# 17 Mit Feature-Branches entwickeln

Wenn alle im Team auf einem gemeinsamen Branch entwickeln, entsteht eine sehr unübersichtliche History mit vielen zufälligen Merge-Commits. Dadurch wird es schwierig, Änderungen für ein bestimmtes Feature oder einen Bugfix[1] nachzuvollziehen.

***Gemeinsam auf einem Branch entwickeln***
*→ Seite 135*

Während der Entwicklung von Features sind kleinschrittige Commits hilfreich, um jederzeit auf einen alten funktionierenden Stand zurückzufallen. Doch wenn man sich einen Überblick über die im Release enthaltenen neuen Features verschaffen will, sind grobgranulare Commits sinnvoller. Bei diesem Workflow werden die kleinschrittigen Commits auf dem *Feature-Branch* und die Release-Commits auf dem `master`-Branch angelegt.

***Commits zusammenstellen***
*→ Seite 39*

Die Integration von Feature-Branches wird mithilfe von *Pull-Requests* durchgeführt. Dadurch können Code-Reviews und weitere Qualitätsmaßnahmen, wie Builds und Tests, vor dem eigentlichen Merge in den `master`-Branch durchgeführt werden.

Dieser Workflow zeigt, wie Feature-Branches so eingesetzt werden, dass

- die Entwicklung von Features untereinander entkoppelt ist,
- Pull-Requests für Code-Reviews und andere Qualitätsmaßnahmen benutzt werden können,
- die Commits, die ein Feature implementieren, einfach aufzufinden sind,
- die First-Parent-History des `master`-Branch nur grobgranulare Feature-Commits beinhaltet, die als Release-Dokumentation dienen können,
- wichtige Änderungen des `master`-Branch während der Feature-Entwicklung benutzt werden können.

[1] In Git werden Features und Bugs unter dem Begriff *Topic* zusammengefasst. Entsprechend wird häufig auch von *Topic-Branches* gesprochen.

## Überblick

Abbildung 17–1 zeigt die Grundstruktur, die beim Arbeiten mit Feature-Branches entsteht. Ausgehend vom `master`-Branch wird für jedes Feature oder jeden Bugfix (nachfolgend werden Bugfixes nicht mehr explizit aufgeführt) ein neuer Branch angelegt. Dieser Branch wird benutzt, um alle Änderungen und Erweiterungen durchzuführen. Sobald das Feature in den `master`-Branch integriert werden soll, wird ein Pull-Request angelegt. Pull-Requests können vor dem Merge mit dem `master`-Branch durch verschiedene Qualitätsmaßnahmen (Code-Reviews, Builds, Tests) überprüft werden.

Der Austausch zwischen Feature-Branches findet ausschließlich über den `master`-Branch statt. Gibt es Abhängigkeiten zwischen Features, so muss das Feature, welches Lieferungen für andere Features bereitstellen muss, zeitlich früher eingeplant werden. Wird die Entwicklung der anderen Features dadurch blockiert, so muss das Feature in kleinere Features aufgeteilt werden, sodass die wichtigen Lieferungen schneller erfolgen (siehe `feature/a1` und `feature/a2`).

Der Entwickler eines Feature-Branch kann sich notwendige Neuerungen des `master`-Branch jederzeit durch einen Merge in den Feature-Branch holen.

## Voraussetzungen

**Feature-basiertes Vorgehen:** Die Planung des Projekts bzw. Produkts muss auf Features basieren, d. h., fachliche Anforderungen werden in Feature-Aufgabenpakete überführt. Features haben untereinander wenige Abhängigkeiten.

**Kleine Features:** Die Entwicklung eines Features muss in Stunden oder Tagen abgeschlossen werden können. Je länger die Feature-Entwicklung parallel zu der restlichen Entwicklung läuft, umso größer ist das Risiko, dass bei der Integration des Features große Aufwände entstehen.

**Zentrale Repository-Verwaltung mit Pull-Requests:** Im Projekt ist eine zentrale Repository-Verwaltung im Einsatz, die Pull-Requests unterstützt.

Workflow kompakt

# Mit Feature-Branches entwickeln

*Jedes Feature oder jeder Bugfix wird in einem separaten Branch entwickelt. Nach der Fertigstellung wird das Feature oder der Bugfix mithilfe eines Pull-Request in den* `master`*-Branch integriert.*

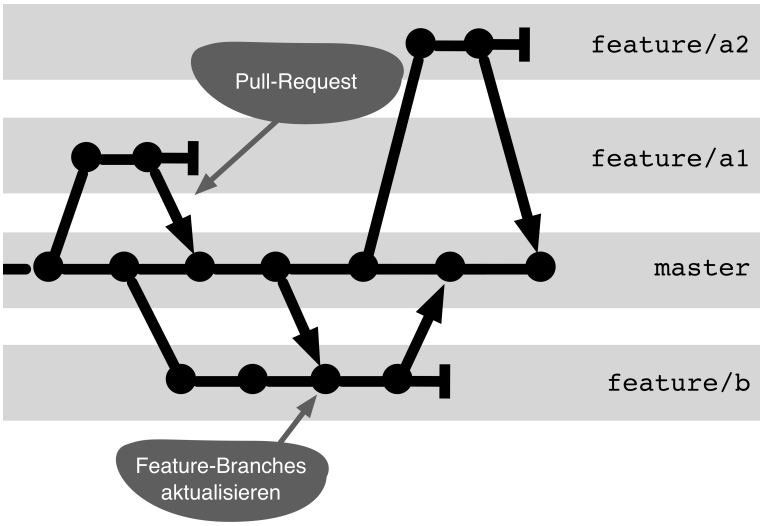


***Abb. 17–1***
*Workflow im Überblick*

## Ablauf und Umsetzung

Für die folgenden Abläufe wird von einem zentralen Repository ausgegangen. Die Entwicklung findet wie immer in einem lokalen Klon statt. Das zentrale Repository wird im Klon über das Remote `origin` angesprochen.

***Ein Projekt aufsetzen***
*→ Seite 123*
***Wie sagt man Git, wo das Remote-Repository liegt?***
*→ Seite 91*

### Feature-Branch anlegen

Sobald ein neues Feature bearbeitet werden soll, wird ein neuer Branch erzeugt. Dabei ist darauf zu achten, dass der Branch ausgehend vom `master`-Branch angelegt wird.

#### Schritt 1: `master`-Branch aktualisieren

Wenn gerade Zugriff auf das zentrale Repository besteht, ist es sinnvoll, als Erstes den `master`-Branch auf den neuesten Stand zu bringen. Dabei kann es zu keinen Merge-Konflikten kommen, da bei Feature-basierten Arbeiten im lokalen Repository nicht auf dem `master`-Branch gearbeitet wird.

***Fetch: Branches aus einem anderen Repository holen***
*→ Seite 99*

```
> git checkout master
> git pull --ff-only
```

**`--ff-only`:** Nur ein Fast-Forward-Merge ist erlaubt. Das heißt, wenn lokale Änderungen vorliegen, wird der Merge abgebrochen.

Falls der Merge mit einer Fehlermeldung abbricht, dann wurde vorab aus Versehen direkt auf dem `master`-Branch gearbeitet. Diese Änderungen müssen als Erstes in einen Feature-Branch verschoben werden (»Commits auf einen anderen Branch verschieben« ab Seite 114).

#### Schritt 2: Feature-Branch anlegen

*Branches verzweigen* → *Seite 59*

Anschließend kann der neue Branch angelegt werden, und die Arbeit kann beginnen:

```
> git checkout -b feature/a1
```

*Tipp: Verwenden Sie einheitliche Namen für Branches.*

Es ist sinnvoll, sich im Team auf eine einheitliche Namensgebung von Feature- und Bugfix-Branches festzulegen. Git unterstützt auch hierarchische Namen für Branches, so werden Feature-Branches häufig mit dem Präfix `feature` begonnen.

Werden Features oder Bugfixes mit einem Tracking-Werkzeug verwaltet (z. B. Redmine[2], Jira[3]), so können deren eindeutige Nummern oder Token für den Branch-Namen verwendet werden.

#### Schritt 3: Feature-Branch zentral sichern

In diesem Workflow werden Pull-Requests verwendet. Damit Pull-Requests in der zentralen Repository-Verwaltung angelegt werden können, müssen die Feature-Branches auch zentral verfügbar sein.

*Austausch zwischen Repositorys* → *Seite 95*

Dazu wird der Branch im zentralen Repository mit dem `push`-Befehl angelegt:

```
> git push --set-upstream origin feature/a1
```

**`--set-upstream`:** Dieser Parameter verknüpft den lokalen Feature-Branch mit dem neuen Remote-Branch. Das heißt, zukünftig kann bei allen `push`- und `pull`-Befehlen auf ein explizites Remote verzichtet werden.

**`origin`:** Das ist der Name des Remote (der Alias für das zentrale Repository), auf dem der Feature-Branch gesichert werden soll.

---

[2] *http://www.redmine.org*

[3] *https://www.atlassian.com/software/jira*

Änderungen an dem lokalen Feature-Branch können zukünftig durch einen einfachen `push`-Befehl zentral gesichert werden:

```
> git push
```

## Alternative: Feature-Branch in zentraler Repository-Verwaltung anlegen

Die meisten zentralen Repository-Verwaltungen erlauben auch das Anlegen von Branches direkt über die Weboberfläche (Abbildung 17–2).

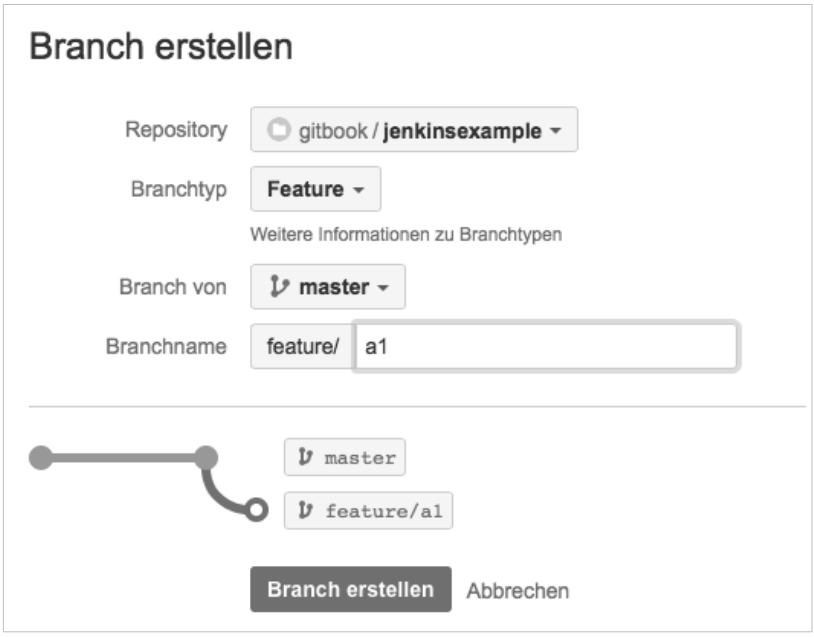


*Abb. 17–2*
*Feature-Branch in Weboberfläche anlegen*

Anschließend einfach mit dem `fetch`-Befehl alle Remote-Branches aktualisieren. Dabei wird man den gerade angelegten Feature-Branch als neuen Remote-Branch sehen.

```
> git fetch
```

Mit dem `checkout`-Befehl erzeugt man den neuen lokalen Feature-Branch.

```
> git checkout feature/a1
```

*Tipp: Tab-Completion bei Branch-Namen*

Wenn Sie mit der Git-Bash arbeiten, können Sie sich die Eingabe des Branch-Namens erleichtern, indem Sie nach den ersten Buchstaben die TAB-Taste drücken. Es wird dann automatisch der Branch-Name vervollständigt. Wenn Sie zweimal TAB drücken, bekommen Sie alle passenden Branches angezeigt.

## Feature in den `master`-Branch integrieren

***Warum Feature-Branches nicht erst kurz vor dem Release integrieren?***
*→ Seite 156*

Wie wir bereits in den Voraussetzungen definiert haben, ist es wichtig, dass Features nicht zu lange parallel existieren. Ansonsten nimmt die Gefahr von Merge-Konflikten und inhaltlichen Inkompatibilitäten stark zu. Selbst wenn das Feature noch nicht in das nächste Release einfließen soll, ist es sinnvoll, die Integration zeitnah durchzuführen und besser mit einem Feature-Toogle die Funktionalität zu deaktivieren.

In diesem Abschnitt wird beschrieben, wie das Feature mithilfe eines Pull-Request in den `master`-Branch integriert wird. Pull-Requests (manchmal auch *Merge-Requests* genannt) werden von vielen zentralen Repository-Verwaltungen, z. B. Atlassian Bitbucket Server[4], GitHub[5] oder GitLab[6], unterstützt. Nachfolgend werden die Schritte exemplarisch mit Atlassian Bitbucket Server vorgestellt.

***Network of Trust***
*→ Seite 301*

Pull-Requests kommen ursprünglich aus der Open-Source-Entwicklung, um kontrolliert Änderungen zu integrieren. Es wird kein einfacher lokaler Merge zwischen Branches durchgeführt, sondern eine Anfrage an einen Integrator bzw. an eine zentrale Repository-Verwaltung versandt, mit der Aufforderung, diesen Merge nach Überprüfung durchzuführen.

Im Kontext von Feature-Branches ermöglichen Pull-Requests die Integration des Feature in den `master`-Branch und können gleichzeitig die Qualität des Features vorab durch automatische Builds oder Reviews erhöhen.

### Schritt 1: Pull-Request anlegen

Nachdem das Feature entwickelt wurde, werden die Commits des Feature-Branch in die zentrale Repository-Verwaltung übertragen.

```
> git push
```

Dann kann man über eine Weboberfläche den Pull-Request anlegen. Dabei gibt man den Quell-Branch (Feature-Branch) und den Ziel-Branch (`master`-Branch) an (Abbildung 17–3).

### Schritt 2: Pull-Request bearbeiten

***Pull-Requests bauen***
*→ Seite 269*

Als Ergebnis entsteht ein neuer Pull-Request, der wiederum über die Weboberfläche bearbeitet werden kann. Wenn erforderlich oder gewünscht, kann ein Code-Review auf den Änderungen des Pull-Request

[4] *www.atlassian.com/software/bitbucket/server*

[5] *https://github.com*

[6] *https://gitlab.com/*

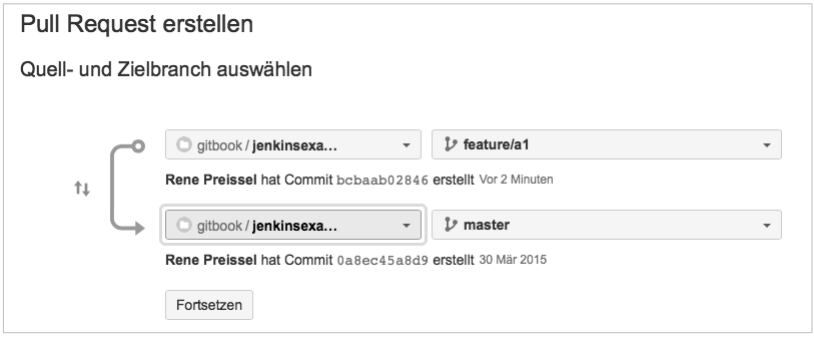


*Abb. 17–3*
*Pull-Request anlegen*

durchgeführt werden[7]. Die Reviewer können direkt in der Weboberfläche den gesamten Pull-Request oder einzelne Codezeilen kommentieren und ihre Zustimmung oder Ablehnung für den Pull-Request hinterlegen. Wenn ein entsprechendes Build-System angebunden ist, kann der Pull-Request gebaut und getestet werden. Die Ergebnisse des Builds und der Tests werden wiederum am Pull-Request festgehalten (Abbildung 17–4).

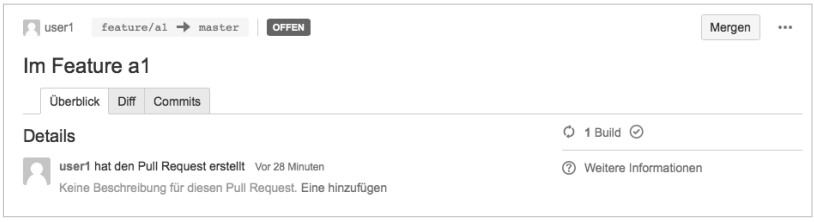


*Abb. 17–4*
*Pull-Request bearbeiten*

Wenn Nacharbeiten am Feature notwendig sind, weil ein Reviewer oder das Build-System Probleme festgestellt hat, müssen diese durch neue Commits auf dem Feature-Branch erledigt werden.

```
> git commit
> git push
```

Sobald die neuen Commits durch Push in das zentrale Repository übertragen wurden, wird der Pull-Request aktualisiert, und es ist für jeden ersichtlich, dass Änderungen erfolgt sind (Abbildung 17–5).

Wenn ein Merge des Feature-Branch mit dem `master`-Branch wegen Konflikten in Dateien nicht möglich ist, wird das ebenso am Pull-Request angezeigt (Abbildung 17–6). Dann muss der Feature-Branch lokal mit dem aktuellen `master`-Branch vereinigt und das entstandene

[7] Einige Repository-Verwaltungen erlauben es, diesen Code-Review als zwingende Voraussetzung für das Mergen zu definieren.

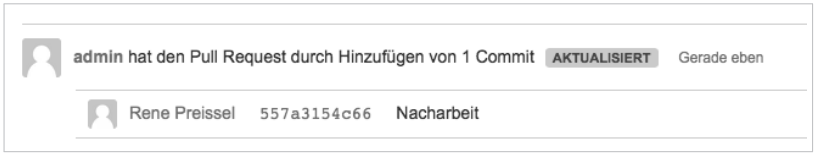


*Abb. 17–5*
*Pull-Request um neue Commits erweitern*

Merge-Commit wiederum durch Push in das zentrale Repository übertragen werden (siehe »Änderungen des `master`-Branch in den Feature-Branch übernehmen« ab Seite 151).


*Abb. 17–6*
*Konflikte im Pull-Request*

### Schritt 3: Pull-Request abschließen

Sind alle Voraussetzungen erfüllt, kann der Pull-Request abgeschlossen werden, d. h., ein Merge wird durchgeführt. Dazu wird der entsprechende Webdialog gestartet (Abbildung 17–7).


*Abb. 17–7*
*Pull-Request mergen*

Alternativ kann man auch den Pull-Request ohne Merge beenden: In Bitbucket Server: Ablehnen bzw. in GitHub: Close pull request. Dann wird der Pull-Request ohne Merge geschlossen und es kann auf dem Feature-Branch weiter gearbeitet werden.

### Schritt 4: Feature-Branch löschen

Typischerweise wird nach dem Merge der Feature-Branch gelöscht. Das Löschen des zentralen Feature-Branch ist direkt über die Merge-Weboberfläche möglich (siehe Abbildung 17–7 unten). Alternativ kann

der remote Feature-Branch auch über die Kommandozeile gelöscht werden:

```
> git push --delete origin feature/a1
```

Das Löschen des lokalen Feature-Branch muss immer manuell passieren:

```
> git checkout master
> git branch -d feature/a1
```

**`-d`:** Löscht den übergebenen Branc.

## Änderungen des `master`-Branch in den Feature-Branch übernehmen

Im besten Fall findet die Entwicklung eines Features unabhängig von anderen Features statt.

Manchmal gibt es jedoch auf dem `master`-Branch wichtige Änderungen, die für die Entwicklung des Features notwendig sind, z. B. große Refaktorierungen oder Neuerungen an grundlegenden Services. Ein anderer typischer Fall ist, dass ein Pull-Request Konflikte zwischen dem `master` und dem Feature-Branch meldet. In solchen Situationen müssen die Änderungen des `master`-Branch in den Feature-Branch übernommen werden.

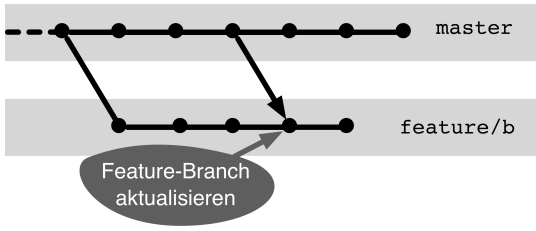


***Abb. 17–8***
*Änderungen aus dem Master- in den Feature-Branch übernehmen*

Abbildung 17–8 veranschaulicht die Situation. Es soll ein Merge vom `master`-Branch in den Feature-Branch durchgeführt werden:

### Schritt 1: `origin/master`-Branch aktualisieren

Als Erstes müssen die Änderungen des `origin/master`-Branch in das lokale Repository importiert werden. Der `fetch`-Befehl aktualisiert alle Remote-Branches:

```
> git fetch
```

#### Schritt 2: Änderungen in den Feature-Branch übernehmen

*Branches zusammenführen* → *Seite 67*

Im zweiten Schritt müssen die Änderungen durch einen Merge in den Feature-Branch übernommen werden:

```
> git merge origin/master
```

Falls es zu Konflikten kommt, müssen diese mit den normalen Mitteln gelöst werden.

Der Zwischenstand kann beliebig oft vom `master`-Branch in den Feature-Branch übernommen werden. Git kann sehr gut mit mehrfachen Merges umgehen. Allerdings wird dadurch die Commit-Historie komplexer und schwerer lesbar.

## Warum nicht anders?

### Warum nicht auf Pull-Requests verzichten?

Kann man den Feature-Branch-Workflow nicht auch ohne Pull-Requests durchführen, d. h., die Integration des Feature-Branch manuell durchführen?

Das funktioniert natürlich auch. Dabei verliert man aber einerseits den Vorteil der automatischen und manuellen Prüfungen vor der Integration. Andererseits ist der manuelle Merge des Feature-Branch in den `master`-Branch nicht trivial, insbesondere wenn man eine richtige und vollständige First-Parent-History des `master`-Branch erzeugen will. Zusätzlich erhält man durch Pull-Requests in der Weboberfläche eine gute Übersicht über das, was im Projekt passiert.

Der Punkt 1 ist leicht zu verstehen. Mit Pull-Request kann man ein Quality-Gate vor der Integration in den `master`-Branch errichten. Es kommen keine Commits in den `master`-Branch, die nicht den definierten Qualitätsansprüchen genügen.

Für Punkt 2 muss man sich z. B. das Szenario anschauen, wenn der lokale Merge des Feature-Branch in den `master`-Branch funktioniert hat, aber der nachfolgende Push abgewiesen wird. In Abbildung 17–9 oben und in der Mitte ist die beschriebene Situation skizziert: Remote wurde das `C`-Commit und lokal das `D`-Commit angelegt.

Würde man jetzt, wie typischerweise in solchen Situationen, einen `pull`-Befehl absetzen, dann entstünde ein neues Merge-Commit `E` (Abbildung 17–9 unten). Damit würde in der First-Parent-History des `master`-Branch das `C`-Commit nicht mehr enthalten sein.

*Branch-Zeiger umsetzen* → *Seite 64*

In der First-Parent-History sollen aber alle Features enthalten sein, deswegen muss bei einem fehlgeschlagenen `push`-Befehl das lokale

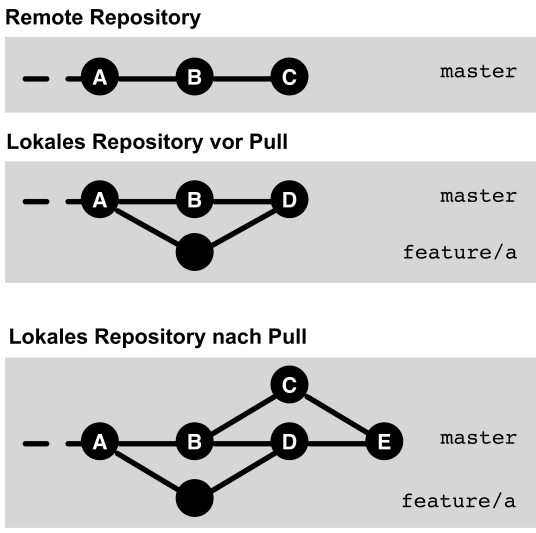


*Abb. 17–9*
*Keine verwendbare First-Parent-History nach dem Pull-Befehl*

Feature-Merge-Commit (`D`-Commit) mit dem `reset`-Befehl entfernt und der Merge mit dem aktuellen `master`-Branch wiederholt werden:

```
> git reset --hard HEAD^
```

Mit genügend Sorgfalt ist dieser Ablauf auch ohne Pull-Requests möglich, doch ist es schwer zu überprüfen, ob jeder Entwickler zu jedem Zeitpunkt die richtige Operation durchführt. Das Ergebnis ist dann häufig eine kaputte bzw. inkorrekte First-Parent-History.

Git-Erweiterungen

## Git-Flow: High-Level-Operationen

*Git-Flow*[8] ist eine Sammlung von Skripten, um den Umgang mit Branches, insbesondere Feature-Branches, zu vereinfachen. Wenn Sie ohne Pull-Request arbeiten wollen, dann sollten Sie sich die Skripte anschauen.
So kann ein neuer Feature-Branch folgendermaßen erzeugt und gleichzeitig aktiviert werden:

```
> git flow feature start feature/a1
```

Am Ende kann der Feature-Branch in den `master`-Branch übernommen und gleichzeitig gelöscht werden.

```
> git flow feature finish feature/a1
```

[7] *https://github.com/nvie/gitflow*

## Warum nach einem Pull-Request nicht auf dem Feature-Branch weiterarbeiten?

Wenn es Abhängigkeiten zwischen Features gibt und die Entwicklung des liefernden Feature lange dauert, teilt man ein Feature in Teillieferungen auf: Aus einem großen Feature werden mehrere kleine Features.

Sobald die erste Teillieferung fertig ist, kann ein Pull-Request erstellt werden, und sobald dieser abgeschlossen ist, können andere Features die Änderungen aus dem `master`-Branch holen.

Jetzt könnte der Entwickler der nächsten Teillieferung auf die Idee kommen, den bereits vorhandenen Feature-Branch für die weitere Entwicklung zu nutzen (Abbildung 17–10).

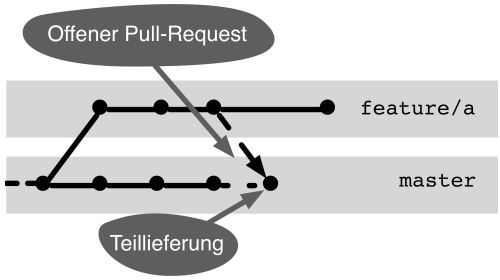


***Abb. 17–10*** *Wiederverwendung eines Feature-Branch*

Das Problem entsteht dabei, sobald Commits für die neue Teillieferung durch Push in das zentrale Repository übertragen werden und der Pull-Request der ersten Teillieferung noch nicht abgeschlossen ist. Dann wird der noch offene Pull-Request aktualisiert und die noch unvollständigen Änderungen werden ggf. schon integriert. Genauso kommt es zu Problemen, wenn Nacharbeiten an der ersten Teillieferung notwendig werden und diese in Konflikt zu den neuen Änderungen stehen.

Deswegen sollte man einen Feature-Branch nicht für weitere Teillieferungen wiederverwenden, sondern einen neuen Branch anlegen. Sollte die Bearbeitung des ersten Pull-Request länger dauern und benötigt man für die zweite Teillieferung die Ergebnisse der ersten Teillieferung, dann ist es möglich, einen neuen Feature-Branch bei dem letzten Commit des ersten Feature-Branch zu beginnen (Abbildung 17–11).

Sind noch Nacharbeiten notwendig, können diese autark auf dem ersten Feature-Branch durchgeführt werden. Die Änderungen sind dann ganz normal über den `master`-Branch in den zweiten Feature-Branch zu holen.

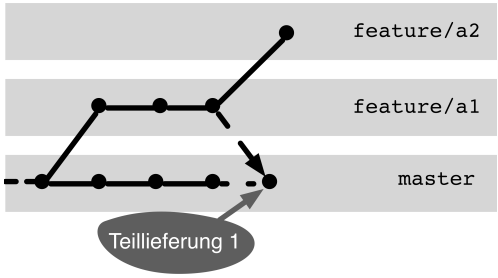


*Abb. 17–11*
*Zweite Teillieferung auf neuen Feature-Branch*

## Warum keinen Rebase des Feature-Branch vor dem Pull-Request durchführen?

Hat man ein größeres Feature entwickelt und mehrfach die Änderungen aus dem `master`-Branch geholt, entsteht eine Historie wie in Abbildung 17–12.

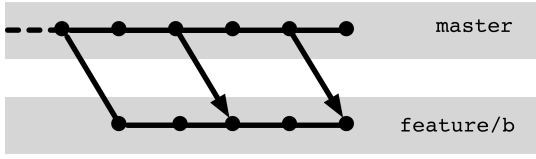


*Abb. 17–12*
*Feature-Branch mit mehrfacher Integration des `master`-Branch*

Man sieht in dem Beispiel zwei Merge-Commits im Feature-Branch. Nach dem Pull-Request sieht es aus wie in Abbildung 17–13.

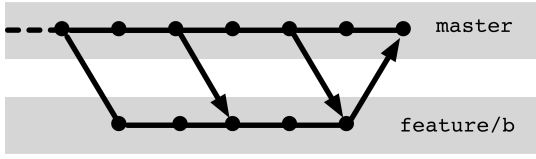


*Abb. 17–13*
*Viele Merge-Commits nach Integration*

Würde man vor dem Erstellen des Pull-Request ein Rebase durchführen, verschwänden die Merge-Commits auf dem Feature-Branch und die Historie würde nach der erfolgreichen Integration wie in Abbildung 17–14 aussehen.

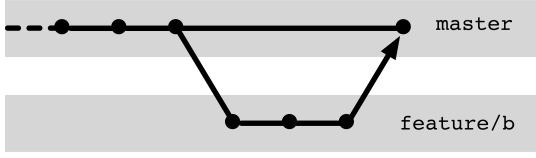


*Abb. 17–14*
*Feature-Branch nach Rebase und Integration*

Der Vorteil dieses Vorgehens ist eine »schönere« Historie, in der man die Feature-Branches sehr gut erkennen kann. Der Nachteil ist, dass nach dem Rebase der `push`-Befehl mit der Option `--force` ausgeführt werden muss, d. h., die bisher vorhandene Historie des Feature-Branch wird überschrieben. Wenn mehr als ein Entwickler an dem Feature arbeitet, erfordert das eine Abstimmung im Team. Außerdem entstehen, wie bei jedem Rebase, Commits, die von keinem Entwickler überprüft worden sind.

Wenn alle im Team sich sehr gut mit Git auskennen und Rebase verstanden haben, ist diese Variante hilfreich für eine »schöne« Historie. Für den Einstieg mit Feature-Branches oder bei wechselnden Teammitgliedern ist die Gefahr von Fehlern vorhanden.

*Tipp: Force-Push für `master`-Branch verbieten*

Falls Sie sich für das Rebasing von Feature-Branches entscheiden, sollten Sie als Vorsichtsmaßnahme unbedingt alle Nicht-Feature-Branches (`master`-Branch etc.) vor dem versehentlichen Überschreiben mittels Force-Push schützen.

## Warum Feature-Branches nicht erst kurz vor dem Release integrieren?

Bei der Arbeit mit Feature-Branches kommt das Release-Management häufig auf die Idee, die Entscheidung, welche Features in das neue Release kommen sollen, erst kurz vor dem Liefertermin zu treffen.

Konzeptionell scheint das mit dem Feature-Branch-Ansatz auch sehr einfach zu gehen. Jedes Feature wird in einem Branch vollständig entwickelt, jedoch noch nicht in den `master`-Branch integriert. Kurz vor dem entscheidenden Tag wird erst beschlossen, welche Features in den `master`-Branch integriert werden sollen.

In einer idealen Welt – mit völlig unabhängigen Features und ohne Programmierfehler – wäre dieses Vorgehen auch anwendbar. In der Realität führt dieses Vorgehen jedoch meistens zu größeren Merge-Konflikten bei der Integration und zu langen Stabilisierungsphasen.

Außerdem wird es komplizierter, abhängige Features zu entwickeln. Normalerweise würde man bei Abhängigkeiten zwischen Features das erste Feature in den `master`-Branch integrieren und das andere Feature könnte sich die Änderungen holen. Bei der Lösung mit der späten Integration müssen die Feature-Branches direkt die Änderungen austauschen (siehe den folgenden Abschnitt). Damit wäre die unabhängige Integration dieser Features kurz vor dem Release unmöglich.

Auch bewährte Prozesse für qualitative Software, wie Continuous Integration und Refaktorierungen, sind bei der späten Integration kaum umzusetzen.

***Integration mit Jenkins*** → *Seite 263*

## Warum nicht direkt Commits zwischen Feature-Branches austauschen?

Der beschriebene Workflow sieht keinen direkten Austausch von Commits zwischen Feature-Branches vor. Die Integration findet immer über Lieferungen im `master`-Branch statt.

Wäre es nicht einfacher, direkt zwischen Feature-Branches Merges durchzuführen?

Der entscheidende Vorteil von Feature-Branches ist die klare Zuordnung von Änderungen zu Features und die klare Zuordnung von Verantwortlichkeiten. Jeder Entwickler übernimmt die Verantwortung für seine Änderungen, und diese werden im Pull-Request durch Code-Reviews und andere Qualitätsmaßnahmen überprüft.

Würde man direkt zwischen Feature-Branches Commits austauschen, dann würde derjenige Entwickler, der als Erstes den Pull-Request stellt, alle Änderungen der anderen integrierten Features mit abgeben müssen. Er wäre also verantwortlich, dafür zu sorgen, dass der Code die entsprechende Qualität hat.

Abgesehen davon würde die Historie unübersichtlicher werden, und es wäre nicht mehr so einfach möglich, die Commits während der Feature-Entwicklung von jenen der Integration zu unterscheiden.

Schritt für Schritt

## Offene Feature-Branches anzeigen

*Es werden alle noch nicht mit dem* `master`*-Branch zusammengeführten Feature-Branches angezeigt.*

### 1. Offene Feature-Branches anzeigen

Arbeitet man konsequent mit Feature-Branches, hat man häufig mehr als einen aktiven Feature-Branch in seinem Repository. Mit dem `branch`-Befehl können alle noch nicht integrierten Branches angezeigt werden:

```
> git branch --no-merged master
```

**`--no-merged master`:** Zeigt alle Branches mit Commits an, die nicht im `master`-Branch enthalten sind, d. h. alle Feature-Branches, die noch nicht mit dem `master`-Branch zusammengeführt wurden.

Schritt für Schritt

## Integrierte Features anzeigen

*Es werden die zuletzt mit dem* `master`*-Branch zusammengeführten Feature-Branches angezeigt.*

### 1. Integrierte Features anzeigen

Arbeitet man konsequent mit Feature-Branches, erhält man mit der First-Parent-History des `master`-Branch eine Auflistung der letzten integrierten Pull-Requests:

```
> git log --first-parent --oneline master
```

**`--first-parent`:** Zeigt nur die Commits an, die über den ersten Parent eines Merge-Commit zu erreichen sind.

**`--oneline`:** Zeigt nur den abgekürzte Commit-Hash und die erste Zeile der Commit-Message an.

Schritt für Schritt

## Alle Änderungen eines integrierten Features anzeigen

*Alle Änderungen des Features werden als Diff angezeigt.*

Insbesondere bei nachträglichen Code-Reviews ist es wichtig, herauszufinden, welche Änderungen vorgenommen wurden.

In Abbildung 17–15 ist ein Beispiel eines Feature-Branch zu sehen. Nachfolgend werden die Commits entsprechend dieser Abbildung referenziert.

### 1. Commits des Features finden

Für einen Diff benötigt man alle Commits im `master`-Branch, die zu dem Feature gehören. Normalerweise wird dies nur ein Merge-Commit sein. Im Beispiel wird das Commit `H` gefunden.

```
> git checkout master
> git log --first-parent --oneline --grep="feature/a"
c52ce0a Pull Request für feature/a
```

**`--grep`:** Sucht in der Log-Meldung nach einem bestimmten Text.

### 2. Diff durchführen

Die Änderungen können angezeigt werden, indem ein Diff des gefundenen Commit mit dem First Parent des Commit durchgeführt wird. Damit werden genau die Änderungen sichtbar, die dieses Feature in Bezug auf den `master`-Branch eingebracht hat. Im Beispiel werden die Änderungen zwischen Commit `G` und `H` angezeigt.

```
> git diff c52ce0a^1 c52ce0a
```

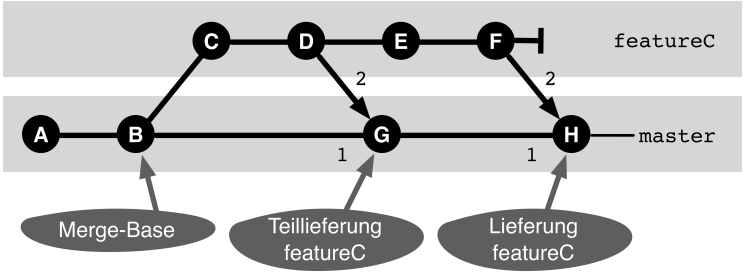


***Abb. 17–15***
*Beispiel eines Feature-Branch*

Schritt für Schritt

## Alle Commits eines Feature-Branch finden

*Alle Commits, die während der Feature-Entwicklung angelegt wurden, werden angezeigt.*

Während eines nachträglichen Code-Reviews will man genauer nachvollziehen, wie Änderungen eines Feature-Branch eingebaut wurden. Dazu kann es sinnvoll sein, alle Commits des bereits integrierten Feature-Branch einzeln anzuschauen.

In Abbildung 17–15 ist ein Beispiel für ein Feature-Branch zu sehen. Im Folgenden werden die Commits entsprechend dieser Abbildung referenziert.

### 1. Merge-Commits des Features finden

Als Erstes benötigt man alle Commits auf dem `master`-Branch, die etwas mit dem Feature zu tun haben. Normalerweise wird dies nur ein Merge-Commit sein. Im Beispiel wird das Commit `H` gefunden.

```
> git checkout master

> git log --first-parent --oneline --grep="feature/a"
c52ce0a Pull-Request für feature/a
```

**`--grep`:** Sucht in der Log-Meldung nach einem bestimmten Text.

### 2. Feature-Branch-Start finden

Um alle Commits anzuzeigen, wird das Commit des `master`-Branch gesucht, von dem der Feature-Branch abgezweigt ist. Ausgangspunkt ist das gefundene Commit des vorherigen Schritts (Commit `H`). Dieses muss ein Merge-Commit sein und hat zwei Parents. Mit dem `merge-base`-Befehl wird das Commit gesucht, das der gemeinsame Ausgangspunkt des ersten (Commit `G`) und des zweiten Parent (Commit `F`) ist – die Merge-Base (Commit `B`).

```
> git merge-base c52ce0a^2 c52ce0a^1
ca52c7f9bfd010abd739ca99e4201f56be1cfb42
```

**3. Feature-Commits anzeigen**

Nachdem der Ausgangspunkt gefunden wurde, können jetzt mit dem `log`-Befehl alle Commits des Feature-Branch angezeigt werden. Dazu wird gefragt, welche Commits notwendig sind, um von der Merge-Base (Commit `B`) auf das letzte Commit des Feature-Branch zu kommen. Das letzte Commit des Feature-Branch ist der zweite Parent (Commit `F`) des Merge-Commit (Commit `H`).

```
> git log --oneline ca52c7f..c52ce0a^2
```