Wichtige Git-Befehle

1. Neues Git-Repository anlegen:
   - Lokales Repository (privat):

     ```
     cd <Name des zu verwaltenden Verzeichnisses>
     git init
     ```

     Diese Variante ist sowohl bei neuen als auch bereits belegten Verzeichnissen möglich.

   - Zentrales Repository (Zugriff für mehrere Team-Mitglieder):

     ```
     mkdir <Repository-Name>.git
     cd <Repository-Name>.git
     git init --bare
     ```

     Die Option `--bare` sorgt dafür, dass nur die Repository-Daten abgelegt werden. Somit kann nicht direkt in dem Repository-Verzeichnis gearbeitet werden! `--bare` sollte immer genutzt werden, wenn in ein Repository zu einem späteren Zeitpunkt „gepusht “ werden können soll.

2. Zentrales Repository klonen, um auf einer eigenen Kopie arbeiten zu können:
   - Lokales Repository klonen:

     ```
     git clone <Pfad-zum-Repository> [Verzeichnis-Name]
     ```

   - Repository von einem Server klonen:

     ```
     git clone <Server-Name>:<Pfad-zum-Repository>
         [Verzeichnis-Name]
     ```

   Die optionale Angabe `[Verzeichnis-Name]` gibt den Namen des Verzeichnisses an, in das das Repository geklont werden soll. Wird die Angabe weggelassen, so wird standardmäßig der Name des Git-Repositories (ohne die Endung `.git`) verwendet.

3. Hat man ein ganz neues leeres Git-Repository geklont, so muss dies zunächst noch initialisiert werden:

   ```
   cd <Repository-Name>
   touch .gitignore
   git add .gitignore
   git commit -a -m "Initial commit"
   git push -u origin master
   ```

   Die einzelnen Befehle werden an späterer Stelle erläutert.

4. Dateien oder ganze Verzeichnisse unter Git-Verwaltung stellen:

   ```
   git add <Datei-/Verzeichnisname>
   ```

Mit `git add` werden Dateien[1] in die sog. *Staging Area* aufgenommen. Die *Staging Area* umfasst alle Dateien, deren aktueller Inhalt beim nächsten *Commit* erfasst werden soll. Somit müssen vor jedem *Commit* immer alle zu erfassenden Dateien zur *Staging Area* hinzugefügt werden. Zu beachten ist, dass tatsächlich nur der Stand in den *Commit* aufgenommen wird, den die Datei zum Zeitpunkt des Aufrufs von `git add` hatte. Alle Änderungen nach `git add` werden **nicht** automatisch zur *Staging Area* hinzugefügt.

Da das Hinzufügen jeder einzelnen Datei mittels `git add` umständlich ist, wird im Folgenden eine kürzere Variante vorgestellt (s. Aktuellen Projektstand sichern).

5. Aktuellen Projektstand sichern:

   Nachdem alle Dateien für den nächsten *Commit* mit `git add` hinzugefügt wurden, so kann dieser abgesetzt werden:

   ```
   git commit
   ```

   Es öffnet sich ein Editor (standardmäßig `vi`), in dem die vorgenommenen Änderungen, Erweiterungen etc. protokolliert werden können. Sollte der `vi` als Standardeditor nicht gewünscht sein, so gibt es zwei Möglichkeiten, dies umzustellen:

   - Man kann die Umgebungsvariable `VISUAL` setzen, die systemweit den Standardeditor innerhalb eines Terminals definiert. Dies hat den Vorteil, dass so auch von anderen Programmen der bevorzugte Editor genutzt wird:

     ```
     export VISUAL=<Editor-Name>
     ```

     Diese Einstellung gilt zunächst nur für die momentan geöffnete Shell. Soll die Variable bei jedem Terminalaufruf gesetzt werden, so kann der Befehl an die `.bashrc` (unter Mac OS X `.profile`) angehängt werden:

     ```
     echo 'export VISUAL=<Editor-Name>' >> ~/.bashrc
     ```

     bzw.

     ```
     echo 'export VISUAL=<Editor-Name>' >> ~/.profile
     ```

   - Alternativ kann die Einstellung auf Git beschränkt werden:

     ```
     git config --global core.editor "<Editor-Name>"
     ```

     Die Option `--global` bewirkt, dass die Einstellung für alle Git-Repositories übernommen wird. Ohne Angabe von `--global` werden Optionen nur in dem Repository angewendet, in dem man sich gerade befindet.

   Häufig ist es erwünscht, eine einmal mit `git add` hinzugefügte Datei bei Änderungen automatisch wieder in die *Staging Area* für den nächsten *Commit* aufzunehmen. Hierfür besteht die Option `-a`:

[1] bei Verzeichnissen analog alle Dateien, die sich innerhalb des Verzeichnisses und dessen Unterverzeichnissen befinden

```
git commit -a
```

Damit ergibt sich folgende vereinfachte Vorgehensweise (unter der Annahme, dass es sich nicht um ein Repository handelt, dass mit anderen Mitarbeitern geteilt wird):

- Neue, bisher noch nicht von Git verwaltete Dateien werden mit `git add` hinzugefügt.
- Nachdem am Pojekt gearbeitet wurde, können alle Änderungen seit dem letzten *Commit* mit Hilfe von `git commit -a` in den nächsten *Commit* aufgenommen werden.

6. Repository-Status abfragen:

```
git status
```

Der Befehl `git status` liefert Informationen über die Dateien der eigenen Arbeitskopie und listet u. a. Dateien, die seit dem letzten *Commit* modifiziert oder neu hinzugefügt wurden. Dateien, die noch gar nicht unter Versionsverwaltung stehen, werden ebenfalls aufgeführt. Ist der momentan ausgecheckte Zweig mit einem Zweig eines anderen Repositories verbunden (*tracking branch*, s. *Git als zentrales Versionsverwaltungssystem*), so wird zusätzlich eine Meldung ausgegeben, wenn der lokale Zweig gegenüber dem entfernten Zweig veraltet oder voraus ist.

`git status` sollte daher bei jedem Arbeitsbeginn und vor jedem *Commit* ausgeführt werden, um sich über den Arbeitsstand innerhalb des Repositories zu vergewissern.

7. Verwaltete Dateien verschieben oder löschen:

Git bietet zur Veränderung des Dateisystems Befehle, die analog zu den von Unix-Systemen bekannten Kommandos funktionieren:

- Datei verschieben:

```
git mv <Quelldatei> <Ziel>
```

- Datei löschen:

```
git rm <Datei>
git rm -r <Verzeichnis>
```

Beide Befehle vermerken die umgesetzten Änderungen automatisch in der *Staging Area*.

8. Verzweigungen erstellen und wieder zusammenführen:

Mit Git ist es sehr einfach Verzweigungen innerhalb des Entwicklungsverlaufs eines Projektes zu erstellen. Zweige bieten die Vorteile:

- Zweige können zur Gruppierung von logisch zusammengehörigen *Commits* genutzt werden.
- Sie bieten die Möglichkeit, parallel zum Hauptprojekt neue Features zu entwickeln und zu testen.
- Arbeiten mehrere Personen an einem Projekt, so helfen sie verschiedene Entwicklungen voneinander zu separieren, sodass sie erst zusammengeführt werden können, wenn sie einen fertigen Zustand erreicht haben. So lassen sich unerwünschte Wechselwirkungen neuer Entwicklungen vermeiden.

Vom aktuellen *Commit* aus einen neuen Zweig erstellen und auschecken:

```
git branch <Zweig-Name>
git checkout <Zweig-Name>
```

Als Kurzform ist auch

```
git checkout -b <Zweig-Name>
```

möglich. Wie man am vorletzten Listing erkennen kann, ist es mit `git checkout <Zweig-Name>` generell möglich, zwischen den momentan existierenden Zweigen hin- und herzuschalten. Sollten sich noch Änderungen im Arbeitsverzeichnis befinden, die noch nicht mit einem *Commit* gespeichert wurden, so werden diese beim Zweigwechsel übertragen, sofern möglich. Andernfalls wirft Git eine Fehlermeldung aus.

Mit Hilfe von

```
git branch -a
```

ist es jederzeit möglich die momentan vorhanden Zweige aufzulisten. Hierzu zählen sowohl lokale wie auch Zweige von eingebundenen Servern (s. *Git als zentrales Versionsverwaltungssystem*). Möchte man nur die lokalen Zweige betrachten, so kann die Option `-a` weggelassen werden.

Soll ein Zweig wieder mit einem anderen zusammengeführt werden, so ist dies mit folgenden Befehlen möglich:

```
git checkout <Zweig-in-den-gemergt-werden-soll>
git merge --no-ff <Zweig-der-gemergt-werden-soll>
```

Die Option `--no-ff` bewirkt die Unterdrückung eines potentiellen *Fast Forward*-Merge, der zu einer nahtlosen Integration des Zweiges in den zweiten Zweig führen würde. Dies hat den Nachteil, dass hierdurch die gewünschte Gruppierung von *Commits* verloren gehen würde.

Da Zweige nur Zeiger auf einen bestimmten *Commit* sind, und die *Commits* des gemergten Zweigs mit einem zweiten Zweig vereint wurden, kann der ursprüngliche Zweig gefahrlos gelöscht werden:

```
git branch -d <Zweig-der-gemergt-wurde>
```

Sollten durch das Löschen hingegen *Commits* verloren gehen, so gibt Git eine Meldung aus und verhindert die Ausführung.

9. Git als zentrales Versionsverwaltungssystem:

   Projekte mit mehreren Mitarbeitern haben häufig ein zentrales Repository, in das alle Änderungen eingepflegt werden und über das sich die verschiedenen Team-Mitglieder den aktuellen Entwicklungsstand des Gesamtprojektes beziehen können. In diesem Zusammenhang gibt es folgende wichtige Arbeitsschritte:

   - Zentrales Repository klonen (s. *Repository klonen*)
   - Neusten Stand des zentralen Repositories abfragen:

     ```
     git fetch
     ```

     Von den hier vorgestellten Befehlen arbeiten bis auf `get fetch` und `git push` alle auf lokalen Daten des Repositories. Somit sollte bei allen anderen Befehlen diesen Abschnitts zuvor immer ein `git fetch` ausgeführt werden!
   - Einen Zweig des entfernten Repositories ausschecken, um lokal darin arbeiten zu können:

     ```
     git checkout --track origin/<Zweig-Name>
     ```

     Hiermit wird ein sog. *tracking branch* angelegt, der direkt mit dem entsprechenden Zweig des entfernten Repositories verknüpft ist. Wie die Synchronisierung zwischen den Zweigen erfolgt, wird im Folgenden näher beschrieben.
   - Neuesten Stand eines entfernten Zweiges in den lokalen *tracking branch* einfließen lassen:

     ```
     (git checkout <Zweig-Name>)
     git fetch
     git merge origin/<Zweig-Name>
     ```

     oder in Kurzform:

     ```
     (git checkout <Zweig-Name>)
     git pull
     ```

     Hierbei können evtl. Konflikte entstehen, wenn parallel dieselben Zeilen einer Datei geändert wurden. Diese werden von Git in der Datei markiert und müssen händisch behoben werden. Welche Dateien betroffen sind, lässt sich mit Hilfe des Kommandos `git status` einsehen. Sind die Konflikte innerhalb einer Datei behoben, so kann diese mit Hilfe eines `git add` als gelöst markiert werden. Nach der Behebung aller vorliegenden Konflikte muss schließlich noch ein `git commit` ausgeführt werden, um den *Merge*-Vorgang abzuschließen.
   - Lokalen Arbeitsstand in das entfernte Repository einpflegen:

     ```
     git push
     ```

- Lokalen (bisher privaten) Zweig im entfernten Repository veröffentlichen:

```
git push -u origin <Zweig-Name>
```

  Die Option `-u` bewirkt, dass der lokale Zweig gleichzeitig als *tracking branch* eingerichtet wird.

- Entfernten Zweig löschen:

```
git push origin :<Zweig-Name>
```

  Werden von einem anderen Team-Mitglied entfernte Zweige gelöscht, so werden sie bei den anderen Mitarbeitern dennoch weiterhin vom Befehl `git branch -a` gelistet (selbst nach einem `git fetch`). Um alle Referenzen auf gelöschte Zweige zu entfernen, muss der Befehl

```
git remote prune origin
```

  ausgeführt werden.

10. Änderungen und *Commits* rückgängig machen:

- Änderungen einer Datei verwerfen:

```
git checkout -- <Pfad-zur-Datei>
```

  Die Angabe von `--` ist nicht notwendig, vermeidet aber Probleme mit Dateien, die wie Zweige benannt sind.

- Alle Änderungen seit dem letzten *Commit* verwerfen:

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

- Den letzten *Commit* verwerfen (**VORSICHT:** Die History wird hierbei geändert, sodass der *Commit* tatsächlich verloren ist!):

```
git reset --hard HEAD~
```

  `git reset` erwartet die Angabe eines *Commits*, auf den der aktuelle Zweig zurückgesetzt werden soll. `HEAD` bezeichnet hierbei den aktuellen *Commit* des ausgecheckten Zweiges. Mittels `~` wird der vorhergehende *Commit* angesprochen.

  Das Kommando `git reset` kann mit Hilfe von Optionen in verschiedenen Variationen eingesetzt werden:

  – `--hard`:
    Setzt die Arbeitskopie auf den angegebenen *Commit* zurück und verwirft die *staging area* inkl. aller weiteren Änderungen, die sich noch nicht in der *staging area* befinden.

  – *(ohne Optionen)*:
    Wie `--hard`, aber ohne Verwerfen der getätigten Änderungen.

– `--soft`:
Setzt den *HEAD-Commit* neu, behält aber alle Einträge der *staging area* und alle modifizierten Dateien.

Sollte ein gelöschter *Commit* bereits mit einem `git push` veröffentlicht worden sein, so reicht kein einfacher weiterer Aufruf von `git push`, um den *Commit* auf dem Server zu löschen. Um den Zweig auf dem Server vollständig zu überschreiben, kann die Option `-f` (force) genutzt werden:

```
git push -f origin <Zweig-Name>
```

Dieses Kommando sollte allerdings mit viel Bedacht eingesetzt werden, da die Änderungen hiermit unwiderruflich verloren gehen. Zusätzlich sollte man sicherstellen, dass in der Zwischenzeit niemand den zu löschenden *Commit* bereits in die eigene Arbeitskopie eingebunden hat!

- Den letzten *Commit* durch Hinzufügen eines weiteren *Commits* rückgängig machen:

```
git revert
```

Der Einsatz von `git revert` bietet den Vorteil, dass keine Einträge der Git-History verloren gehen. Andererseits wird hiermit allerdings die History aufgebläht.

11. Dateien von der Versionsverwaltung ausschließen:

Git bietet zwar die Möglichkeit mittels `git add` ganz gezielt Dateien zu selektieren, die unter Versionsverwaltung gestellt werden sollen, jedoch kann es beispielsweise beim Hinzufügen ganzer Verzeichnisse sinnvoll sein, dass von vornherein bestimmte Dateien bzw. Dateitypen ausgeschlossen werden. Dies bietet zusätzlich den Vorteil, dass diese Dateien anschließend auch nicht mehr in der Ausgabe von `git status` als *untracked files* erscheinen und somit nicht mehr die Ausgabe unnötig aufblähen.

Zum gezielten Ignorieren von Dateien und Verzeichnissen wird im obersten Verzeichnis des Repositories eine Datei `.gitignore` angelegt, in der jede Zeile eine Regel enthält.

```
touch .gitignore
nano .gitignore
```

Eine solche `.gitignore`-Datei könnte beispielsweise folgendermaßen aussehen:

```
# ignoriere gedit Backup-Dateien:
*~
# schliesse Dateien aus, die aus dem Quellcode generiert werden:
*.o
*.so
# ignoriere die Datei TODO in jedem Verzeichnis:
TODO
# manual.pdf nur im root-Verzeichnis ignorieren:
```

```
/manual.pdf
# Verzeichnis build ignorieren:
build/
```

Die Datei darf sie von der *Bash* gewohnten Zeichen für reguläre Ausdrücke enthalten, wie beispielsweise:

- `*`: trifft auf beliebig viele Zeichen (auch 0) zu
- `?`: Platzhalter für genau ein Zeichen
- `[0-9]`: Eine der Ziffern von 0 bis 9

Verzeichnisse werden mit Hilfe eines nachgestellten Slash (`.../`) markiert. Soll sich eine Angabe nur auf das Verzeichnis beziehen, in der sich die Datei `.gitignore` befindet, so wird ein führender Slash verwendet. Eine Umkehrung (also Dateien, die trotz einer Regel doch nicht ignoriert werden sollen) ist mit vorangestelltem Ausrufezeichen (`!`) möglich.

Wie bereits angedeutet, kann es mehr als eine `.gitignore` geben. Legt man zusätzliche `.gitignore`-Dateien in den Unterverzeichnissen des Repositories an, so überschreiben diese die Regeln der `.gitignore` des Root-Verzeichnisses.

Möchte man trotz einer Ausschlussregel eine Datei unter Versionsverwaltung stellen, so muss nicht für jede Datei eine Ausnahme erstellt werden. Es reicht die Option `-f`/`--force` zu benutzen:

```
git add -f <Datei>
```

Um Dateien, die bereits versioniert werden, nachträglich zu ignorieren, reicht es nicht aus, nur die `.gitignore` anzupassen. Zusätzlich muss die Datei explizit aus der bestehenden Verwaltung entfernt werden:

```
git rm --cached <Datei>
```

Die Option `--cached` bewirkt, dass die Datei nur aus der Versionsverwaltung entfernt, aber nicht von dem Datenträger gelöscht wird.

12. Aktuellen Dateiinhalt mit dem Stand des letzten *Commits* vergleichen:

```
git diff <Datei>
```

13. Versionsgeschichte betrachten:

```
git log
```

Es werden textuell alle vorhergenden *Commits* mit *Commit*-Nachricht angezeigt. Zusätzlich kann die Option `--graph` angegeben werden, mit der über ASCII-Art die Verzweigungen unter den *Commits* dargestellt werden.

14. Änderungen seit dem letzten *Commit* temporär aufheben:

```
git stash
```

Mit Hilfe von *Stashing* kann zum Arbeitsstand des letzten *Commits* zurückgesprungen werden (z. B. um zur letzten lauffähigen Version zurückzukehren). An späterer Stelle kann man die aufgehobenen Änderungen mit einem Aufruf von

```
git stash pop
```

wieder aufspielen.

15. Tag erstellen:

    Git bietet die Möglichkeit *Commits* mit einem Namen zu versehen, über den sie besser wieder aufgefunden werden können. Dies ist z. B. sinnvoll, um abgeschlossene Programmversionen der Entwicklung im Repository hervorzuheben. Ein Tag für den letzten getätigten *Commit* des aktuellen Zweigs wird erstellt mit:

    ```
    git tag -a <Tag-Name>
    ```

    Soll ein älterer *Commit* nachträglich mit einem Tag versehen werden, so kann man sich dessen SHA-1 Hash mit Hilfe von `git log` heraussuchen, wie einen gewöhnlichen Zweig auschecken und taggen:

    ```
    git checkout <SHA-1 Hash>
    git tag -a <Tag-Name>
    ```

    Alle erstellten Tags kann man auflisten lassen, wenn man `git tag` ohne Optionen einsetzt:

    ```
    git tag
    ```

    Zu beachten ist, das Tags **nicht** automatisch an ein entferntes Git-Repository übertragen werden, wenn der Befehl `git push` ausgeführt wird. Hierzu muss zusätzlich die Option `--tags` mitgegeben werden:

    ```
    git push --tags
    ```

    Das Holen von Tags geschieht hingegen automatisch mit Hilfe von `git fetch`.

16. Arbeitsverzeichnis (*working copy*) aufräumen:

    - Alle Dateien löschen, die nicht unter Versionsverwaltung stehen (ignorierte Verzeichnisse und alle Dateien der `.gitignore`-Liste bleiben bestehen):

      ```
      git clean -f
      ```

    - Zusätzlich Verzeichnisse löschen, die nicht unter Versionsverwaltung stehen:

      ```
      git clean -fd
      ```

- Sollen neben nicht versionierten Dateien auch alle Dateien gelöscht werden, die auf der `.gitignore` stehen, so muss der Parameter `-x` angegeben werden:

```
git clean -fx
```

  Dieses Variante des Kommandos ist vor allen Dingen dann nützlich, wenn Build-Dateien gelöscht werden sollen. Es ist also mit einem `make clean` vergleichbar.
  Zu beachten ist, dass das Kommando nur Dateien ausgehend von dem Verzeichniss löscht, in dem der Befehl abgesetzt worden ist.

17. Aktuellen Projektstand exportieren:

    Git bietet die Möglichkeit, die Daten eines bestimmten *Commits* zu exportieren. „Exportieren“ bedeutet, dass alle zu Dateien, die zum Zeitpunkt des *Commits* versioniert waren, extrahiert werden. Alle übrigen Dateien und das `.git`-Verzeichnis werden herausgefiltert:

```
git archive <Zweig-Name bzw. Commit> --output=<Datei>
   --prefix=<Projekt-Name>/
```

    Als Ausgabe erstellt Git ein Archiv, dessen Typ anhand der Dateiendung gewählt wird.

    Die Option `--prefix` sorgt dafür, dass allen Dateien des Repositories das gewählte Präfix vorangestellt wird und kann somit dazu genutzt werden, um alle Daten in ein gemeinsames Hauptverzeichnis innerhalb des Archivs zu packen. Dies sollte immer gemacht werden, damit beim Entpackvorgang nicht versehentlich der gesamte Inhalt im aktuellen Arbeitsverzeichnis landet und Dateien überschrieben werden. Wichtig ist, dass das Präfix hierzu **immer** mit einem `/` abgeschlossen werden muss, damit es als vorangestelltes Verzeichnis interpretiert wird.

    Das Kommando eignet sich vor allen Dingen gut, um ein Archiv des aktuellen Releases zu erstellen:

```
git archive master --output=release.tgz --prefix=project/
```

18. Farbige Kommandoausgaben:

    Die Ausgaben der Befehle `git status` und `git diff` sind übersichtlicher, wenn sie farblich dargestellt werden. Farben können mit dem Befehl

```
git config --global color.ui true
```

    aktiviert werden.

Gitflow

Dieser Abschnitt beschreibt eine bewährte Vorgehensweise, wie im Team an einem gemeinsamen Projekt gearbeitet werden kann[2]. Es wird hierbei davon ausgegangen, dass bereits ein leeres Repository auf einem Server existiert, zu dem alle Team-Mitglieder Zugang haben.

Konzept:

Das hier vorgestellte Arbeitskonzept macht massiven Gebrauch vom Git Verzweigungsmodell. Unterschieden wird hierbei zwischen *long running branches*, die für die gesamte Lebensdauer des Projektes bestehen und *topic branches*, die nur eine gewisse Verwendung existieren. Folgende Zweige werden verwendet:

- **`long running branches`**:
  - **`master`**: Der `master`-Zweig ist der Standard-Zweig in jedem Git-Repository und wird standardmäßig ausgecheckt, wenn ein Repository geklont wird. Im Hinsicht auf Veröffentlichungen sollte der `master`-Zweig daher lediglich fertige Versionen (Releases) beinhalten. Daher wird **nie** im `master` gearbeitet und es werden auch keine *Commits* im `master` abgesetzt. Es ist lediglich erlaubt `develop`- oder `hotfix`-Zweige in den `master` per Merge einfließen zu lassen.
  - **`develop`**: Der `develop`-Zweig ist Ausgangspunkt der Programmierarbeiten und enthält die aktuellen Entwicklungen für das nächste Release. Auch in diesem Zweig wird nicht direkt gearbeitet, sondern es werden Unterzweige angelegt, in denen neue Features entwickelt und getestet werden können. Dies garantiert, dass sich neue Features in der Entwicklungsphase nicht gegenseitig beeinflussen und dass der `develop`-Zweig lediglich für sich abgeschlossene Teilentwicklungen enthält, sodass er immer eine gültige Basis für weitere Verzweigungen bereitstellt. Wurden alle Änderungen für das nächste Release in `develop` gesammelt, so kann die Versionsnummer inkrementiert und das Ergebnis in den `master`-Zweig gemergt werden.
- **`topic branches`**:
  - **`feature-***`: `feature`-Zweige werden von `develop` abgeleitet und enthalten die Entwicklung und das Testen neuer Features. Je nach Große können in weiteren Zweigen (`subfeature-xxx`) Unterpunkte ausgearbeitet werden. Ist das zu entwickelnde Feature abgeschlossen, so kann der Zweig wieder über einen Merge mit `develop` verbunden und anschließend gelöscht werden.
  - **`hotfix-***`**: Sollte ein kritischer Fehler in einer Release-Fassung auftreten, so kann ausgehend vom `master` ein `hotfix`-Zweig erstellt werden, in dem das Problem unabhängig von weiteren Entwicklungen behoben werden kann. Abschließend wird das Ergebnis nach Inkrementierung der Versionsnummer

[2]Das Gitflow-Modell wurde ursprünglich von Vincent Driessen in einem Blog Post beschrieben: `http://nvie.com/posts/a-successful-git-branching-model/`

sowohl in den `master`- als auch `develop`-Zweig gemergt, um das Problem sowohl in der aktuellen Version als auch in der aktuellen Entwicklungsfassung zu beheben.

Workflow:

1. Neues leeres Repository vorbereiten:

   Wichtig ist, dass ein neues Repository über `master`- und `develop`-Zweige verfügt, die mit dem Server synchronisiert werden:

```
# Klonen
git clone <Server-Name>:<Pfad-zum-Repository>
cd <Repository-Name>

# Initialisieren
touch .gitignore
git add .gitignore
git commit -a -m "Initial commit"
git push -u origin master

# Develop-Zweig einrichten
git checkout -b develop
git push -u origin develop
```

   Alle anderen Team-Mitglieder müssen dementsprechend den neuen `develop`-Zweig auschecken (über den `master` verfügen sie schon standardmäßig):

```
# Klonen
git clone <Server-Name>:<Pfad-zum-Repository>
cd <Repository-Name>

# Develop auschecken
git checkout --track origin/develop
```

   Anschließend werden mit Hilfe von `git push` lokale Änderungen am `develop`-Zweig immer automatisch zum Server übertragen.

2. Neues Feature beginnen:

   Neue Features gehen immer von der aktuellen Entwicklung des `develop`-Zweiges aus. Soll beispielsweise ein Feature *Settings Dialog* programmiert werden, so ist folgendermaßen vorzugehen:

```
git checkout develop
git pull
# <evtl auftretende Merge-Konflikte loesen>
git checkout -b feature-settings-dialog
```

Anschließend kann in dem neuen Zweig gearbeitet und Änderungen mit *Commits* festgehalten werden.

Sollen weiter Team-Mitglieder an demselben Feature mitarbeiten können, so muss der `feature`-Zweig auf dem Server veröffentlicht werden,

```
git push -u origin feature-settings-dialog
# (Autovervollstaendigung mit <tab> nutzen
```

sodass er von den anderen ausgecheckt werden kann:

```
git checkout --track origin/feature-settings-dialog
```

3. Feature abschließen:

   Ist ein Programmteil fertig gestellt, so kann er wieder zurück in den `develop`-Zweig gemergt werden:

```
git checkout develop
git pull
# <evtl auftretende Merge-Konflikte loesen>
git merge --no-ff feature-settings-dialog
git push origin develop
git branch -d feature-settings-dialog
```

   Die Option `--no-ff` sollte beim Merge auf jeden Fall angegeben werden, damit die Verzweigungsstruktur in der History des Repositories erhalten bleibt.

   Wurde der Zweig auf dem Server veröffentlicht, so muss auch hier noch gelöscht werden:

```
git push origin :feature-settings-dialog
```

   Alle anderen Team-Mitglieder rufen daraufhin

```
git remote prune origin
```

   auf, damit der gelöschte Zweig auch bei ihnen lokal nicht mehr angezeigt wird.

4. Neues Release veröffentlichen:

   Wurden alle Features für das nächste Release im `develop`-Zweig gesammelt, so kann der neue Stand mit einem Tag versehen und im `master` veröffentlicht werden:

```
git checkout develop
git pull
# <Versionnummer erhoehen>
git commit -a
git checkout master
git merge --no-ff develop
git tag -a release-xx.yy.zz
git push
git push --tags
```

5. Hotfix vornehmen:

   Tritt im aktuellen Release ein schwerer Fehler auf, so sollte er sofort in einem `hotfix`-Zweig behoben werden:

```
git checkout master
git checkout -b hotfix-***
# <Fehler beheben>
git commit -a
# <Versionnummer erhoehen>
git commit -a
git checkout master
git merge --no-ff hotfix-***
git checkout develop
git merge --no-ff hotfix-***
git push
git branch -d hotfix-***
```

Workflow visuell: