer kann Git alle Änderungen problemlos zusammenführen. Gibt es in den beiden Branches unterschiedliche Änderungen an der gleichen Codestelle oder hat eine Entwicklerin eine Datei gelöscht, während ein anderer Entwickler etwas an ihr verändert hat, entstehen Konflikte. Nur zu verständlich, denn woher soll Git wissen, was nun die richtige Version ist? Solche Konflikte müssen von Hand, also von den Entwicklerinnen und Entwicklern selbst gelöst werden.

Schauen wir uns ein Beispiel an. Wir befinden uns im Branch `mein-feature-branch` und mergen mithilfe des Kommandos `git merge main` den Stand des Haupt-Branchs `main` dort hinein. Dabei tritt ein Konflikt auf. Tritt ein solcher Konflikt während eines Merges auf, hält Git den Vorgang an und vermeldet es in der Konsole.

```
Auto-merging [filename]
CONFLICT (content): Merge conflict in <Dateiname>
Automatic merge failed; fix conflicts and then commit the result.
```

Nun heißt es nicht zu verzagen, denn Merge-Konflikte gehören zur Arbeit mit Git einfach dazu. Es erfordert etwas Übung, und es gibt auch ein paar Strategien, um die Anzahl und den Umfang dieser Konflikte zu begrenzen.

Zunächst nutzt du am besten das Kommando `git status`, um zu sehen, welche Dateien betroffen sind und wie ihr Status ist. Wie immer gibt Git sachdienliche Hinweise dazu, was als Nächstes zu tun ist.

```
$ git status
 On branch mein-feature-branch
 You have unmerged paths.
   (fix conflicts and run "git commit")

 Unmerged paths:
   (use "git add ..." to mark resolution)

 both modified:     <Dateiname>

 no changes added to commit (use "git add" and/or "git commit -a")
```

Öffne nun die betroffene Datei. Wenn du diese in einem Texteditor deiner Wahl öffnest, ist die konfliktbehaftete Stelle markiert.

```
Eine Datei, in der Dinge stehen
<<<<<<< HEAD
Zeile im Branch main geändert.
=======
Zeile im Feature-Branch geändert.
>>>>>>> feature-branch
```

`HEAD` zeigt auf den letzten Commit im aktuellen Branch, mit `<<<<<<< HEAD` ist also der Stand in unserem Ziel-Branch `main` gekennzeichnet. Die Änderung, die in Konflikt mit dieser Änderung steht, ist durch `=======` abgetrennt. Das `>>>>>>> feature-branch` verrät uns, aus welchem Branch die Änderung kommt.

Um den Konflikt zu lösen, musst du nun entscheiden, welche Änderung an dieser Stelle richtig ist. Ist es eine von beiden oder eine Kombination? Um bei unserem Beispiel zu bleiben: Angenommen, der Stand aus dem Feature-Branch wäre richtig, dann sollte am Ende in der Datei, die den Merge-Konflikt enthält, nur noch folgende Zeile übrig sein:

```
Zeile im Feature-Branch geändert.
```

Alle anderen Markierungen des Merge-Konflikts müssen entfernt werden. Sobald alles bereinigt ist, speicherst du die Änderungen im Editor, und es geht zurück in die Konsole. Dort kannst du erneut den Status abfragen und anschließend die Datei, die den Konflikt beinhaltet hat, durch `git add Datei.txt` als gelöst markieren und

dem Index hinzufügen. Den Merge schließt du dann über das Kommando `git commit` ab.

Merge-Konflikte können auch in mehreren Dateien gleichzeitig auftreten. Dann müssen die Konflikte nach und nach in den jeweiligen Dateien gelöst, anschließend dem Index hinzugefügt und abschließend muss der Commit erstellt werden.

Konflikte mit einem Merge-Tool lösen

Gerade in komplexeren Fällen kann es hilfreich sein, ein Merge-Tool zu Hilfe zu nehmen, das eine grafische Oberfläche zum Lösen von Konflikten bietet. Solche Tools sind z.B. `meld` oder `kdiff3`. Diese müssen installiert werden, macOS bringt FileMerge bereits mit. Das Tool der Wahl kann dann in der Git-Konfiguration als präferiertes Merge-Tool hinterlegt werden.

```
$ git config --global merge.tool kdiff3
$ git config --global merge.tool meld
```

Aufgerufen wird das Merge-Tool durch die Eingabe von `git merge tool`. Die Oberflächen der meisten Tools sehen ziemlich antiquiert aus, lass dich davon nicht abschrecken. Alle bieten eine Darstellung der beiden Versionen einer Datei – also die beiden Versionen, die miteinander in Konflikt stehen. Diese werden manchmal auch als `ours` und `theirs` bezeichnet, was etwas verwirrend ist. `ours` ist der Stand des Branchs, in dem du dich befindest und in den du die Änderungen der anderen hineingemergt hast. `theirs` ist dementsprechend der Stand des Branchs, den du in deinen oder »unseren« Branch gemergt hast.

In den meisten Merge-Tools werden die Konflikte farblich hervorgehoben, und sie zeigen jeweils drei Bereiche an: den Stand des Ziel-Branchs, den des Branchs, dessen Stand in den Ziel-Branch übertragen wird, und die Version, für die du dich entschieden hast. Im Beispiel von FileMerge, das im Screenshot abgebildet ist, befindet sich links der Stand des Feature-Branchs, rechts der des Branchs `main` und unten die gemergte Version. Die Richtung des Pfeils verdeutlicht, welche Variante gerade ausgewählt ist. Unten rechts kann die Variante gewählt werden, es ist aber auch möglich, beide auszuwählen oder das Resultat im unteren Fenster zu editieren.

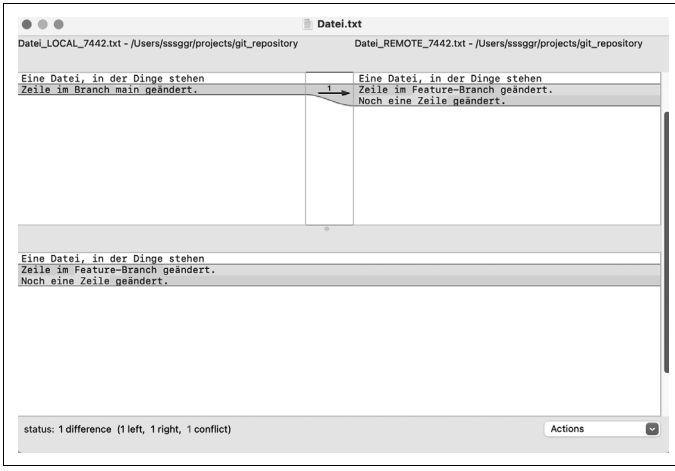

Abbildung 3-11: Ein Screenshot des Merge-Tools FileMerge

Viele Texteditoren haben bereits eine gute Git-Unterstützung integriert, oder es stehen entsprechende Plug-ins zur Verfügung. Sie heben solche Merge-Konflikte optisch hervor und bieten ein paar Optionen an, um sie zu lösen.

Abbildung 3-12: Die Ansicht eines Merge-Konflikts im Editor Visual Studio Code

Automatisierte Tests als doppelter Boden

Natürlich kann es durch Merges – sei es durch Git oder manuell – zu Problemen kommen, wenn Codeteile doch nicht zusammenpassen oder Fehler beim Mergen passieren. Das lässt sich durch eine gute Abdeckung mit automatisierten Tests abfedern. Durch Tests fallen solche Fehler schnell auf, in der Regel schon beim Mergen. Mehr dazu findest du in Kapitel 9 im Abschnitt *Kontinuierlich integrieren und ausliefern mit Git*.

Merge abbrechen

Manchmal hat man sich beim Mergen vertan, die Konflikte sind zu groß, und es gibt vielleicht bessere, kleinere Zwischenschritte. Dann lässt sich ein Merge – wie übrigens auch alle anderen Git-Operationen – einfach abbrechen: `git merge --abort`.

Die Git-Historie durch Rebasing glätten

Häufiges Mergen führt zu vielen Merge-Commits. Zu diesen Merge-Commits gibt es unterschiedliche Positionen. Für die eine Seite sind sie ein integraler Teil von Git und machen nachvollziehbar, wann Versionen zusammengeführt wurden. Andere empfinden sie als störend, da sie die Git-Historie durch unnötige Commits unübersichtlich machen. Rebasing ist eine Möglichkeit, die Zahl der Merge-Commits zu reduzieren.

Diese Option ist allerdings mit äußerster Vorsicht einzusetzen, denn im Unterschied zu einem Merge bleibt bei einem Rebase die Historie nicht erhalten, sondern wird komplett neu geschrieben. Abbildung 3-13 zeigt die beiden Varianten Merge und Rebase im Vergleich.

Die Commits aus dem Ursprungs-Branch – in Abbildung 3-10 der Branch `feature` – werden im Ziel-Branch neu erstellt. Bei einem Rebase tut Git so, als wären die übertragenen Commits im Ziel-Branch entstanden. Es erstellt die Commits neu im Ziel-Branch, sodass ein linearer Verlauf zustande kommt.

So ist es später einfacher, den Projektverlauf nachzuvollziehen. Der Preis für den linearen Verlauf ist allerdings der Verlust der Sicherheit, den dir die Git-Historie bietet. Ein Rebase ist fehleranfällig.