Software Engineering - Versionskontrolle (mit Git)

Eine allererste Einführung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de
Version: 1.0

Die Folien basieren in Teilen auf Folien von Dr. Helm und Prof. Dr. Hermann.

Alle Fehler sind meine eigenen.

Folien: [HTML] https://delors.github.io/se-versionskontrolle/folien.de.rst.html
[PDF] https://delors.github.io/se-versionskontrolle/folien.de.rst.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues
Kontrollfragen: https://delors.github.io/se-versionskontrolle/kontrollfragen.de.rst.html

1. Versionsmanagement und Versionskontrollsysteme

Versionsmanagement

Beobachtung

1

- Softwareentwicklung ist Teamarbeit
- Viel (indirekte) Kommunikation nötig
- Entwicklungswissen muss dokumentiert werden
- Software besteht aus vielen Dokumenten:
- Lastenheft
- Pflichtenheft
- Analyse- und Designdokument
- Programmcode (Quellcode, Skripte, Konfigurationsdateien, ...)
- Testdokumentation
- Codedokumentation

2

- Verschiedene Personen greifen (gleichzeitig) auf Dokumente zu
- Oft bearbeiten verschiedene Personen gleichzeitig (unabhängig voneinander) das selbe Dokument

Versionsmanagement

- Wo ist die aktuelle Version?
- Was ist die zuletzt lauffähige Version?
- Wo ist die Implementierungsversion vom 01. April 2016?

  Und welche Dokumente beziehen sich auf diese Version?
- Welche Version wurde dem Kunden präsentiert?

Änderungsmanagement

- Was hat sich seit letzter Woche geändert?
- Wer hat diese Änderung gemacht?
- Warum wurde diese Änderung gemacht?

Einfache Lösungen, die oft verwendet werden:

- Austausch der Dokumente via USB-Stick / Festplatte
- Austausch der Dokumente via Mail
- Austausch über Netzwerkfestplatte

Zusätzlich müssen dann noch Konventionen und Regeln im Team definiert werden.

Warnung

Just, don't do it!

Unterstütztes Versionsmanagement - Motivation

🔎 Beobachtung

- „Einfache Lösungen“ um Versionen zu verwalten erzeugen neue Probleme
- Konventionen und Regeln werden nicht eingehalten
- Koordination ist aufwendig und führt zu Verzögerungen
- Varianten und Konfigurationen werden von Hand verwaltet
- Versions- und Änderungsfragen nicht bzw. nur schwer beantwortbar
- Geistige Kapazität wird mit „Kleinkram“ verschwendet

Schlussfolgerung

Konventionen müssen technisch erzwungen werden!

Versionskontrollsysteme (VCS) - Überblick

Zweck

Versionskontrollsysteme verwalten mehrere Versionen des Codes.

- Erlauben es mehreren Personen gleichzeitig am selben Projekt zu arbeiten
- Änderungen unterschiedlicher Personen werden teil-automatisch integriert
- Erhält Historie von Änderungen

Arten

Zentralisierte VCS:
synchronisieren alle Änderungen in einem zentralen Repository (Subversion, . . .)

! Keine Offline-Nutzung möglich.

Dezentralisierte VCS:
können mehrere, unabhängige Repositories haben (Git, Mercurial, . . .)

Bemerkung
Dezentralisierte VCS (insbesondere Git) sind heute am weitesten verbreitet.

Konsistenzmechanismen

Optimistische Mechanismen:

- System erlaubt gleichzeitiges Bearbeiten des Dokuments durch verschiedene Personen
- System erkennt und integriert die Änderungen (Merging)
- Evtl. funktioniert das nicht automatisch; dann muss der Konflikt manuell beseitigt werden

Pessimistische Mechanismen:

- System verbietet gleichzeitiges Bearbeiten des Dokuments durch verschiedene Personen (Sperrprotokolle)

Beide Mechanismen haben Vor- und Nachteile

✓Sperren serialisiert die Arbeit

! Mergen kann in seltenen Fällen komplex werden und zu Fehlern führen

2. Versionskontrolle mit Git

VCS - Git - einfache Verwendung

Repository auf Server einrichten

1. Git repository einrichten (Beispielsweise über Web-Frontend wie https://github.com)
2. Lokale Kopie des Remote-repositories „Klonen”: `git clone <repo-URL> [lokales Verzeichnis]`

Repository lokal anlegen

In einem beliebigen Verzeichnis: `git init`

Datei neu versionieren

1. Dateien dem Repository hinzufügen

   ```
   git add <Dateipfade>
   ```

   Dateien landen dann in der sogenannten „Staging Area“.

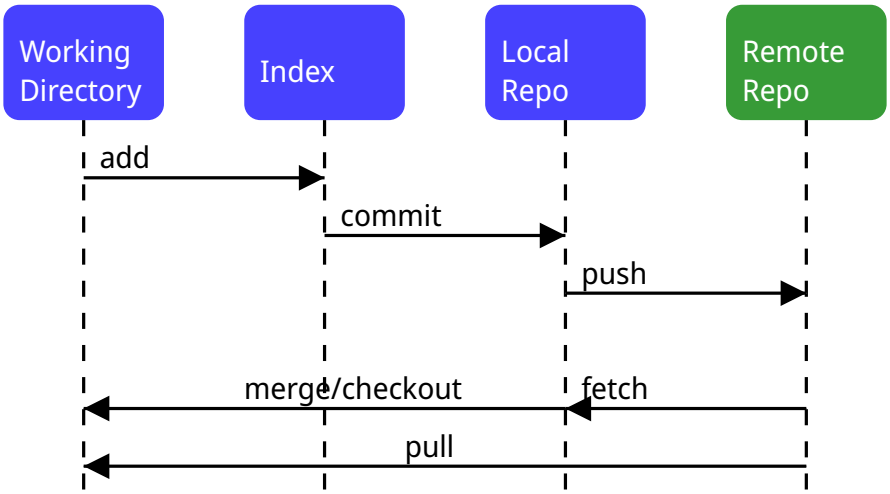
   Die Staging Area (oder Index) hält alle Änderungen, Hinzufügungen und Löschungen von Dateien, die Teil des nächsten Commits werden sollen.

2. gestagete Änderungen committen

   ```
   git ci
   ```

   Dies fügt die Änderungen dem lokalen Repository hinzu.

VCS - Git - Prozess



VCS - Git - Commits - Beispiel

```
commit ace47c68a2deaa6290344a5f9c2d7749d01f0efc
Author: Michael Eichberg <mail@michael-eichberg.de>
Date:   Wed Jan 22 17:43:28 2025 +0100

    encrypted presenter notes

diff --git a/renaissance/css/core/slide-view.css b/renaissance/css/core/slide-view.css
index 21d433b..03f010a 100644
--- a/renaissance/css/core/slide-view.css
+++ b/renaissance/css/core/slide-view.css
@@ -47,6 +47,12 @@
         /* The height will be computed by JavaScript depending on the mode. */
     }

+
+
+    /* Presenter Notes */
+    ld-presenter-note-marker[data-encrypted="true"] {
+        display: none;
+    }
 }

...
```

VCS - Git - Commits

Commits beschreiben eine atomare Änderung des Codes

- Hashcode, um den Commit zu identifizieren
- Autor und Zeit des Commits
- Beschreibung der Änderung
- Änderung als Diff: Hinzugefügte und entferne Zeilen je Datei

VCS - Git - Hilfreiches

Zwischenspeichern von Änderungen

Aktuelle Änderungen zwischenspeichern und Working Copy resetten:

```
git stash
```

Hilfreich z. B. wenn man vergessen hat, Änderungen von *Remote* zu pullen. Ein pull könnte lokale Änderungen überschreiben, mit `git stash` werden diese Änderungen aber zunächst sicher beiseite gelegt.

Änderungen vom *Stash* in *Working Copy* zurückspielen:

```
git stash pop
```

Änderungen in der Working Copy zurücksetzen

```
git reset --hard
```

Setzt alle Änderungen in der Working Copy auf den letzten Commit zurück (z. B. nach einem „Fehlversuch“).

Metadaten setzen

Username und Emailadresse als Metadaten für Commits setzen:

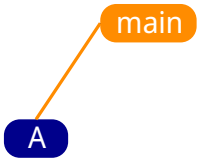
```
git config user.name <name>
```

```
git config user.email <e-mail>
```

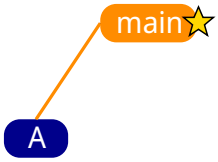
Git Branches

Git verwaltet Versionen von Dokumenten mittels Commits in Branches.

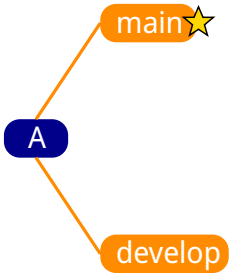
Initiales Setup - main ist aktuell auf dieser Version



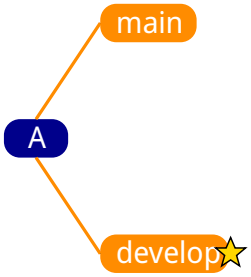
main ist der aktuell ausgecheckte Branch



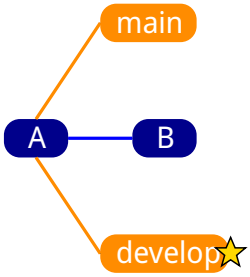
`git branch develop`



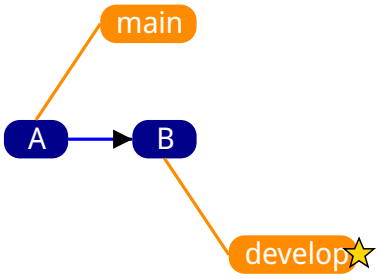
`git checkout develop`



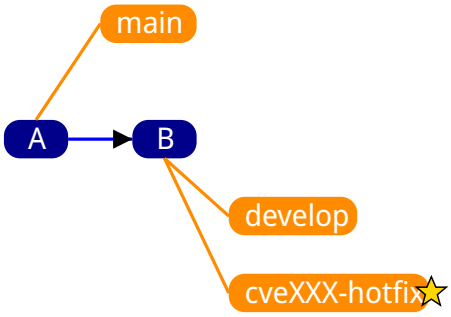
`git commit` von B



`git commit` von B setzt den *aktuellen* Branch weiter



`git checkout -b cveXXX-hotfix`



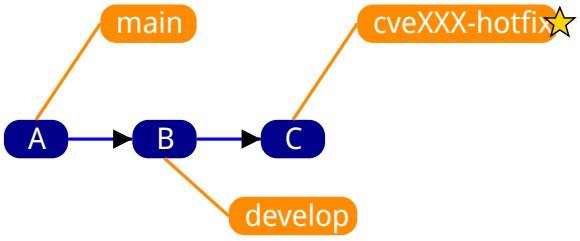
Bemerkung

`git checkout -b cveXXX-hotfix`

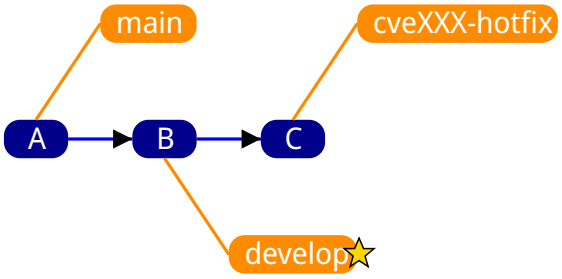
ist lediglich eine Abkürzung für:

```
git branch cveXXX-hotfix
git checkout cveXXX-hotfix
```

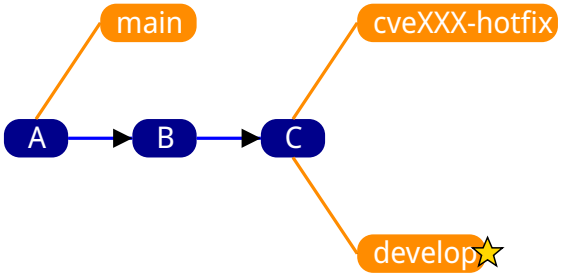
`git commit` von C



`git checkout develop`



Fast-forward Merge `git merge cveXXX-hotfix`



Sonderfälle

- wenn es auf beiden Branches Änderungen gab, dann kann ein Merge ggf. fehlschlagen und muss manuell gemerged werden.
- Um Änderungen auf ein remote Repository zu schieben bzw. davon zu holen muss man git push und git pull verwenden. Dabei kann es auch zu Konflikten kommen, die manuell gelöst werden müssen.

Git-Flow

Git-Flow ist eine Konvention zur Nutzung von Branches in einer sinnvollen Art und Weise.[1]

Mindestens fünf Arten von Branches:

main: enthält stets die zuletzt veröffentlichte Version

develop: enthält aktuelle Entwicklungsversion

feature/topic branches: zur Entwicklung individueller Features

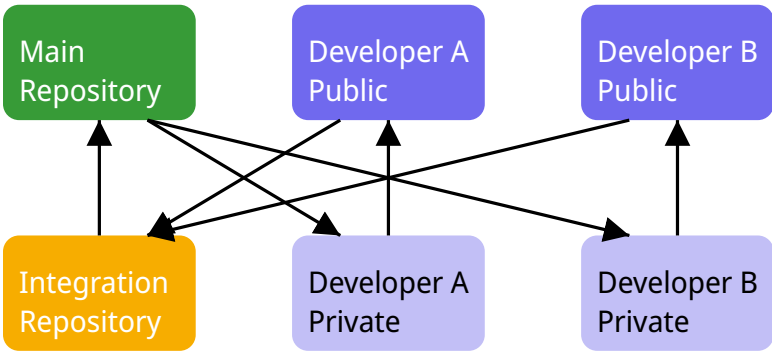
hotfix branches: zur Implementierung dringender Bugfixes

release branches: zum Vorbereiten eines Releases

---

[1] Erstmals dokumentiert durch Vincent Driessen http://nvie.com/posts/a-successful-git-branching-model/

Dezentralisierte VCS - Verteiltes Arbeiten



(Verteilte) Workflows beschreiben, wie Personen Änderungen zwischen verteilten Repositories synchronisieren.

- Hängt von Projekt und Organisationsstruktur ab
- Workflows unterscheiden öffentliche und private Repositories
- In den meisten Workflows gibt es ein ausgezeichnetes Repository als *ground truth*

Achtung!

Die Commit-Historie des blessed repository niemals verändern!

GIT Befehle

- Erstellen eines neuen lokalen Repositories: `git init`
- Lokalen Klon von entferntem Repositories anlegen: `git clone <Repository-URL>`
- Geänderte Dateien anzeigen: `git status`
- Zeilenweise Änderungen anzeigen: `git diff (<Datei-Pfad>)`
- Änderungshistorie ansehen: `git log`
- Commit ansehen: `git show <Commit-Hash>`
- Dateien dem nächsten Commit hinzufügen: `git add (--all|<Datei-Pfad>)`
- Commit anlegen: `git commit (-m " <Beschreibung>")`
- Neuen Branch anlegen: `git checkout -b <Branch-Name>`
- Aktuellen Branch wechseln: `git checkout <Branch-Name>`
- Commits eines anderen in den aktuellen ziehen: `git merge <Branch-Name>`
- Commits vom entfernten zum lokalen Repository holen: `git fetch`
- Commits vom lokalen zum entfernten Repository schieben: `git push`
- Kombination von `git fetch` und `git merge`: `git pull`

.gitignore

Die Datei .gitignore listet alle Arten von Dateien und Verzeichnissen auf, die von Git ignoriert werden sollen. Dies sind typischerweise alle Artefakte, die automatische generiert werden als Teil des Entwicklungsprozesses.

- Kommentare beginnen mit #
- Leerzeilen sind erlaubt
- jede nicht-leere Zeile, die kein Kommentar ist, beschreibt ein Muster
- Wildcards (*) sind erlaubt
- ! am Anfang negiert ein Muster
- "/" separiert Verzeichnisse

Beispiel

```
*.bak
*.class
*.jar
target/

# "Editors"
.vscode/
.zed/
.idea/
```

Übung

2.1. Eine erste Übung mit GIT

Installieren Sie Git auf Ihrem System, falls es nicht verfügbar sein sollte.

1. Erstellen Sie ein neues Verzeichnis und legen Sie darin ein neues lokales Repository mit Hilfe von `git init` an.
2. Entpacken Sie die Datei https://delors.github.io/se-versionskontrolle/exercise/RPN.zip in dem Verzeichnis.
3. Führen Sie einen initialen Commit durch mit Hilfe von `git add` und `git commit`.
4. Compilieren Sie die Sourcen mit `javac`.
5. Legen Sie eine `.gitignore` Datei an, um sicherzustellen, dass die Binärdateien nicht in das Repository gelangen.
6. Nutzen Sie `git status`, um sich zu vergewissern, dass die Binärdateien ignoriert werden.
7. fügen Sie die `.gitignore` Datei Ihrem Repository hinzu.
8. Erstellen Sie einen neuen Branch mit dem Namen `feature/bugfix` und wechseln Sie auf den neuen Branch.
   (Nutzen Sie git status, um zu verifizieren, dass Sie auf dem neuen Branch sind)
9. Ändern Sie die Datei `RPN.java`, um den Bug im Switch statement (`case "* " => case "*"`) zu beheben.
10. Committen Sie die Änderungen.
11. Wechseln Sie zurück auf den `main` Branch.
12. Mergen Sie den Branch `feature/bugfix` in den `main` Branch mit Hilfe von `git merge`.
13. Löschen Sie den Branch `feature/bugfix` mit Hilfe von `git branch -d`.
14. Erstellen Sie einen neuen Branch develop und wechseln Sie auf diesen Branch.
15. Ändern Sie die Reihenfolge der Methoden `pop` und `peek` in der Klasse `Stack`.
16. Committen Sie die Änderungen.
17. Wechseln Sie zurück auf den `main` Branch.
    (Führen Sie noch keinen Merge durch!)
18. Entfernen Sie die `{}` Klammern um die `throw new NoSuchElementException()` Anweisungen.
19. Committen Sie die Änderungen.
20. Führen Sie einen Merge von `develop` in `main` durch.
21. Öffnen Sie die Datei `ds/Stack.java` und beheben Sie den Merge-Konflikt.
22. Committen Sie die Änderungen.
23. Nutzen Sie `git log --oneline --graph --all` um sich die Commit-Historie anzusehen.
24. Wechseln Sie zurück zum `develop` Branch.
25. Führen Sie einen Merge von `main` in `develop` durch.