# WSE1

## Werkzeuge im Software Engineering

## 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)

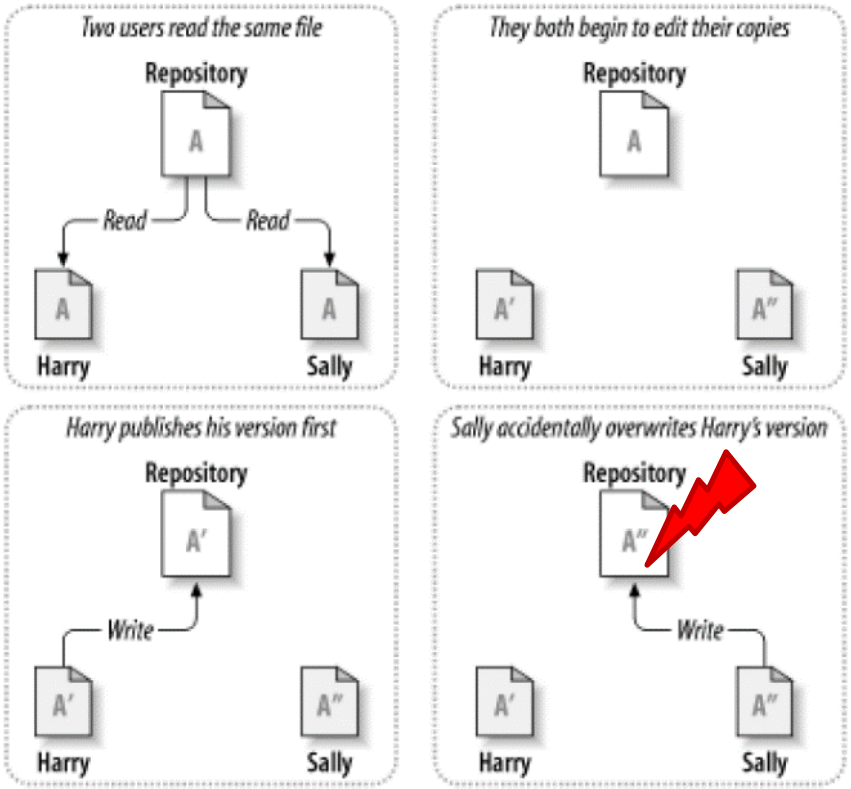
# Hauptaufgaben von VCS

- 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

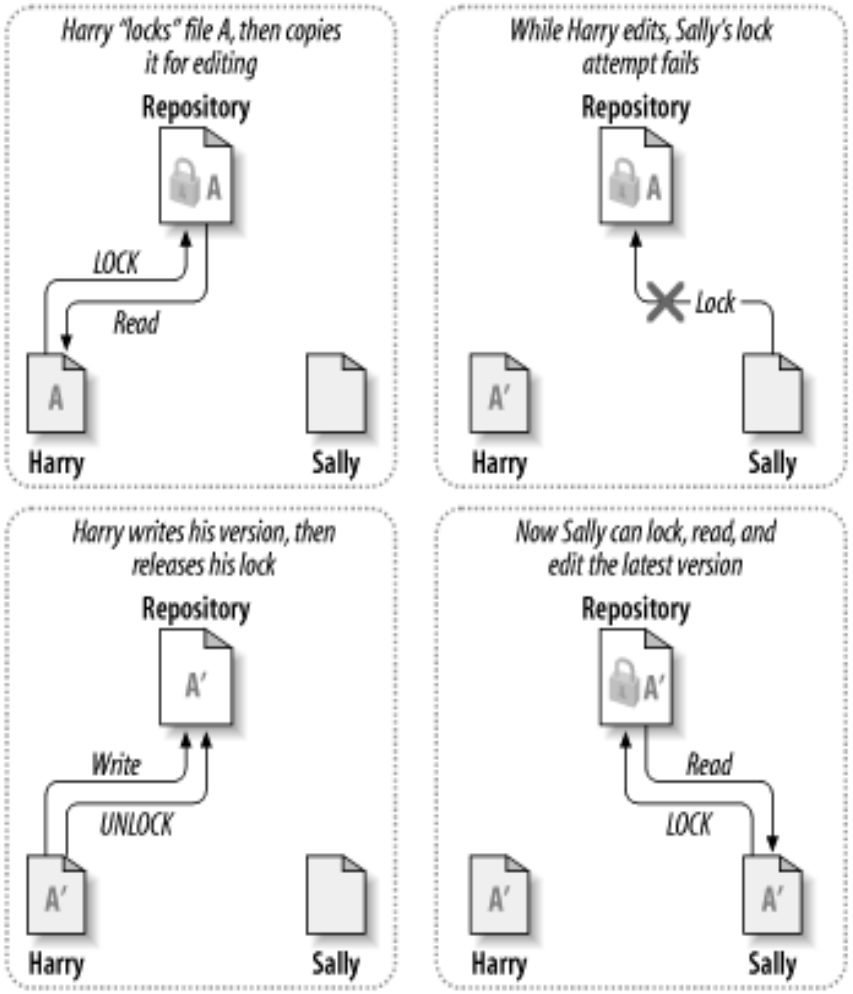
- Archiv (Repository)
  - Speicherung der Dateien in eigenem Dateiformat oder Datenbank
  - nur Speicherung der Änderungen zwischen den Versionen (Platzersparnis)
  - Zugriff auf Dateien nur via VCS
- Arbeitskopie (Working Copy)
  - lokale Kopie einer Version zur Bearbeitung
- Update der Arbeitskopie (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

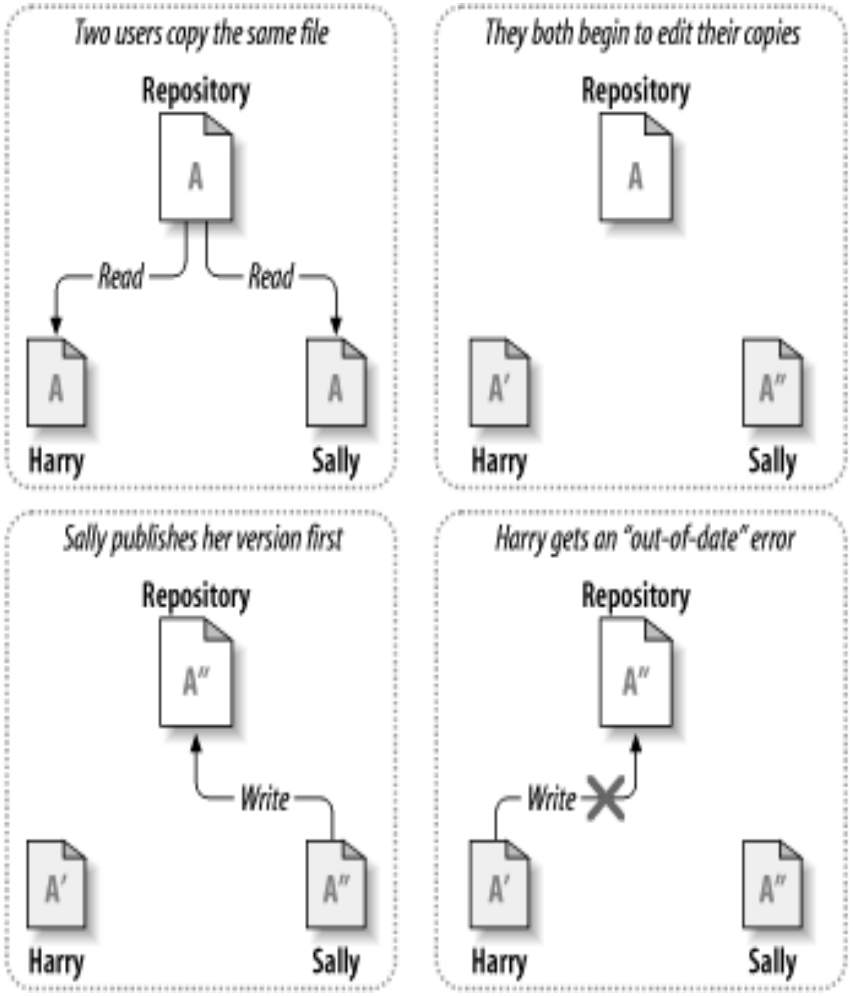
# Strategien – Lock-Modify-Unlock




- Sperren einer Datei vor der Bearbeitung
- Freigabe der Sperre nach der Bearbeitung

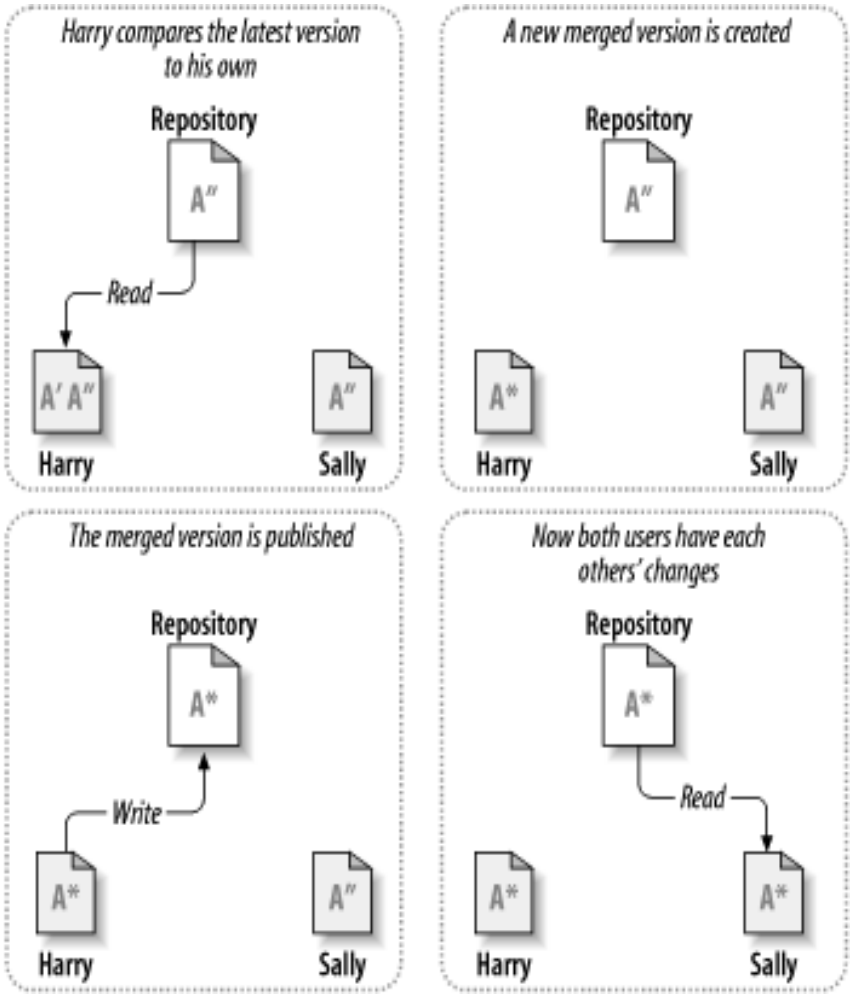
- Probleme:
  - Vergessen von Sperren
  - unnötige Serialisierung des Workflow
  - trügerischer Eindruck von Sicherheit
    - Gefahr von semantischer Inkompatibilität

# 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

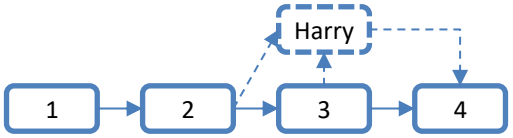



- Behebung von Konfliktsituationen
  - manuell (wenn sich geänderte Bereiche überschneiden)
- Vergleichen der letzten Version mit der Arbeitskopie
- Erstellen einer gemeinsamen Version zur Auflösung des Konflikt von Hand
- Aktualisierung des Repository

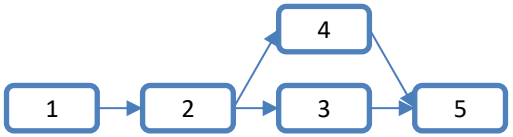
# Entwicklung von VCS

- 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:



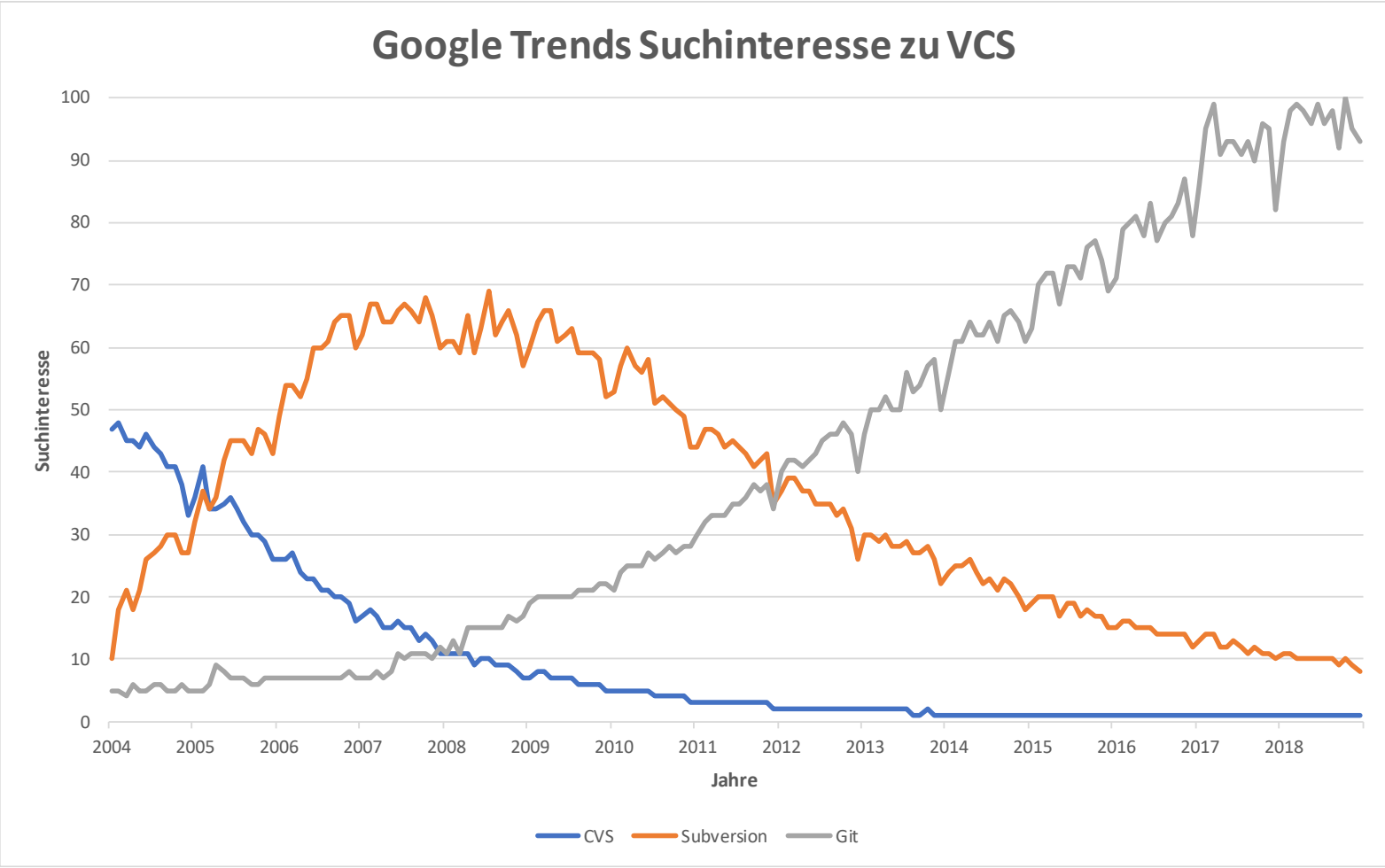

commit before merge:



# Geschichte von Git

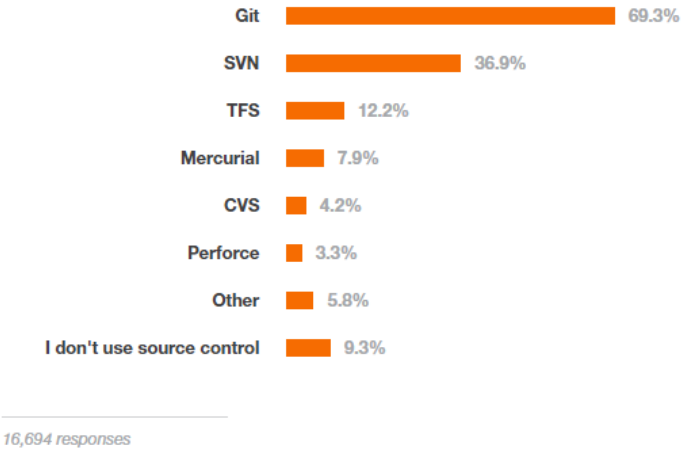

- **Ende 1990er**
  - erste  verteilte VCS entstehen
- **1998**
  - erstes ernstzunehmendes System: BitKeeper (proprietär)
- **2002**
  - BitKeeper wird für die Linux Kernel Entwicklung verwendet ("Linus did not scale")
- **2005**
  - Lizenzänderung von BitKeeper führt zum Bruch mit der Linux Community 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

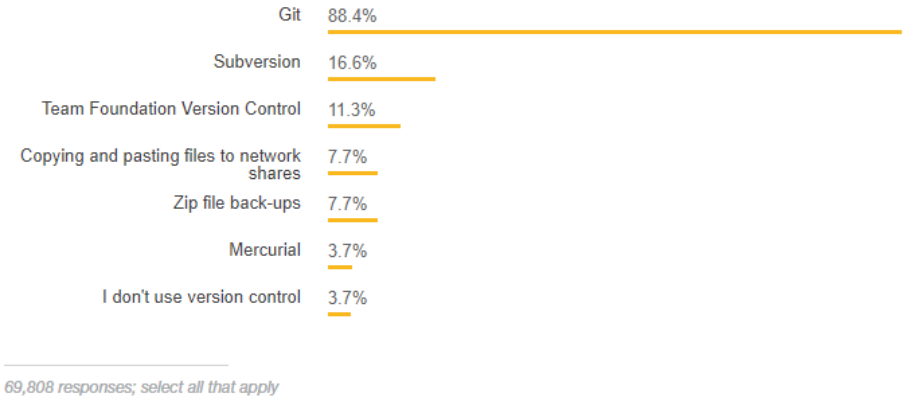


# Verbreitung von VCS

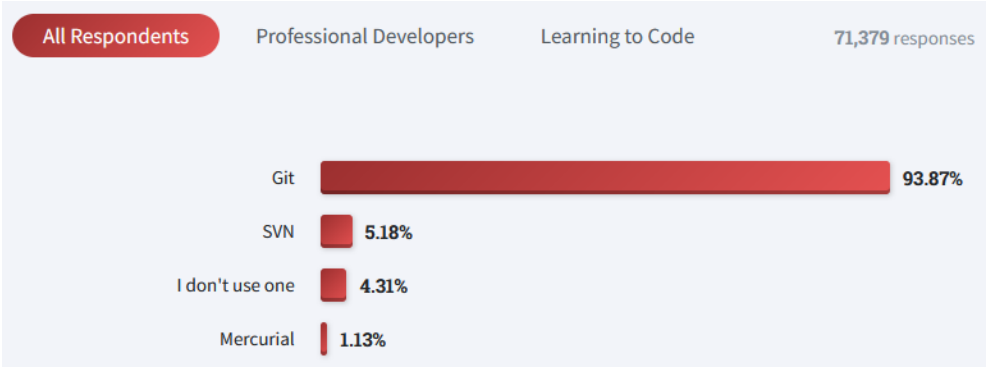
**Stack Overflow Developer Survey 2015:**




**Stack Overflow Developer Survey 2018:**




**Stack Overflow Developer Survey 2022:**

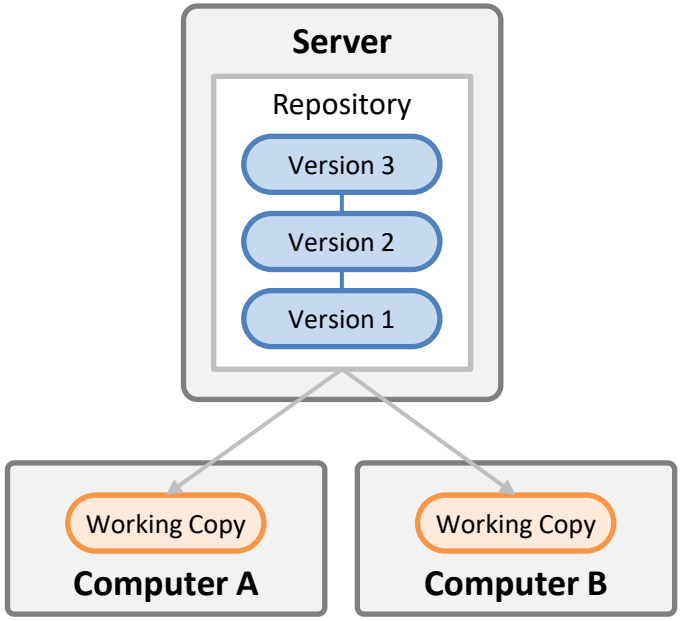


# Motivation für verteilte VCS

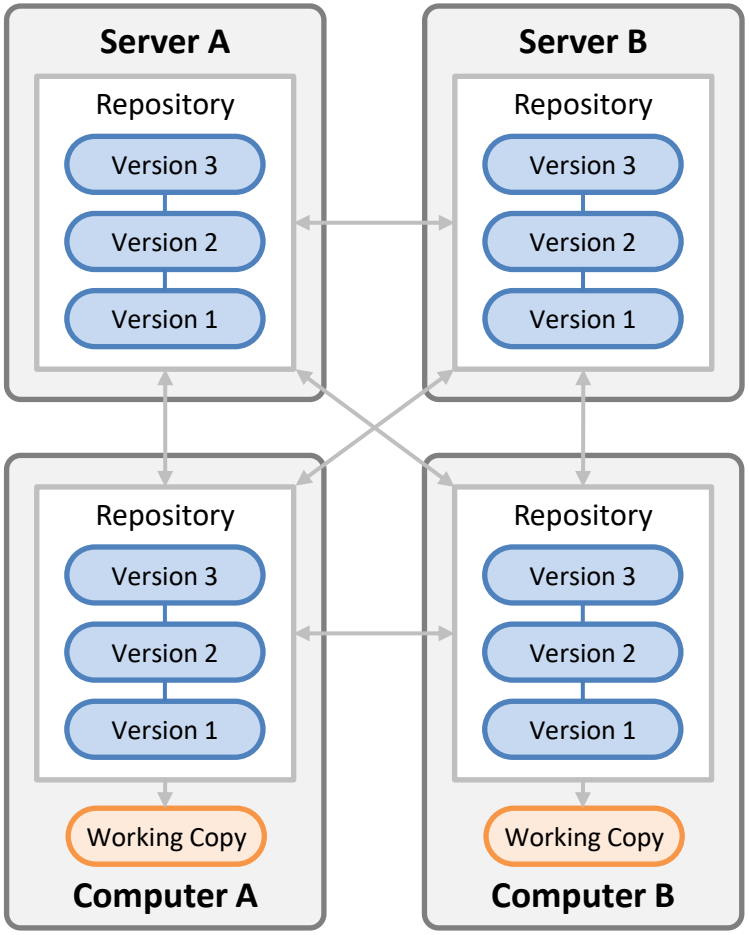
- zentrale VCS sind von zentralem VCS Server abhängig
  - Verbindung zum zentralen VCS Server ist für fast alle Operationen erforderlich
  - offline zu arbeiten ist daher nicht möglich
  - Netzwerkkommunikation macht viele Operationen langsam
  - zentraler VCS Server ist ein Single Point of Failure
- 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:**

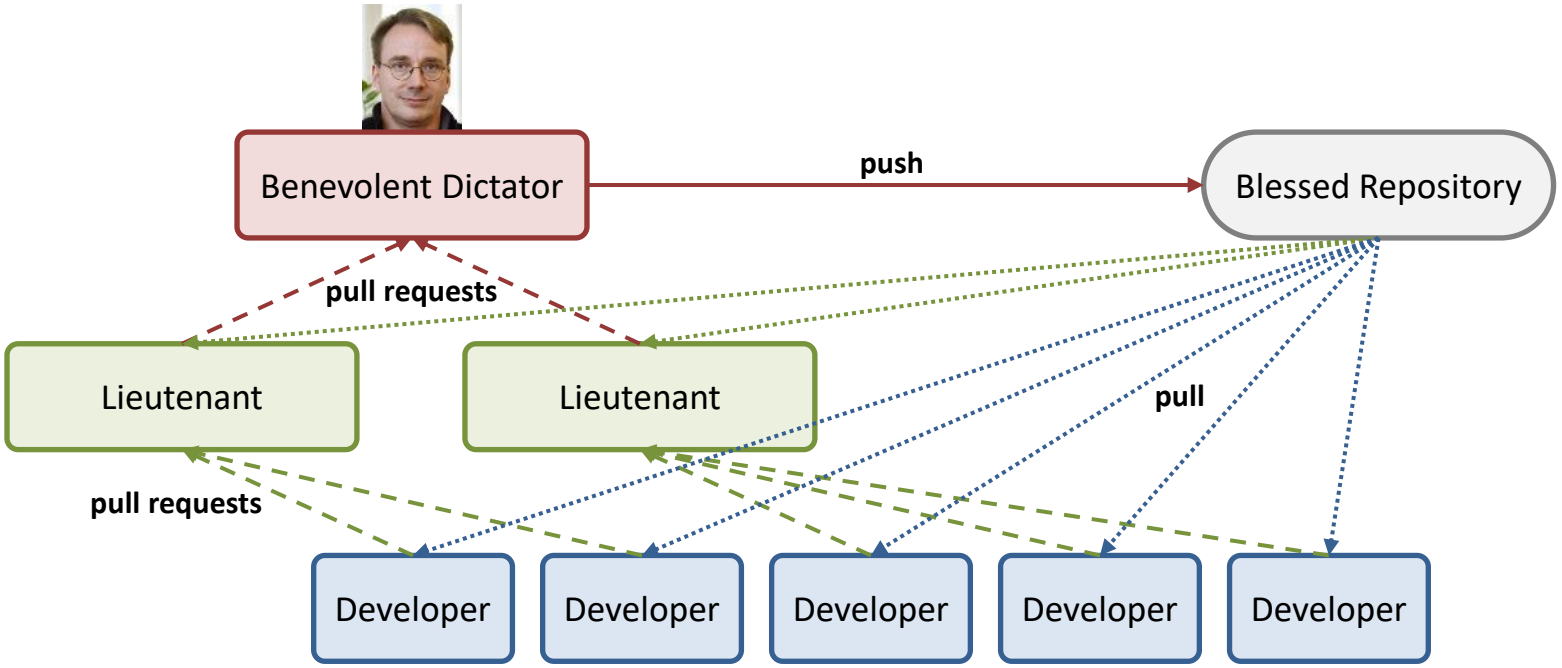
- zentraler Server hält Repository
- Clients halten nur lokale Arbeitskopie
- für viele Operationen ist eine Verbindung zum Server erforderlich
- Erzeugen neuer Version ist ohne Netzwerkverbindung nicht möglich
- zentrales Repository macht die Synchronisation und Integration von Änderungen leichter
-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

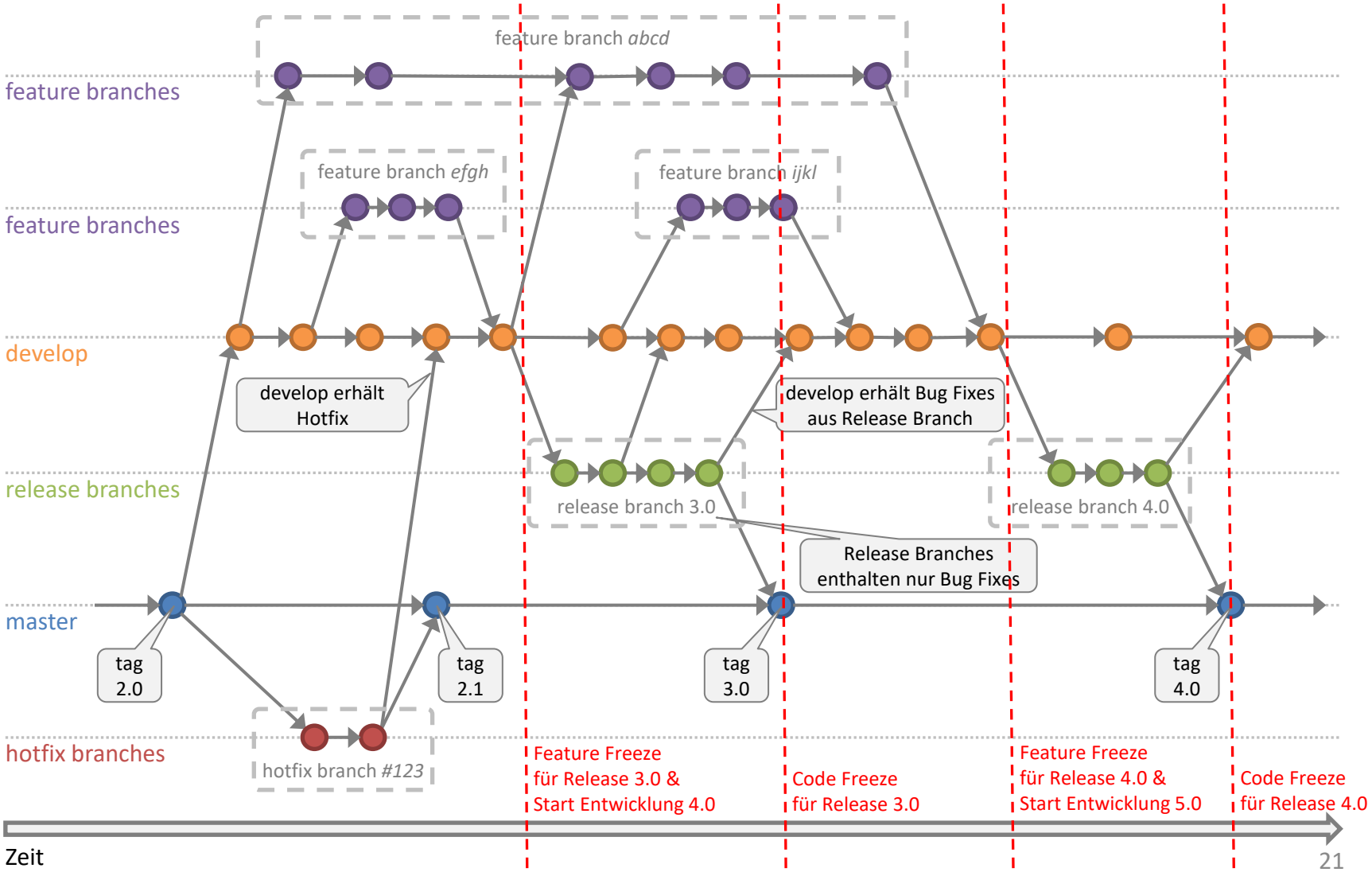
# Tags

- 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

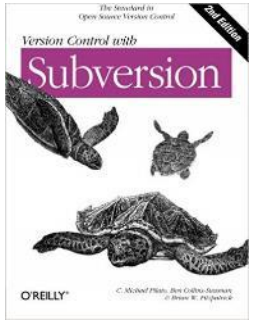
# Branching Beispiel

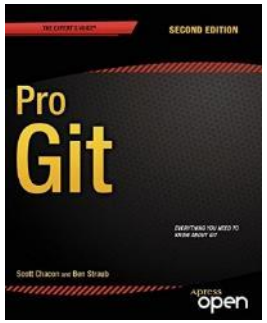


# Integration Hell

- 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