Git

Dezentrale Versionsverwaltung im Team - Grundlagen und Workflows

von
René Preißel, Bjørn Stachmann

2., aktualisierte und erweiterte Auflage

Thematische Gliederung:

Software Engineering

dpunkt.verlag 2013

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 130 0

Inhaltsverzeichnis: Git – Preißel / Stachmann

16 Mit Feature-Branches entwickeln

Wenn alle im Team auf einem gemeinsamen Branch entwickeln, entsteht eine sehr unübersichtliche First-Parent-Historie mit vielen Merge-Commits. Dadurch wird es schwierig, Änderungen für ein bestimmtes Feature oder einen Bugfix[1] nachzuvollziehen. Insbesondere bei Codereviews und bei der Fehlersuche ist es hilfreich, genau zu wissen, welche Codezeilen für ein Feature geändert wurden. Durch den Einsatz von Feature-Branches kann man diese Informationen durch Git verwalten lassen. 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. Die grobgranulare Historie des `master`-Branch kann gut als Grundlage für die Release-Dokumentation dienen. Auch die Tester werden die grobgranularen Commits mit klarem Feature-Bezug begrüßen. Dieser Workflow zeigt, wie Feature-Branches eingesetzt werden, sodass

Gemeinsam auf einem Branch entwickeln
→ Seite 127

Commits zusammenstellen
→ Seite 27

- die Commits, die ein Feature implementieren, einfach aufzufinden sind,
- die First-Parent-Historie des `master`-Branch nur grobgranulare Feature-Commits beinhaltet, die als Releasedokumentation dienen können,
- Teillieferungen von Features möglich sind und
- wichtige Änderungen des `master`-Branches 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 16-1 zeigt die Grundstruktur, die beim Arbeiten mit Feature-Branches entsteht. Ausgehend vom `master`-Branch wird für jedes Feature oder jeden Bugfix (nachfolgend werden Bugs 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, muss ein Merge durchgeführt werden. Dabei muss darauf geachtet werden, dass der Merge immer ausgehend vom `master`-Branch angestoßen wird und dass Fast-Forward-Merges verhindert werden. Dadurch entsteht eine klare First-Parent-Historie auf dem `master`-Branch, die nur Merge-Commits von Features beinhaltet.

Branches zusammenführen *→ Seite 53*

Fast-Forward-Merges *→ Seite 59*

Gibt es Abhängigkeiten zwischen Features oder wird ein Feature inkrementell entwickelt, dann werden Teillieferungen in den `master`-Branch integriert, und danach wird auf dem Feature-Branch weiterentwickelt.

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

Voraussetzungen

Featurebasiertes Vorgehen: Die Planung des Projekts bzw. Produkts muss auf Features basieren, d. h., fachliche Anforderungen werden in Feature-Aufgabenpakete überführt. Features haben untereinander eine sehr geringe Überschneidung.

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.

Lokale Regressionstests: Bevor das neue Feature in den `master`-Branch integriert wird, müssen lokale Regressionstests auf dem Rechner des Entwicklers ausgeführt werden können. Dabei wird überprüft, ob die Änderungen des Features mit den Änderungen anderer Features zusammenarbeiten und ob es keine unerwünschten Seiteneffekte gibt. Falls es keine solchen lokalen Regressionstests gibt, werden Fehler häufig erst im integrierten `master`-Branch entdeckt. Die Behebung dieser Fehler führt zu einer nicht featurebezogenen Verzweigung der Historie, und damit ist der Hauptvorteil von Feature-Branches dahin.

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 in den `master`-Branch integriert.

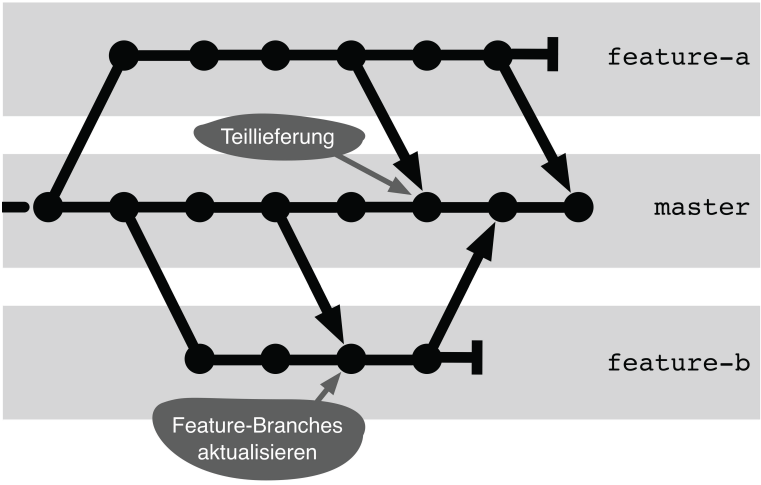


Abb. 16-1
Workflow im Überblick

16.1 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 111*

Im nachfolgenden Ablauf wird der `push`-Befehl benutzt, um lokale Änderungen in das zentrale Repository zu übertragen.

Bei der Arbeit mit Feature-Branches hat man häufig mehrere Branches im lokalen Repository. Damit der `push`-Befehl ohne Branch-Parameter nur den gerade aktiven Branch zum Remote überträgt, kann man die `push.default`-Option setzen.

```
> git config push.default upstream
```

Der Standardwert `matching` würde alle lokalen Branches übertragen, für die es einen gleichnamigen Remote-Branch gibt. Man müsste also bei jedem `push`-Befehl den Branch explizit angeben, um nur diesen zu übertragen.

Feature-Branch anlegen

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

Schritt 1: `master`-Branch aktualisieren

Abholen von Daten
→ Seite 75

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

```
> 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.

Branches umpflanzen
→ Seite 68

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 105).

Schritt 2: Feature-Branch anlegen

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

Branches verzweigen
→ Seite 45

```
> git checkout -b feature-a
```

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, z. B. `feature/a`.

Häufig werden Features und Bugfixes mit einem Tracking-Werkzeug verwaltet (z. B. Bugzilla[2], Mantis[3]). Diese Werkzeuge vergeben eindeutige Nummern oder Tokens für Features und Bugs. Diese Nummern können im Branchnamen verwendet werden.

[2] http://www.bugzilla.org
[3] http://www.mantisbt.org

Schritt 3: Optional: Feature-Branch zentral sichern

Häufig werden Feature-Branches nur lokal angelegt, insbesondere wenn sie nur eine kurze Lebenszeit haben.

Wenn die Implementierung eines Features jedoch länger dauert, die Sicherung der Zwischenergebnisse besonders wichtig ist oder mehrere Entwickler an einem Feature arbeiten sollen, dann kann der Branch auch im zentralen Repository gesichert werden.

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

Austausch zwischen Repositorys → *Seite 73*

```
> git push --set-upstream origin feature-a
```

`--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.

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

```
> git push
```

Feature in den `master`-Branch integrieren

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 mit dem `master`-Branch integriert wird. Dabei ist es wichtig, dass das notwendige Merge immer im `master`-Branch ausgeführt wird. Ansonsten erhält man keine sinnvolle First-Parent-Historie im `master`-Branch.

Schritt 1: `master`-Branch aktualisieren

Vor dem eigentlichen `merge`-Befehl muss der lokale `master`-Branch auf den aktuellsten Stand gebracht werden. Hierbei kann es zu keinen Konflikten kommen, da auf dem lokalen `master`-Branch nicht gearbeitet wird.

Abholen von Daten → *Seite 75*

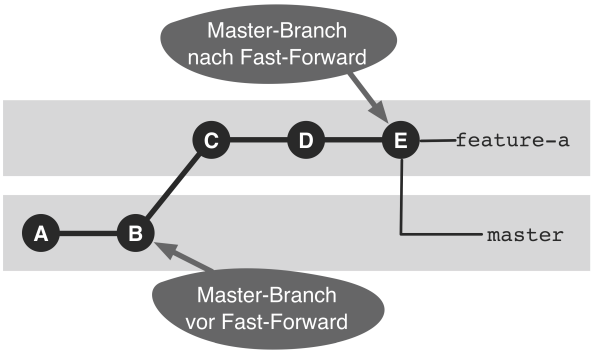
```
> git checkout master
> git pull --ff-only
```

Schritt 2: Merge des Feature-Branch durchführen

Branches zusammenführen
→ Seite 53

Die Änderungen des Feature-Branch werden mit einem `merge`-Befehl in den `master`-Branch übernommen. Damit die First-Parent-Historie des `master`-Branch als Feature-Historie dienen kann, muss ein Fast-Forward-Merge verhindert werden.

Abb. 16-2
Probleme mit Fast-Forward und der First-Parent-Historie



Fast-Forward-Merges
→ Seite 59

Abbildung 16-2 veranschaulicht die Probleme, die mit einem Fast-Forward-Merge auftreten können. Vor dem Merge zeigt der `master`-Branch auf das `B`-Commit und der Feature-Branch auf das `E`-Commit. Nach einem Fast-Forward-Merge zeigt der `master`-Branch nun auch auf das `E`-Commit. Die First-Parent-Historie des `master`-Branch würde jetzt die Zwischen-Commits `D` und `C` beinhalten.

Der folgende Befehl führt einen Merge durch und verhindert einen Fast-Forward-Merge:

```
> git merge feature-a --no-ff  --no-commit
```

`--no-ff`: Dieser Parameter verhindert einen Fast-Forward-Merge.
`--no-commit`: Es soll noch kein Commit durchführt werden, da die nachfolgenden Tests fehlschlagen könnten.

Abb. 16-3
Merge ohne Fast-Forward

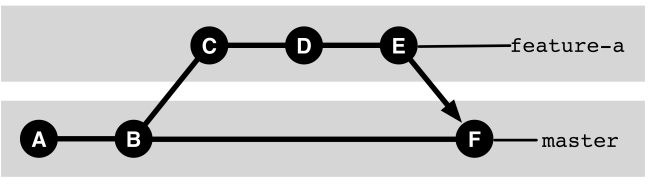


Abbildung 16-3 zeigt die Commit-Historie des Beispiels, wenn Fast-Forwards unterdrückt werden. Das neue Merge-Commit `F` ist zu sehen.[4] Die First-Parent-Historie des `master`-Branch beinhaltet nun nicht die Commits `C` bis `E`.

Während dieses Merge kann es zu Konflikten kommen, wenn andere Features die gleichen Dateien wie das lokale Feature geändert haben. Diese Konflikte müssen mit den normalen Mitteln behoben werden.

Bearbeitungskonflikte → *Seite 56*

Schritt 3: Regressionstests durchführen und Commit anlegen

Nachdem der Merge durchgeführt wurde, müssen die Regressionstests durchgeführt werden. Dabei wird überprüft, ob das neue Feature Fehler in anderen Features hervorruft.

Wenn die Tests zu Fehlern führen, müssen diese analysiert werden. Für die Fehlerbehebung wird der gerade durchgeführte Merge mit dem `reset`-Befehl wieder verworfen:[5]

Branch-Zeiger umsetzen → *Seite 49*

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

`--hard`: Alle Änderungen werden im Stage-Bereich und im Workspace verworfen.

`HEAD`: Der aktuelle Branch wird auf das letzte abgeschlossene Commit zurückgesetzt.

Anschließend wird der Feature-Branch wieder aktiviert. Die Fehler werden dort behoben, und dann wird wieder bei Schritt 1 dieses Ablaufs begonnen.

Falls es zu keinen Fehlern im Regressionstest kommt, kann das Commit abgeschlossen werden.

```
> git commit -m "Lieferung feature-a"
```

Tipp: Legen Sie den Merge-Kommentar einheitlich fest.

Um die Historie des `master`-Branch als Dokumentation zu benutzen, sollte der Kommentar des Merge-Commits einheitlich festgelegt werden. Insbesondere ist es sinnvoll, die eindeutige Identifikation des Features, z. B. die Nummer, mit unterzubringen. Dadurch wird es später einfach möglich sein, mit dem `log`-Befehl und dem Parameter `--grep` nach den Features auf dem `master`-Branch zu suchen.

[4] Dieses Commit ist durch die Option `-no-commit` noch nicht endgültig festgeschrieben, sondern nur vorbereitet.

[5] Git besitzt die Möglichkeit, einmal durchgeführte Konfliktauflösungen zu speichern und diese automatisch anzuwenden, wenn derselbe Konflikt noch einmal auftritt. Dazu gibt es den `rerere`-Befehl. Der Abschnitt »ReReRe – Konfliktauflösungen automatisieren« (Seite 148) beschreibt, wie ReReRe angewendet wird.

Schritt 4: `master`-Branch in das zentrale Repository übertragen

Nach den letzten Schritten liegt im lokalen Repository die Zusammenführung des Features mit dem `master`-Branch vor. Im nächsten Schritt muss der `master`-Branch mit dem `push`-Befehl in das zentrale Repository übertragen werden.

```
> git push
```

Falls es bei diesem Befehl zu Fehlern kommt, dann wurde in der Zwischenzeit bereits ein anderes Feature in den `master`-Branch integriert und ein Fast-Forward-Merge ist nicht mehr möglich. Normalerweise würde man jetzt einen `pull`-Befehl absetzen und die Änderungen lokal zusammenführen. Dabei würde aber die *First-Parent-Historie* nicht mehr nutzbar sein.

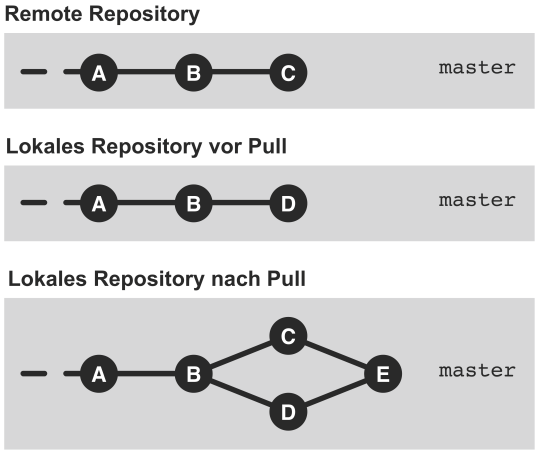


Abb. 16-4
Keine verwendbare First-Parent-Historie nach dem pull-Befehl

In Abbildung 16-4 – oben und in der Mitte – ist die beschriebene Situation skizziert: Remote wurde das `C`-Commit und lokal das `D`-Commit angelegt. Würde man jetzt einen `pull`-Befehl absetzen, dann entstünde ein neues Merge-Commit `E` (siehe Abbildung 16-4 – unten). Damit würde in der First-Parent-Historie des `master`-Branch das `C`-Commit nicht mehr enthalten sein.

Branch-Zeiger umsetzen *→ Seite 49*

In der First-Parent-Historie sollen aber alle Features enthalten sein, deswegen muss bei einem fehlgeschlagenen `push`-Befehl das lokale Feature-Merge-Commit mit dem `reset`-Befehl entfernt werden:[6]

[6] Auch hier bietet es sich wieder an, mit dem `rerere`-Befehl zu arbeiten, um bei einem erneuten Merge die Konflikte nicht noch mal lösen zu müssen – siehe »ReReRe – Konfliktauflösungen automatisieren« (Seite 148).

```
> git reset --hard ORIG_HEAD
```

`ORIG_HEAD`**:** Referenziert das Commit, das vor dem Merge im aktuellen Branch aktiv war.

Anschließend muss wieder mit Schritt 1 dieses Ablaufes begonnen werden, d. h., das neue Commit wird mit dem `pull`-Befehl aus dem `master`-Branch geholt.

Falls der `push`-Befehl erfolgreich war, ist das neue Feature jetzt im zentralen Repository enthalten.

Schritt 5: Feature-Branch löschen oder weiterbenutzen

Variante 1: Feature-Branch löschen

Wenn die Entwicklung des Features nach dem Zusammenführen mit dem `master`-Branch abgeschlossen ist, kann der Feature-Branch gelöscht werden.

Branch löschen
→ Seite 50

```
> git branch -d feature-a
```

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

Falls das Löschen zu einer Fehlermeldung führt, dann wurde typischerweise vergessen, den Feature-Branch mit dem `master`-Branch zusammenzuführen. Die Option `-d` löscht einen Branch nur dann, wenn alle Commits eines Branch durch einen anderen Branch referenziert werden. Möchte man einen Feature-Branch löschen, ohne alle Commits in den `master`-Branch zu übernehmen, kann man die Option `-D` benutzen.

Falls der Feature-Branch im zentralen Repository gesichert wurde, muss der Branch dort ebenso gelöscht werden.

```
> git push origin :feature-a
```

Achtung! Der Doppelpunkt vor dem Namen des Branch ist wichtig. Der Befehl bedeutet: Kopiere nichts in den Feature-Branch.

Variante 2: Feature weiterentwickeln

Falls die Entwicklung des Features noch nicht abgeschlossen ist (d. h., die erste Integration mit dem `master`-Branch war nur eine Teillieferung), kann der Feature-Branch weiterbenutzt werden.

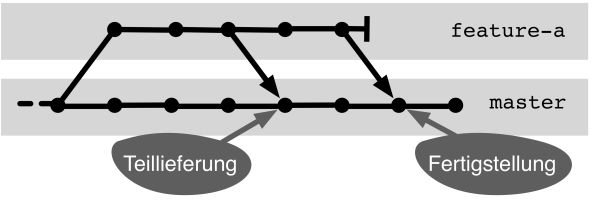


Abb. 16-5
Weiterarbeiten mit Feature-Branch

Aktiver Branch
→ Seite 47

In Abbildung 16-5 ist die Weiterarbeit nach einer Teillieferung skizziert. Es wird einfach der Feature-Branch weiterbenutzt.

```
> git checkout feature-a
```

Sobald die nächste Lieferung fertig ist, wird die normale Integration mit dem `master`-Branch erneut durchgeführt. Dabei ist Git so schlau, nur die Änderungen der neuen Commits in den `master`-Branch zu übernehmen.

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. Dann müssen diese Änderungen des `master`-Branch in den Feature-Branch übernommen werden.

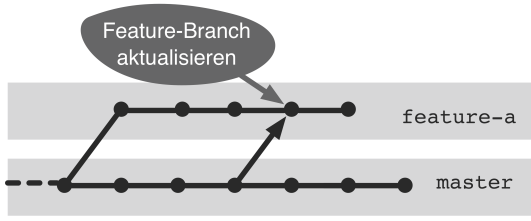


Abb. 16-6
Änderungen aus dem Master- in den Feature-Branch übernehmen

Abbildung 16-6 veranschaulicht die Situation. Es soll ein Merge vom `master`-Branch in den Feature-Branch durchgeführt werden.

Schritt 1: `master`-Branch aktualisieren

Als Erstes müssen die Änderungen des `master`-Branch in das lokale Repository importiert werden.

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

`--ff-only`: Es wird nur Fast-Forward-Merge zugelassen. Damit wird verhindert, dass aus Versehen ein Merge auf dem `master`-Branch durchgeführt wird.

Schritt 2: Änderungen in den Feature-Branch übernehmen

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

Branches zusammenführen
→ Seite 53

```
> git checkout feature-a
> git merge --no-ff master
```

`--no-ff`: Fast-Forward-Merge verbieten. Ein Fast-Forward-Merge kann an dieser Stelle nur passieren, wenn direkt vor diesem Merge der Feature-Branch mit dem `master`-Branch zusammengeführt wurde. Ein Fast-Forward-Merge würde dann die First-Parent-Historie des Feature-Branch zerstören.

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

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

Git-Erweiterungen

Git-Flow: High-Level-Operationen

Git-Flow[7] ist eine Sammlung von Skripten, um den Umgang mit Branches, insbesondere Feature-Branches, zu vereinfachen.
So kann ein neuer Feature-Branch folgendermaßen erzeugt und gleichzeitig aktiviert werden:

```
> git flow feature start feature-a
```

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

```
> git flow feature finish feature-a
```

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