

SPM3

Moderne Entwicklungsprozesse und Werkzeuge

Organisatorisches

Stefan Wagner, Philipp Fleck

Inhalt

- professionelle Softwareentwicklung ist ein komplexer und dynamischer Prozess mit vielen unterschiedlichen Aufgaben und Akteuren
 - agile Methoden erhöhen die Flexibilität in der Projektabwicklung
- Vielzahl an Software Werkzeugen zur Unterstützung verfügbar:
 - integrierte Entwicklungsumgebungen
 - Versionsverwaltungssysteme
 - Issue-Tracking-Systeme
 - Spezifikations- und Entwurfssysteme
 - Build Server
 - Test Frameworks
 - Codegeneratoren
 - statische Codeanalysatoren
 - Profiler
 - Projektmanagementsysteme
 - etc.
- Computer Aided Software Engineering (CASE)

Inhalt

- Verständnis von agilen Methoden und Umgang mit solchen Werkzeugen ist in der Praxis unerlässlich
 - Umgang mit solchen Werkzeugen wird daher im Rahmen der Studienprojekte geübt
- viele dieser Werkzeuge sind frei verfügbar und haben individuelle (und oftmals subjektive) Stärken und Schwächen
- IT Team am Campus Hagenberg bietet Versionsverwaltungssysteme an:
 - Git / GitLab
 - Subversion
- andere Werkzeuge können oftmals von kleinen Gruppen auch kostenlos genutzt werden:
 - Azure DevOps
 - Jira Software
 - Trello
 - ...
- virtuelle Projektserver können von Projektgruppen bei der IT beantragt werden:
 - Administration und Betrieb des Servers wird von der Projektgruppe selbst übernommen
 - auf den Projektservern können Werkzeuge von der Projektgruppe individuell eingesetzt werden

Inhalt

- Lehrveranstaltungen zur Vorbereitung des Einsatzes von Prozessen und Werkzeugen:
 - 3. Semester:
SPM3VO & SPM3UE: Moderne Entwicklungsprozesse und Werkzeuge
 - Vorlesung zu Methoden der agilen Softwareentwicklung
 - Übung zum Einsatz von Werkzeugen bei der Durchführung agiler Softwareprojekte
 - Versionsverwaltung und Task-Management anhand einiger exemplarischer Werkzeuge
 - Vorbereitung für den konkreten Einsatz in Projekten
 - 4. Semester:
SPW4VO & SPW4UE: Software-Entwicklungsprozesse – Testautomatisierung und Continuous Delivery
 - geblockt im Laufe des 4. Semesters
 - Vorlesung und Übung (jeweils 1 SWS)
 - Unit Testen, testgetriebene Softwareentwicklung, Testautomatisierung, CI/CD
 - Vorbereitung für den konkreten Einsatz in den Studienprojekten
- Ziele dieser Lehrveranstaltungen:
 - Einführung in den Einsatz von Task-Management Werkzeugen zur Unterstützung agiler Softwareprojekte
 - Azure DevOps, Jira Software, GitLab, ...
 - Einführung in den Einsatz von Versionsverwaltungssystemen
 - Git, GitHub, GitLab
 - Einführung in die Entwicklung und Automatisierung von Softwaretests und der gesamten Deployment Pipeline

SPM3

Moderne Entwicklungsprozesse und Werkzeuge

Versionsverwaltung

Stefan Wagner

Warum Versionsverwaltung?

- Softwareentwicklung ist ein dynamischer Prozess
 - stetige Weiterentwicklung
 - lange Entwicklungszeiträume
 - viele Produktversionen
 - viele und wechselnde Entwickler
 - örtliche Dezentralität
- zentrale Frage:
WER hat WAS, WANN und WARUM geändert?
- Lösung:
Versionsverwaltungssysteme (Version Control Systems, VCS)
 - wer? Autor eines Commit
 - was? Dateien bzw. Differenz eines Commit
 - wann? Zeitstempel eines Commit
 - warum? Beschreibung eines Commit (Commit Message)

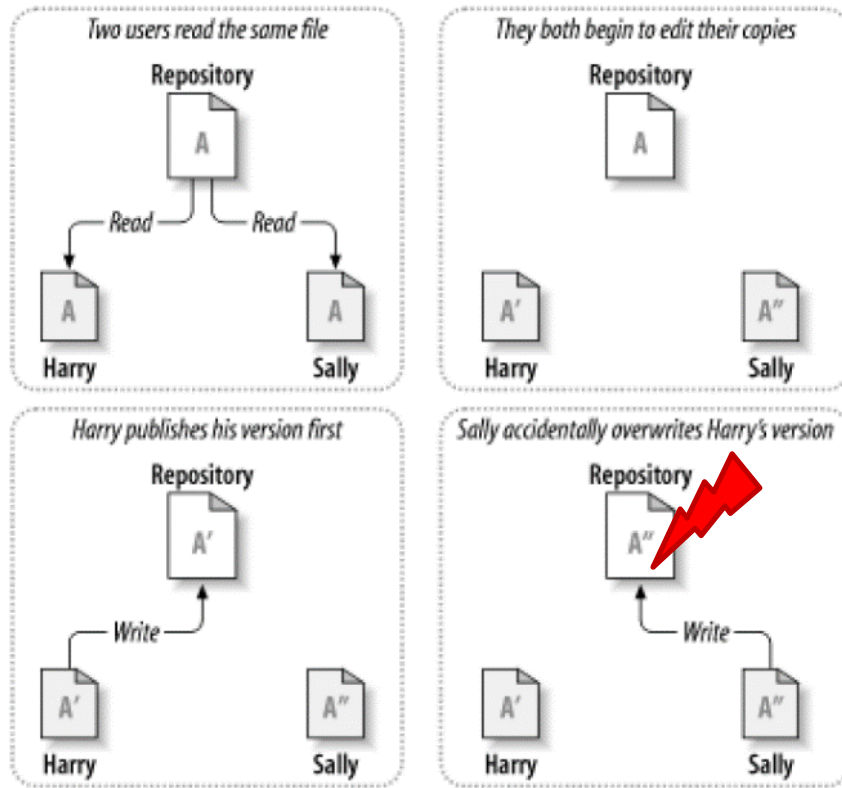
Hauptaufgaben von VCS

- Protokollierung von Änderungen
- Wiederherstellung von alten Versionen
- Archivierung von alten Versionen (Releases)
- Koordinierung des Zugriffs mehrerer Entwickler
- gleichzeitige Entwicklung mehrerer Entwicklungszweige (Branches)

Aufbau von VCS

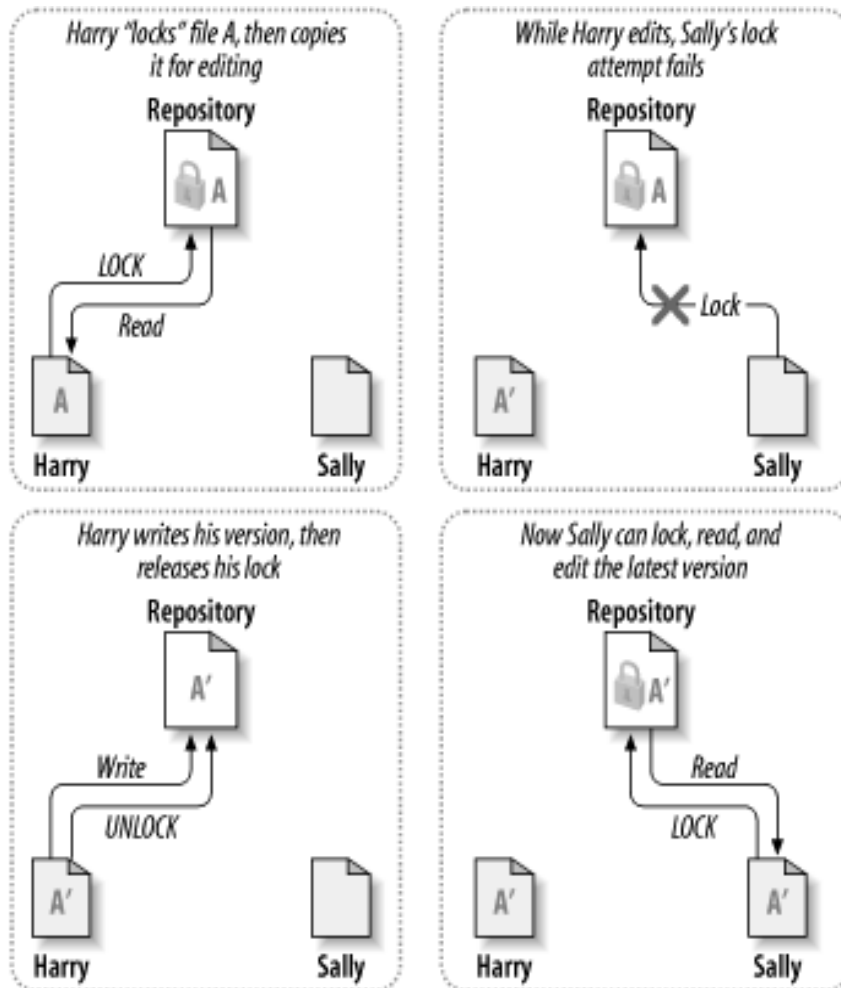
- Archiv (Repository)
 - Speicherung der Dateien in eigenem Dateiformat oder Datenbank
 - nur Speicherung der Änderungen zwischen den Versionen (Platzersparnis)
 - Zugriff auf Dateien nur via VCS
- Arbeitskopie (Working Copy)
 - lokale Kopie einer Version zur Bearbeitung
- Update der Arbeitskopie (Update, Checkout, Pull)
 - Aktualisieren der Arbeitskopie auf eine neuere Version
- Update des Repository (Commit)
 - Übertragung der Änderungen in der Arbeitskopie an das Repository und gleichzeitiges Anlegen einer neuen Version

Problem bei gleichzeitiger Bearbeitung



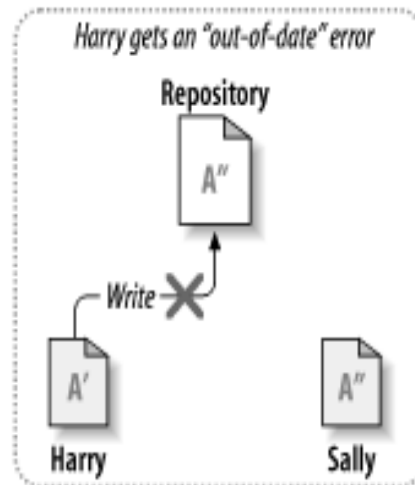
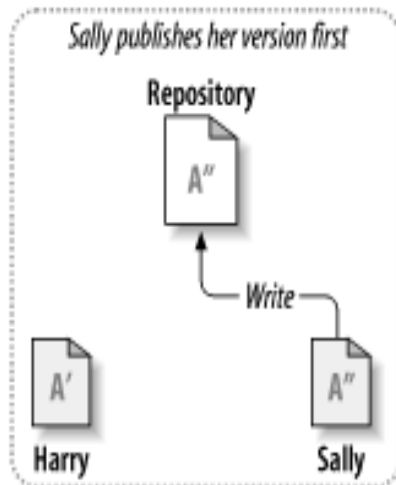
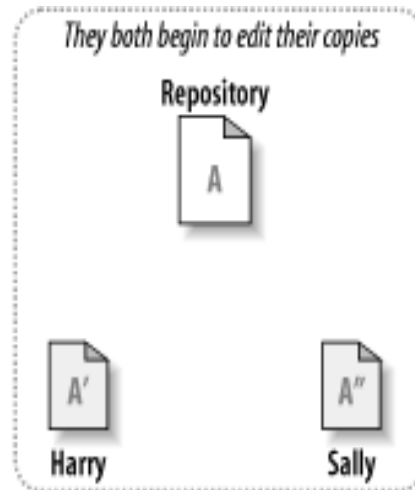
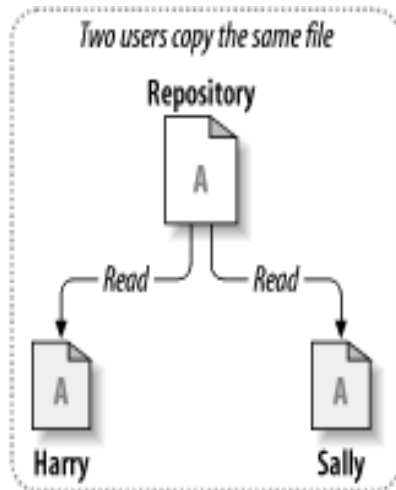
- gleichzeitige Erstellung von zwei neuen Versionen
- eine Version wird zurückgeschrieben
- zweite Version wird zurückgeschrieben und überschreibt dabei erste Version
- aktuelle Version im Repository enthält nur noch Änderungen der zweiten Version
- Änderungen der ersten Version gehen verloren

Strategien – Lock-Modify-Unlock



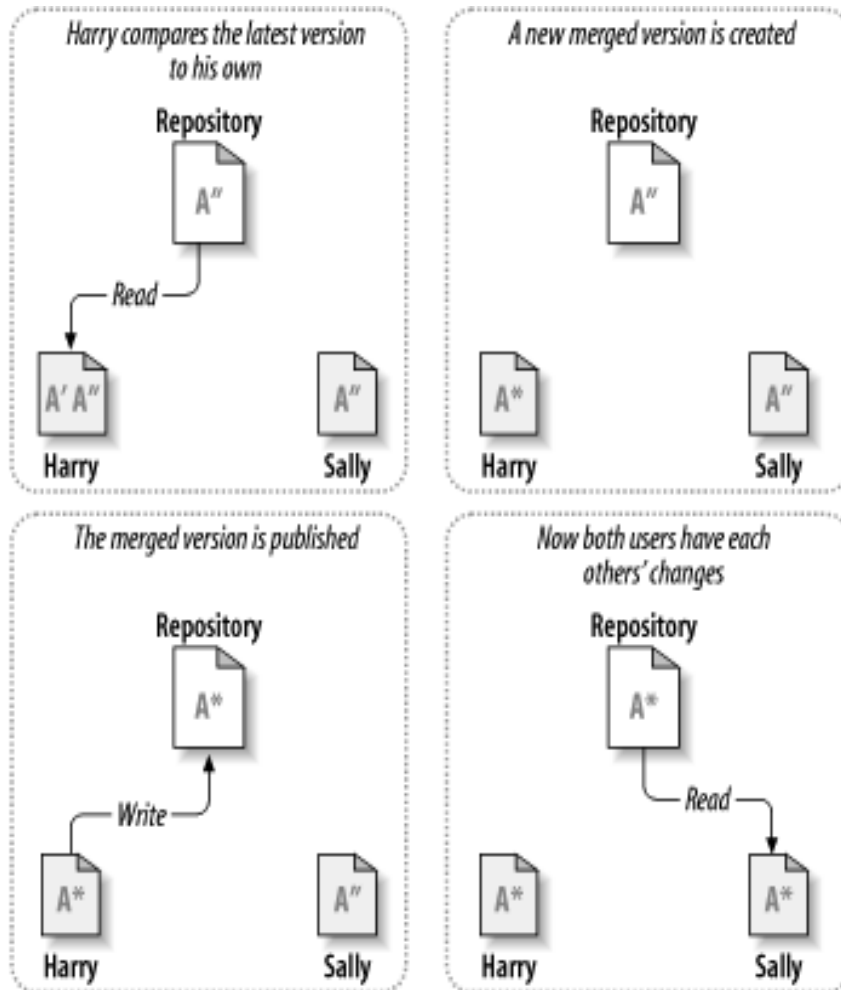
- Sperren einer Datei vor der Bearbeitung
- Freigabe der Sperre nach der Bearbeitung
- Probleme:
 - Vergessen von Sperren
 - unnötige Serialisierung des Workflow
 - trügerischer Eindruck von Sicherheit
 - Gefahr von semantischer Inkompatibilität

Strategien – Copy-Modify-Merge



- Anlegen von beliebig vielen Arbeitskopien
- gleichzeitige Änderung der Arbeitskopien
- Aktualisierung der lokalen Änderungen im Repository
- Behebung von Konfliktsituationen
 - automatisch (wenn sich geänderte Bereiche nicht überschneiden)

Strategien – Copy-Modify-Merge

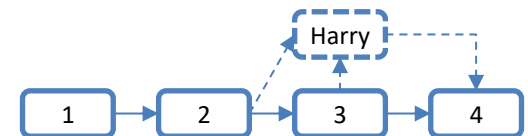


- Behebung von Konfliktsituationen
 - manuell (wenn sich geänderte Bereiche überschneiden)
- Vergleichen der letzten Version mit der Arbeitskopie
- Erstellen einer gemeinsamen Version zur Auflösung des Konflikt von Hand
- Aktualisierung des Repository

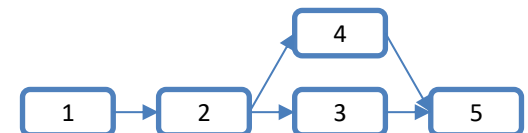
Entwicklung von VCS

- 1. Generation:
 - ein lokales Repository
 - Operationen auf einzelnen Dateien
 - Locks zur Synchronisation von Änderungen
- 2. Generation:
 - ein zentrales, entferntes Repository
 - Operationen auf Dateigruppen
 - *merge before commit* Strategie:
vor jedem Commit muss konsistenter Zustand hergestellt werden
- 3. Generation:
 - mehrere verteilte Repositories
 - Operationen auf Dateigruppen und Änderungsmengen
 - *commit before merge* Strategie:
Commits sind jederzeit möglich, Konsistenz wird anschließend mit Merge hergestellt

merge before commit:



commit before merge:

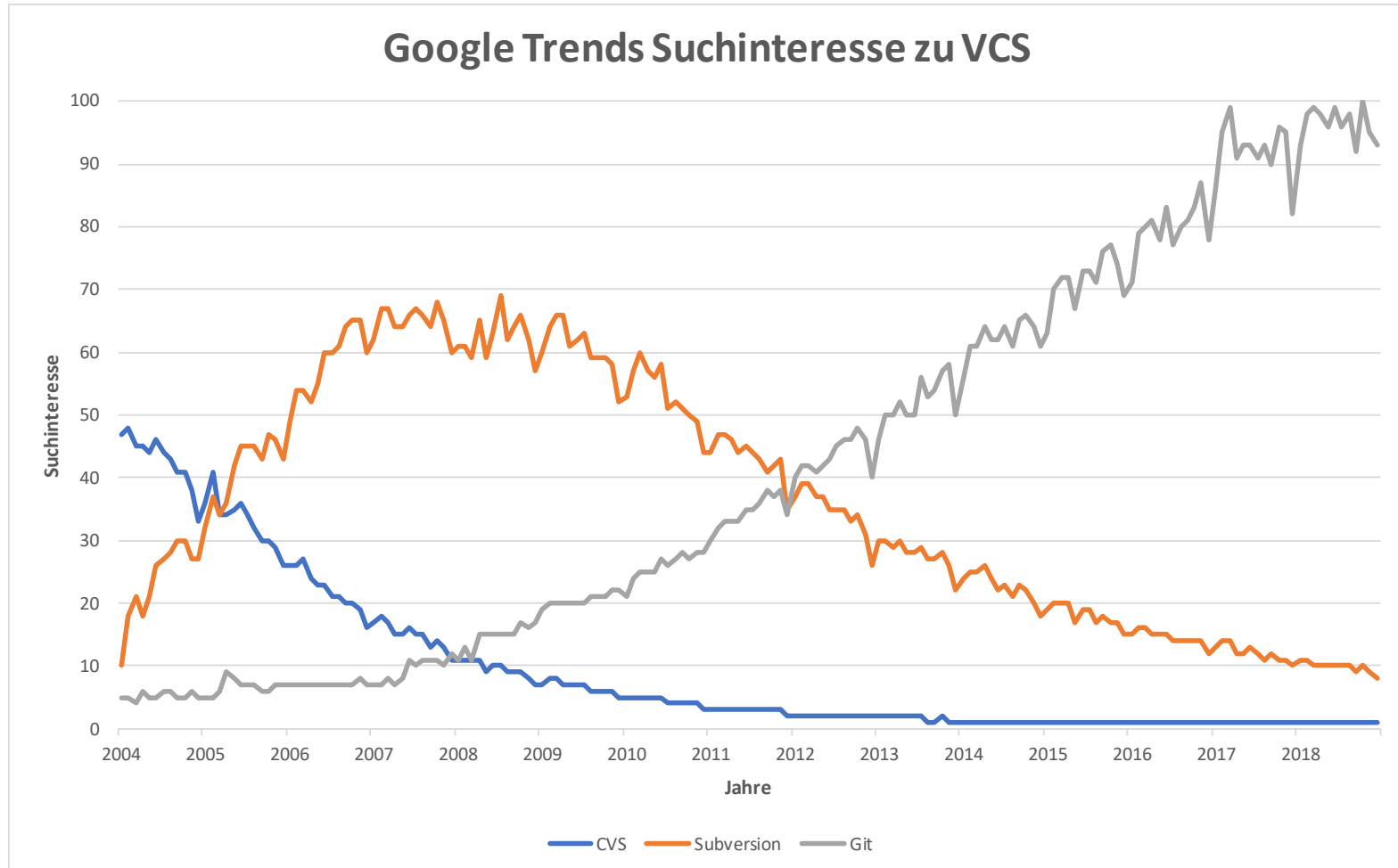


Geschichte von Git



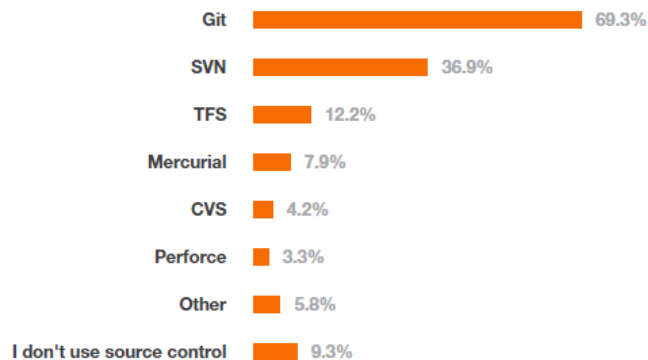
- **Ende 1990er**
 - erste verteilte VCS entstehen
- **1998**
 - erstes ernstzunehmendes System: BitKeeper (proprietär)
- **2002**
 - BitKeeper wird für die Linux Kernel Entwicklung verwendet ("Linus did not scale")
- **2005**
 - Lizenzänderung von BitKeeper führt zum Bruch mit der Linux Community und damit zur Entwicklung von Git
- **Dezember 2005**
 - Release von Git 1.0
- **ab 2005**
 - Start des Booms von verteilten VCS
 - Open Source Projekte wechseln von SVN/CVS zu verteilten VCS (vor allem Git, aber auch Mercurial oder Bazaar)
- **2008**
 - github.com geht online
- **Mai 2014**
 - Release von Git 2.0
- **September 2018**
 - Release von Git 2.19

Verbreitung von VCS



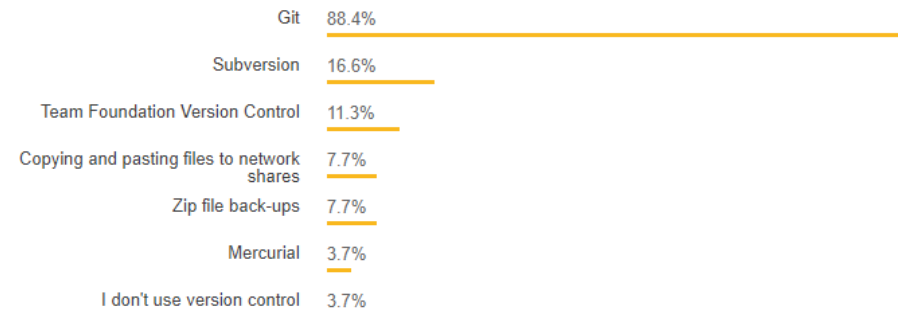
Verbreitung von VCS

Stack Overflow Developer Survey 2015:



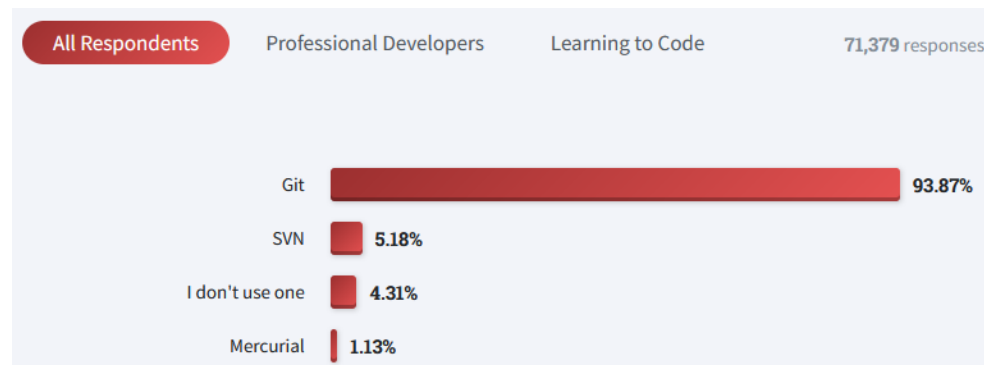
16,694 responses

Stack Overflow Developer Survey 2018:



69,808 responses; select all that apply

Stack Overflow Developer Survey 2022:

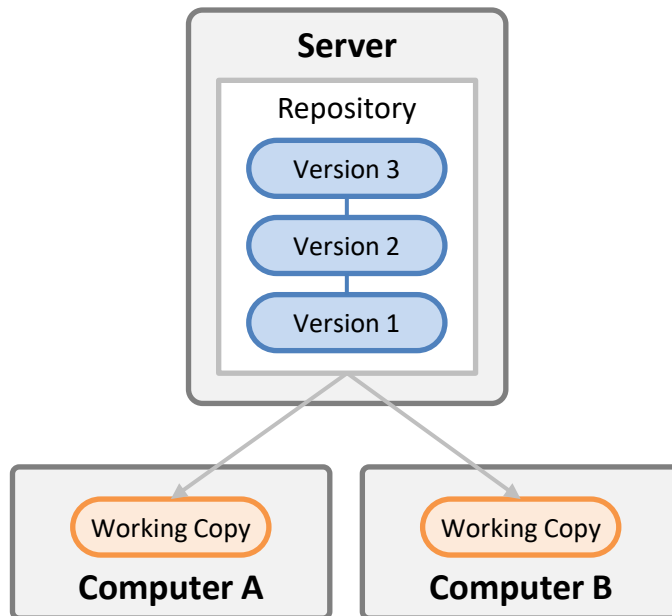


Motivation für verteilte VCS

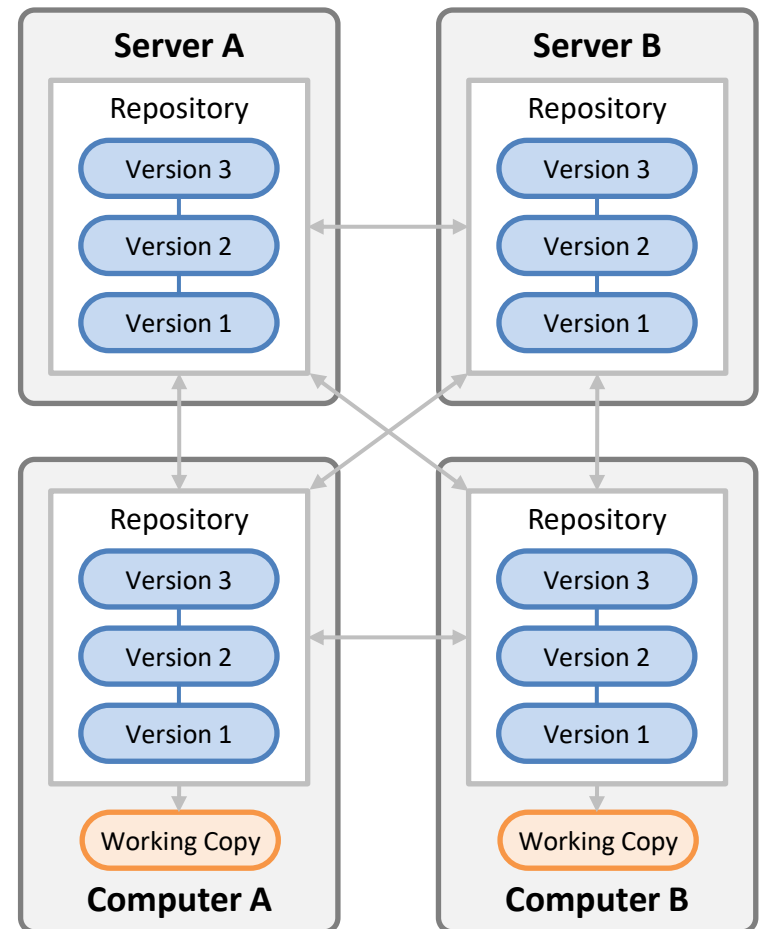
- zentrale VCS sind von zentralem VCS Server abhängig
 - Verbindung zum zentralen VCS Server ist für fast alle Operationen erforderlich
 - offline zu arbeiten ist daher nicht möglich
 - Netzwerkkommunikation macht viele Operationen langsam
 - zentraler VCS Server ist ein Single Point of Failure
- zentrale VCS skalieren nicht gut
 - alle Personen eines Projekts benötigen lesenden und schreibenden Zugriff auf den zentralen VCS Server
 - Last kann nicht einfach auf mehrere Server verteilt werden
 - wer den zentralen VCS Server kontrolliert, kontrolliert das gesamte Projekt
- zentrale VCS sind daher insbesondere für Open Source Projekte mit einer großen und lose gekoppelten Entwicklergemeinschaft nicht gut geeignet

Zentrale vs. verteilte VCS

zentrales VCS:



verteiltes VCS:



Zentrale vs. verteilte VCS

zentrale VCS:

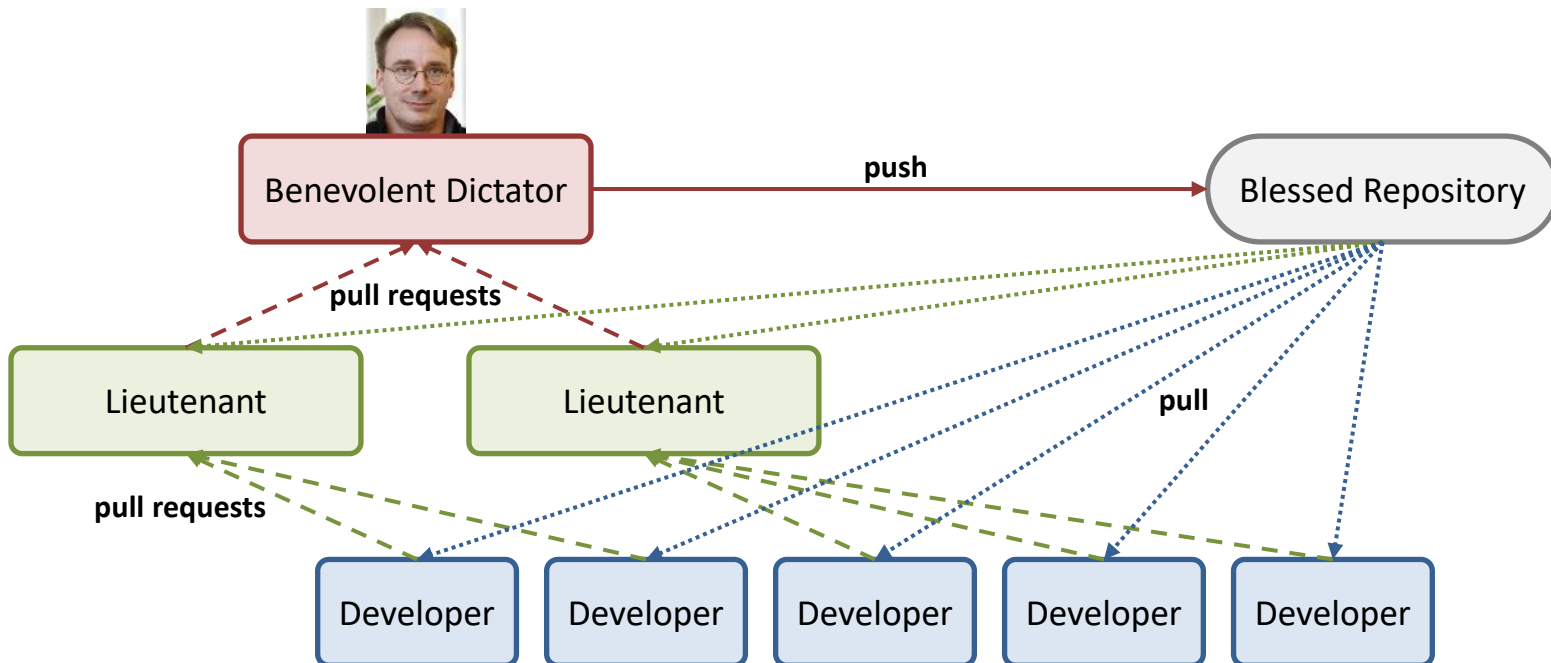
- zentraler Server hält Repository
- Clients halten nur lokale Arbeitskopie
- für viele Operationen ist eine Verbindung zum Server erforderlich
- Erzeugen neuer Version ist ohne Netzwerkverbindung nicht möglich
- zentrales Repository macht die Synchronisation und Integration von Änderungen leichter
- ideal für stark vernetzte und eng zusammenarbeitende Teams

verteilte VCS:

- P2P Modell
- kein zentraler Server erforderlich
- kein zentrales Repository
- Clients halten gesamtes Repository und lokale Arbeitskopie
- viele Operationen erfolgen nur lokal und sind daher sehr schnell
- Erzeugen neuer Versionen ist ohne Netzwerkverbindung möglich
- verteilte Repositories machen die Synchronisation und Integration von Änderungen komplexer
- ideal für schwach vernetzte und lose zusammenarbeitende Gruppen (z.B. Open Source Projekte)

Entwicklungsmodell des Linux Kernel

- Benevolent Dictator Workflow
 - skaliert für sehr große Projekte mit hunderten Entwicklern
 - strenge, hierarchische Organisation
 - mehrstufige Struktur ermöglicht Delegation
 - kein schreibender Zugriff auf fremde Repositories erforderlich



Grundlegende Operationen eines VCS

- **Create**
 - neues Repository erzeugen
- **Checkout**
 - Arbeitskopie erzeugen
- **Commit**
 - Änderungen von der Arbeitskopie in das Repository übertragen
- **Update**
 - Änderungen vom Repository in die Arbeitskopie übertragen
- **Add**
 - Dateien zur Versionsverwaltung hinzufügen
- **Edit**
 - Dateien bearbeiten
- **Delete**
 - Dateien löschen
- **Rename**
 - Dateien umbenennen
- **Move**
 - Dateien verschieben
- **Status**
 - Änderungen in der Arbeitskopie auflisten
- **Diff**
 - Änderungen in einer Datei anzeigen
- **Revert**
 - Änderungen in der Arbeitskopie zurücknehmen
- **Log**
 - Historie der Versionen im Repository anzeigen
- **Tag**
 - spezifische Version mit einem sprechenden Namen versehen
- **Branch**
 - neuen Entwicklungszweig erzeugen
- **Merge**
 - Änderungen in einem Entwicklungszweig auf einen anderen übertragen
- **Resolve**
 - bei Merge entstandene Konflikte auflösen
- **Lock**
 - Dateien explizit sperren (z.B. für Binärdateien)

Die Bezeichnung und Bedeutung dieser Operationen ist in den verschiedenen VCS teilweise unterschiedlich.

Branching & Merging

- wenn viele Entwickler gemeinsam an einem Entwicklungszweig arbeiten, steigt die Wahrscheinlichkeit wechselseitiger Konflikte und Blockaden
- ein Rechenbeispiel:
 - 0.1% Wahrscheinlichkeit, dass ein Entwickler mit einem Commit einen Build Break verursacht
 - 10 Commits durchschnittlich pro Tag und Entwickler
 - Entwicklung mit 5 Entwicklern:
 - 50 Commits pro Tag
 - ca. alle 20 Tage ein Build Break 😊
 - Entwicklung mit 100 Entwicklern:
 - 1.000 Commits pro Tag
 - **jeden Tag ein Build Break** ☹
- Branches erlauben die gleichzeitige und unabhängige Weiterentwicklung unterschiedlicher Entwicklungszweige
- Entwicklung kann auf kleinere Teams aufgeteilt werden, die sich nicht beeinflussen
- Entwicklung und Integration werden dadurch entkoppelt
- Merge bezeichnet die Übernahme von Änderungen aus einem Entwicklungszweig in einen anderen
- dabei entstehen ev. Konflikte, die manuell aufgelöst werden müssen

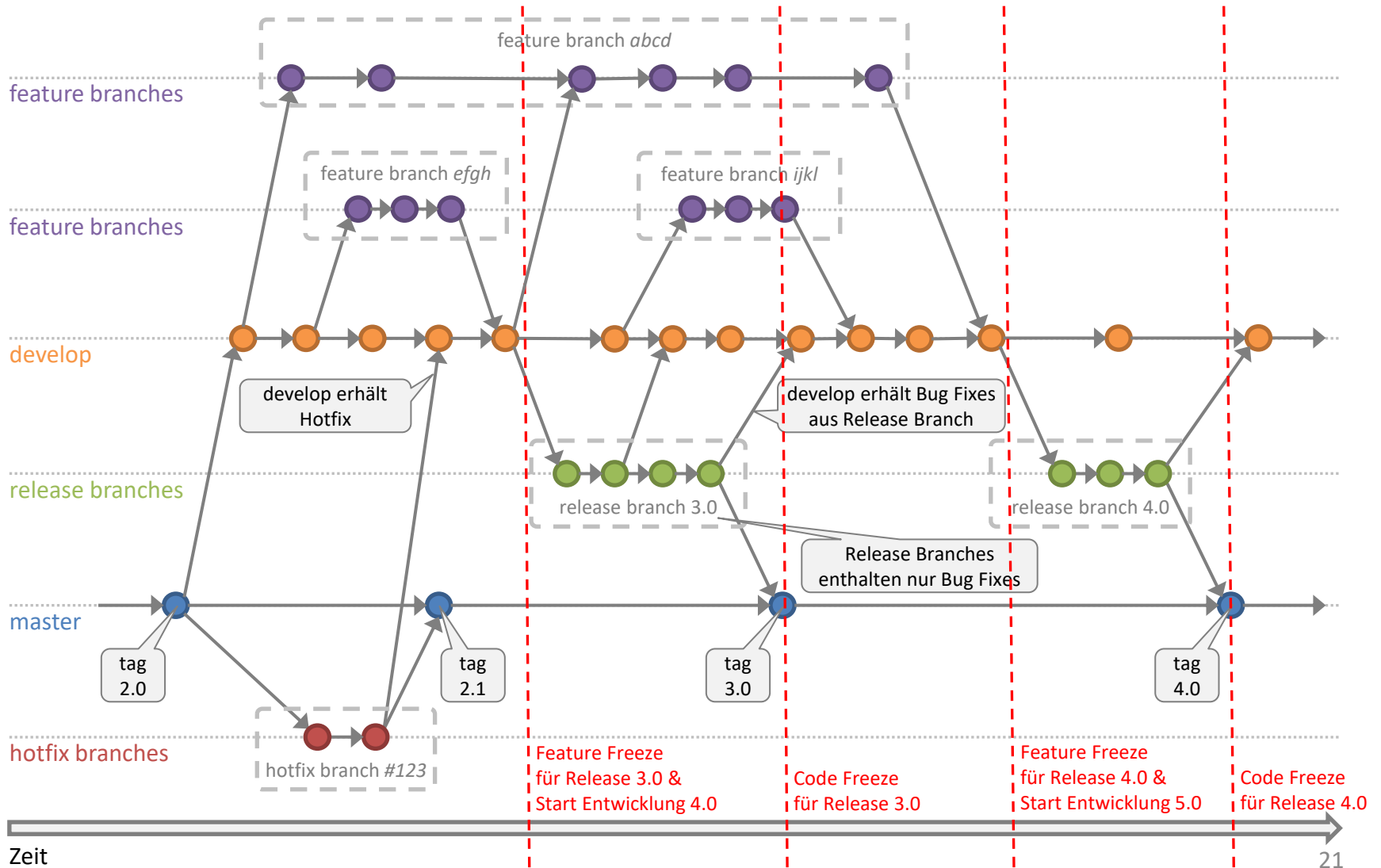
Tags

- Tags dienen zur Markierung bestimmter Versionen mit einem sprechenden Namen
- Archivierung eines Entwicklungsstands zu einem bestimmten Zeitpunkt (z.B. Releases)
- Tags sind statisch und werden nicht weiterentwickelt

Branch Arten

- Master Branch / Stable Branch / Production Branch
 - enthält nur stabile und getestete Versionen (z.B. Releases)
- Development Branch
 - enthält aktuelle Entwicklung für das nächste Release
- Feature Branch / Topic Branch / Refactoring Branch
 - enthält Entwicklung eines spezifischen Features oder Refactorings für das nächste oder zukünftige Releases
- Release Branch
 - enthält nur Bug Fixes und dient zur Stabilisierung (d.h. Tests) des nächsten Release
- Hotfix Branch
 - enthält dringenden Bug Fix für ein bereits veröffentlichtes Release

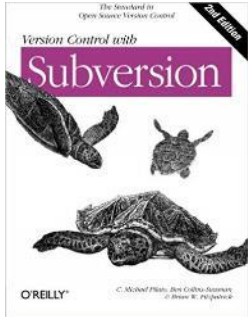
Branching Beispiel



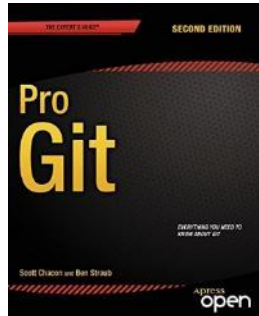
Integration Hell

- Branches mit langer Entwicklungszeit divergieren vom Hauptentwicklungsstrang
- Integration solcher Branches kann aufgrund von Konflikten und semantischen Inkonsistenzen sehr aufwändig und fehleranfällig sein
- Änderungen vom Hauptentwicklungsstrang sollten in Branches regelmäßig übernommen werden
- dadurch wird Divergenz reduziert und die Integration in den Hauptentwicklungsstrang am Ende des Branches wesentlich vereinfacht

Quellen



B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato:
Version Control with Subversion
<http://svnbook.red-bean.com>



S. Chacon, B. Straub:
Pro Git
<https://git-scm.com/book/en/v2>



E. Sink:
Version Control by Example
<https://ericsink.com/vcbe/index.html>



R. Preißel, B. Stachmann:
Git: Dezentrale Versionsverwaltung im Team – Grundlagen und Workflows
dpunkt.verlag

SPM3

Moderne Entwicklungsprozesse und Werkzeuge

Git

Stefan Wagner

Git



Git vs. Subversion

Git

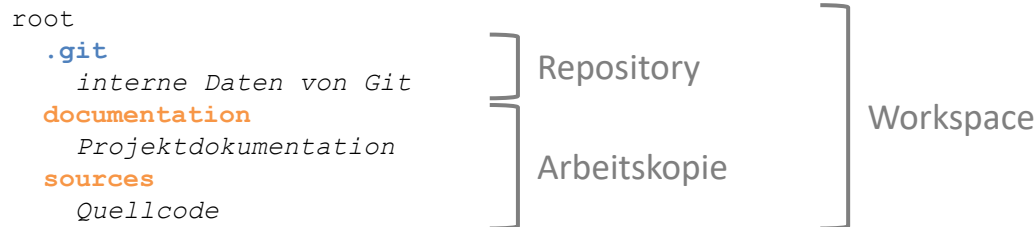
- dezentrales VCS
- Versionen werden mit Hashes referenziert
- Verzeichnisse werden nicht versioniert
- Arbeitskopie umfasst immer alle Dateien einer Version
- Branches/Tags sind Zeiger auf Commits
- Branch/Tag umfasst immer alle Dateien einer Version
- Berechtigungsmanagement nur für ganze Repositories möglich

Subversion

- zentrales VCS
- Versionen werden mit Revisionsnummer referenziert
- auch Verzeichnisse werden versioniert
- partielle Arbeitskopie eines Teils einer Version möglich
- Branches/Tags sind Kopien von Verzeichnissen
- Branch/Tag eines Teils einer Version möglich
- Berechtigungsmanagement innerhalb eines Repositories möglich

Git Workspaces

- Git Workspace bezeichnet ein Verzeichnis, in dem Dateien mit Git versioniert und verwaltet werden
- Git Workspace umfasst:
 - ein Repository mit der gesamten Historie aller Versionen inkl. aller Branches/Tags/etc.
 - einen Stand der versionierten Dateien, mit denen gearbeitet werden kann
- jeder Workspace enthält auf oberster Ebene ein Verzeichnis *.git*, in dem Git die gesamten Daten des Repositories verwaltet
- Dateien im Workspace können beliebig strukturiert werden:



- *bare* Repositories haben keinen umschließenden Workspace
 - dienen zur Ablage auf Servern, auf denen nicht direkt gearbeitet wird
 - enthalten nur die Daten des Repositories, also den Inhalt des *.git* Verzeichnisses
 - Namenskonvention: bare Repositories enden auf *.git*

Anlegen eines Git Workspace

- Git Workspace kann entweder neu angelegt oder von einem bestehenden Repository kopiert werden
- [git init](#)
 - erstellt einen neuen Git Workspace
 - mit der Option `-b` (`--initial-branch`) kann der Name des initialen Branch festgelegt werden (z.B. `main` statt `master`)
 - Option `--bare` legt nur ein bare Repository an
 - Beispiel: `git init -b main`
- [git clone](#)
 - kopiert ein bestehendes Git Repository und erstellt dafür einen neuen Workspace
 - mit der Option `-o` (`--origin`) kann der Name des Ursprungs festgelegt werden
 - Option `-b` (`--branch`) gibt den Namen des Branch an, der ausgecheckt werden soll
 - Beispiel: `git clone https://github.com/microsoft/vscode`

Einstellungen

- Git speichert Einstellungen in Konfigurationsdateien
- [git config](#)
 - Option `-l` (`--list`) zeigt alle vorhandenen Einstellungen an
 - Option `--unset` entfernt eine Einstellung
 - 3 Ebenen:
 - System (`--system`)
 - systemweite Einstellungen für alle Benutzer
 - unter `/etc/gitconfig` (Linux) oder `<GitInstallDir>\etc\gitconfig` (Windows)
 - Benutzer (`--global`)
 - benutzerspezifische Einstellungen für alle Repositories dieses Benutzers
 - unter `<UserHomeDir>/.gitconfig`
 - Repository (`--local` = Standard)
 - individuelle Einstellungen für ein einzelnes Repository
 - unter `<WorkspaceDir>/.git/config`
 - spezifischere Ebenen überlagern allgemeinere Ebenen: Repository → Benutzer → System
 - Option `--show-origin` zeigt an, wo eine Einstellung definiert wurde
- initial sollten zumindest Name und E-Mail-Adresse eingestellt werden:

```
git config --global user.name "Dave Developer"
git config --global user.email dave@developers.com
```
- mit *alias.<command>* können individuelle Kurzformen für Befehle festgelegt werden:

```
git config --global alias.unstage 'reset HEAD --'
git config --global alias.graph 'log --all --decorate --oneline --graph'
git config --global alias.guiclient '!gitk'
```

Git Repositories

- Git Repositories enthalten:
 - Dateiinhalte (*Blobs*)
 - textuell oder binär
 - werden unabhängig von Dateinamen gespeichert
 - Verzeichnisse (*Trees*)
 - verknüpfen Dateinamen mit Dateiinhalten
 - können weitere Verzeichnisse enthalten
 - Versionen (*Commits*)
 - speichern einen Stand des Wurzelverzeichnisses, d.h. des gesamten Arbeitsverzeichnisses
 - enthalten zusätzlich noch Metadaten (Autor, Committer, Zeitstempel, Beschreibung, Verweis auf Vorgängerversion)
- für jeden Blob/Tree/Commit wird ein Hashwert gespeichert, über den das jeweilige Element referenziert werden kann
- Hashwerte von Commits entsprechen den Revisionsnummern in Subversion
- identische Inhalte werden nur einmal gespeichert
- Daten können nachträglich nicht verändert werden, da sich durch eine Änderung des Inhalts auch der Hashwert ändern würde

Datenstruktur von Commits

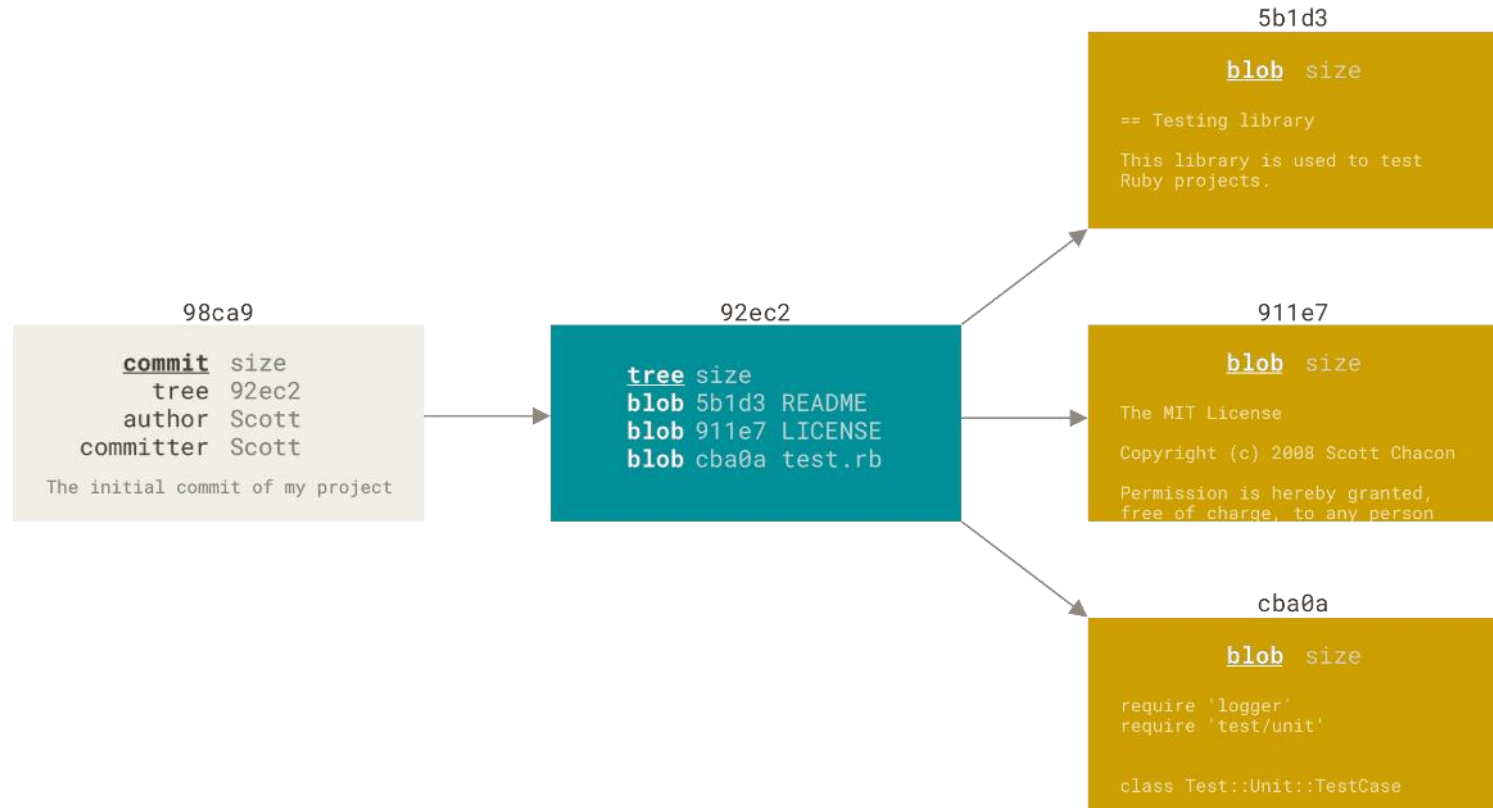


Abbildung aus S. Chacon, B. Straub: Pro Git

Datenstruktur von Commits

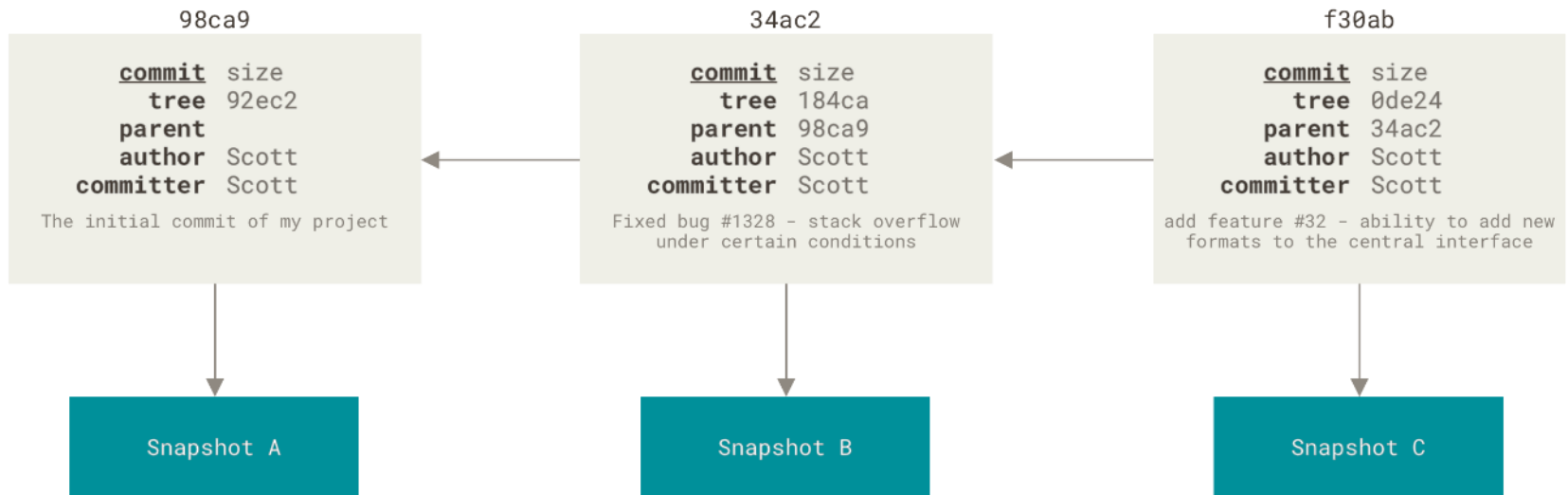


Abbildung aus S. Chacon, B. Straub: Pro Git

Plumbing vs. Porcelain

- Git hat sehr viele verschiedene Befehle
- manche dieser Befehle sind Standardbefehle (porcelain), mit denen laufend gearbeitet wird
- andere sind Basisbefehle (plumbing), die selten oder nie direkt benötigt werden
- ob ein Befehl zu den Standardbefehlen oder zu den Basisbefehlen gehört, ist leider oft nicht sofort zu erkennen
- Beispiel für einen Basisbefehl: [git cat-file](#)
 - gibt den Typ und den Inhalt von Objekten im Repository an
 - Option `-t` gibt den Typ des Objekts aus
 - Option `-p` gibt den Inhalt des Objects aus
 - Beispiel: `git cat-file -p HEAD`
- kurze Erklärung einiger Plumbing Commands: [A Plumber's Guide to Git](#)

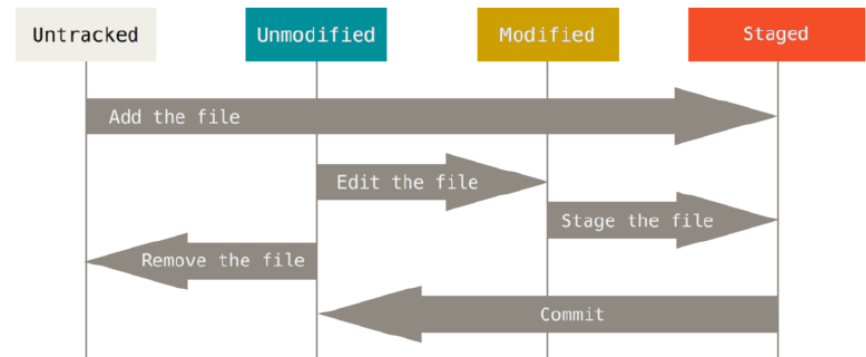
Typischer Arbeitsablauf mit Git

- lokales Repository aktualisieren:
[git pull](#)
- Änderungen durchführen und überprüfen:
[git status](#), [git diff](#)
- Änderungen zur Staging Area hinzufügen:
[git add](#)
- Änderungen in das lokale Repository schreiben:
[git commit](#)
- Änderungen an das entfernte Repository übertragen:
[git push](#)

Zustände von Dateien

- Dateien in einem Git Workspace können folgende Zustände haben:

- **untracked**
 - nicht unter Versionskontrolle
- **unmodified**
 - versioniert und nicht verändert
- **modified**
 - versioniert und verändert, aber nicht für den nächsten Commit vorgesehen
- **staged**
 - versioniert, verändert und für den nächsten Commit vorgesehen



- alle Änderungen, die vom nächsten Commit erfasst werden sollen, müssen mit [git add](#) in die Staging Area (Index) übertragen werden
 - Staging Area wird unter `.git/index` gespeichert
- nachdem eine Änderung in die Staging Area übertragen wurde, werden nachträgliche Änderungen nicht automatisch übernommen
 - falls eine Datei nach dem Übertragen in die Staging Area nochmals verändert wird und diese neue Änderungen ebenfalls vom nächsten Commit erfasst werden sollen, muss `git add` nochmals aufgerufen werden

Abbildung aus S. Chacon, B. Straub: Pro Git

Verschieben und Löschen von Dateien

- Verschiebungen, Umbenennungen oder Löschungen von Dateien müssen ebenfalls in die Staging Area übertragen werden, um vom nächsten Commit erfasst zu werden
- [git rm](#)
 - entfernt eine Datei aus dem Arbeitsverzeichnis und trägt die Löschung in die Staging Area ein
 - Option *--cached* trägt Löschung nur in die Staging Area ein und behält aber die Datei im Arbeitsverzeichnis
 - anstatt *git rm* kann auch ein normales *rm* mit anschließendem *git add* verwendet werden

<code>git rm code.c</code>	entspricht	<code>rm code.c</code>
		<code>git add code.c</code>

- [git mv](#)
 - verschiebt eine Datei im Arbeitsverzeichnis und trägt Verschiebung in die Staging Area ein
 - Verschiebungen/Umbenennungen von Dateien werden von Git auch automatisch aufgrund des gleichen Dateiinhalts (d.h. Hashwerts) erkannt
 - anstatt *git mv* kann auch ein normales *mv* mit anschließendem *git add* verwendet werden

<code>git mv old.c new.c</code>	entspricht	<code>mv old.c new.c</code>
		<code>git add old.c</code>
		<code>git add new.c</code>

Überprüfen von Änderungen

- [git status](#)
 - zeigt den Zustand der Dateien im Arbeitsverzeichnis an
 - Option `-s` (`--short`) zeigt Kurzform (ähnlich zu `svn status`)
- [git diff](#)
 - zeigt die inhaltlichen Änderungen in einer Datei im *unified diff* Format an
 - ohne weitere Option wird das Arbeitsverzeichnis mit der Staging Area verglichen
 - Option `--cached` vergleicht die Staging Area mit dem letzten Commit

Ignorieren von Dateien

- Dateien, die nicht versioniert werden sollen, können in *.gitignore* Dateien eingetragen werden
 - jedes Verzeichnis kann eine eigene *.gitignore* Datei enthalten
 - üblich ist jedoch nur eine *.gitignore* Datei im Wurzelverzeichnis des Workspace
- eine zu ignorierende Datei bzw. ein zu ignorierendes Verzeichnis pro Zeile
- Einträge ohne "/" zu Beginn werden rekursiv in allen Unterverzeichnissen angewendet
- Pfad ist relativ zu dem Verzeichnis, in dem die *.gitignore* Datei gespeichert ist
- Ausnahmen können mit "!" angegeben werden
- Platzhalter können verwendet werden, um mehrere Dateien auszunehmen:
 - * entspricht einer beliebigen Zeichenkette (inkl. der leeren Zeichenkette)
 - ** entspricht beliebigen verschachtelten Unterverzeichnissen
 - ? entspricht einem einzelnen Zeichen
 - [abc] entspricht einem Zeichen aus der angegebenen Menge (z.B. "a", "b" oder "c")

Ignorieren von Dateien

- Beispiel für eine *.gitignore* Datei:

.gitignore

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

- weitere Beispiele für viele Entwicklungsumgebungen und Programmiersprachen auf GitHub:
<https://github.com/github/gitignore>

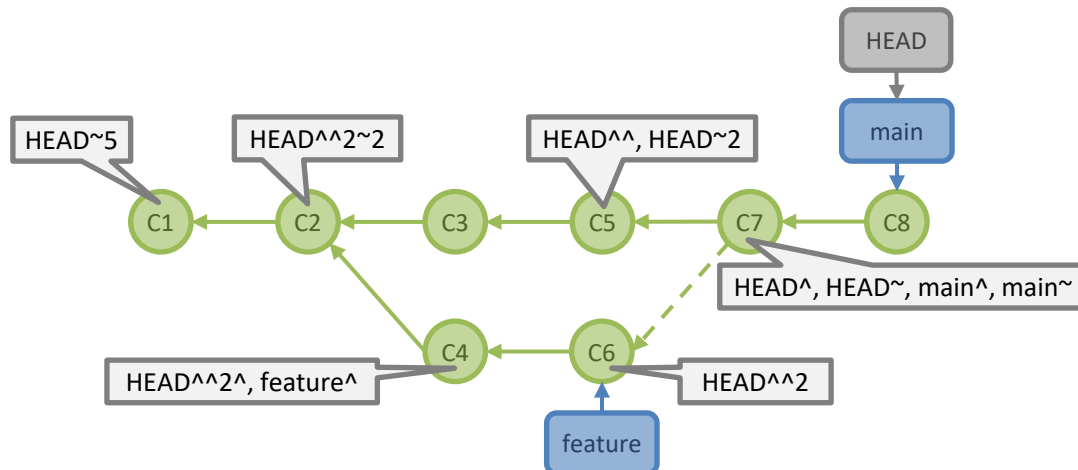
Beispiel aus S. Chacon, B. Straub: Pro Git

Speichern von Versionen im lokalen Repository

- [git commit](#)
 - speichert Änderungen in der Staging Area als neuen Commit im lokalen Repository
 - Option *-m* (*--message*) zur Angabe einer Beschreibung, ansonsten wird automatisch ein Editor gestartet
 - Option *-a* (*--all*) überträgt alle aktuellen Änderungen automatisch in die Staging Area und führt anschließend einen Commit durch
 - Option *--amend* ergänzt letzten Commit mit den aktuellen Änderungen in der Staging Area

Referenzieren von Commits

- Git verwendet Namen als Referenzen, um auf bestimmte Commits zu verweisen
- jeder Branch bzw. Tag ist eine solche Referenz
- HEAD referenziert den aktuellen Branch oder einen Commit (*detached head*)
- Zirkumflex (^) und Tilde (~) bezeichnen den Vorgänger
 - z.B. HEAD^ oder HEAD~ steht für den vorletzten Commit des aktuellen Branch
 - beide Zeichen können auch wiederholt werden
z.B. HEAD^^ oder HEAD~~ steht für den vorvorletzten Commit
 - beide Zeichen können mit einer Zahl kombiniert werden
 - bei ^ steht die Zahl für das jeweilige Elternteil
z.B. HEAD^2 = 2. Elternteil von HEAD
 - bei ~ steht die Zahl für Anzahl der Vorgänger
z.B. HEAD~3 = HEAD~~~



Anzeigen der Historie

- [git log](#)
 - listet die Historie vergangener Commits auf
 - Option *--oneline* zeigt eine kompaktere Darstellung mit nur einer Zeile pro Commit
 - Option *--graph* zeigt zusätzlich eine textuelle Visualisierung des Commitgraphen
 - Option *--decorate* fügt zusätzlich Referenzen auf Commits hinzu (d.h. Namen von Branches bzw. Tags)
 - neben einigen vordefinierten Standardformaten kann das Format der Ausgabe auch individuell angegeben werden (Option *--format=<format>*)
 - zahlreiche weitere Optionen um Historie zu filtern
 - Option *-p* (*--patch*) zeigt zusätzlich die konkreten Änderungen als unified diff an (vgl. *git diff*)

Anzeigen der Historie

- [git show](#)
 - zeigt Informationen zu einem Commit an (Autor, Datum, Commit Message, Änderungen, etc.)
 - Option *--oneline* zeigt die Metainformationen zum Commit in einer kompakteren Darstellung in nur einer Zeile an
 - Option *--name-status* zeigt nur die geänderten Dateinamen und den Status (hinzugefügt, geändert, gelöscht) ohne detaillierte Änderungen an
- [git diff](#)
 - zeigt die Änderungen zwischen zwei Commits bzw. einem Commit und der Arbeitskopie oder der Staging Area an
 - bei der Angabe von zwei Commits werden die Änderungen vom ersten zum zweiten Commit angezeigt
 - Option *--name-status* zeigt nur die geänderten Dateinamen und den Status (hinzugefügt, geändert, gelöscht) ohne detaillierte Änderungen an
 - Option *--cached* vergleicht die Staging Area mit dem letzten Commit

Befehl	Änderungen von →	auf
git diff	Staging Area	Arbeitskopie
git diff --cached	HEAD	Staging Area
git diff HEAD	HEAD	Arbeitskopie
git diff HEAD^ HEAD	Vorgänger von HEAD	HEAD
git diff branch1 main	branch1	main
git diff branch1...main	Ursprung von branch1	main

Zurücknehmen von Änderungen

- [git checkout](#) bewegt HEAD und aktualisiert das Arbeitsverzeichnis
 - kann auch verwendet werden, um einzelne Dateien im Arbeitsverzeichnis wiederherzustellen
- [git reset](#) bewegt den Branch auf den HEAD zeigt
 - mehrere Funktionsarten:
 - Option `--soft` bewegt nur den Branch/HEAD
 - Option `--mixed` ändert zusätzlich die Staging Area (Standardmodus)
 - Option `--hard` ändert zusätzlich das Arbeitsverzeichnis
- [git restore](#) stellt Dateien wieder her und verwirft Änderungen
 - Ziel der Wiederherstellung kann angegeben werden
 - Option `-W` (`--worktree`) stellt Datei im Arbeitsverzeichnis wieder her
 - Option `-S` (`--staged`) stellt Datei in der Staging Area wieder her
 - Optionen können auch gemeinsam verwendet werden
- [git revert](#) erzeugt einen neuen Commit der inhaltlich einem alten Commit entspricht
 - wenn der zurückzunehmende Commit bereits gepusht wurde, muss `git revert` verwendet werden, da die Historie nicht verändert werden darf

Zurücknehmen von Änderungen

- Änderungen im Arbeitsverzeichnis rückgängig machen:

```
git restore <file>
```

```
git checkout -- <file>
```

```
git reset --hard (Achtung: alle Änderungen)
```

- Staging einer Datei rückgängig machen:

```
git restore --staged <file>
```

```
git reset <file>
```

- Commit rückgängig machen:

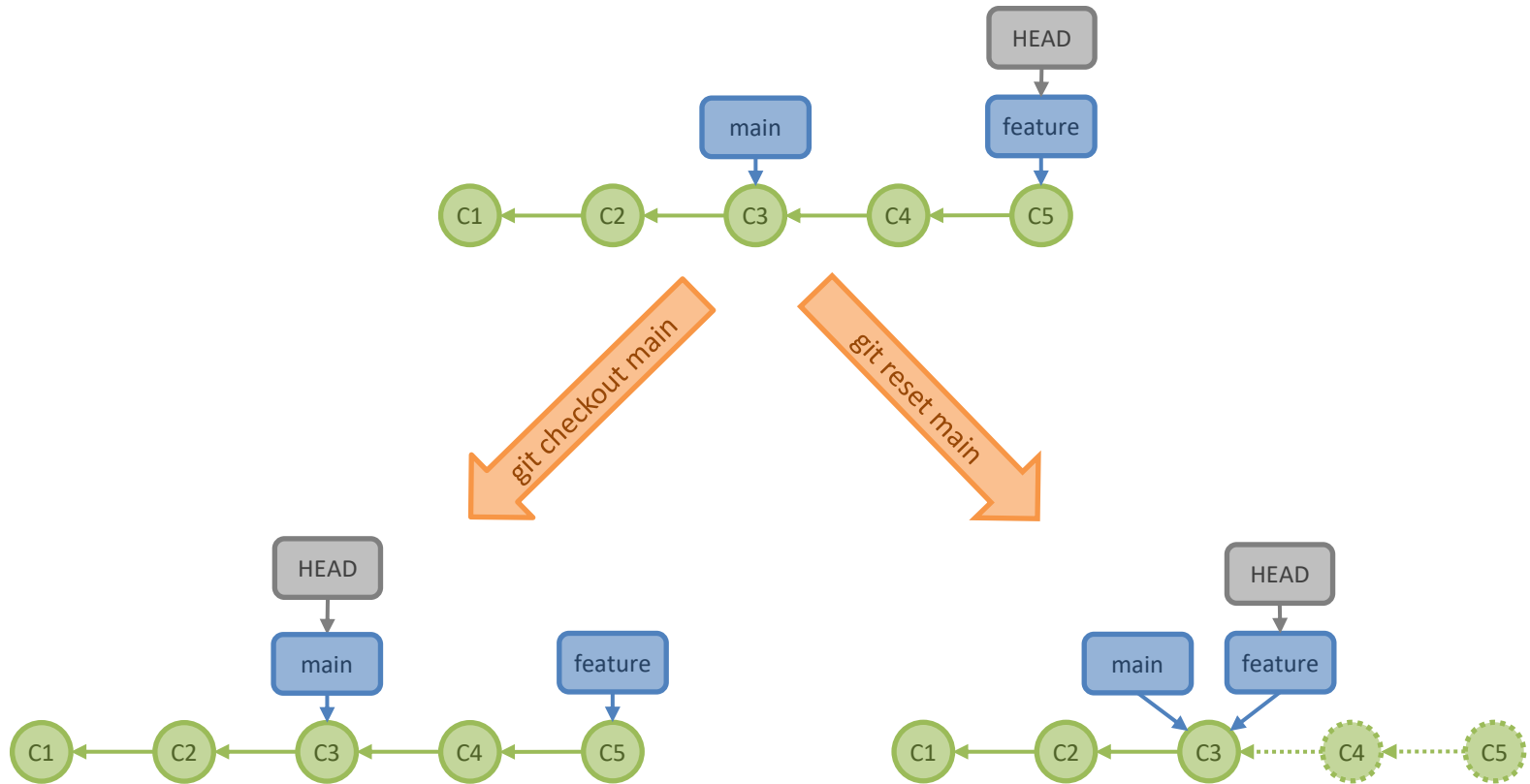
```
git commit --amend
```

```
git reset --soft HEAD^ gefolgt von git commit
```

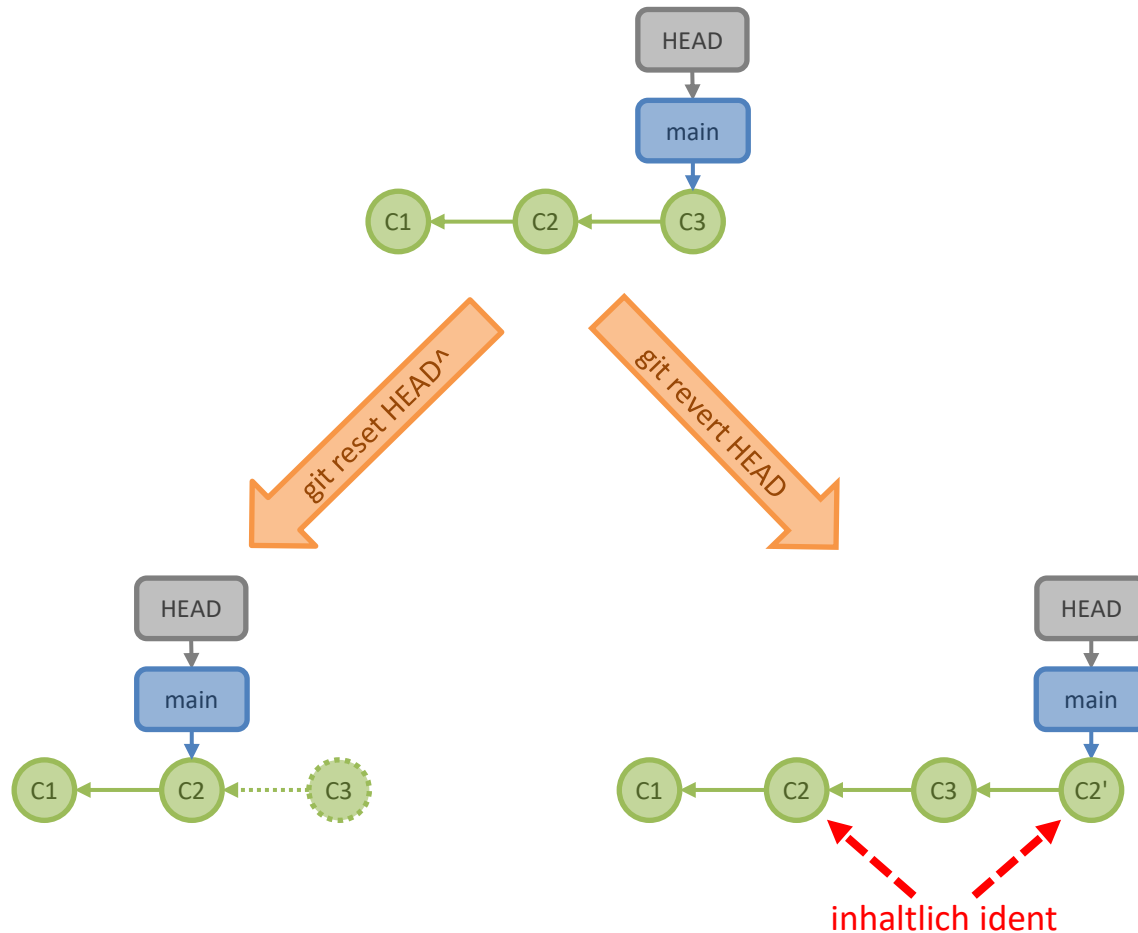
- alten Commit wiederherstellen:

```
git revert HEAD
```

Reset vs. Checkout



Reset vs. Revert



Zurücknehmen von Änderungen

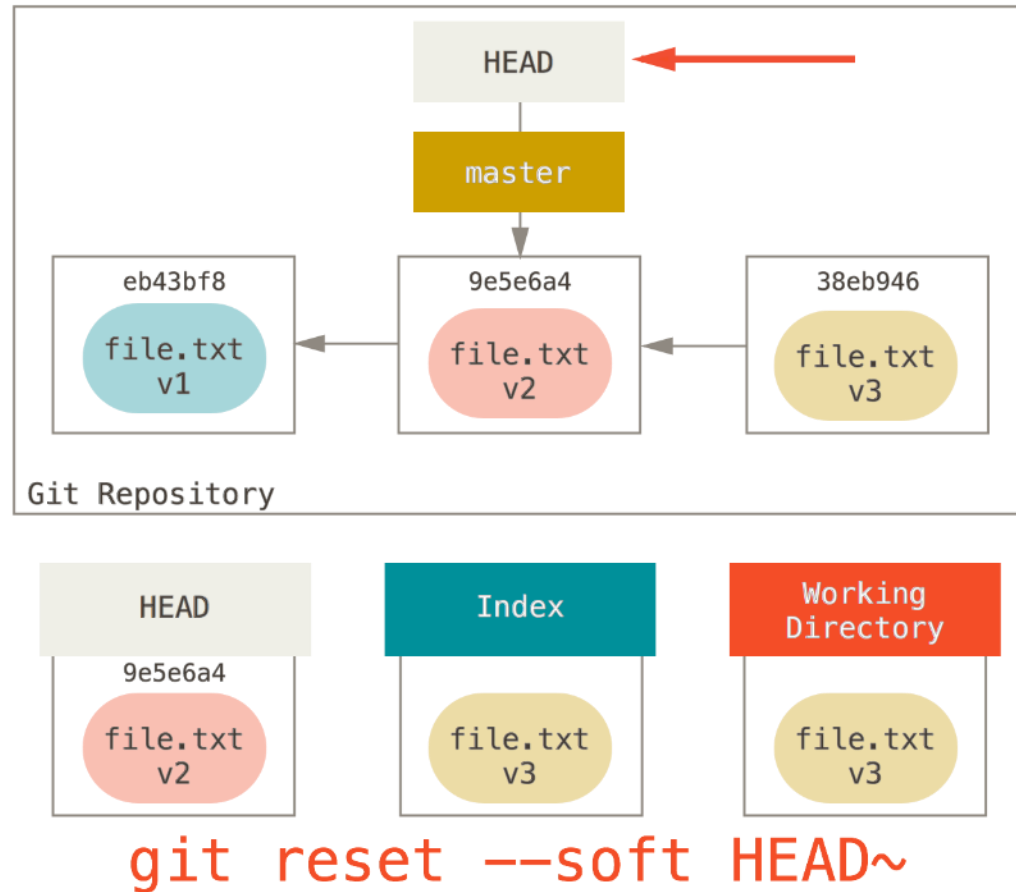
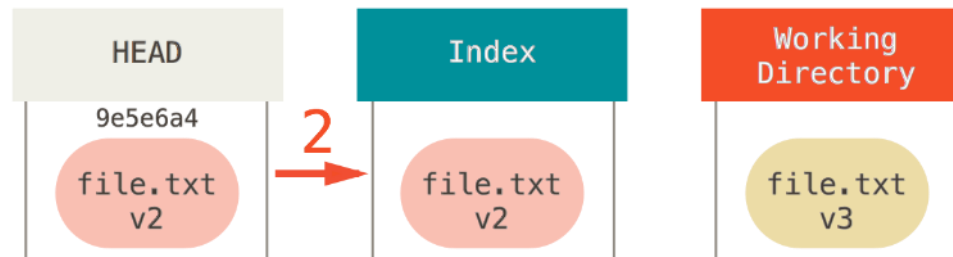
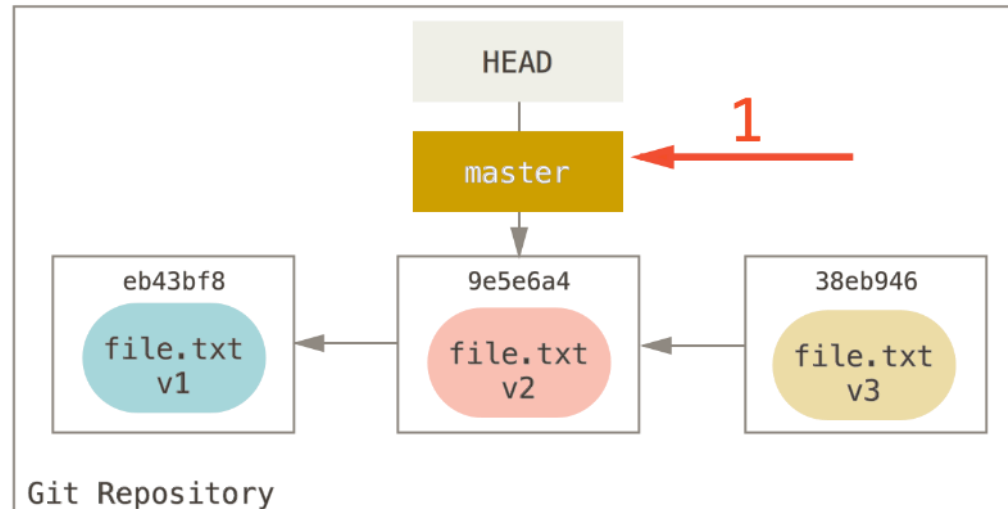


Abbildung aus S. Chacon, B. Straub: Pro Git

Zurücknehmen von Änderungen



`git reset [--mixed] HEAD~`

Abbildung aus S. Chacon, B. Straub: Pro Git

Zurücknehmen von Änderungen

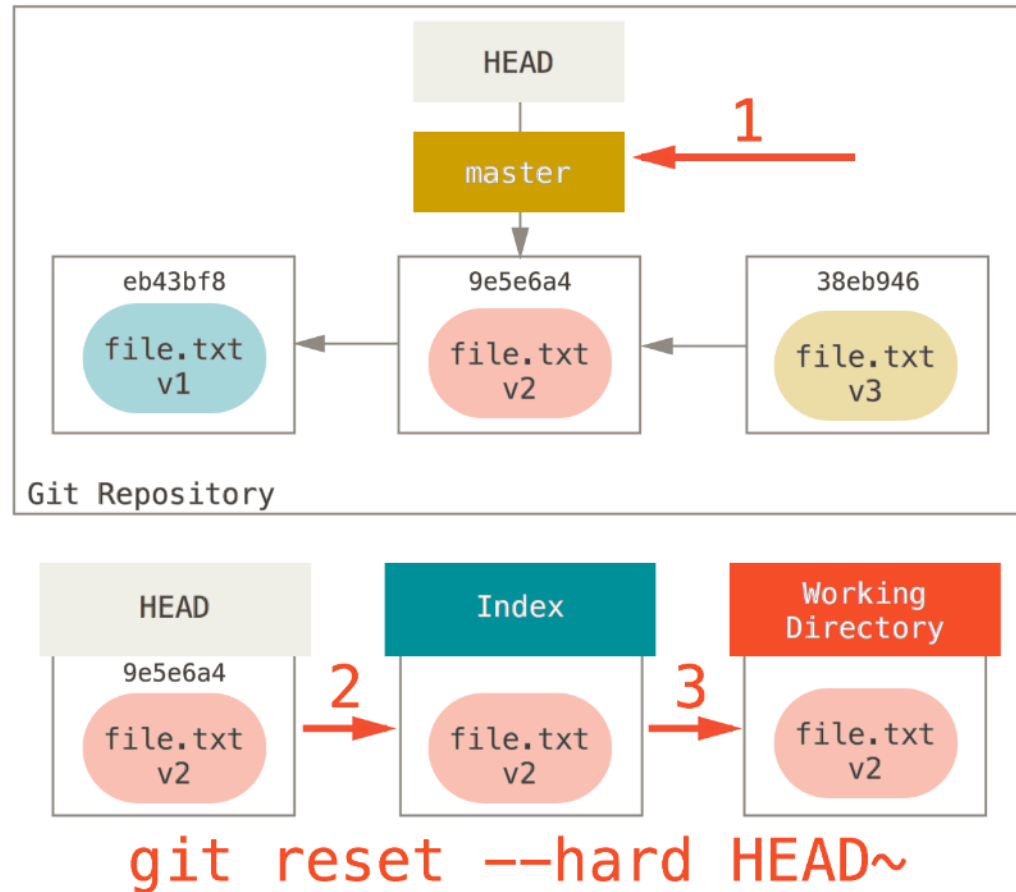
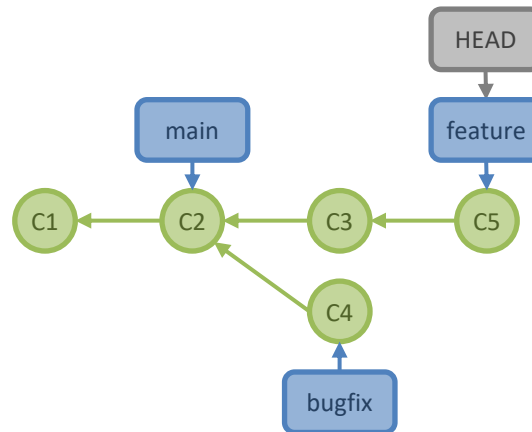


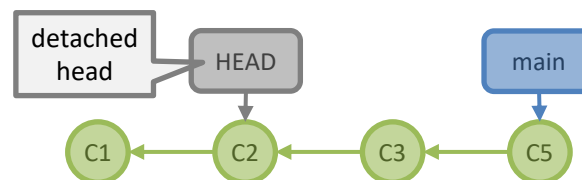
Abbildung aus S. Chacon, B. Straub: Pro Git

Branches

- Branches sind Zeiger auf Commits
- HEAD Zeiger zeigt auf aktuellen Branch, an dem gerade entwickelt wird
- bei einem Commit wird der Branch, auf den HEAD zeigt, automatisch auf den neuen Commit gesetzt

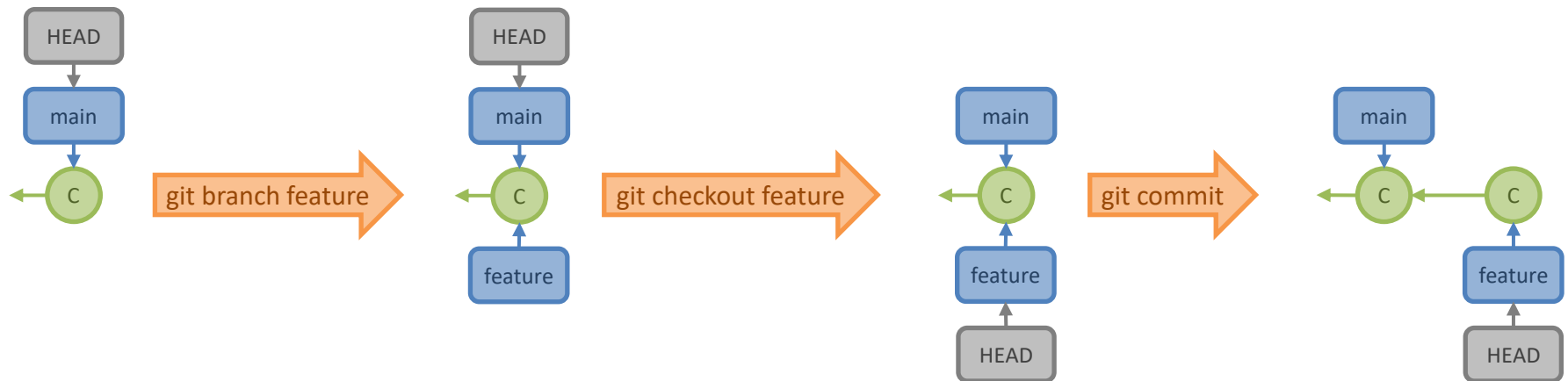


- HEAD kann auch direkt auf einen Commit gesetzt werden ([detached head](#))
- bei einem Commit wird dann HEAD direkt auf den neuen Commit gesetzt
- kann z.B. verwendet werden, um Branches auf alten Commits zu erzeugen



Branches

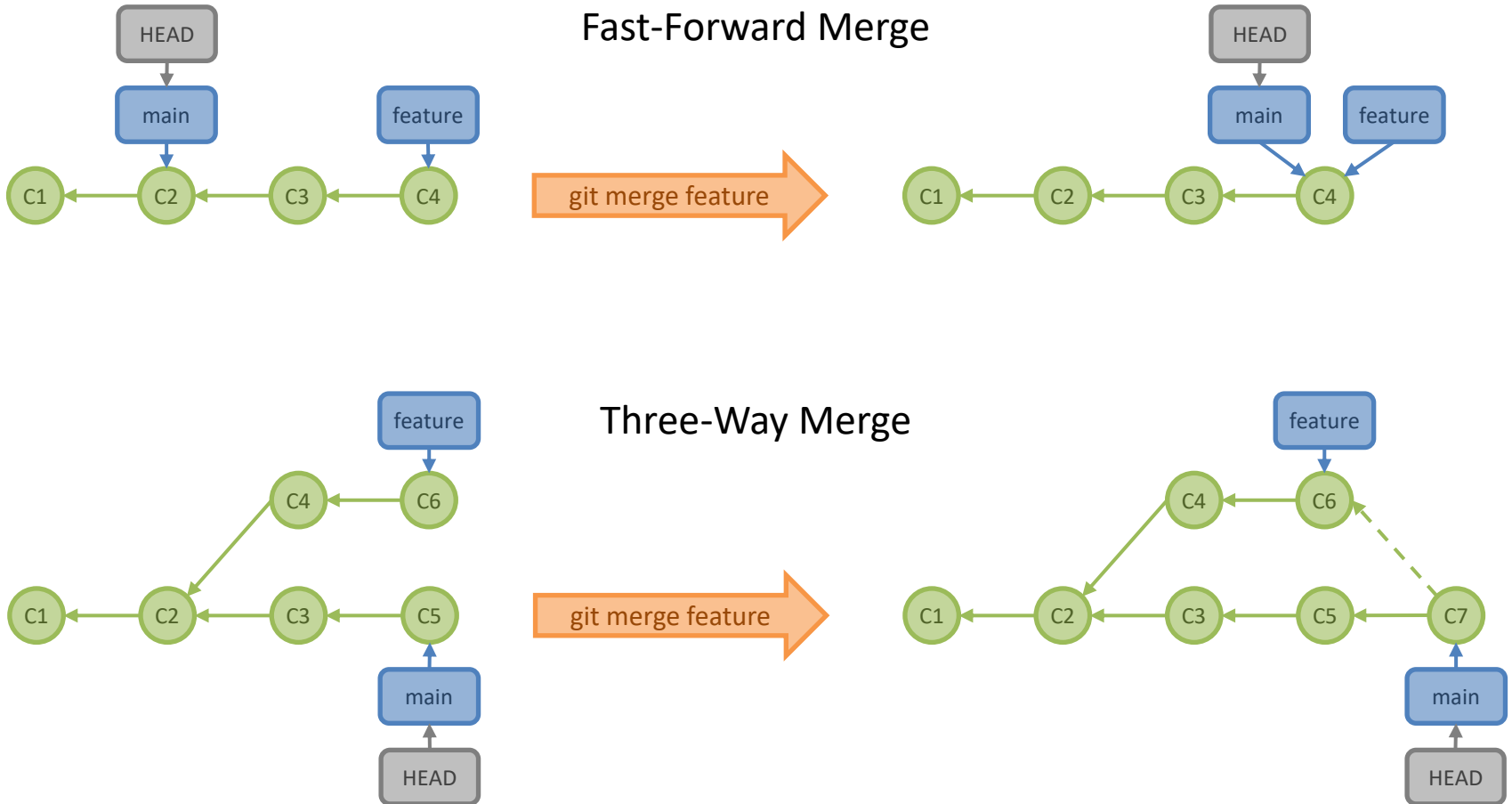
- [git branch](#)
 - legt einen neuen Branch an
- [git checkout](#)
 - setzt HEAD auf einen bestimmten Branch und aktualisiert das Arbeitsverzeichnis entsprechend
 - Option *-b* legt einen neuen Branch an, falls der angegebene Branch nicht existiert
 - kann auch verwendet werden, um eine einzelne Datei zurückzusetzen
- [git switch](#)
 - Alternative zu *git checkout* (seit Version v2.23)
 - gleiche Funktionalität wie *git checkout*, einzelne Dateien können aber nicht zurückgesetzt werden
 - Option *-c* (*--create*) legt einen neuen Branch an, falls der angegebene Branch nicht existiert
 - Option *-d* (*--detach*) setzt HEAD auf einen beliebigen Commit (*detached head*)



Merge

- [git merge](#)
 - integriert einen anderen Branch (Quellbranch) in den aktuellen Branch (Zielbranch)
- 2 mögliche Arten:
 - **Fast-Forward Merge:**
 - Quellbranch ist direkter Nachfolger des Zielbranch
 - Zielbranch muss nur auf den Commit gesetzt werden, auf den Quellbranch zeigt
 - **Three-Way Merge:**
 - Quellbranch und Zielbranch haben sich auseinanderentwickelt
 - Merge-Commit erforderlich, der beide Branches wieder zusammenführt
- bei einem Three-Way Merge entsteht ein neuer Merge-Commit mit zwei Vorgängern
- bei einem Three-Way Merge können Konflikte austreten, die manuell aufgelöst werden müssen
- Option `--no-ff` verhindert Fast-Forward Merges, wodurch immer ein Merge-Commit erstellt wird

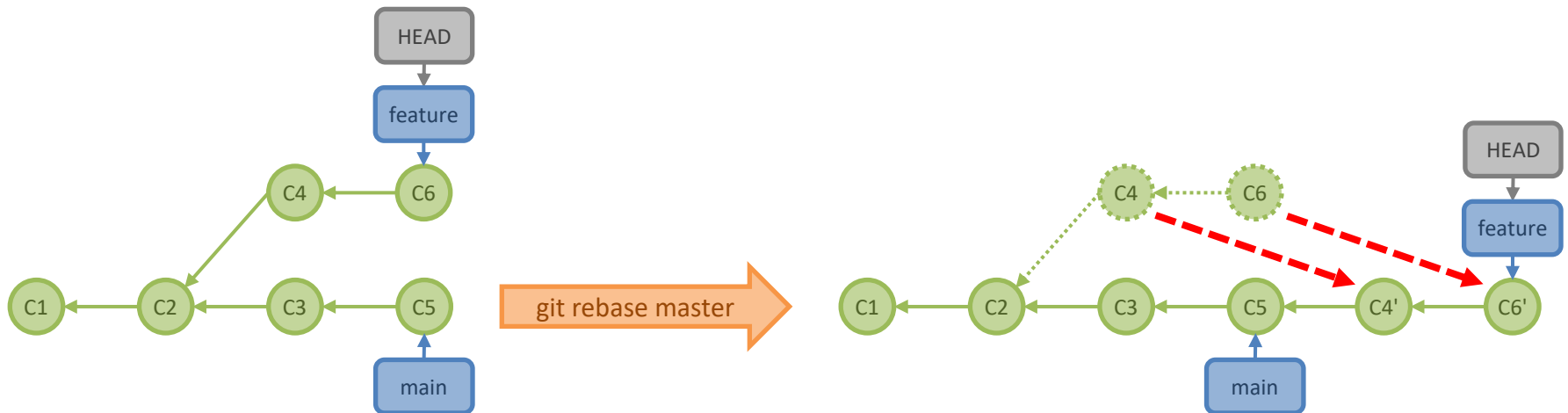
Merge



Rebase

- [git rebase](#) zieht Änderungen eines anderen Branches auf den Zielbranch (Replay)
- Zeiger des Branches liegt damit vor dem Zeiger des Zielbranches
- anschließender Merge auf dem Zielbranch ist damit nur ein Fast-Forward Merge
- Vorteil: hält die Historie des Zielbranch linear und somit einfach und übersichtlich
- Achtung: Rebase sollte immer nur auf lokalen Commits angewendet werden, die noch nicht in ein gemeinsames Repository übertragen wurden, da durch Rebase die Historie verändert wird

Rebase



Cherry Picking

- manchmal müssen einzelne Commits bereits vorzeitig in einen anderen Branch übernommen werden, noch bevor ein Merge/Rebase des gesamten Branch durchgeführt werden kann
 - z.B. wenn in einem Feature Branch ein dringender Bugfix implementiert wurde, bevor die Entwicklung des Features abgeschlossen ist
- dieses selektive Übernehmen und Anwenden von Commits wird als *cherry picking* bezeichnet
- [git cherry-pick](#)
 - überträgt einen oder mehrere Commits auf den aktuellen HEAD
 - Option `-n` (`--no-commit`) überträgt Änderungen nur in das aktuelle Arbeitsverzeichnis und in die Staging Area, führt aber keinen Commit durch
 - bei einem späteren Rebase werden bereits übernommene Commits automatisch ausgelassen



Stashing

- Wechsel auf einen anderen Branch schlägt fehl, wenn Änderungen im aktuellen Arbeitsverzeichnis nicht kompatibel sind
- falls die Änderungen im Arbeitsverzeichnis noch nicht mit einem Commit gespeichert werden sollen, müssen sie zwischengelagert werden
- [git stash](#)
 - speichert aktuelle Änderungen im Arbeitsverzeichnis und in der Staging Area in einem Stack ab
 - stellt einen sauberen Stand gemäß des aktuellen HEAD wieder her
 - die abgespeicherten Änderungen können später wiederhergestellt werden (auch auf einem anderen Branch)
 - *git stash list* listet gespeicherte Einträge auf
 - *git stash show* zeigt Details zu einem gespeicherten Eintrag
 - *git stash apply* überträgt einen Eintrag auf das aktuelle Arbeitsverzeichnis
 - *git stash drop* entfernt einen Eintrag
 - *git stash clear* löscht alle gespeicherten Einträge

Veränderungen der Historie

- Git erlaubt es, nachträglich Veränderungen an der Historie vorzunehmen (vgl. z.B. *git reset*, *git commit --amend*)
 - Bereinigung und Vereinfachung der Historie ist somit auch post factum noch möglich
- *git rebase* mit der Option *-i* (*--interactive*) ermöglicht ein Rebase, bei dem interaktiv festgelegt werden kann, was genau mit den einzelnen Commits geschehen soll
 - Optionen für jeden Commit sind:
 - *pick*: Commit wird übernommen
 - *reword*: Commit wird übernommen, aber die Commit Message wird verändert
 - *squash*: Commit wird mit dem vorherigen Commit verschmolzen
 - *drop*: Commit wird ausgelassen
- **ACHTUNG: eine Änderung der Historie sollte nur dann gemacht werden, wenn noch nicht gepusht wurde**

Reflog als Sicherheitsnetz

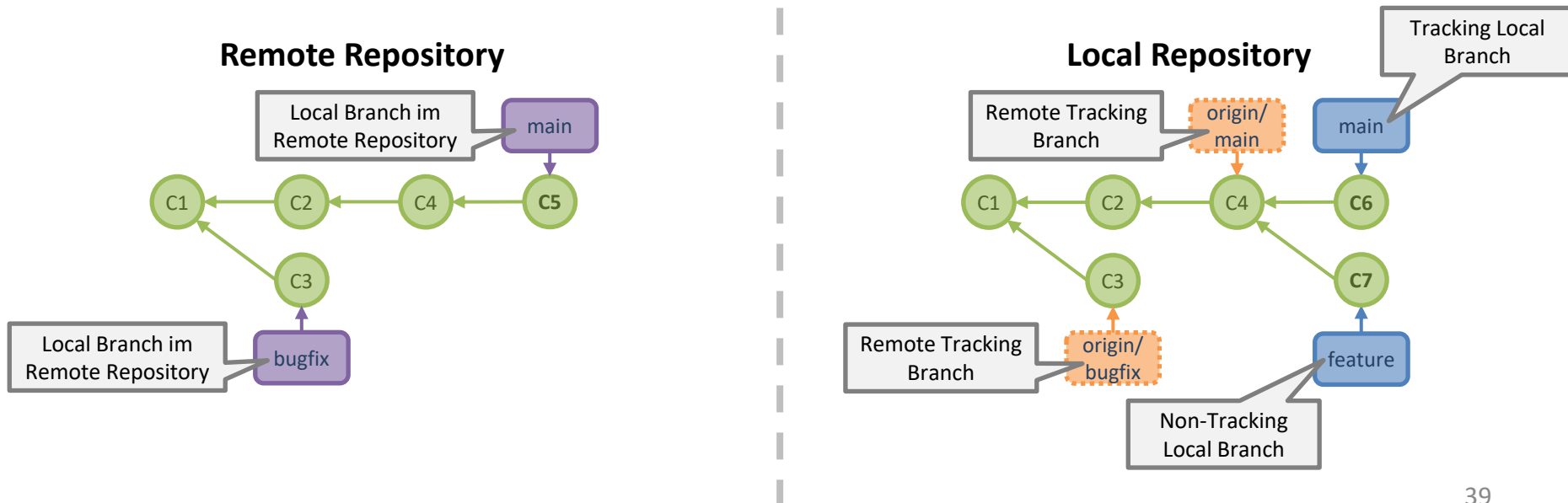
- Git protokolliert im Reflog alle Aktionen, die am HEAD durchgeführt werden
- kein Commit wird tatsächlich verändert oder geht verloren
 - z.B. *git commit --amend* verändert nicht den letzten Commit sondern erzeugt einen neuen Commit, der alte Commit kann im Reflog immer noch gefunden werden
- mit Hilfe des Reflog kann jede Aktion auch wieder rückgängig gemacht werden
- [git reflog](#) zeigt die Einträge im Reflog an
 - mit HEAD@{1}, HEAD@{2}, etc. kann auf die alten HEADs zurückgegriffen werden

Austausch mit einem entfernten Repository

- [git clone](#) erstellt eine lokale Kopie eines entfernten Repositories
- entferntes Repository wird meist als *origin* bezeichnet
- lokales Repository muss regelmäßig mit dem entfernten Repository abgeglichen werden
- Branches im lokalen und im entfernten Repository können sich unterschiedlich entwickelt haben und müssen wieder integriert werden
- ein lokales Repository kann mit mehreren entfernten Repositories interagieren
- [git remote](#) dient zur Verwaltung der entfernten Repositories
 - *git remote -v* listet alle entfernten Repositories auf
 - *git remote show* zeigt Details zu einem entfernten Repository an
 - *git remote add* fügt ein entferntes Repository hinzu
 - *git remote remove* entfernt ein entferntes Repository

Remote Tracking Branches

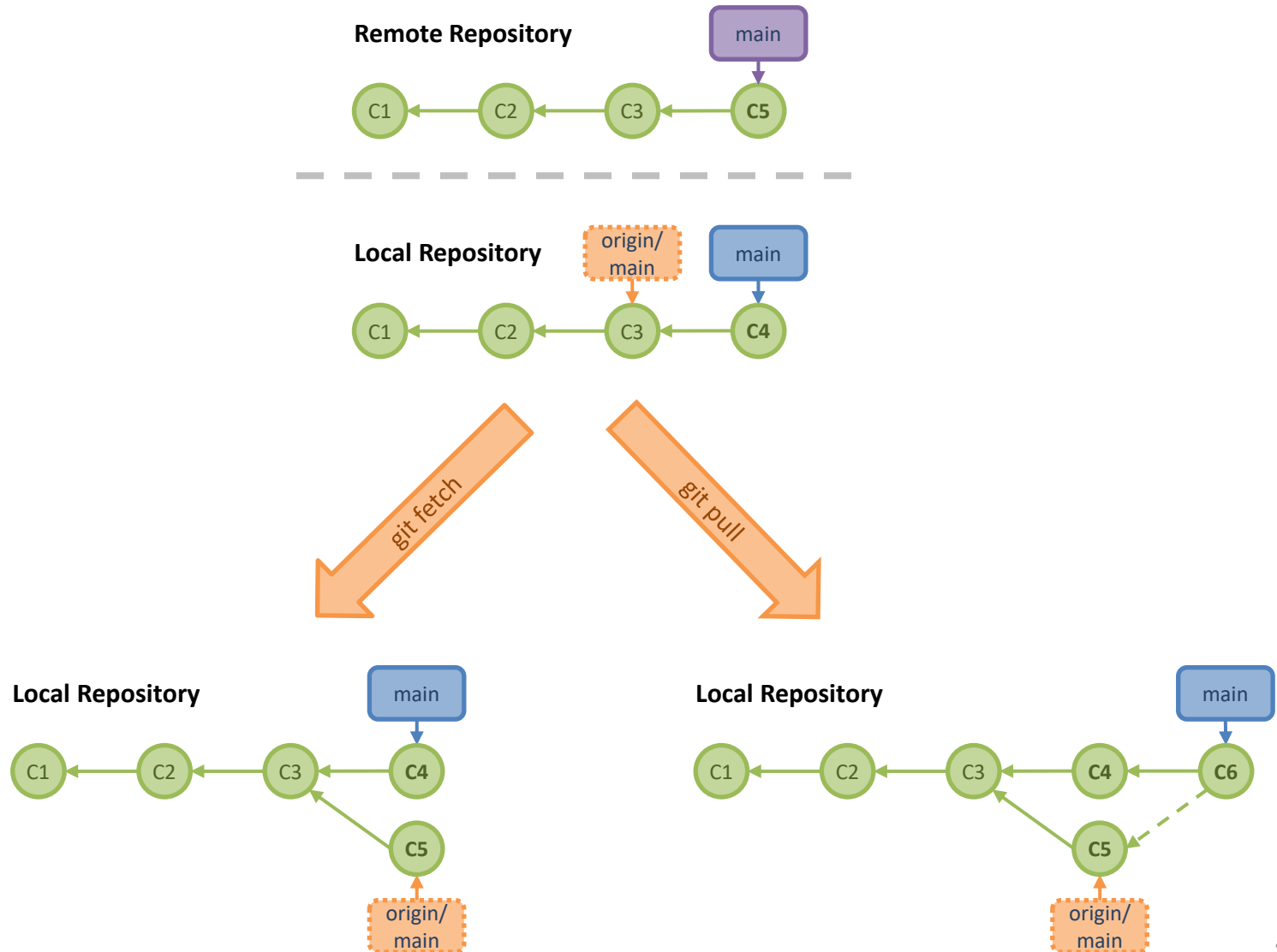
- Branches im lokalen und im entfernten Repository müssen abgeglichen werden
- Remote Tracking Branches verbinden einen lokalen Branch mit einem Branch im entfernten Repository
 - z.B. *origin/main* entspricht dem Branch *main* im entfernten Repository
- Remote Tracking Branches repräsentieren den Stand eines Branches im entfernten Repository zum Zeitpunkt des letzten Abgleichs
- Remote Tracking Branches werden beim Abgleich automatisch aktualisiert
- Commits können nicht direkt auf Remote Tracking Branches geschrieben werden



Austausch mit einem entfernten Repository

- [git fetch](#) holt Änderungen von einem entfernten Repository
 - alle Remote Tracking Branches werden aktualisiert
 - Änderungen werden aber nicht automatisch mit lokalem Branch zusammengeführt
 - Integration muss anschließend manuell mit einem Merge gemacht werden
- [git pull](#) holt Änderungen von einem entfernten Repository wie *git fetch* und integriert diese sofort mit dem lokalen Branch
- [git push](#) überträgt lokale Änderungen in ein entferntes Repository
 - push ist nur erlaubt, wenn lokale Änderungen direkte Nachfolger des Branch im Remote Repository sind
 - Option *-u* (*--set-upstream*) erzeugt einen Branch im entfernten Repository
 - Option *-d* (*--delete*) löscht einen Branch im entfernten Repository
 - Option *-f* (*--force*) erzwingt einen Push, selbst wenn lokale Änderungen keine direkten Nachfolger des Branch im Remote Repository sind
 - **ACHTUNG: force push ist sehr problematisch, da dadurch Commits ausgehängt werden können, die von andere Personen noch verwendet werden**

Fetch vs. Pull



SPM3

Moderne Entwicklungsprozesse und Werkzeuge

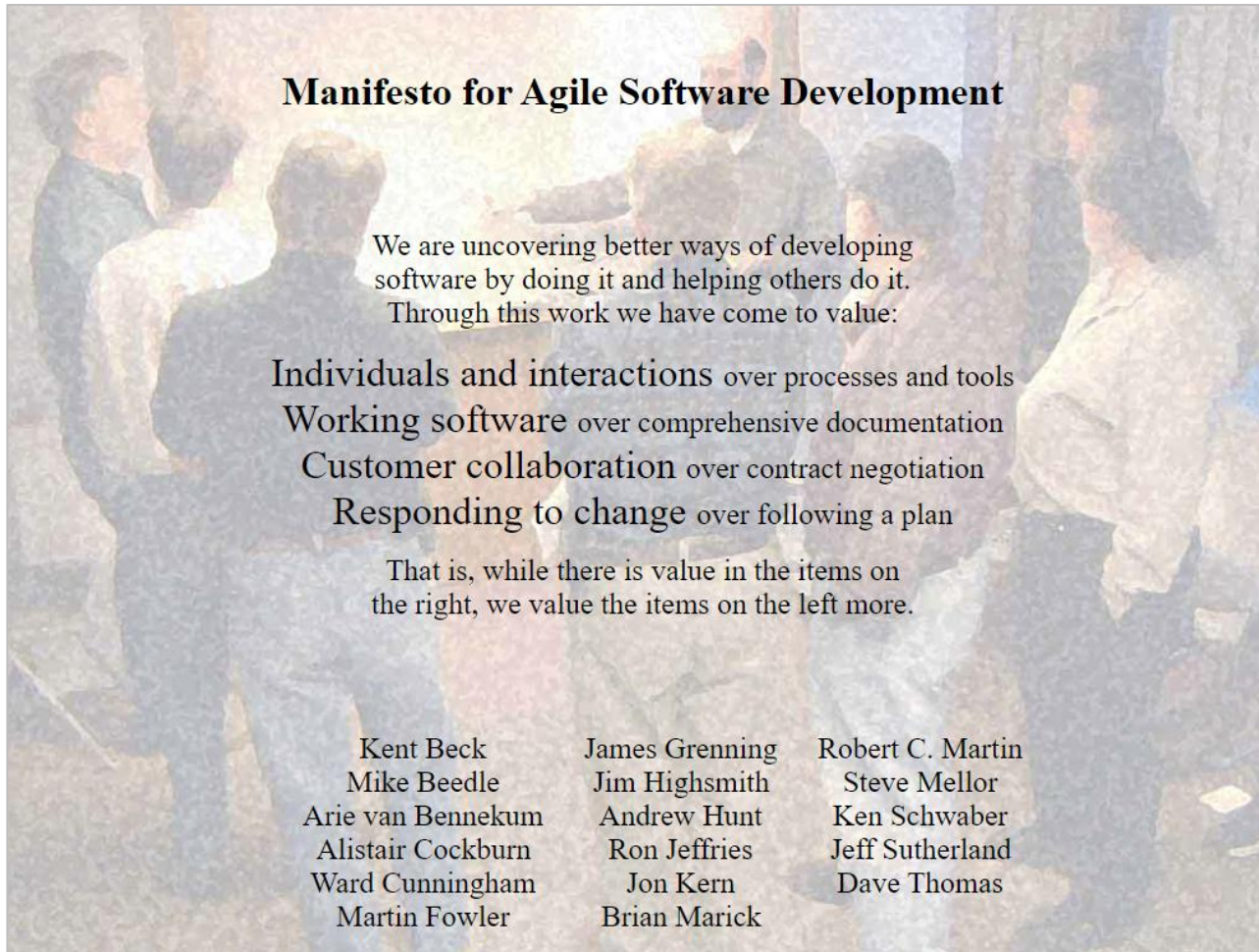
Task-Management

Stefan Wagner, Johannes Karder, Philipp Fleck

Motivation für agile Methoden

- schwergewichtige Prozessmodelle für Softwareentwicklung in den 90er Jahren
 - überbetonte Prozesse, d.h. Einhaltung des Prozesses ist wichtiger als eigentliches Ergebnis
 - immer umfangreichere Prozessdokumentation
 - immer größerer Overhead für die Abwicklung des Prozesses
 - Prozesse sind starr und wenig flexibel

Agile Manifesto im Jahr 2001



<https://agilemanifesto.org>

Agile Softwareentwicklung

- Prinzipien
 - frühe und kontinuierliche Auslieferung wertvoller Software zur Zufriedenstellung des Kunden
 - Anforderungsänderungen willkommen heißen
 - regelmäßige Auslieferung funktionierender Software
 - tägliche Zusammenarbeit von Fachexperten und Entwicklern
 - Vertrauen in motivierte Individuen
 - Informationsvermittlung im direkten Gespräch
 - funktionierende Software als wichtigstes Fortschrittsmaß
 - nachhaltige Entwicklung bei gleichmäßigem Tempo
 - ständiger Fokus auf technische Exzellenz und gutes Design
 - Einfachheit und Maximierung nicht getaner Arbeit
 - selbstorganisierende Teams
 - regelmäßige Reflektion

Agile Softwareentwicklung

- Umsetzung durch iterativ-inkrementelle Entwicklung
 - Entwicklung in kurzen Iterationen
 - Ergebnis einer Iteration ist funktionierende Software (Produktinkrement)
 - Feedback zum Produktinkrement nach jeder Iteration
 - zwischen Iterationen können Ziele und Prioritäten geändert werden
 - immer nur die nächste Iteration wird im Detail geplant
 - Projektfortschritt wird regelmäßig ermittelt und geprüft
 - regelmäßige Reflexion und Prozessverbesserung

Agile Softwareentwicklung

Vorteile

- kurze Releasezyklen
- Produktinkremente mit überprüfbarer Qualität
- stärkere Kundeneinbindung
- regelmäßige und intensive Kommunikation im Team
- weniger Overengineering
- regelmäßige Reviews
- testgesteuerte Entwicklung

Nachteile

- höher qualifiziertes Personal erforderlich
- regelmäßige Anforderungsänderungen
- schlechte Skalierbarkeit
- inkrementelles Design, keine Design-Reviews
- starker Fokus auf Code und nicht auf Design

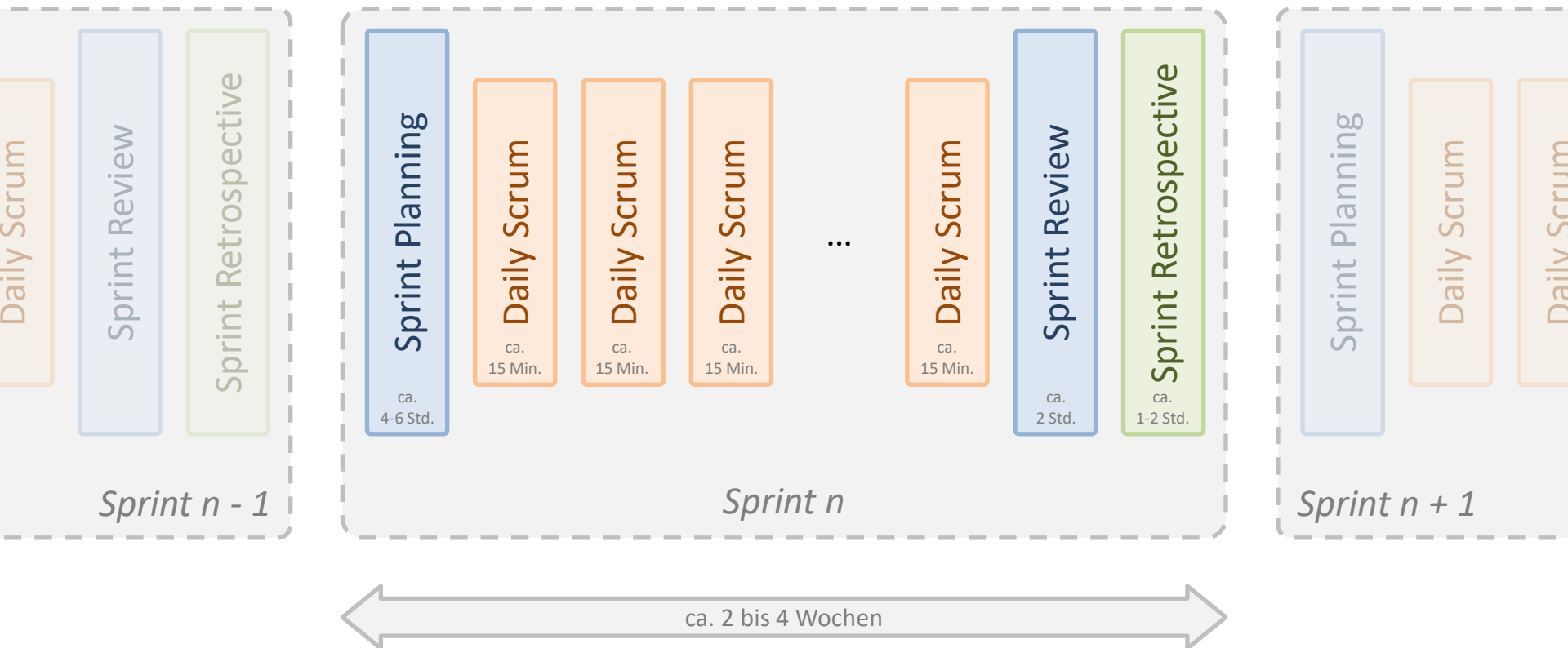
Agile Softwareentwicklung

- zahlreiche Vorgehensmodelle:
 - Scrum
 - Crystal
 - ASD (Adaptive Software Development)
 - XP (Extreme Programming)
 - FDD (Feature-driven Development)
 - Lean Development
 - DSDM (Dynamic Software Development Method)
 - ...
- in Studienprojekten (und auch in vielen Unternehmen) wird meist Scrum oder zumindest ein Scrum-ähnlicher Prozess eingesetzt

Scrum Begriffe

- **Story**
 - Beschreibung einer Menge von Anforderungen aus Sicht eines Anwenders, um einen bestimmten Nutzen zu erzielen
- **Product Backlog**
 - Board mit allen Stories
- **Sprint**
 - eine Iteration, in der Stories erledigt werden
- **Task**
 - Aufgabe, die zu einer Story gehört
- **Sprint Backlog**
 - Board mit allen Tasks des aktuellen Sprints
- **Burndown Chart**
 - visualisiert den geschätzten restlichen Aufwand und die verbleibende Zeit
- **Product Increment**
 - Ergebnis eines Sprints in Form einer konkreten Weiterentwicklung des Produkts
- **Impediment**
 - Aspekt, der den Projektfortschritt behindert

Scrum Ablauf



Scrum in Studienprojekten

- Rollen:
 - **Auftraggeber (AG)**
 - externe Person vom Auftraggeber
 - Ansprechperson für Product Owner
 - sollte am Anfang und Ende eines jeden Sprints möglichst persönlich anwesend sein
 - priorisiert Stories und nimmt Sprints ab
 - **Product Owner (PO)**
 - Produktverantwortung und Schnittstelle zum AG
 - meist 2 Personen zu jeweils 50%
 - erstellt, verfeinert und schätzt Stories
 - definiert Akzeptanzkriterien und nimmt Stories und Tasks ab
 - **Scrum Master (SM)**
 - Prozessverantwortung
 - 1 Person und 1 Stellvertreter
 - moderiert Meetings
 - räumt Hindernisse aus dem Weg
 - **Development Team (Team)**
 - erstellt und schätzt Tasks
 - führt Sprints durch und entwickelt Produkt
 - **Projektcoach (PC)**
 - LVA-Leiter, der die Projektgruppe begleitet
 - hilft bei Problemen und gibt fachliche Inputs

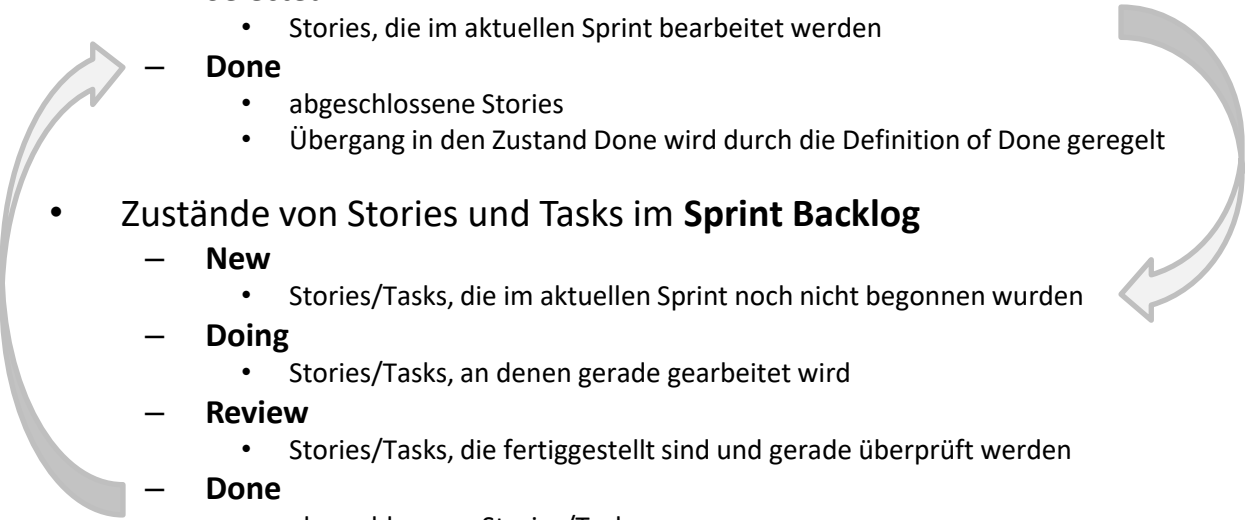
Scrum in Studienprojekten

- Vorbedingung:
 - ausreichend viele Stories sind im Product Backlog in ausreichend detaillierter Qualität beschrieben
- Ablauf eines Sprints:
 - **AG Meeting:**
 - Sprint Review des vergangenen Sprints, Abnahme durch PO/AG
 - ev. nicht abgeschlossene Stories des letzten Sprints werden wieder ins Product Backlog aufgenommen
 - Stories werden vom PO vorgestellt und diskutiert
 - Sprintziel wird festgelegt
 - Stories werden vom AG/PO priorisiert und vom Team anhand der Kapazität für den nächsten Sprint ausgewählt
 - wesentliche Regel: AG/PO bestimmt was gemacht werden soll, Team bestimmt wieviel davon gemacht werden kann
 - **Sprint Retrospektive:**
 - Erfassen der Sprint Retrospective Items, Ableiten von konkreten Verbesserungsmaßnahmen, Erstellen der Focus List
 - **Sprint Planning:**
 - SM erstellt Sprint Backlog für den aktuellen Sprint
 - Team bricht Stories in Tasks herunter, die beschrieben und geschätzt werden
 - Kapazität des Sprints wird nochmals geprüft und gegebenenfalls werden die übernommenen Stories nochmals angepasst
 - **Sprint Durchführung:**
 - gemeinsame Abstimmung am Beginn jedes Termins (Daily Scrum, ca. 15 Minuten), was wurde gemacht?, was wird heute gemacht? gibt es Hindernisse?
 - Aktualisierung des Sprint Backlogs, der Aufwandsschätzungen und des Burndown Charts
 - **Product Backlog Refinement:**
 - Stories für die nächsten Sprints werden von POs erstellt, verfeinert und grob geschätzt
 - erfolgt parallel zur Durchführung des aktuellen Sprints

Boards

- **Product Backlog**
 - Erfassung und Verwaltung von Stories (sprintübergreifend)
 - Stories werden von PO gesammelt, schrittweise verfeinert, grob geschätzt und priorisiert
 - ausreichend verfeinerte Stories können je nach Priorität für den nächsten Sprint ausgewählt werden
- **Sprint Backlog / Scrum Board**
 - eigenes Board für jeden Sprint
 - Erfassung und Verwaltung von Tasks eines Sprints
 - im Sprint Planning werden Stories für den nächsten Sprint ausgewählt und in das Scrum Board des Sprints übernommen
 - Stories werden anschließend vom Team in Tasks heruntergebrochen und geschätzt
 - Auswahl der Stories wird gegebenenfalls noch korrigiert, falls Sprint zu viele oder zu wenige Tasks enthält
 - Tasks durchlaufen während des Sprints verschiedene Zustände (minimal z.B. New, Doing, Review, Done)
- **Focus List**
 - positive und negative Punkte, die im aktuellen Sprint besonders im Auge behalten werden sollen
 - Ergebnis aus der Sprint Retrospektive
 - sollte beim Daily Scrum neben dem Scrum Board dargestellt werden und immer gut sichtbar sein
- **Impediment Board**
 - Erfassung und Verwaltung von Hindernissen
 - SM verfolgt die Auflösung der Hindernisse
- **Retrospective Board**
 - Erfassung positiver und negativer Faktoren des letzten Sprints
 - Ableitung von Verbesserungen für zukünftige Sprints
 - für die Abwicklung mit einem Werkzeug nicht so gut geeignet, besser mit Post-its und Whiteboard bzw. Flipchart

Lebenszyklus von Stories und Tasks

- Zustände der Stories im **Product Backlog**
 - **New**
 - neue Ideen/Stories, die noch unstrukturiert und nicht genauer beschrieben sind
 - **Refinement**
 - Stories, die vom PO gerade verfeinert werden
 - **Ready**
 - ausreichend verfeinerte Stories, die in einem der kommenden Sprints bearbeitet werden können
 - Übergang in den Zustand Ready wird durch die Definition of Ready geregelt
 - **Selected**
 - Stories, die im aktuellen Sprint bearbeitet werden
 - **Done**
 - abgeschlossene Stories
 - Übergang in den Zustand Done wird durch die Definition of Done geregelt
 - Zustände von Stories und Tasks im **Sprint Backlog**
 - **New**
 - Stories/Tasks, die im aktuellen Sprint noch nicht begonnen wurden
 - **Doing**
 - Stories/Tasks, an denen gerade gearbeitet wird
 - **Review**
 - Stories/Tasks, die fertiggestellt sind und gerade überprüft werden
 - **Done**
 - abgeschlossene Stories/Tasks
- 

Zustandsübergänge von Stories

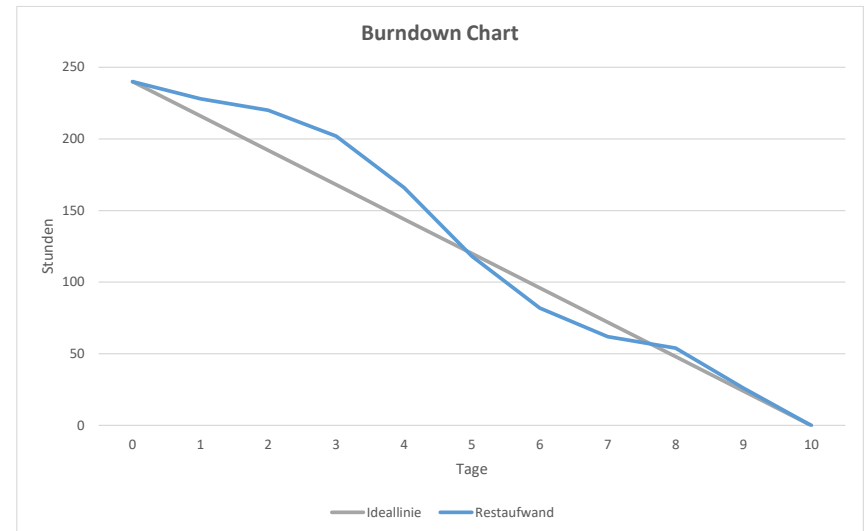
- **Definition of Ready (DoR)**
 - legt fest, was eine Story erfüllen muss, um ev. in den nächsten Sprint aufgenommen werden zu können
 - verständliche Beschreibung
 - Akzeptanzkriterien, Tests, konkrete Ergebnisse (Artefakte)
 - Aufwandsschätzung
 - adäquate Größe
 - Angabe von Abhängigkeiten
- **Definition of Done (DoD)**
 - legt fest, was eine Story bzw. ein Task erfüllen muss, um abgenommen werden zu können
 - Richtlinien eingehalten
 - Akzeptanzkriterien erfüllt
 - Unit Tests vorhanden und erfüllt
 - Peer Review durchgeführt
 - Code integriert
 - Artefakt erstellt

Tipps zum Schätzen

- Größe von Stories und Tasks:
 - eine Story muss in max. einem Sprint erledigt werden können
 - ein Task sollte in max. einem Termin (4 Einheiten) erledigt werden können
 - Stories und Tasks müssen entsprechend feingranular (d.h. konkret) heruntergebrochen werden
- Zeiteinheiten für Aufwandsschätzungen:
 - sollten nicht zu feingranular sein (Scheingenauigkeit)
 - je unsicherer die Schätzung, desto gröber die Zeiteinheit
 - Einheiten besser geeignet als Stunden
 - Schätzen kann auch in abstrakten Einheiten erfolgen (Story Points), eignet sich aber für Studienprojekte meist nicht so gut
- Kapazität eines Sprints:
 - Overhead muss berücksichtigt werden (PO-Tätigkeit, Daily Scrums, Planning, Vorbereitung der AG-Meetings)
 - Puffer muss eingeplant werden (üblich zwischen 10% und 20%)
 - Beispiel:
 - Projektgruppe mit 8 Personen, 4 Einheiten pro Termin
 - somit insgesamt **32 Einheiten pro Termin**
 - 5 produktive Termine für einen Sprint (AG Meeting und Sprint Planning zählen nicht dazu)
 - **Sprintkapazität (brutto):** $32 * 5 = 160 \text{ Einheiten}$
 - PO-Tätigkeit: 1 Person ($2 * 50\%$) = 20 Einheiten
 - Daily Scrums: 0,33 Einheiten pro Termin für das gesamte Team = ca. 16 Einheiten
 - Vorbereitung AG-Meeting: 2 Einheiten für das gesamte Team = 16 Einheiten
 - **Sprintkapazität (netto):** $160 - 20 - 16 - 16 = 108 \text{ Einheiten} - 10\% \text{ Puffer} = 97 \text{ Einheiten}$ (= 61% von 160)

Burndown Chart

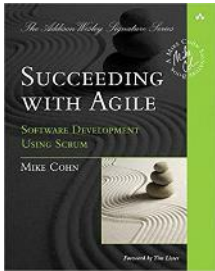
- Visualisierung des Fortschritts und Überwachung des geschätzten Restaufwands
 - x-Achse: Tage bis zum Ende des Sprints
 - y-Achse: Maßeinheit (z.B. Stunden, Story Points, offene Stories/Tasks, etc.)
 - wird im Zuge des Daily Scrum aktualisiert und besprochen
- Burndown Chart sollte möglichst klar und einfach sein
 - Restaufwand liegt auf/unter der Ideallinie
→ Sprint verläuft nach Plan
 - Restaufwand liegt über der Ideallinie
→ Sprint ist in Verzug



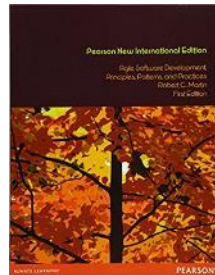
Hauptaufgaben eines Task Management Systems

- Verwaltung des Product Backlog
- Erfassung und Verwaltung von Stories
 - eindeutige ID
 - Beschreibung
 - Aufwandsschätzung
 - Akzeptanzkriterien (Checklisten)
 - ev. Abhängigkeiten
- Verwaltung der Sprint Backlogs
- Erfassung und Verwaltung der Tasks
 - eindeutige ID
 - Beschreibung
 - initiale Aufwandsschätzung
 - aktuelle Schätzung des Restaufwands
- Auswertungen und Reporting
 - Burndown Chart
 - falls möglich, Berücksichtigung dass im Studienprojekt nicht jeden Tag gearbeitet wird

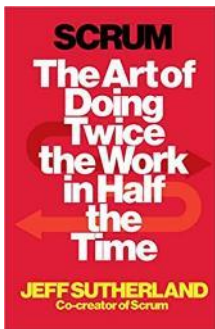
Literatur



M. Cohn:
**Succeeding with Agile:
Software Development Using Scrum**
Addison Wesley



R. Martin:
Agile Software Development, Principles, Patterns, and Practices
Pearson



J. Sutherland:
Scrum: The Art of Doing Twice the Work in Half the Time
Random House Business



H. Koschek:
**Geschichten vom Scrum:
Von Sprints, Retrospektiven und agilen Werten**
dpunkt.verlag