Angewandte Softwareentwicklung
Versionskontrollsysteme

WS 2014/2015

Markus Berg
Hochschule Wismar
Fakultät für Ingenieurwissenschaften
Bereich Elektrotechnik und Informatik

markus.berg@hs-wismar.de
http://mmberg.net

>> *http://twbk.de/<ID>*

Haben sie bereits ein Versionskontrollsystem genutzt?

A	**Ja**
B	**Nein**

Ausgangssituation

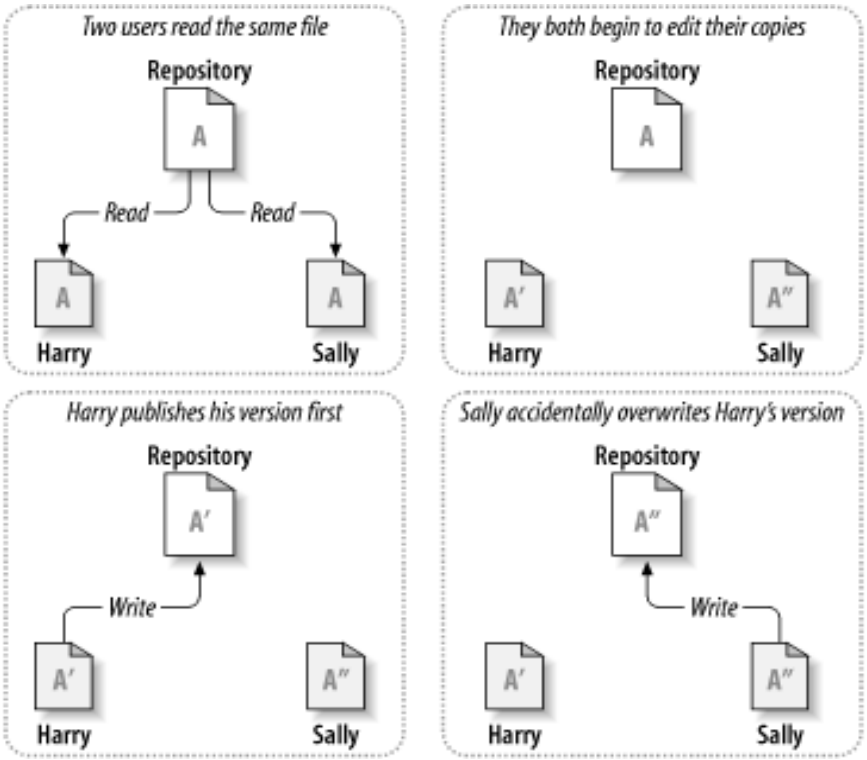
- Dateien durchlaufen im Laufe ihres Lebens verschiedene Versionen
- Änderungen müssen teilweise rückgängig gemacht werden
- Änderungen verschiedener Autoren werden zusammengefügt
- Dateien müssen bei verschiedenen Autoren verteilt werden

Selbst ist der Unwissende

- Lebenslauf_1.doc
- Lebenslauf_2.doc
- Lebenslauf_2014.doc
- Rede.doc
- Rede.old
- Rede2.doc
- Rede3.doc
- Rede_final.doc
- Rede_wirklich_final.doc
- Bericht.doc
- Bericht_Alex_03082014.doc
- Bericht_Alex_Markus_04082014.doc

Mehrere Personen: Zentraler Speicher

- Problem: Änderungen werden versehentlich überschrieben



Mehrere Personen: Dezentraler Speicher

- Person A erstellt das Dokument (V1) und schickt es an B und C
- Person B ändert das Dokument (V2-1) und schickt es an A
- Person A ändert das Dokument (V3) und schickt es an B und C
- Zwischenzeitlich hat Person C das Dokument V1 geändert und schickt es an A (V2-2)

Konflikte = CHAOS!!!

Mehrere Dateien?

- Praktikum1_V1
  - src
    - Main.java
    - Car.java
  - bin
  - praktikum.jar
- Praktikum1_V2
  - src
    - Main.java
    - Car.java
  - bin
  - praktikum.jar

Welche Dateien wurden geändert?

Was wurde geändert?

Wer hat es geändert?

Änderungen rückgängig machen!

Ziele

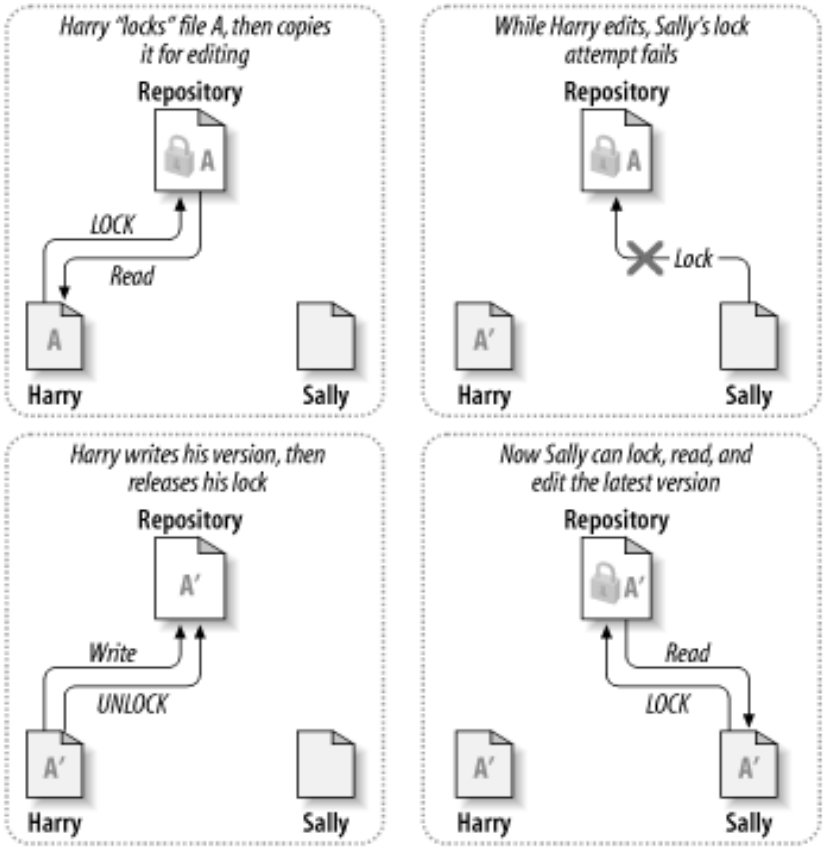
- Sichern von (möglichst lauffähigen) Zwischenständen (Backup)
- Versionierung und Überblick über die Änderungen
- Erstellen von Releases
- Arbeiten im Team
- Unbeabsichtigtes Überschreiben verhindern
- Ständige Bereitstellung der aktuellsten Versionen sowie einer Historie über alle vergangenen Änderungen
- Kein manuelles Verteilen der Dateien an alle Bearbeiter (z.B. per E-Mail)

Lösung

- Versionsverwaltungssysteme
- Trennung von Arbeitskopie und verwalteten Versionen
- Es werden nur die Änderungen + Metadaten (Autor, Datum, Commit-Message) gespeichert
- Versionsnummern werden automatisch generiert

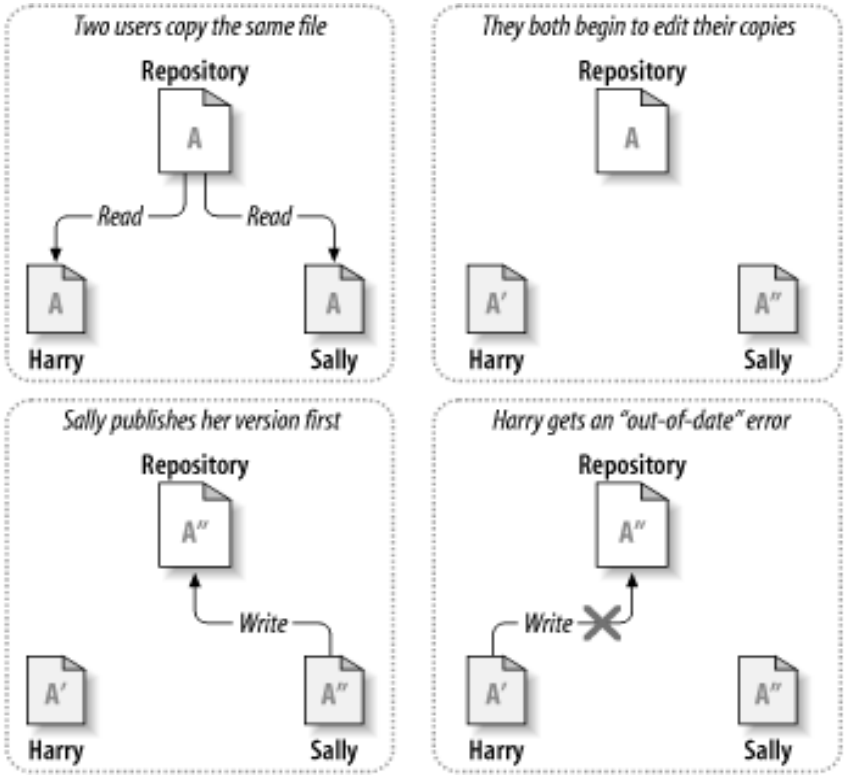
Erste Schritte: RCS

- „Revision Control System“
- Lokales Repository
- Working Copy und Repository liegen beide im (lokalen) Dateisystem
- Hauptsächlich für Single User Betrieb
- Auf Servern (mehrere User) gegenseitiges Sperren
  - Es können nicht mehrere Personen an der gleichen Datei arbeiten

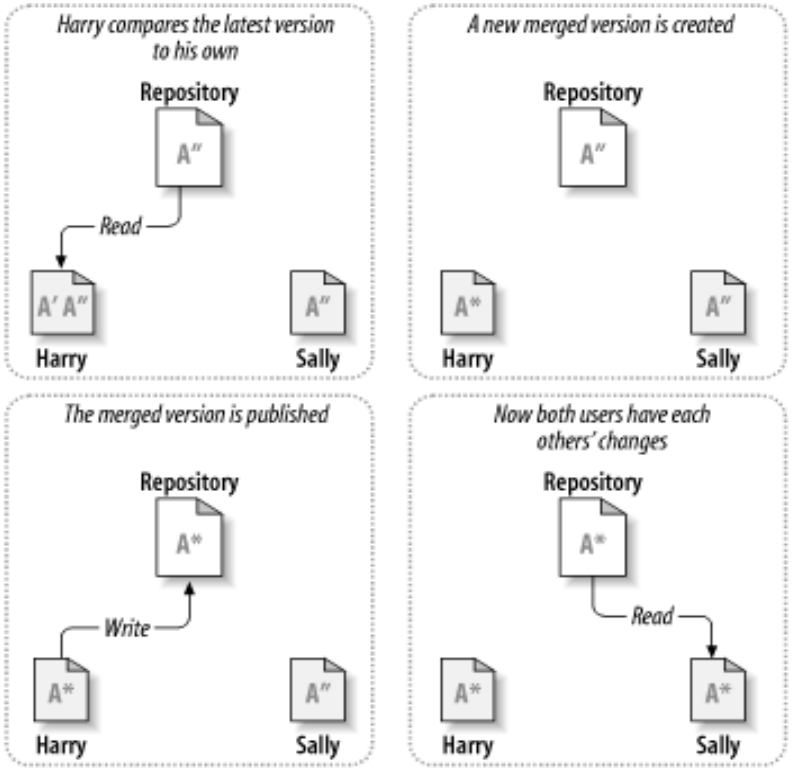


Idee: Kopieren, ändern, zusammenfassen

- 1.



- 2.



CVS (Concurrent Versions System)

- Jeder Bearbeiter hat seine eigenen lokalen Dateien (working copy)
- Jeder Bearbeiter entscheidet wann er seine lokalen Änderungen in das zentrale Repository überträgt und damit allen anderen zur Verfügung stellt
- Hierbei kann es zu Konflikten zwischen Working Copy und Repository kommen (später mehr)
- Andere Entwickler können ihre lokalen Dateien über das Repository aktualisieren
- History nur auf dem Server (Vergleiche nur online)

Von CVS zu SVN

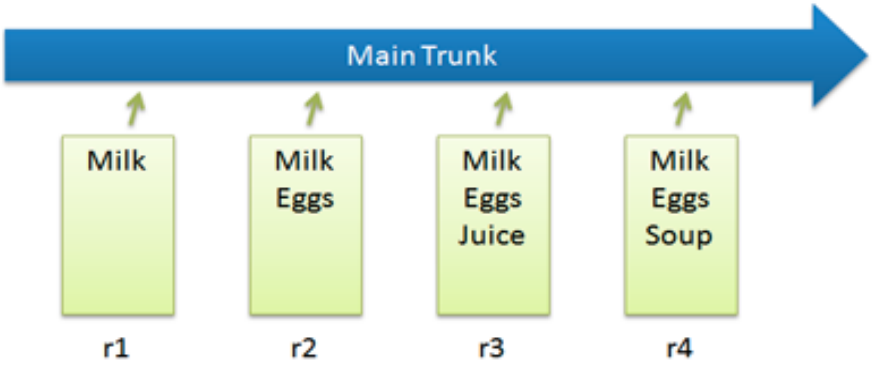
- Unterschied:
  - CVS versioniert jede Datei einzeln
  - SVN versioniert das gesamte Projekt/Repository („Changeset“)
    - Somit wird der Zustand aller Dateien eines Projektes festgehalten
    - Bsp.:
      - Verzeichnis „src“: V5 (→V6)
        - ClassAInterface.java : V1 (→ V6)
        - ClassA.java : V5 (→ V6)
        - ClassB.java : V1
      - Ein Verzeichnis hat die höchste Versionsnummer der enthaltenen Dateien
      - Nun wird das Interface angepasst, was Änderungen in ClassA nach sich zieht
      - Beide Dateien erhalten nun bei einem Commit V6
        - Und nicht wie bei CVS V2 und V6. Dieser Satz an Änderungen bezieht sich auf ein Changeset, da beide Änderungen zueinander in Beziehung stehen.
      - Class B bleibt unberührt
      - Bei Update wird die größte Version kleiner/gleich der angeforderten Version ausgeliefert

Vokabular (I)

- Repository
  - Zentraler Speicher, der die Daten versioniert vorhält („Projektarchiv“)
- Working Copy
  - Lokale Kopie von Daten aus dem Repository, die lokal verändert werden können
- Revision
  - Bestimmte Version der Daten im Repository
- Head
  - Aktuellste Version im Repository
- Trunk
  - Hauptentwicklungsstrang („Stamm“)

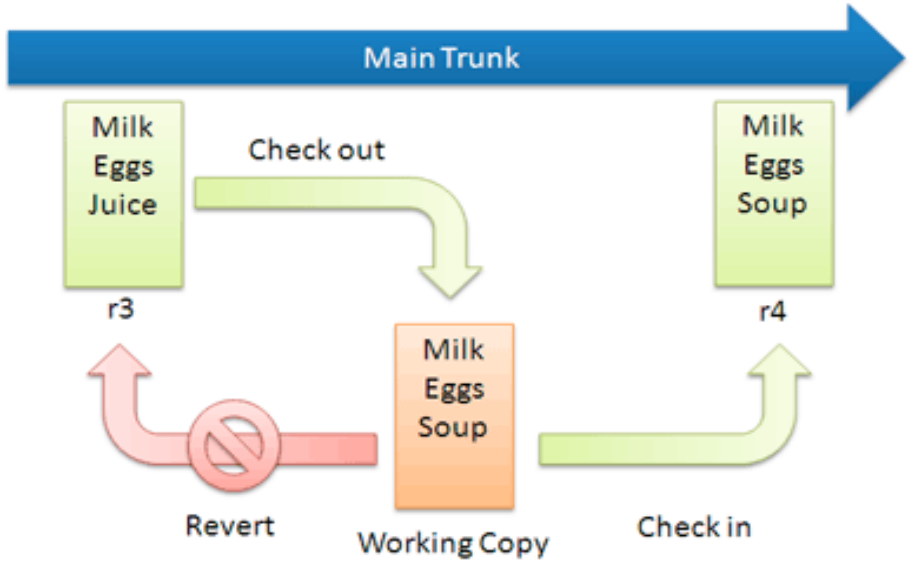
SVN: Checkin & Checkout/Update

Basic Checkins



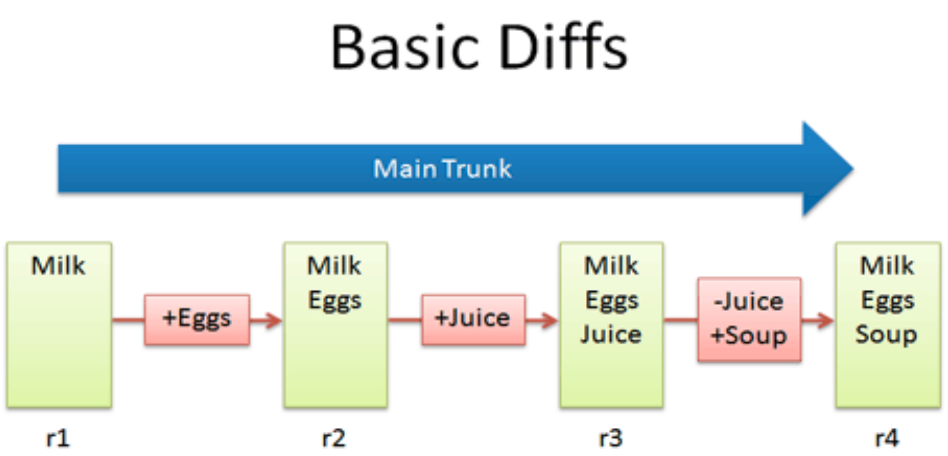
- Übertragen der lokalen Änderungen in das Repository

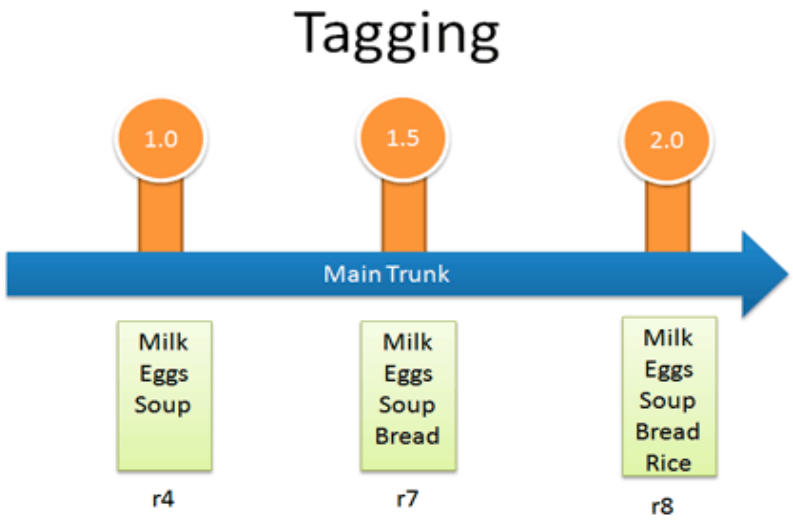
Checkout and Edit



- Checkout = initiales Anlegen der Arbeitskopie
- Checkin=Commit=Hochladen der Änderungen ins Repository
- Update = Aktualisieren der Arbeitskopie mit Änderungen aus dem Repository

SVN: Diffs & Tags



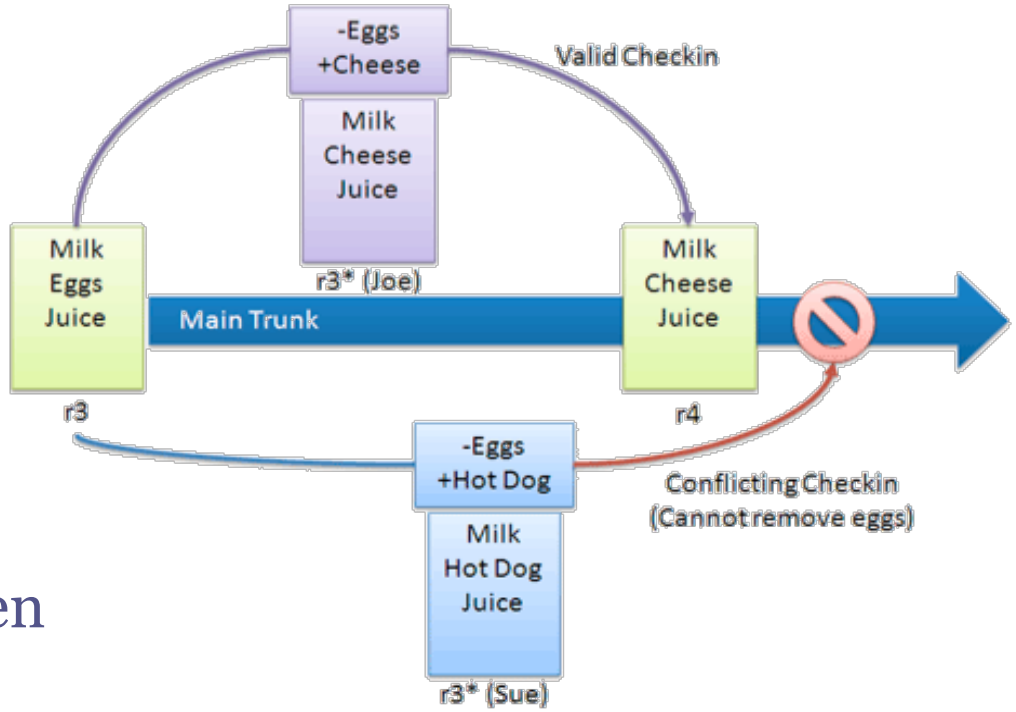


- Ermitteln des Unterschieds zwischen zwei Versionen
- z.B. zwischen r1 und r2 aber auch zwischen r2 und r4

- Revisionen werden mit Tags ausgezeichnet (i.A. Releases)
- Nach Weiterentwicklung sind Releases einfach auffindbar

Konflikte

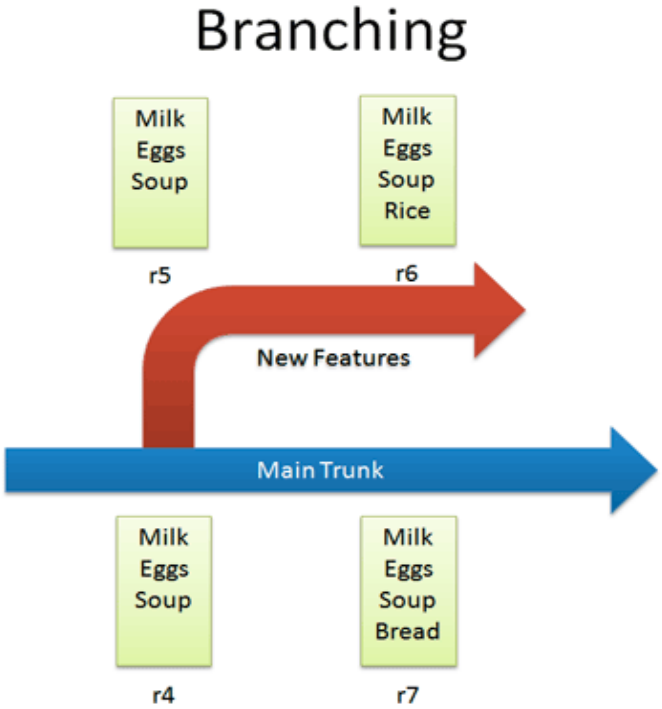
- Konflikte entstehen wenn zwei Personen die gleiche Datei bearbeiten
- Konflikte können teilweise automatisch gelöst werden
  - Binärdateien können nicht gemerged werden
  - Textdateien: Merge erfolgt zeilenweise
    - Bei Änderungen in der gleichen Zeile kein Merge möglich
  - Merge erfolgt beim Update, nicht beim Commit



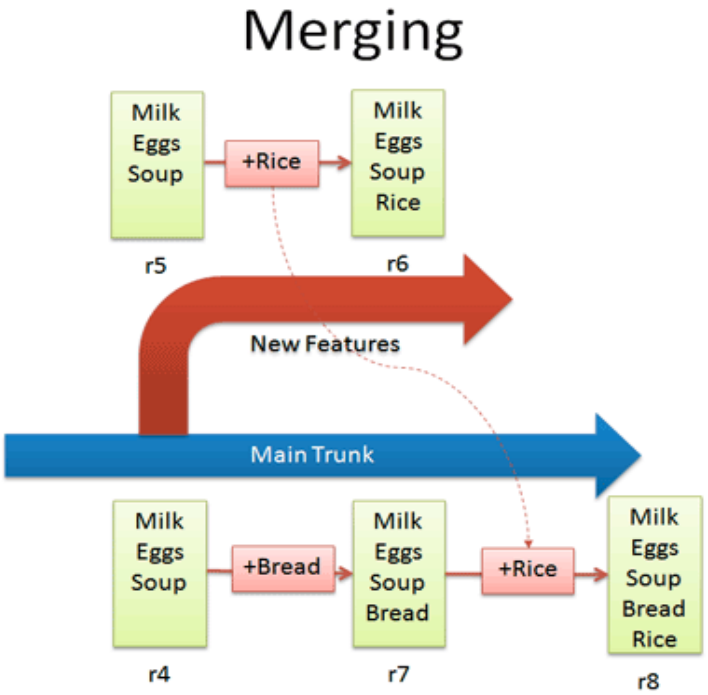
Konflikte lösen

- Bei einem Konflikt erstellt SVN 3 Dateien zusätzlich zur Arbeitskopie von „filename“
  - filename.mine (Datei mit eigenen Änderungen + Konfliktmarkierungen)
  - filename.r`Oldrev` (alte Datei vor der Änderung)
  - filename.r`Newrev` (neue Datei aus dem Repo)
- Revert
  - Lokale Änderungen verwerfen
  - Es entsteht keine neue Version
- Resolve (mit Angabe einer Datei ohne Konflikte)
  - Die 3 temporären Konfliktdateien werden gelöscht
  - Die angegebene Datei wird als neue nicht konfliktbehaftete Version markiert (resolved) und an das Repository übermittelt (commit)

SVN: Branching & Merging



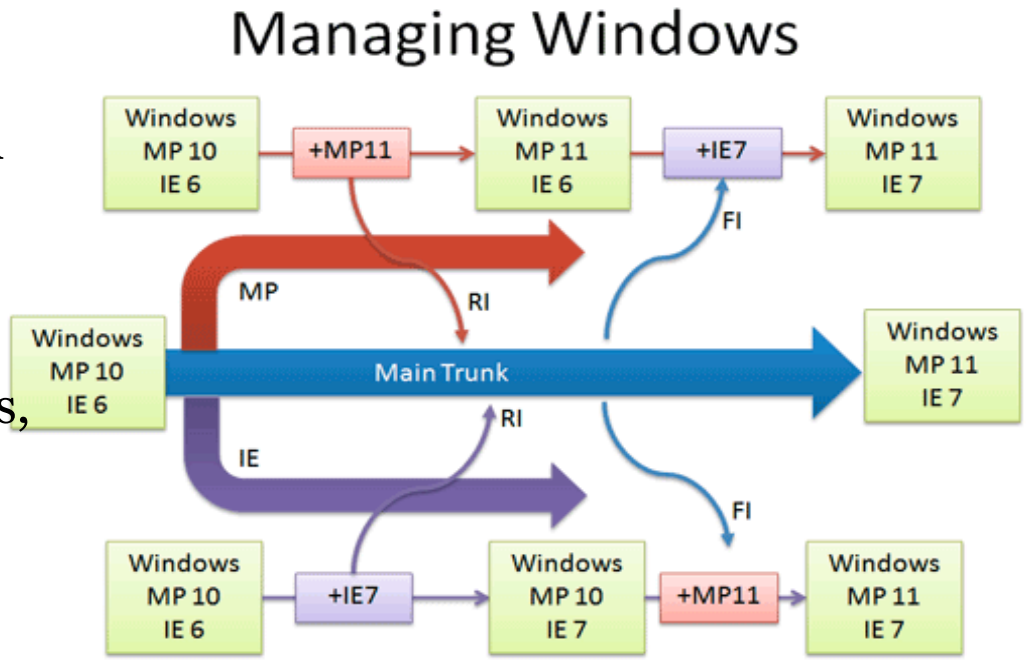
- Erstellen einer isolierten Kopie `svn copy`
- Weiterentwicklung ohne die Hauptentwicklungslinie zu beeinflussen



- r8 = Diff zw. r5 und r6 dem Head (r7) des Trunk hinzufügen
- in den Trunk wechseln, dann:
- `svn merge -r5:6 <branchURL>`

SVN: Integration

- … am Beispiel vom Windows:
  - Team 1 entwickelt den Media Player, Team 2 den Internet Explorer
  - Beide Teams arbeiten isoliert in ihren Branches und fügen neue Features dem Trunk hinzu
- Reverse Integration
  - Neue Features (im Sinne von Releases, keine Zwischenstände) vom Branch dem Trunk hinzufügen
- Forward Integration
  - Neue Features vom Trunk beziehen und dem eigenen Branch hinzufügen



Wirkweisen und Prinzipien

- Löschen
  - Lokal gelöschte Dateien werden nicht im Repository gelöscht sondern bei einem Commit lediglich aus der aktuellen Version entfernt
  - Somit können gelöschte Dateien wieder hergestellt werden
- Gleichzeitiges Bearbeiten
  - … ist möglich
  - … und kann zu Konflikten führen
  - Erst update, ggf. edit/resolve, dann commit

>> *http://twbk.de/<ID>*

Was ist ein Branch?

A	**Hauptentwicklungsstrang**
B	**Entwicklungszweig**

>> *http://twbk.de/<ID>*

Was ist mergen?

A	**Neue Version ins Repository übertragen**
B	**Einen Konflikt lösen**
C	**Zwei Branches bzw. Trunk/Branch kombinieren**

Vokabular (II)

- Branch
  - isolierter Teilentwicklungszweig
- Tag
  - eine bestimmte Revision, die mit einer ID gekennzeichnet wird, meist ein Release
- Check in / Commit
  - Lokale Änderungen ins Repository übertragen
- Check out
  - Ein Projekt initial aus dem Repository laden
- Update
  - Arbeitskopie aktualisieren (mit Änderungen aus dem Repository)
- Merge
  - Zwei Entwicklungslinien zusammenfügen
- Diff
  - Differenz zwischen zwei Revisionen
- Revert
  - Lokale Änderungen verwerfen
- Resolve
  - Konflikt lösen (Review der Änderungen und Entscheidung welche Änderungen übernommen werden)

Was gehört nicht unter Versionskontrolle?

- Automatisch generierte Dateien
  - z.B. exe oder jar-Files
  - bei LaTex z.B. das erzeugte PDF

Clients

- Über Kommandozeile
- Windows: Tortoise SVN (visuell, Integration in Windows-Explorer)
- Mac: svnX
- In IDEs integriert (z.B. Eclipse, Netbeans)

Grundlegende Befehle

- Repository anlegen
  - svnadmin create <pfad>
- Checkout
  - svn co <url>
- Dateien hinzufügen (d.h. unter Versionskontrolle setzen)
  - svn add <file>|<dir>
- Commit
  - svn ci (im entsprechenden Verzeichnis)
- Update
  - svn up (im entsprechenden Verzeichnis)
- Diff
  - svn diff (im entsprechenden Verzeichnis)
- Lokale Änderungen verwerfen
  - svn revert <file>

(empfohlene) Verzeichnisstruktur

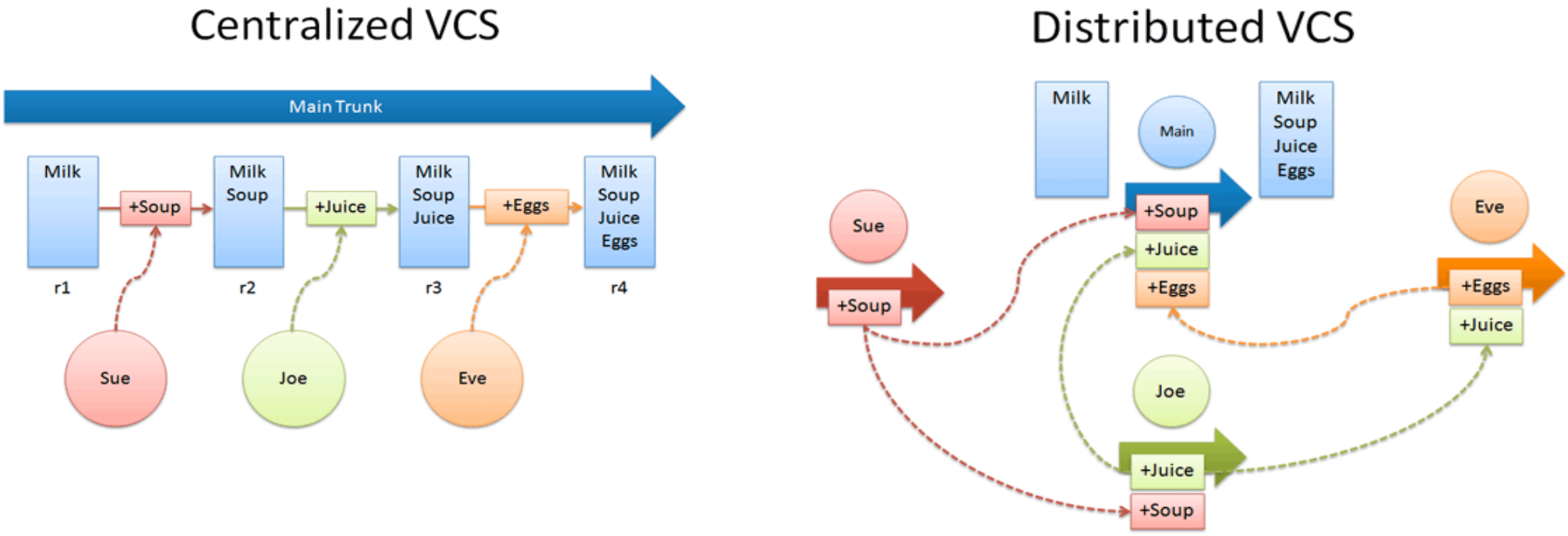
- Trunk
- Branches
- Tags

- (In SVN ist alles eine „billige“ Kopie, d.h. ein Link auf eine bestimmte Version plus evtl. Änderungen)
- Branches und Tags werden mit `svn copy` erzeugt

Dezentrale Versionsverwaltung

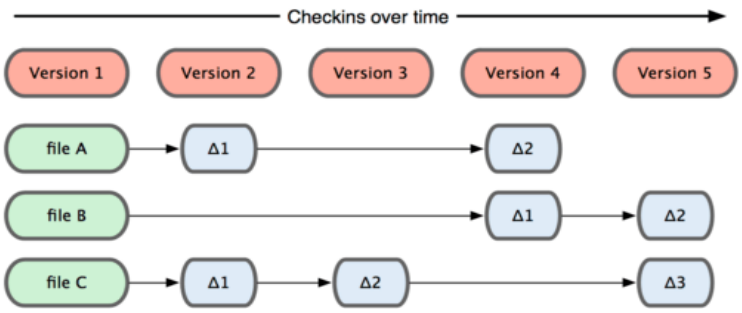
- Bisher:
  - Lokale Versionsverwaltung (RCS)
  - Zentrale Versionsverwaltung (CVS, SVN)
- Neuer Ansatz:
  - Dezentrale Versionsverwaltung (z.B. GIT)
- Kein zentrales Repository
- Jeder Benutzer hat sein eigenes lokales Repository (+ globales Repository für alle)
- Vorteil: Offline arbeiten möglich
  - Versionieren wenn keine Netzverbindung besteht
  - Nachträgliche Zusammenführung der Versionen
  - Die lokalen Repositories werden mit dem zentralen regelmäßig synchronisiert
- Vorteil:
  - Geschwindigkeit
  - Die gesamte Versionsgeschichte ist lokal verfügbar

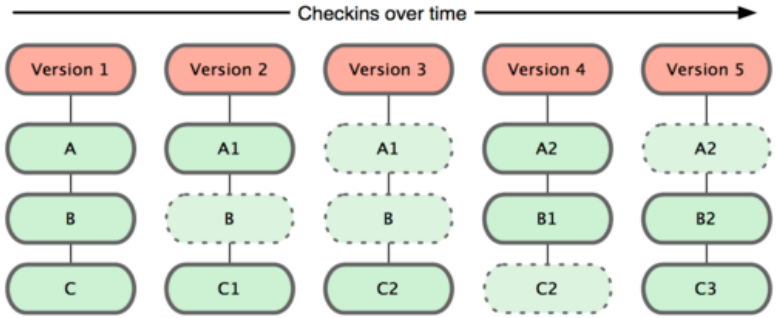
Zentral vs. verteilt



GIT

- u.a. bekannt durch GitHub, Linus Torvalds,…
- Hauptfokus: Branches und deren Zusammenfügen
- Versionsnummern sind Hashes (SHA-1)
  - z.B. 24b9da6552252987aa493b52f8696cd6d3b00373
- GIT speichert immer ganze Dateien und nicht nur Änderungen (schneller)
  - „Snapshots“
  - Unveränderte Dateien werden nicht doppelt gespeichert





GIT

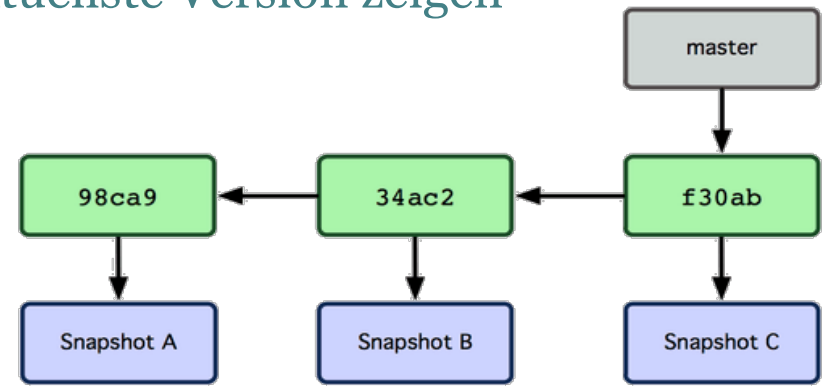
- Arbeitskopien werden durch das Klonen von Repositories erzeugt (git clone) oder durch das Anlegen eines neuen Repos (git init)
- Eine lokale Revision im lokalen Repository erstellen und sie dem globalen Repository hinzufügen sind zwei getrennte Arbeitsschritte
- Dies erlaubt die (offline) Aufzeichnung eines Arbeitsschrittes und die spätere (online) Übertragung in das globale Repo („Remote“)

GIT: Änderungen übermitteln

- Geänderte Dateien werden zunächst in eine Staging Area verschoben (git add) und für den nächsten Commit vorgemerkt
  - Nicht mit svn add verwechseln!!!
- Erst danach werden sie dem (lokalen) Repository hinzugefügt (git commit)
  - Logisch zusammengehörige Änderungen der Staging Area hinzufügen und committen („Changeset“)
  - Es können lokal Revisionen erstellt werden, ohne dass sie direkt dem zentralen Repository hinzugefügt werden müssen
- Änderungen bis jetzt nur im lokalen Repo
- Übertragen in entferntes (i.d.R. globales) Repo mit git push
  - Sogenanntes „Remote“
  - Sinn: Kollaboration
  - git push origin master
    - Master-Branch an Remote namens „origin“ übertragen

GIT: Branches

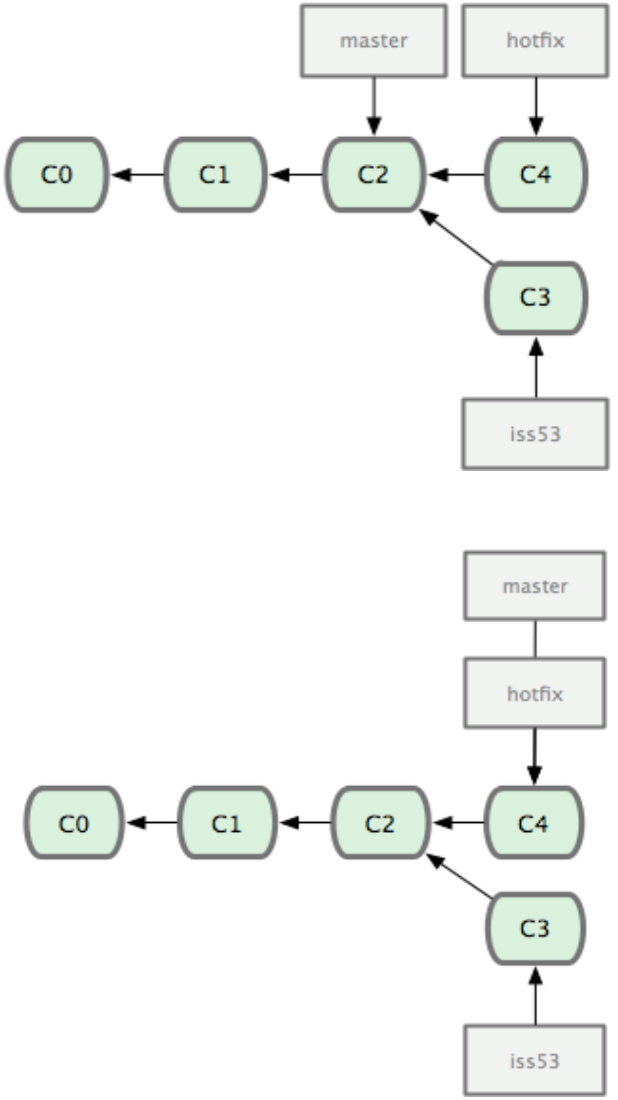
- Entwicklung in Branches (der Master ist selbst ein Branch)
- Für jedes Feature (im Idealfall) sollte ein eigener Branch erstellt werden, der dann mit dem Master zusammengeführt wird
- Branches sind Pointer auf Revisionen (d.h. Snapshots)
  - Master sollte auf aktuellste Version zeigen



- Branches erzeugen und zu ihnen wechseln sind getrennte Operationen
  - Erstellen: git branch <name>
  - Wechseln: git checkout <branch>

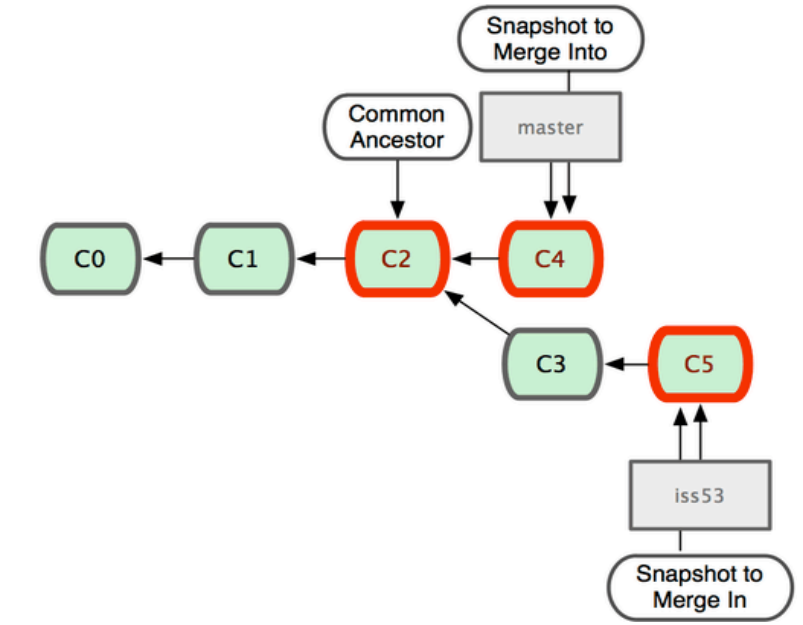
GIT: Warum Branching?

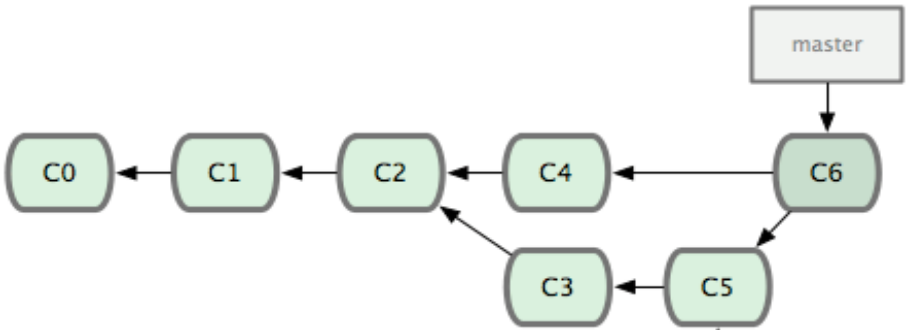
- Lauffähige Version im master-Branch
- Neues Feature wird in iss53 entwickelt
- Problem im Master tritt auf
  - Hotfix erstellen, ohne dass Änderungen durch iss53 stören (Projekt u.U. nicht lauffähig)
- Wenn Hotfix erfolgreich, dem Master hinzufügen (merge)
- Weiterentwickeln von iss53…



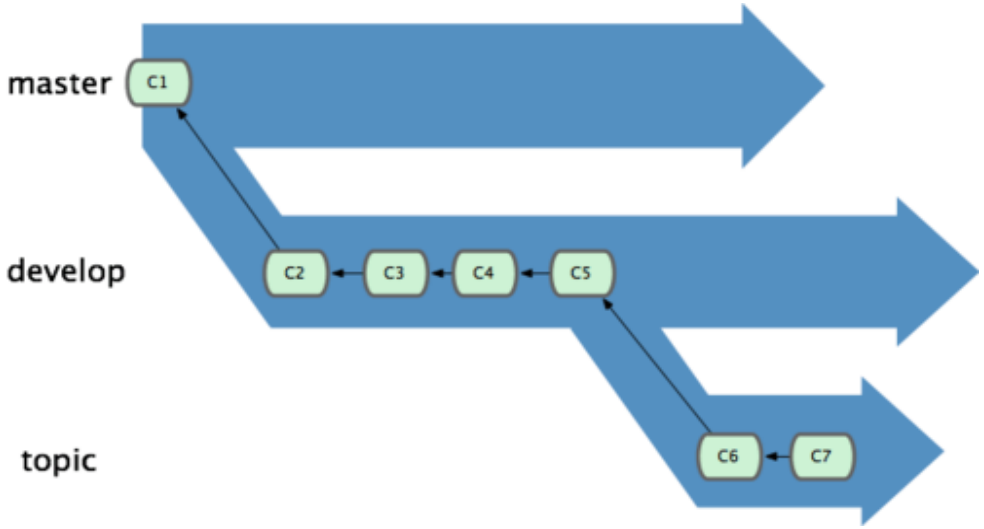
GIT: Branching & Merging

- Arbeit an iss53 ist beendet
- Der neue Master ist aber kein Vorfahre von iss53 mehr
- Merge wird komplizierter
  - 3-Way-Merge
  - C6 hat zwei Vorfahren
  - Iss53-Branch kann nun gelöscht werden, da er in den Master integriert wurde



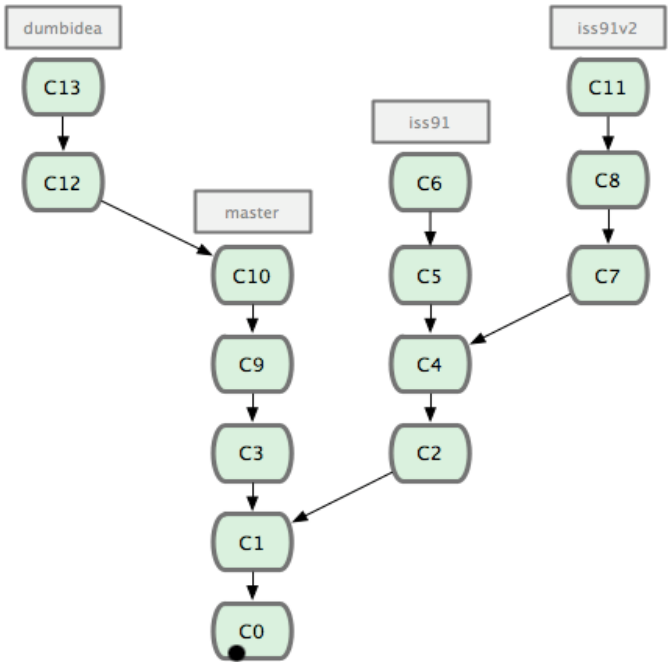


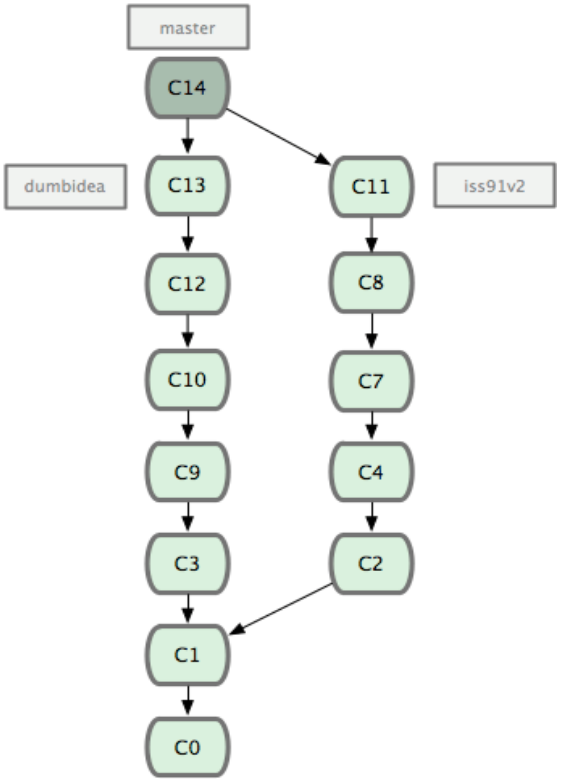
GIT: Branching Workflows



- Verbreitete Organisation:
  - Master-Branch für lauffähige Version (stabil)
  - Development-Branch für experimentelle Version bzw. laufende Entwicklung (test)
  - Einzelne topic/feature-Branches für die einzelnen in der Entwicklung befindlichen Features (instabil bzw. nicht lauffähig)

GIT: Branching Workflows



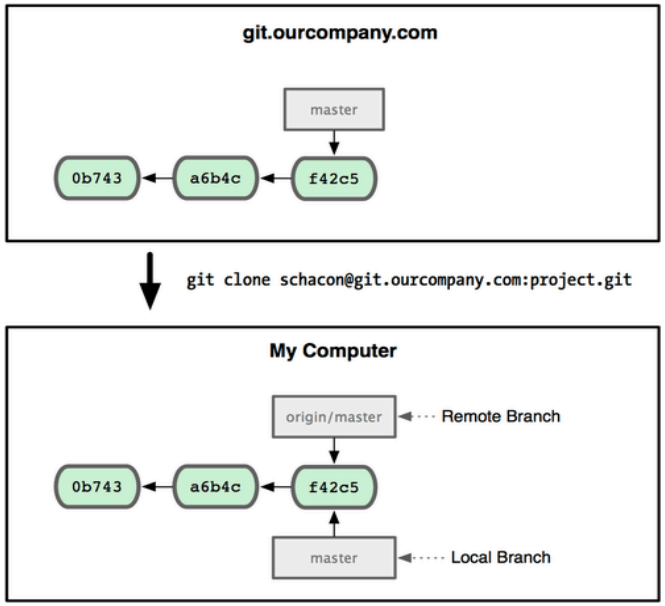


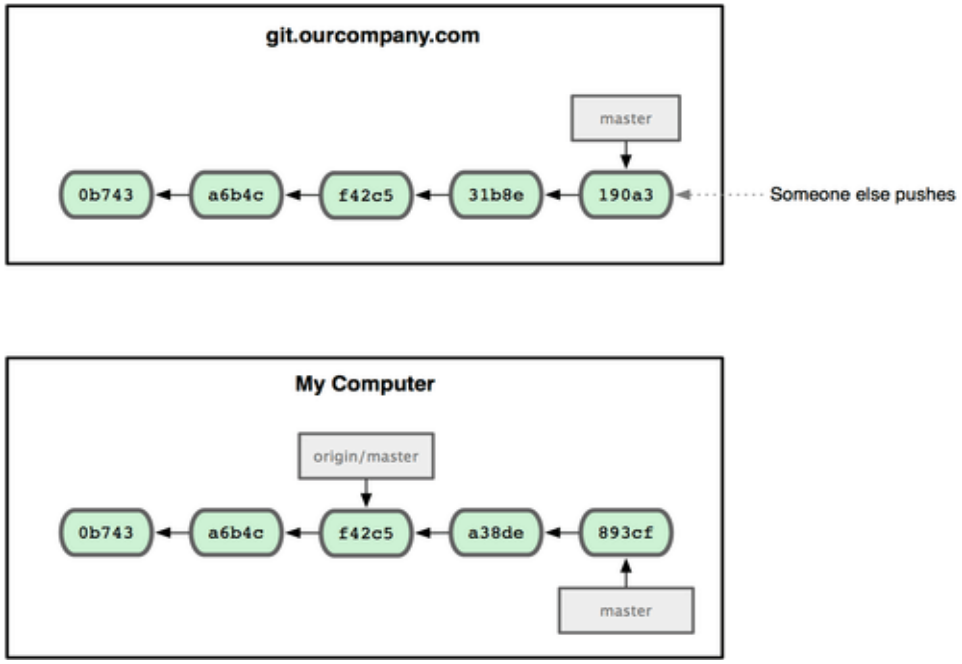
- Iss91 wird in zwei Varianten entwickelt
- Variante 2 setzt sich durch
- Eine „dumme“ Idee ist genial und wird ebenfalls übernommen

GIT: Remotes

- = Entfernte Repositories
- Es ist möglich mehrere Remotes zu haben
  - Änderungen verschiedener Entwickler erhalten
- Beim „clone“ wird automatisch ein Verweis auf das Quellrepo angelegt
  - Dieses Remote nennt sich „origin“
  - Es werden automatisch „tracking-branches“ angelegt
    - D.h. das lokale master bezieht sich auf origin/master
    - Bei push und pull müssen daher keine Argumente angegeben werden (wenn der aktuelle Branch einen anderen „trackt“)

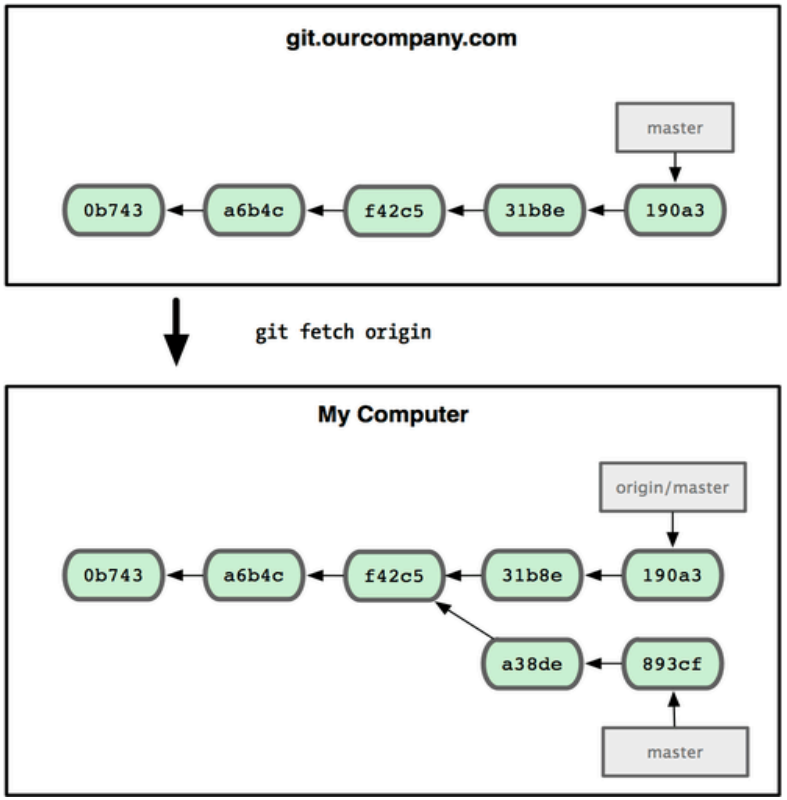
Remote Branching





- Repo klonen
- entferntes und lokales Repo laufen auseinander

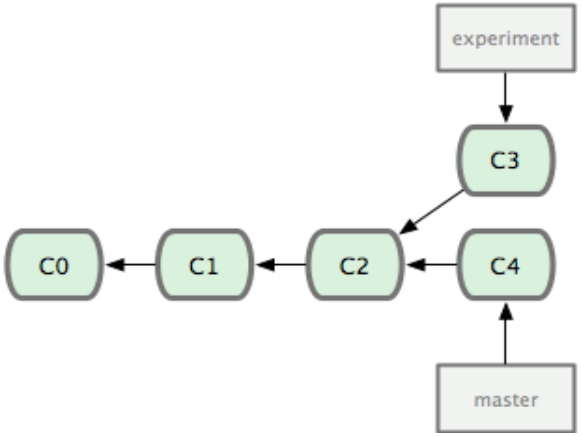
Remote Branching



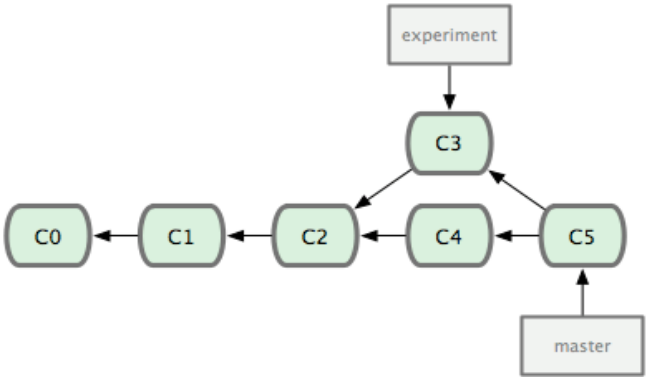
- Lokales Repo aktualisieren, d.h. Änderungen des origin/ master-Branches in lokales Repo aufnehmen
- F42c5 ist die letzte gemeinsame Revision
- Branches sind auseinander gelaufen
- Merge erforderlich!
- Alternativ: pull
  - fetch + merge

Änderungen zusammenführen

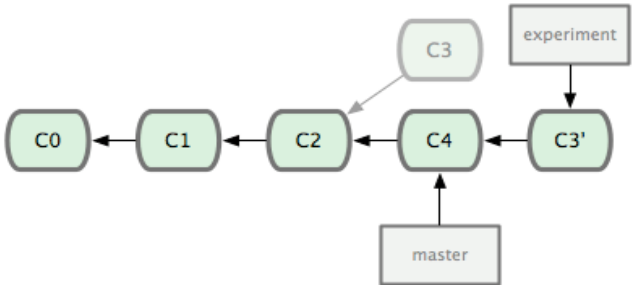
- Ausgangssituation:



- Merge: 3-Way-Merge:
  - Änderungen zw. C2 und C3 sowie C2 und C4 werden zusammengefasst und erzeugen eine neue Revision C5
  - C5=merge(merge(C2,C3),merge(C2,C4))

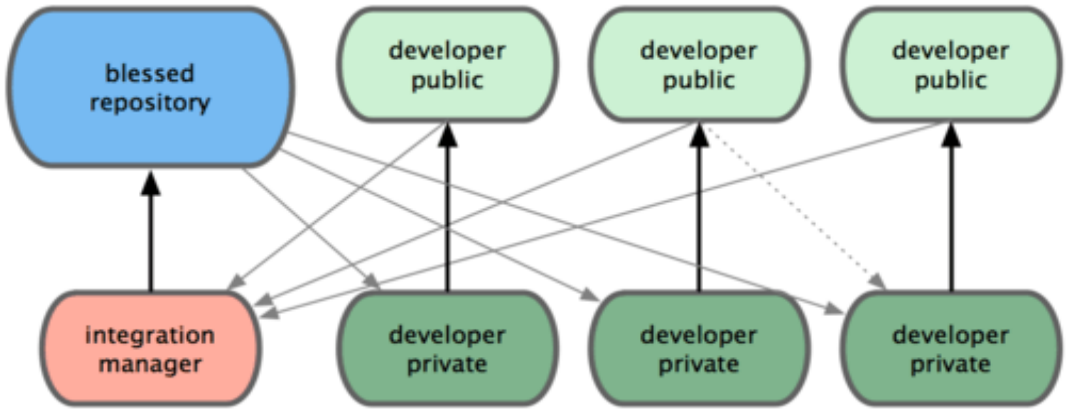


- Rebase: Patch erzeugen:
  - Änderungen zwischen C2 und C3 ermitteln und C4 hinzufügen
  - C3‘= C4+diff(C2,C3)



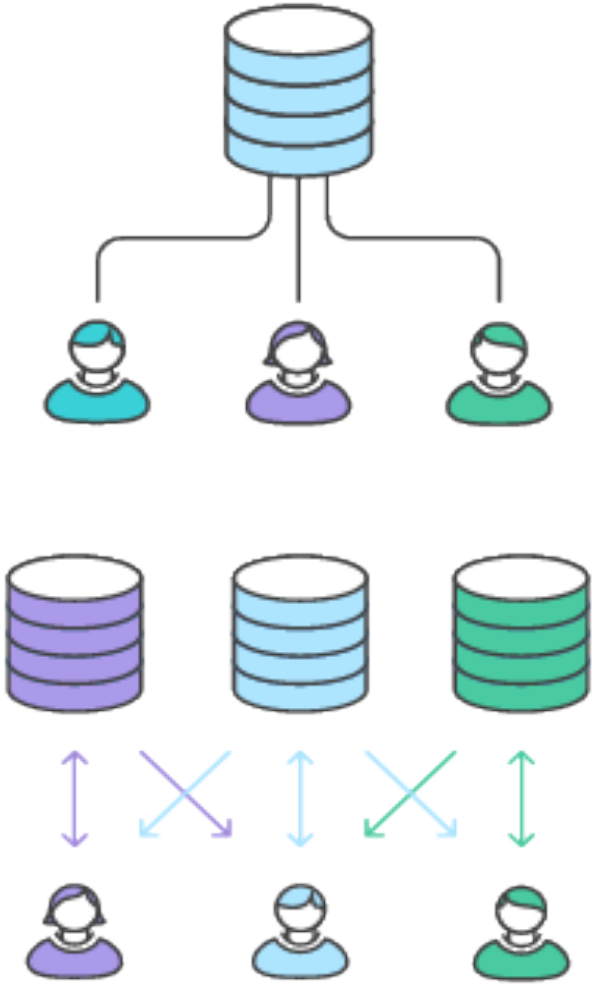
Integration-Manager Workflow

- z.B. GitHub
  - Entwickler forkt öffentliches Projekt vom offiziellen Repository
    - Erzeugt öffentliches Repository (Kopie des Projektes)
  - Entwickler klont sein neues Repository
    - Erzeugt lokales Repository
  - Entwickler committet Änderungen lokal
  - ...und pusht sie in sein öffentliches Repository
  - Wenn er fertig ist, sendet er einen Pull Request an den Projektinhaber (integration manager)
  - Dieser kann nun Änderungen des Entwicklers von dessen öffentlichem Repository fetchen (in sein lokales Repo), mergen und anschließend in das öffentliche offizielle Repository pushen



SVN vs GIT

- Zentral vs. dezentral
- Online vs. offline arbeiten
- git push ~ svn commit
- SVN trunk ~ GIT master
- GIT: keine separaten Verzeichnisse für Branches
- GIT: das Working Directory selbst ist ein Branch
- GIT ist sehr flexibel und erlaubt verschiedene Workflows, u.a. auch einen zentralen Ansatz
- SVN Versionsnummern vs. GIT Hashes



Quellen bzw. weiterführende Literatur

- http://www3.cs.fau.de/Lehre/DOLINUX/SS2009/8-versionsverwaltung.pdf
- http://betterexplained.com/articles/a-visual-guide-to-version-control/
- http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/
- http://svnbook.red-bean.com
- http://git-scm.com/book/en/Getting-Started-Git-Basics
  - http://git-scm.com/book/en/Git-Basics
  - http://git-scm.com/book/en/Git-Branching
- GIT Tutorials:
  - https://www.atlassian.com/git
  - https://www.codeschool.com/courses/try-git

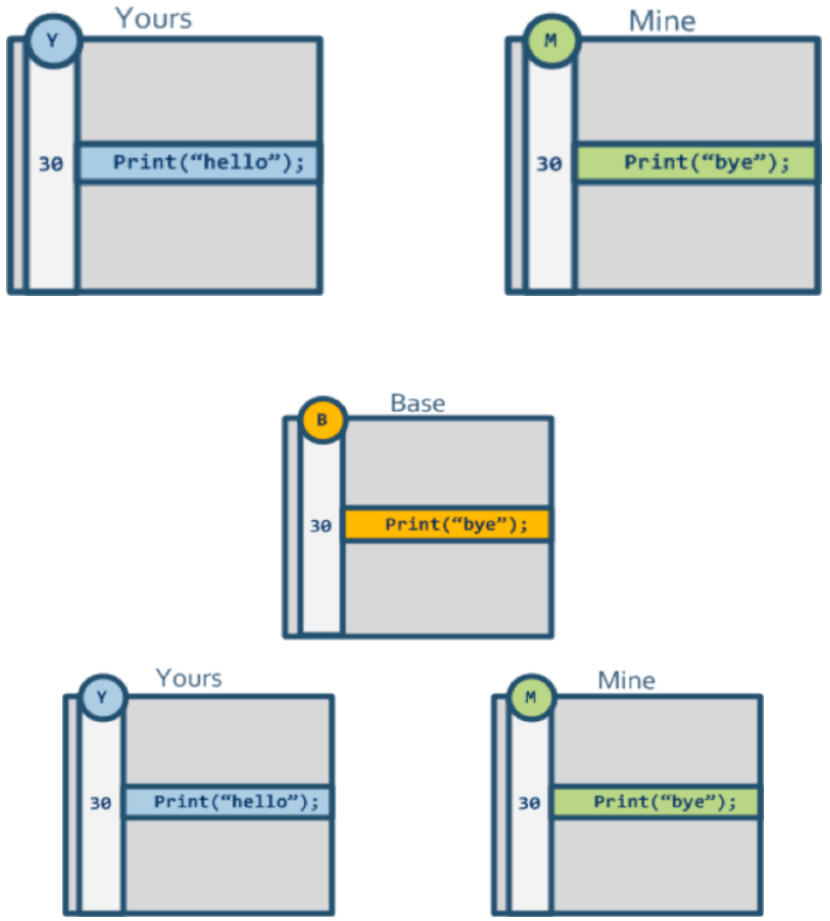
Anhang

Nachträgliche Ergänzungen zur Vorlesung

Merge-Varianten

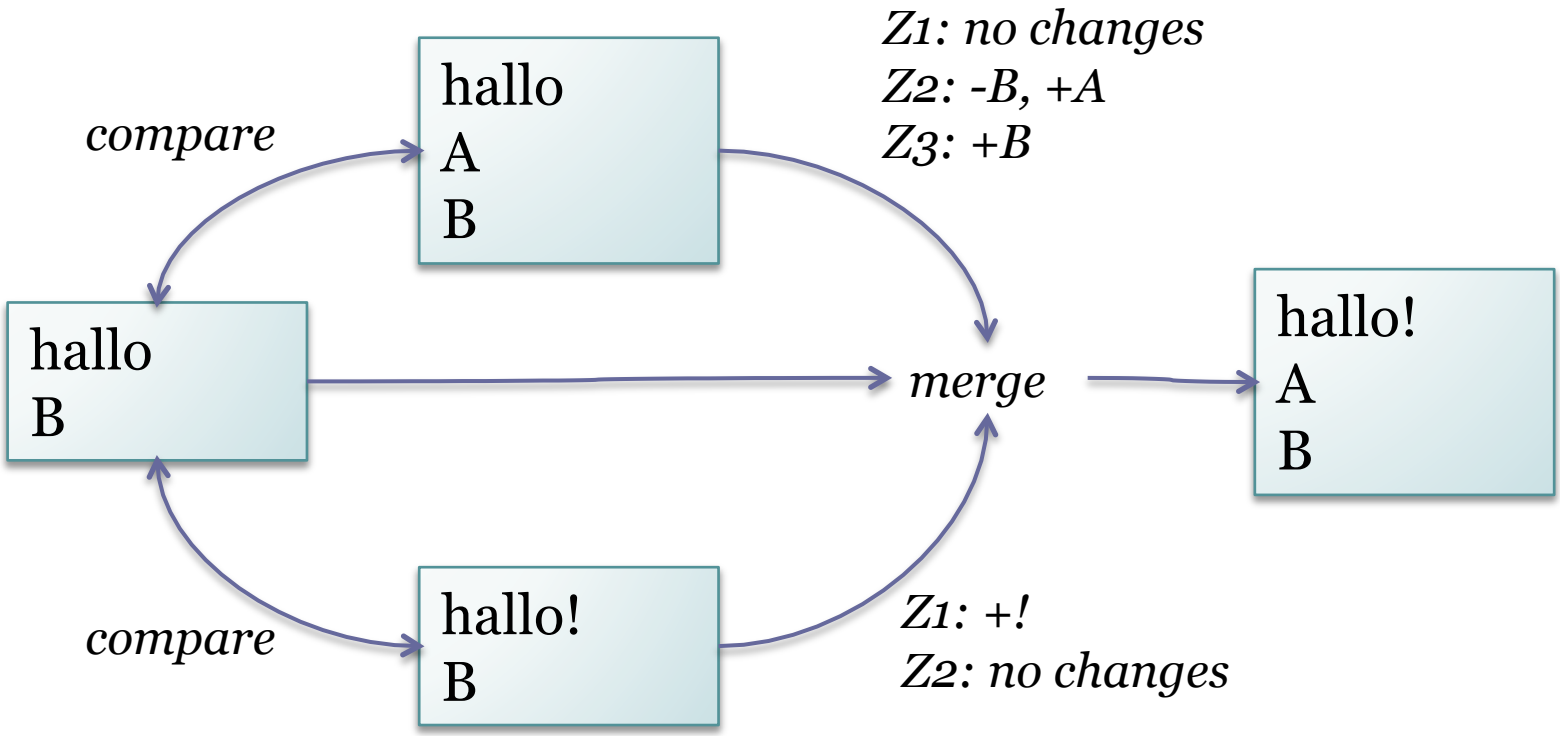
- Zwei-Wege-Merge erkennt, dass Änderungen stattgefunden haben, erkennt jedoch keinen zeitlichen Verlauf, d.h. welche der beiden Versionen das Original und welche die Änderung (und damit die Endversion) ist

- Drei-Wege-Merge vergleicht die veränderten Versionen mit einem gemeinsamen Vorgänger und kann so ermitteln, welche der beiden Versionen die eigentliche Änderung und somit gültige Endversion ist



aus: http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

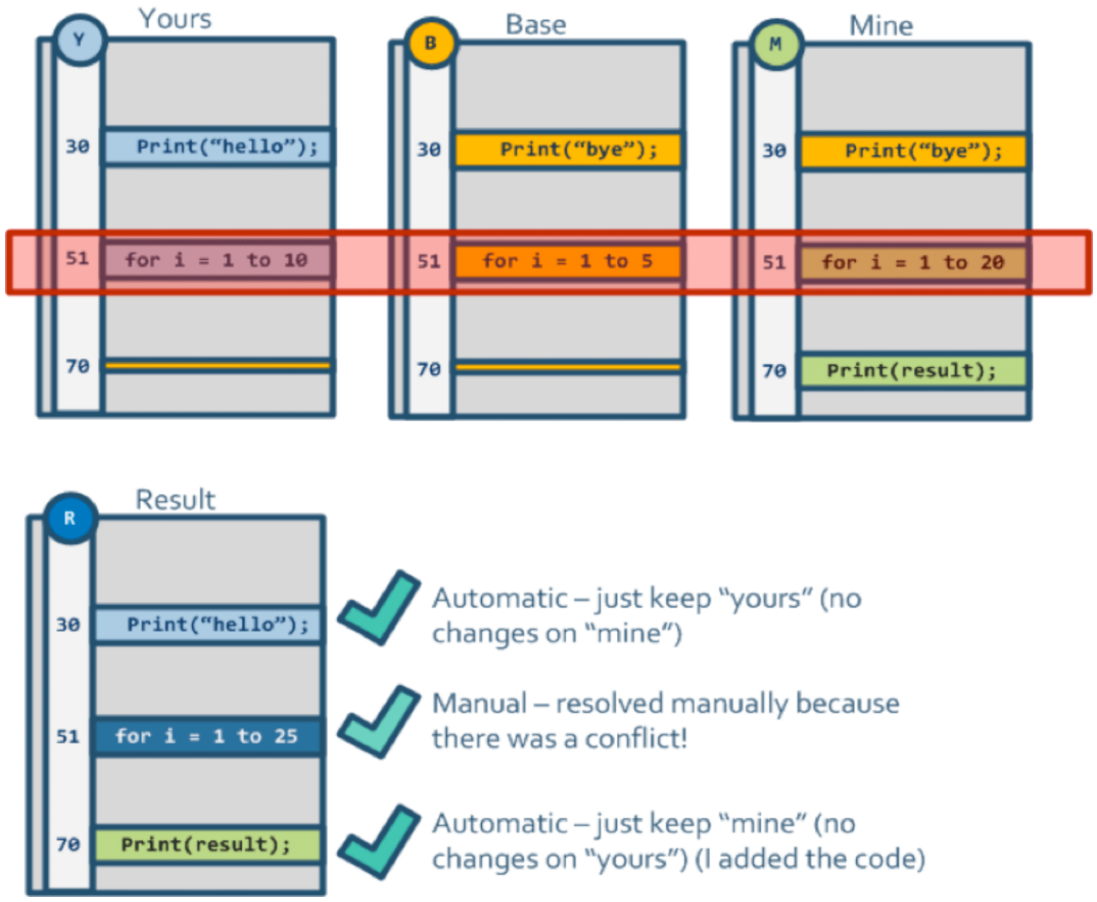
Drei-Wege-Merge



→ Kein Konflikt da keine Änderungen in gleicher Zeile

Drei-Wege-Merge

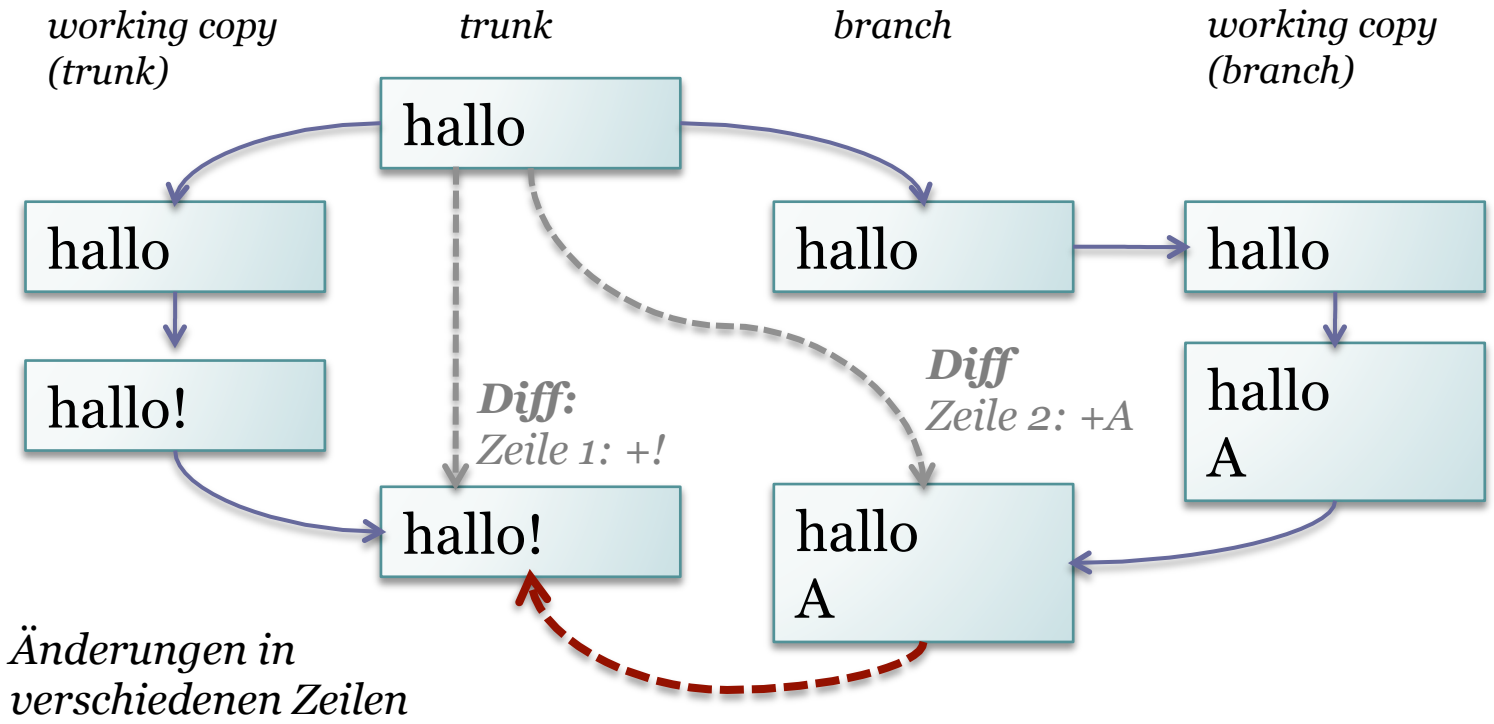
- Beispiel:



aus: http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

Problem

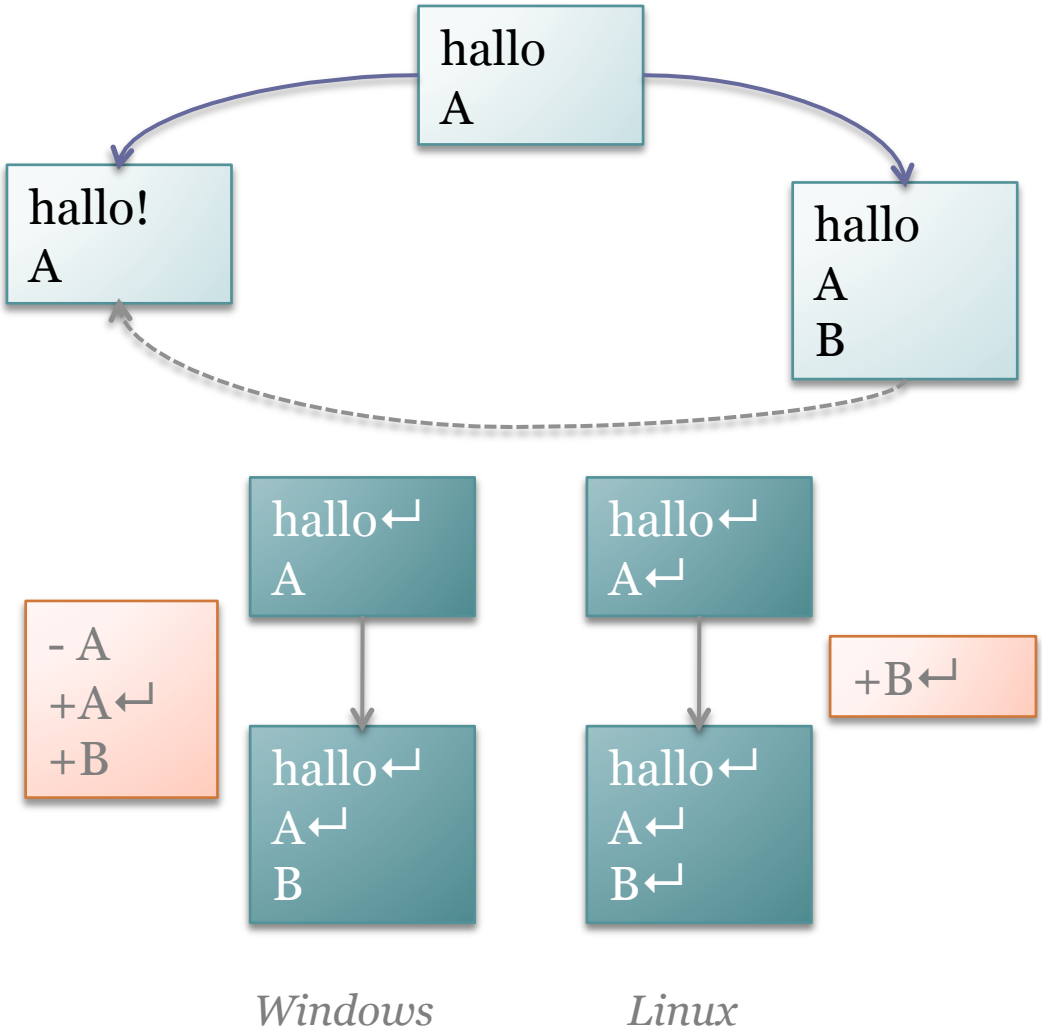
- Warum schlägt folgender Versuch fehl?
  (hier am Bsp. des Merge; identisch bei Update)



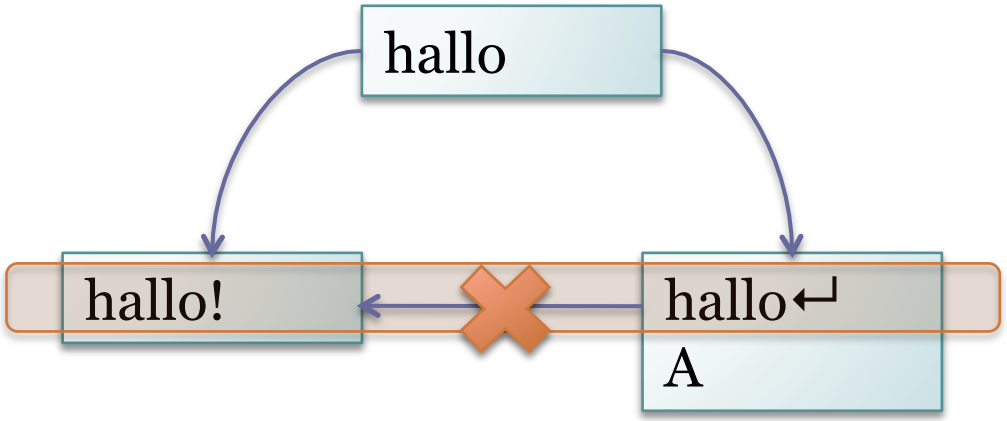
Merge resultiert in Konflikt!

Info: Zeilenende

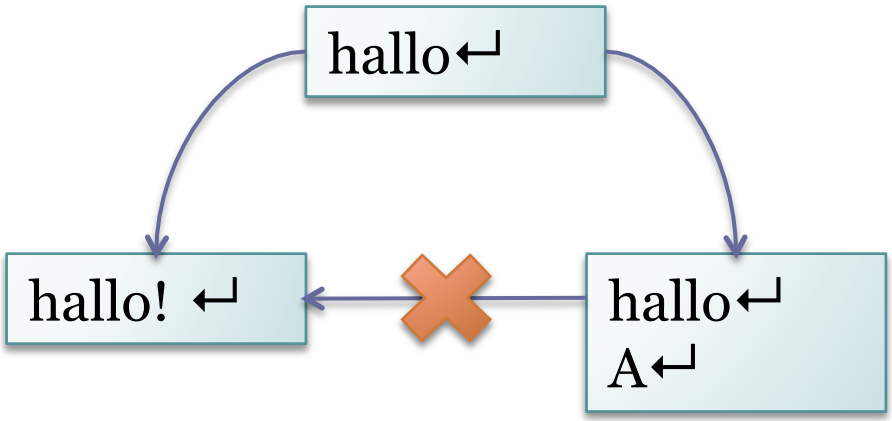
- Unterschiedliches Verhalten bzgl. Zeilenende bei Windows und Linux (nicht zu verwechseln mit der Tatsache dass Windows \r\n und Linux \n schreibt)
- Vereinfacht gesagt: Windows markiert Zeilenumbruch, Linux das Zeilenende
- Unter Windows findet beim Einfügen des B eine Änderung in Zeile 2 (Zeilenumbruch) und 3 („B“) statt.
- Unter Linux ist das Ende der zweiten Zeile bereits markiert und die folgende Änderung wirkt sich nur auf Zeile 3 aus.



Vermutung I



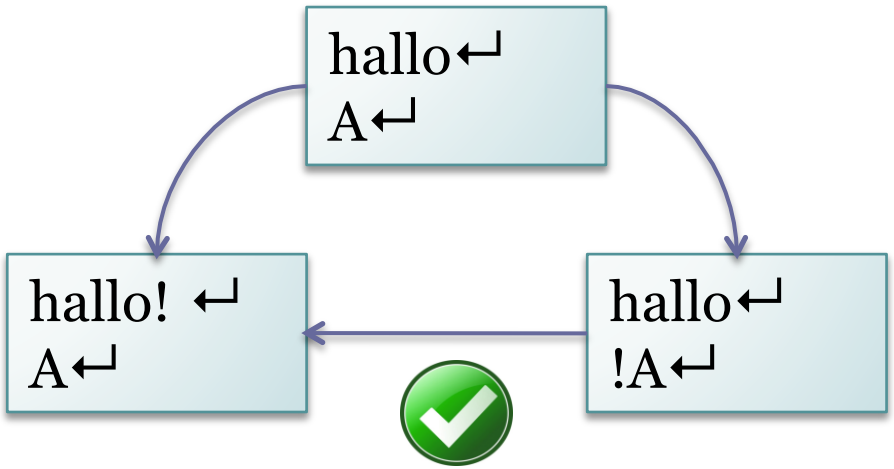
Zeilenwechsel und Ausrufezeichen = Änderungen in gleicher Zeile



Allerdings besteht das Problem auch hier
→ Vermutung widerlegt

Vermutung II

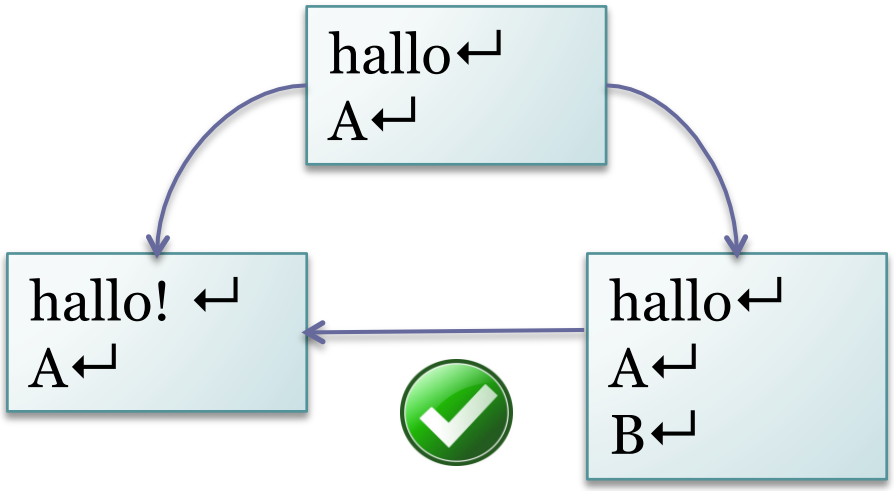
- Änderungen dürfen nicht direkt aufeinanderfolgend sein (am Ende und am Anfang der nächsten Zeile)



Änderungen ebenfalls direkt aufeinanderfolgend, allerdings keine neue Zeile
→ Vermutung widerlegt

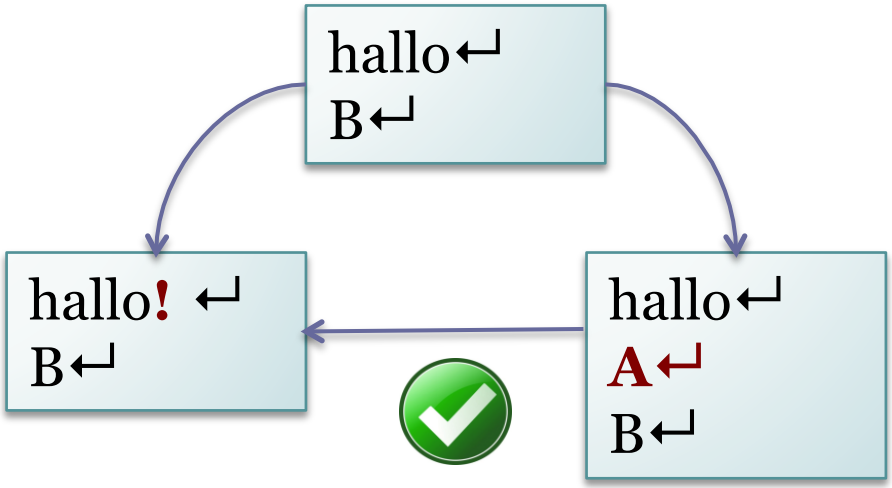
Vermutung III

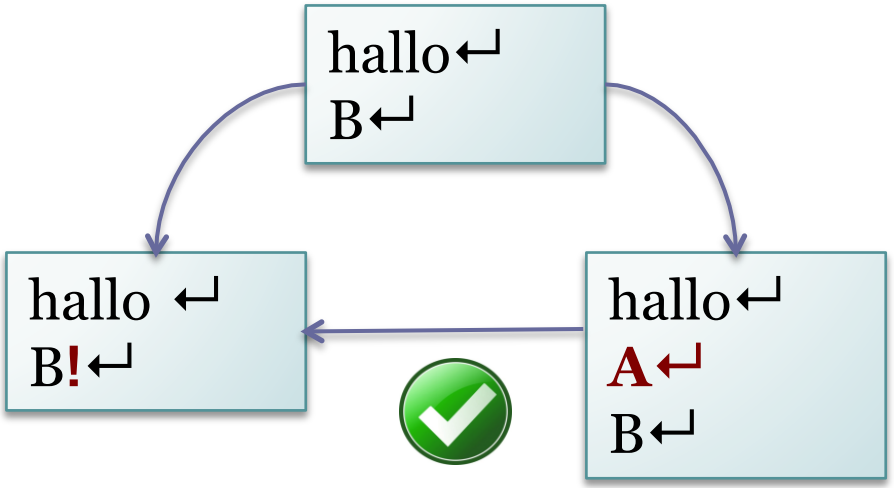
- Wenn eine neue Zeile eingefügt wird, darf sich die vorherige Zeile nicht ändern
  - A ist nun statisch und ändert sich nicht. Die Änderungen erfolgen in Zeile 1 und 3.

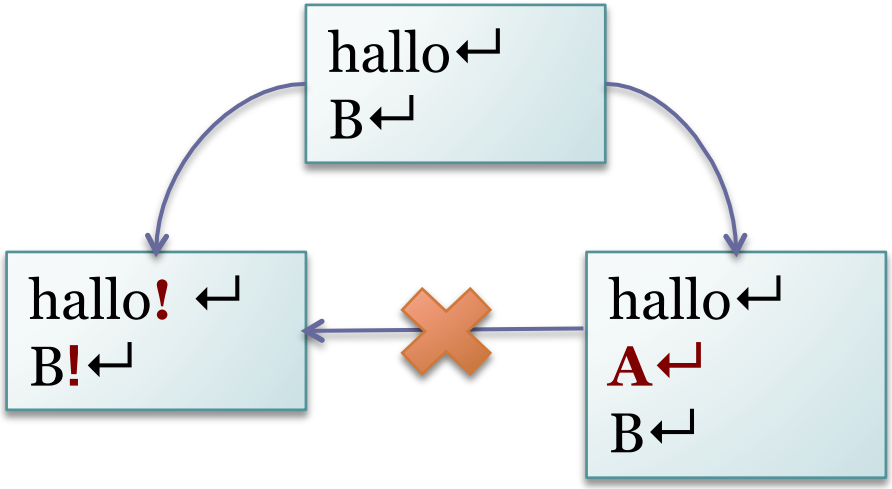


Neue Zeile, aber unveränderte Zeile dazwischen

Weiterer Test und Schlussfolgerung







→ Beim Einfügen einer neuen Zeile muss entweder die vorherige oder die nachfolgende unverändert bleiben.

Merge-Algorithmen

- … unterscheiden sich
- Mit kdiff ist folgender Merge problemlos möglich (schlägt bei SVN fehl)