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. Spätere Leser*innen der Commit-Historie interessiert viel mehr 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 anschauen. 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 da-