Kommando	Erklärung
git branch	Alle lokalen Branches auflisten.
git branch -a	Alle lokalen und Remote-Branches auflisten.
git branch -r	Alle Remote-Branches auflisten.

In der Ausgabe ist der aktuell ausgecheckte Branch mit einem Sternchen gekennzeichnet.

```
$ git branch
  mein-feature-branch
  bugfix-branch
* main
```

Gibst du zusätzlich das Flag `-v` an, wird dir in der Liste hinter dem Namen des Branchs jeweils noch der letzte Commit in eben diesem Branch angezeigt.

```
$ git branch -a -v
  feature-branch      ef9226d Correct spelling
* main                aeb07f9 [ahead 3] Correct command to
  switch branches
  remotes/origin/HEAD -> origin/main
  remotes/origin/main 972cdb6 Move cheat sheet to the end
```

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.

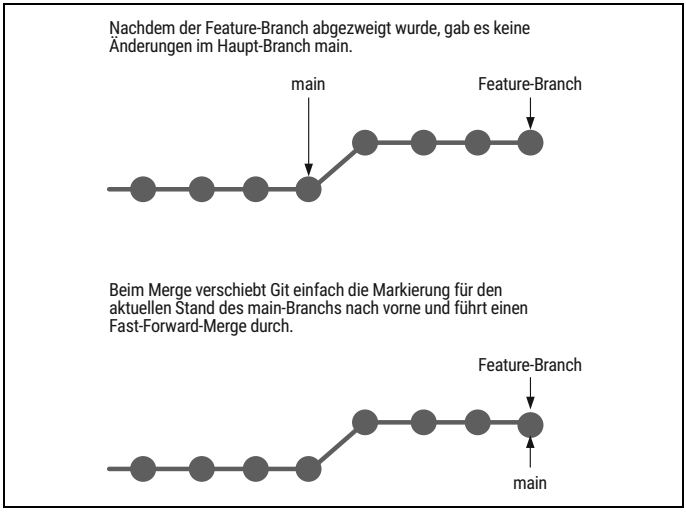

Abbildung 3-9: Ein Fast-Forward-Merge in Git

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-Wege-Merge* durch, bei dem ein Merge-Commit erstellt wird. Dieser Commit enthält die neu hinzugefügten Änderungen.

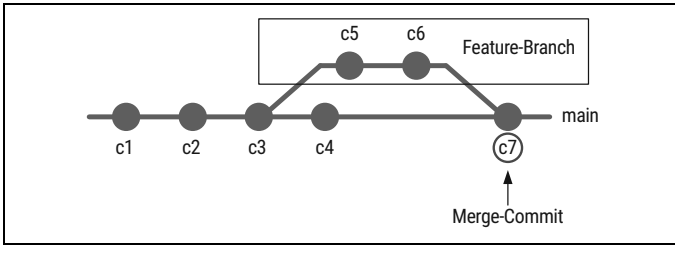

Abbildung 3-10: Gab es Änderungen im main-Branch, führt Git einen 3-Wege-Merge durch und erstellt einen Merge-Commit.

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 [filename]
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:     [filename]

  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 und anschließend dem Index hinzugefügt werden, 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 mergetool`. Die Oberflächen der meisten Tools sehen ziemlich antiquiert aus, lass dich davon nicht abschrecken. Alle bieten eine Darstellung zweier Versionen einer Datei – also der 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 Änderun-

gen der anderen hineingemergt hast. `theirs` ist dementsprechend der Stand des Branchs, den du in deinen oder »unseren« Branch gemergt hast.

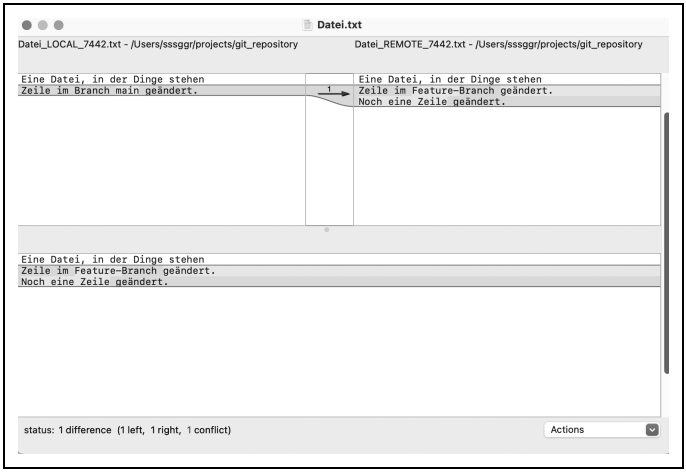

Abbildung 3-11: Ein Screenshot des Merge-Tools FileMerge

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.

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.

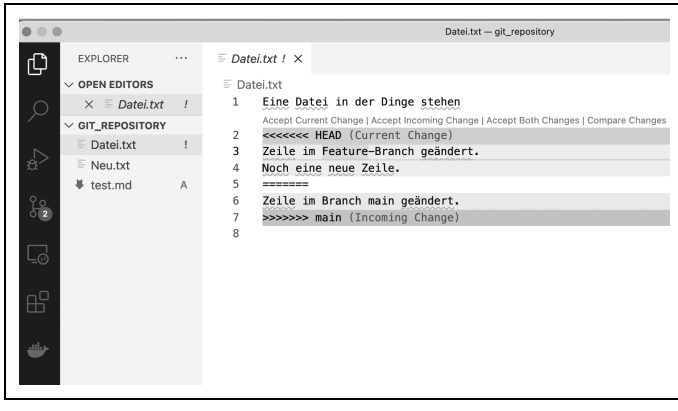

Abbildung 3-12: Die Ansicht eines Merge-Konflikts im Editor Visual Studio Code

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« auf Seite 146.

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

KAPITEL 6

Good Practices

Im folgenden Kapitel geht es um Praktiken, die ich bei der Arbeit mit Git als wichtig oder nützlich empfinde. Ganz bewusst verwende ich an dieser Stelle nicht den Begriff *Best Practices*, impliziert dieser doch, dass es eine Lösung gibt, die für alle die richtige ist. Das glaube ich nicht. Vielmehr denke ich, dass es wichtig ist, zu überlegen, welchen Wert eine Praktik oder ein bestimmter Workflow hat. Davon ausgehend kann ich überlegen, ob ein Einsatz im konkreten Fall sinnvoll ist, und mir dann ein individuelles Set an Good Practices für den spezifischen Anwendungsfall zusammenstellen.

Gute Commits

Commits sind ein zentraler Baustein bei der Arbeit mit Git, deshalb wird ihnen hier ein ganzer Abschnitt gewidmet. Aus den Commits entsteht die Git-Historie. Sie erlaubt es, am Ende nachzuvollziehen, was wann warum geändert wurde. Und damit die Commits dieses Versprechen auch wirklich einlösen können, lohnt es sich, gut zu überlegen, was in einem Commit zusammengefasst wird und wie es beschrieben werden sollte.

Kleine logische Einheiten

Ein Commit sollte Änderungen am Code in kleinstmöglichen Schritten abbilden, das meint in Schritten, die in sich abgeschlossen sind und nichts kaputt machen. Das heißt, die Software sollte erfolgreich gebaut werden können, und es sollte kein Test fehlschlagen. Bei einer größeren Änderung arbeitest du dich also Schritt für

Schritt durch die notwendigen Modifikationen. Für jeden einzelnen Schritt erstellst du einen Commit.

Hier eine Beispiel-Commit-Nachricht für einen ziemlich großen Commit.

```
Add sign out

Update dependencies, add missing unit test for current user method,
rename variable and add button to sign out
```

Schon in der Beschreibung wird deutlich, dass hier ganz schön viel passiert. Das eben genannte Beispiel ließe sich auch in vier kleinere Commits aufteilen.

```
Update dependencies

Add missing unit test for current user method

Rename variable

Add button to sign out
```

Mir hilft es, zu überlegen, wie ich meine Änderungen zusammenfassen kann. Fällt es mir schwer, eine prägnante und kurze Beschreibung zu finden, ist das oft ein guter Indikator dafür, besser noch mal zu überprüfen, ob sich der Commit nicht doch in mehrere Teile, also mehrere Commits aufteilen ließe. Wenn ich z. B. bei einer Zusammenfassung wie *Methode refaktorisiert, Bugfix gemacht und dieses Features hinzugefügt* lande, ist etwas faul. Kleinere Änderungen, wie z. B. die Korrektur von Tippfehlern oder das Extrahieren einer Methode, gehören immer in einen eigenen Commit.

Warum diese Kleinteiligkeit?

- Änderungen kleiner logischer Einheiten sind leichter nachzuvollziehen: beim Eintauchen in die Historie eines Projekts und auch bei einem Code-Review.
- Es gibt weniger bzw. überschaubare Merge-Konflikte.
- Wurden mit einem Commit Bugs eingeführt, sind diese viel leichter zu finden.

- Das Rückgängigmachen von Änderungen wird einfacher, denn ich kann dabei sehr selektiv vorgehen. Enthält ein Commit mehrere Änderungen, kann es natürlich passieren, dass beim Rückgängigmachen so eines Commits auch Änderungen verloren gehen, die man unter Umständen gern behalten hätte.

Gute Nachrichten schreiben

Sind die Commits, wie im vorangegangenen Abschnitt beschrieben, sinnvolle, überschaubare Einheiten, müssen diese nun noch gut beschrieben werden. Der Lohn ist eine schnell zu verstehende und leicht nachvollziehbare Commit-Historie, die es dir, deinem zukünftigen Ich und anderen aktuellen und zukünftigen Teammitgliedern merklich erleichtert, nachzuvollziehen, wann was warum geändert wurde.

Commits und ihre Beschreibung sind für das Verständnis von Softwareprojekten von zentraler Bedeutung. Da Englisch in der Softwareentwicklung das Hauptverständigungsmittel ist, ist es üblich, die Commit-Nachrichten auf Englisch zu verfassen. Das macht Projekte zukunftssicher, denn so könnte künftig auch ein internationales Team gut mit der Codebasis arbeiten. Außerdem gibt es für viele der im Bereich der Softwareentwicklung gängigen englischen Begriffe im Deutschen keine eindeutigen oder eher merkwürdige Übersetzungen.

Eine Commit-Nachricht wie *Add Feature XYZ* oder *Refactoring* ist ziemlich nichtssagend, und ein halbes Jahr später weißt du vermutlich selbst nicht mehr, um was es genau ging und welche Motivation hinter der Änderung stand. Beliebt sind auch Emojis, die allerdings noch größeren Interpretationsspielraum bieten. So sollte die Commit-Historie daher nicht aussehen:

```
7dc5391 Bugfix
6f58e98 WIP
98a8b5d Minor changes
f8d1e31 More Code
04c8470 What the hell is going on?
```

So ist es schon besser:

```
7dc5391 Update dependencies
6f58e98 Simplify current_task method
f8d1e31 Change syntax for ignoring attributes
04c8470 Add upload file method
```

Nimm dir also genug Zeit, um eine gute Commit-Nachricht zu schreiben. Die Abfolge der Commits sollte im Idealfall eine Geschichte erzählen. Daher ist es wichtig, Commit-Nachrichten nicht als lästige Notwendigkeit zu behandeln, sondern sich Mühe beim Verfassen zu geben. Es sollten natürlich keine Romane werden, aber es ergibt häufig durchaus Sinn, etwas mehr Text als nur eine Betreffzeile zu schreiben.

Ein Template für Commit-Nachrichten verwenden

Es kann hilfreich sein, ein Template für Commit-Nachrichten zu nutzen. Das enthält z. B. Erinnerungsstützen, die festhalten, welche Fragen die Commit-Nachricht klären soll. Um so ein Template zu definieren, legst du in deinem Home-Verzeichnis eine Datei mit dem Namen *.gitmessage* ab. Bei mir sieht diese Datei so aus:

```
$ cat ~/.gitmessage

# 50-character subject line
#
# 72-character wrapped longer description.
# This should answer:
#
# * Why was this change necessary?
# * How does it address the problem?
# * Are there any side effects?
#
# Include a link to the ticket, if any.
```

Anschließend musst du in der Git-Konfiguration noch hinterlegen, dass das Template verwendet werden soll.

```
$ git config --global commit.template
~/.gitmessage
```

Erstellst du nun einen Commit, ist das Editorfenster, das sich für die Commit-Nachricht öffnet, nicht mehr leer, sondern enthält den Text deines Templates.

Die Commit-Nachricht soll erklären, warum eine Änderung gemacht wurde. Was genau geändert wurde, kannst du dir im Diff anschauen. Du musst also nicht jeden Schritt deiner Änderung detailliert ausführen. Wer später die Commit-Historie liest, interessiert sich viel mehr für die Motivation hinter der Änderung. Löst sie vielleicht ein Problem, und, wenn ja, welches war das? Wurden andere Implementierungen in Erwägung gezogen und warum verworfen? Gibt es Seiteneffekte oder Dinge, die in späteren Commits adressiert werden sollten?

In der Git-Historie bekannter Projekte stöbern

Gute Commit-Nachrichten zu schreiben, ist gar nicht so einfach und erfordert etwas Übung. Um ein Gefühl für gute Commit-Nachrichten zu bekommen, lohnt es sich, die Repositories von Open-Source-Projekten anzuschauen und dort in der Git-Historie zu lesen. Du könntest dir z.B. die Commit-Historie von Git selbst einmal ansehen. Aber vielleicht interessiert dich auch, wie sie bei deiner Lieblings-Library aussieht.

Eine einheitliche Formatierung macht vieles leichter

Formatierungsregeln sorgen für Einheitlichkeit und machen es leichter, sich auf den Inhalt zu konzentrieren, anstatt sich von unterschiedlichen Darstellungsformen ablenken zu lassen. Daher gibt es einige Regeln, die du beherzigen solltest:

- Die Betreffzeile und die weitere Beschreibung der Änderung sollten durch eine Leerzeile voneinander getrennt werden.
- Die Betreffzeile sollte nicht mehr als 50 Zeichen haben. Wird sie zu lang, bringt sie die Änderung schlicht nicht auf den Punkt. An vielen Stellen, wie beispielsweise in der GitHub-Weboberfläche, wird außerdem nur eine bestimmte Länge angezeigt. Ausführlicher kannst du in der Beschreibung werden.
- Der Betreff sollte mit Großbuchstaben beginnen, und am Ende der Zeile sollte kein Punkt gesetzt werden. Dadurch liest sich die Betreffzeile wie die Überschrift eines Artikelabschnitts. Die darauffolgende Beschreibung ist ein normaler Textabschnitt.

Inhalt