Software Engineering II

2 Konfigurationsmanagement

SS 2020

Prof. Dr. Dirk Müller

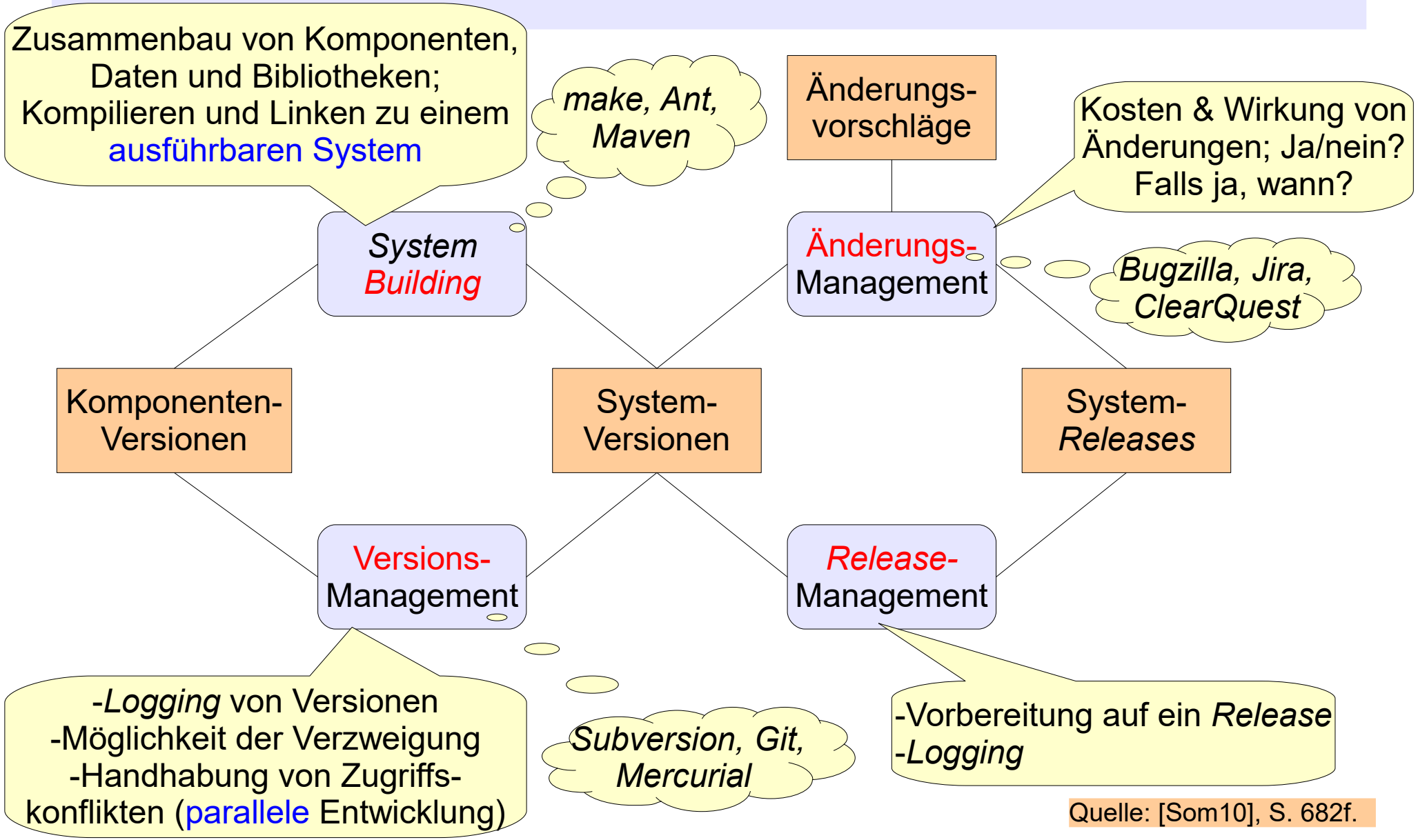
Konfigurationsmanagement

- KM als Oberbegriff für
  - Versionsmanagement
  - *Release*-Management
  - Änderungsmanagement
  - *System Building*
- immer größere Bedeutung
  - *David Parnas* beschrieb SW-Technik als „*multi-person construction of multi-version programs*“.
  - verteiltes, paralleles Arbeiten an Projekten
  - Gute Werkzeugunterstützung ist grundlegend.
  - Beispiel: *Linux*-Kernel-Versionen durch *Git* verwaltet
- Ursprünge und weitere Verwendung in anderen Branchen
  - Militär, Automobilbau [4], S. 5
- hier: *Software Configuration Management* (SCM)

Motivation und Grundidee

- keine stabilen Anforderungen => verschiedene Versionen
- Ergebnis der Nutzung agiler Entwicklungsmethoden: verschiedene Versionen der Implementierung, der Testfälle, des Handbuchs, etc.
- bei komplexen Systemen: Gefahr
  - vergeblichen Aufwands beim Ändern der falschen Version
  - der Auslieferung einer falschen Version an Kunden
  - nicht mehr auffindbaren benötigten Quellcodes im Speicher
- "*Configuration management is the management of an evolving software system.*" [Som10], S. 702
  - Fokus auf die Evolution, nicht auf Prozessmanagement oder Qualitätssicherung
  - Änderung von SW in einer kontrollierten Art
  - *Logging* (Metadaten zu durchgeführten Änderungen)

Aktivitäten



Quelle: [Som10], S. 682f.

Was soll gemanagt werden?

- Software-Element
  - maschinenlesbares Artefakt (Datei) zugehörig zu einem SW-Projekt (Entwurf, Quellcode, Testdaten, Handbuch, etc.)
- Quell-Element
  - manuell erstellt, typischerweise mittels eines Editors

sollte gemanagt werden

auch: *Build*-Skripte

- abgeleitetes Element
  - vollautomatische Erstellung durch Werkzeuge

normalerweise nicht gemanagt

- Konfigurations-Item
  - Artefakt, das unter Konfigurationskontrolle gestellt wurde
  - eindeutiger Name
  - typischerweise verschiedene Versionen
- Version
  - Instanz eines Konfigurations-Items mit einem Zeitstempel
  - eindeutige ID: Name des Konfigurations-Items + Versionsnummer

Konfigurationen und *Baselines*

- Konfiguration
  - Menge von Versionen mit kompatiblen Schnittstellen für einen bestimmten Zweck
  - neue Version eines einzigen Konfigurations-Items führt zu einer neuen Konfiguration
- *Baseline*
  - bewährte Konfiguration, i.e., Stabilität bestätigt
  - ermöglicht einfache Wiederherstellung eines früheren Systemstatus
- *Release*
  - Konfiguration mit dem Zweck der Auslieferung an den Kunden, typischerweise eine *Baseline*

Verwaltung der Konfigurations-Items

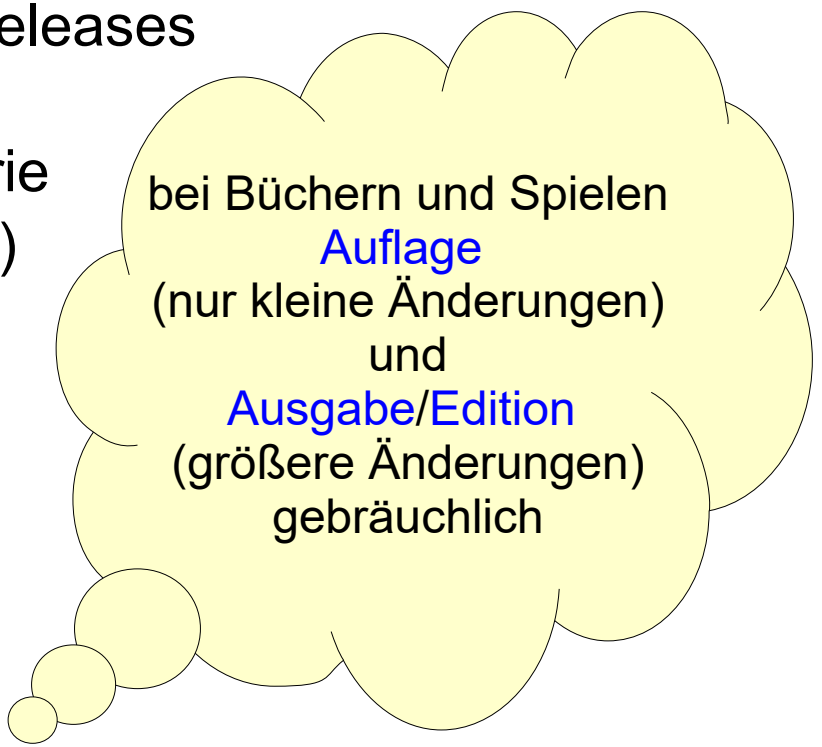
- Instanzen der Konfigurations-Items in einem *Repository* (spezialisierte Datenbank, verwaltetes Verzeichnis)
- Aufbau eines *Repository* entspricht der Projektstruktur, aber ohne temporäre Ordner/Dateien
- effektive Arbeit nur mittels herstellerspezifischer Formate im *Repository* möglich
- *Repositorys* oft über eine Schnittstelle direkt in die Entwicklungswerkzeuge (IDEs) integriert
- *Repositorys* setzen häufig intern Versionskontrolle ein:
  - nach jeder Änderung an einer Datei wird eine neue Version im *Repository* gespeichert
  - nach und nach entsteht eine Versionshistorie, alle jemals vorgenommenen Änderungen werden dokumentiert
  - Löschen als Spezialfall einer Änderung

Typische Arbeitsschritte

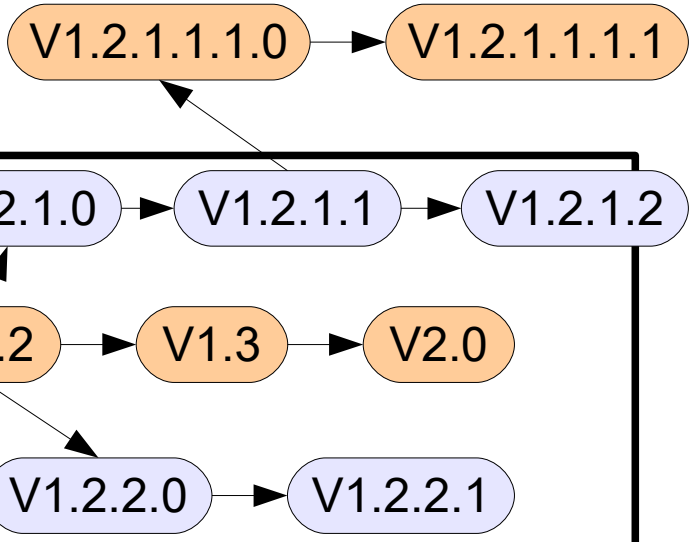
- Arbeit im Projekt nicht mit Dateien im *Repository*, sondern mit Arbeitskopien in einem lokalen Arbeitsbereich (ebenfalls vom Versionsverwaltungssystem verwaltet)
- *Repository* liefert auf Anforderung eine Arbeitskopie (*Check-out*)
- *Repository* aktualisiert die Arbeitskopie bei Bedarf (*Update*)
- Bestätigung von Änderungen zur Übertragung in das *Repository* (*Check-in*, *Commit*)

Versions-Management

- Zweck
  - Identifikation von Versionen und Releases
  - Speicher-Management
  - Aufzeichnung der Änderungshistorie
  - unabhängige Entwicklung (parallel)
  - Projektunterstützung
- Prinzipien
  - Delta-Kodierung
  - pessimistisch vs. optimistisch
- Typen
  - lokal
  - zentral
  - verteilt



Versionierungs-Schemata auf Systemebene



- am gebräuchlichsten
  - release.level
- Zweige
  - release.level.branch.sequence
- Zweig eines Zweigs
  - release.level.branch.sequence.branch.sequence
- feinste Granularität: *Build*-Nummer (oder *Build*-Datum)
- Reifegrad
  - Alphaversion, Betaversion, *Release*-Kandidat (RC), *Release* (Endversion)
- Markennamen mit Versionen können Lücken bzw. Sprünge aufweisen
  - Aufholen von Nummerierungsvorsprüngen der Konkurrenz

Quelle: [Bal08], S. 436f.

Semantic Versioning

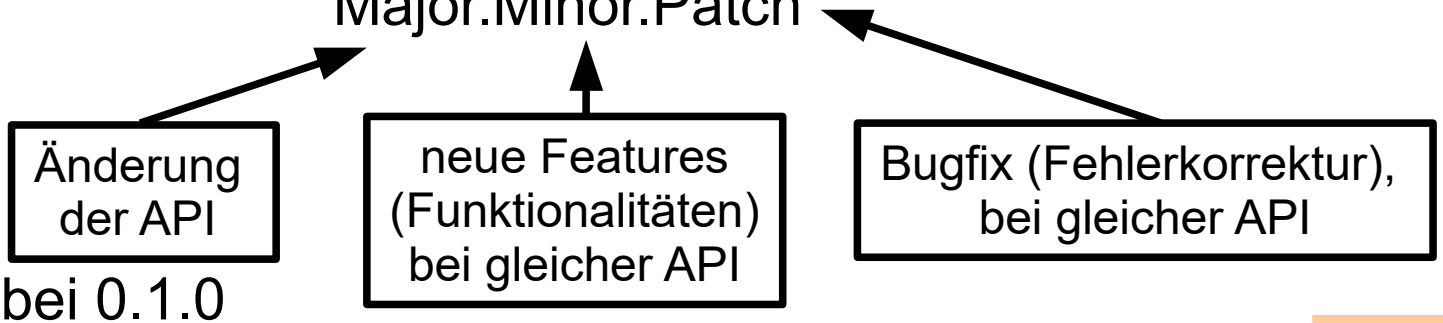
- **Risiken und Probleme bei Projekten mit vielen Abhängigkeiten**
  - *Version Lock*: Unfähigkeit ein Paket zu aktualisieren, ohne dass alle abhängigen Pakete dieses Pakets ebenfalls aktualisiert werden müssen
  - *Version Promiscuity*: Paket gibt vor, mit mehr zukünftigen Versionen seiner abhängigen Pakete kompatibel zu sein, als angemessen ist

zu strikt

zu lasch

Dependency Hell

- Versuch der Standardisierung
  - Major.Minor.Patch

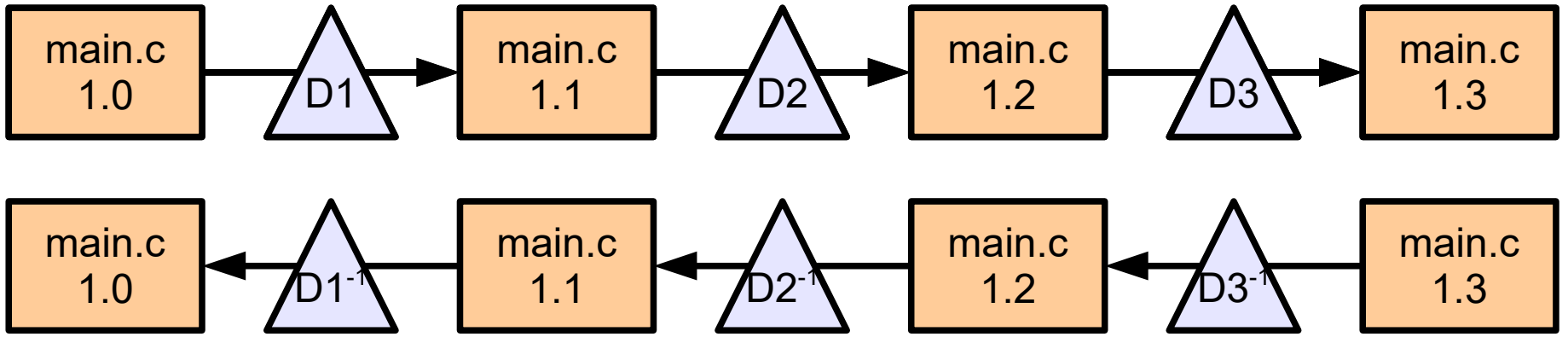


  - Start bei 0.1.0
  - jeweils keine führenden Nullen zugelassen

Quelle: [7]

Delta-Codierung

- neuste Version vollständig gespeichert (Rückwärtsdelta)
- ältere Versionen durch schrittweise Anwendung der inversen Delta-Operation zu erhalten

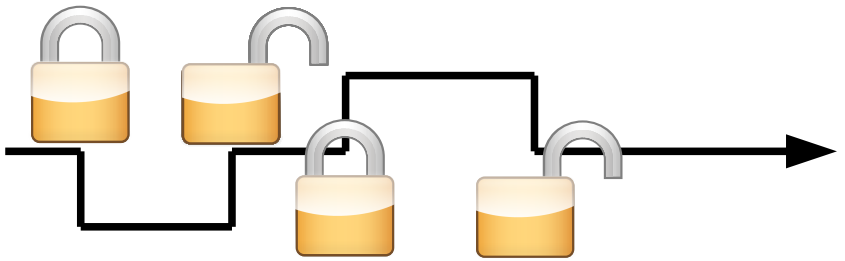


- gute Kompressionsrate nur für sich wenig unterscheidende aufeinanderfolgende Versionen
  => für Binär-Dateien zunächst nicht geeignet
  - aber: moderne Algorithmen in neueren Tools wie z. B. *Subversion*

Pessimistisch vs. optimistisch

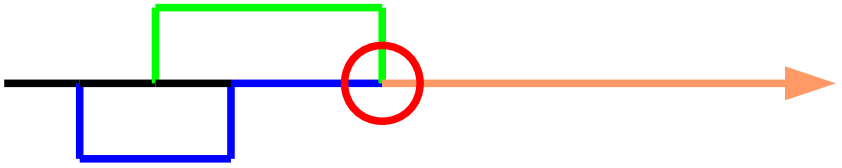
Pessimistisch:
Lock-Modify-Write

- nur 1 Nutzer kann eine Datei zu einer bestimmten Zeit editieren (sequenziell)
- explizites Sperren/Entsperren
  - Blockierung aller anderen Nutzer



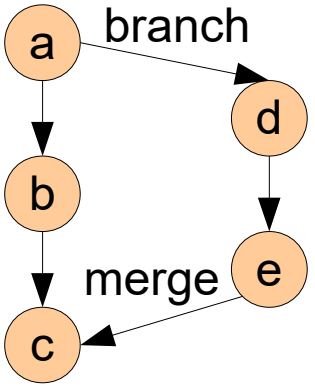
Optimistisch:
Copy-Modify-Merge

- Nutzer können jederzeit eine Arbeitskopie erhalten
- bei sich widersprechenden Änderungen: Varianten müssen verschmolzen werden
  - manuell
  - automatisch, problematisch für Binärdateien



Drei-Wege-*Merge*

- traditioneller Ansatz des automatischen Verschmelzens
- zwei Versionen einer Datei mit 1 gemeinsamen Vorfahren
- mögliche Resultate: Anzahl der Änderungen bzgl. des Vorfahrens
  - 0 keine: bleibt wie bisher
  - 1 eine: nutze geänderte Version
  - 2 beide:
    - gleiche Änderungen: nutze geänderte Version
    - unterschiedliche Änderungen: typischerweise manuelle Konfliktauflösung (Nutzereingriff)



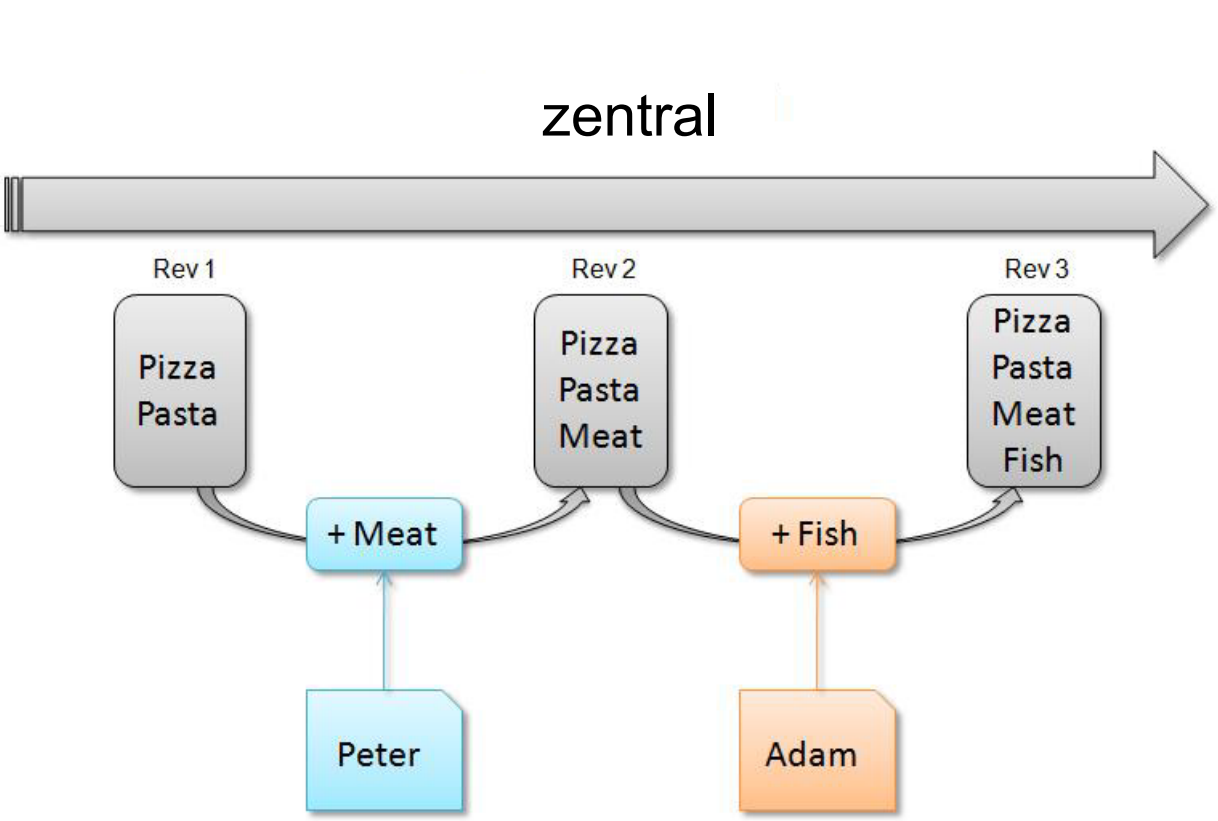
- Rekursives Drei-Wege-*Merge* [3]
  - komplizierte Situation mit mehr als 1 gemeinsamen Vorfahren
  - Konstruktion eines virtuellen Vorfahrens durch Verschmelzung zweier Vorfahren als Vorverarbeitung
  - Versionsbaum muss vollständig bekannt sein
  - z. B. in *Git* verwendet [2]

Lokal – zentral – verteilt

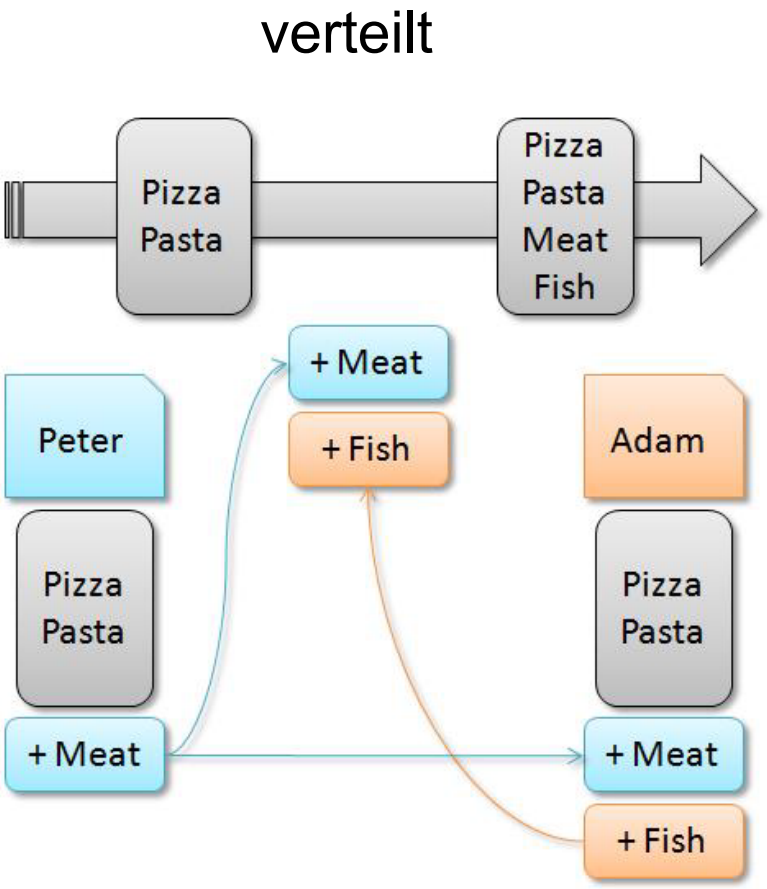
- lokal
  - meist nur auf einzelnen Dateien, Delta-Codierung in Datei selbst
  - pessimistisch/optimistisch hier nicht relevant
  - z. B. *Source Code Control System* (SCCS, 1972), *Revision Control System* (RCS, 1985)
- zentral
  - *Client-Server*-Architektur
  - *Repository* mit einer linearen Versionshistorie auf dem *Server*
  - pessimistisch and optimistisch unterstützt
  - z. B. *Concurrent Versions System* (CVS, 1989), *Subversion* (2004)
- verteilt
  - verteiltes Repository mit DAG als Historie
  - offizielles Repository (z. B. auf https://github.com)
  - nur optimistisch, schnelles fein-granulares Verschmelzen möglich
  - z. B. *Git* (2005) [2], *BitKeeper* (2000), *Mercurial* (2005)

Praxis der agilen SW-Entw. der kontinuierlichen Integration gut unterstützt

Zentraler und verteilter Ansatz



Sequenzialisierung anhand von Zeitstempeln



geänderte Dateien anhand von Checksummen (*Hash*-Werten) erkannt

Quelle: [5]

Verteilt vs. zentral

- Vorteile verteilter Versionskontrolle gegenüber zentraler
  - Verzweigungen und Zusammenführungen (*Merges*) einfacher
  - Projektmitarbeit ohne explizite Schreibrechte auf Server möglich
  - Arbeit kann offline erledigt werden, da gesamte Historie lokal vorhanden => unabhängiger vom Netzwerk, schneller
  - robuster, zentrales *Repository* nicht mehr *Single Point of Failure*
- Nachteile
  - Verzweigungen zu einfach, Gefahr der übermäßigen Anwendung
  - Netzwerk wird anfangs stärker beansprucht, da gesamte Historie mit heruntergeladen werden muss
  - evtl. unübersichtlich mit (zu) vielen *Repositorys*

Quelle: [5]

Ansible

- 2012 erschienen, aktuelle Version 2.9.6 vom 15.03.2020
- für *Unix*-artige BS, seit Version 1.7 indes auch Kontrolle von *Windows*-Hosts unterstützt (über *PowerShell*-Befehle)
- quelloffenes Automatisierungswerkzeug
- Anwendung: Administration, Softwareverteilung, Konfigurationsmanagement
- Referenzen: *Fedora-Linux*-Distribution, *HP Deutschland*, Universität Thessaloniki, *Spotify*, *NASA*
- 2015 Übernahme durch *RedHat Inc.*
- in *Python* geschrieben
- seit 2016 Konfiguration von Netzwerk-HW (*Arista*, *Cisco*, *Juniper*)

Ansible: Einführungsbeispiel

- *OpenSSH* mit *Python* ab 2.4
- parallele Verteilung an Hosts möglich => sehr schnell
- kompatibel zu *Kerberos* und *Docker*
- *Inventory*-Datei mit einer Liste aller zu verwaltenden Rechner

```
[atomic]
192.168.100.100
192.168.100.101
[webserver]
192.168.1.110
192.168.1.111
```

- Was macht dann dieser Einzeiler?

```
ansible -i inventory all -a "useradd -G admins foo"
```

- Antwort: Anlegen des Benutzers `foo` und Aufnahme in die Gruppe `admins` auf allen Rechnern
- *Playbooks* zur Beschreibung/Sicherstellung eines Zustands

Quelle: [6]

Ansible Playbooks

```
---
- hosts: atomic
  tasks:
  - name: install kubernets yum repo file
    template: src=/srv/kubernetes.conf
dest=/etc/yum.repos.d/kubernetes.conf
  - name: ensure kubernetes is at the latest version
    yum: pkg=kubernetes state=latest
  - name: ensure kube-apiserver is active
    service: name=kube-apiserver state=started
```

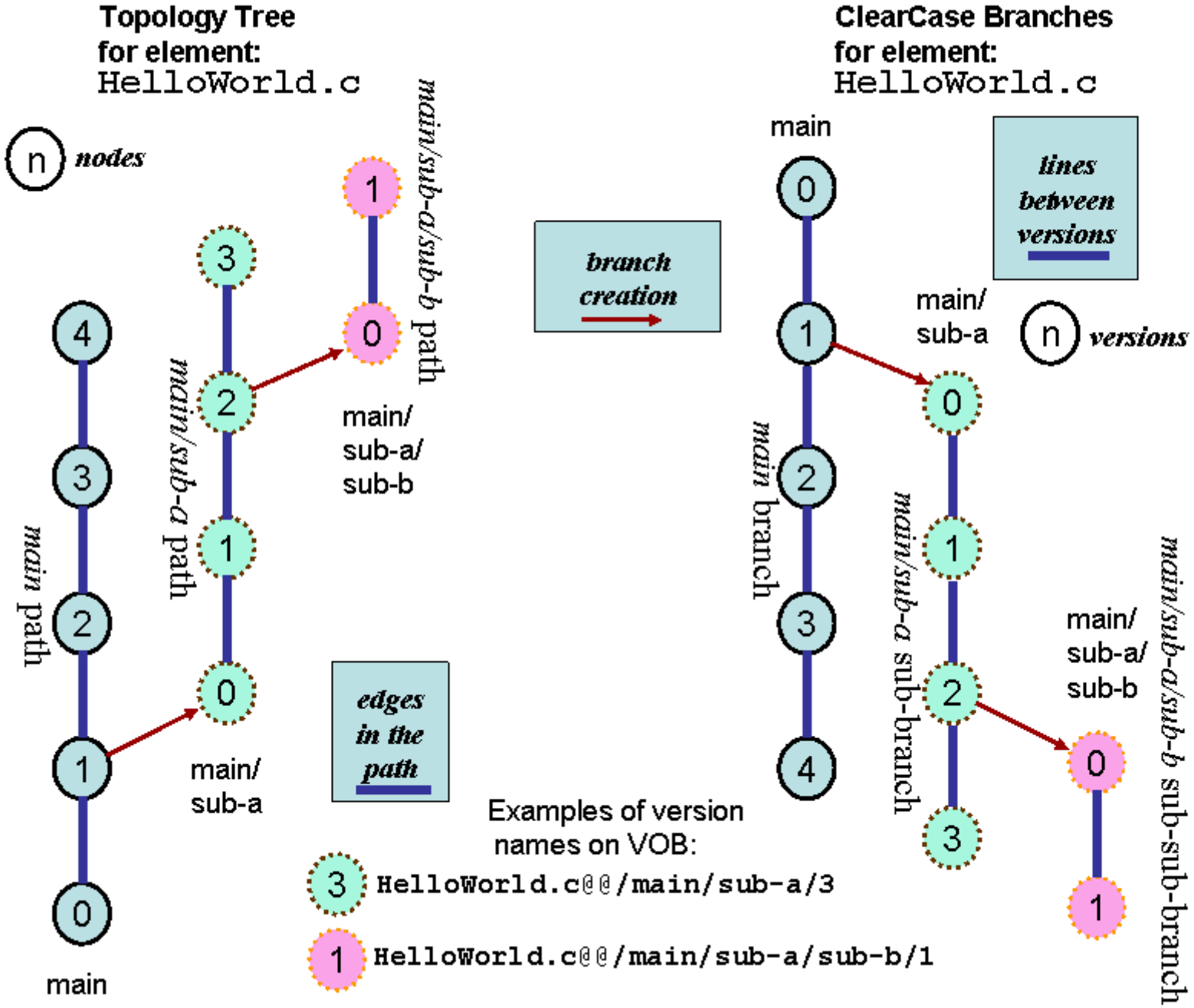
```
# ansible-playbook -i inventory atomic.yaml
PLAY [all] ********************************************************************
GATHERING FACTS ***************************************************************
ok: [192.168.100.100]
ok: [192.168.100.101]
TASK: [install kubernets yum repo file] ***************************************
changed: [192.168.100.100]
changed: [192.168.100.101]
TASK: [ensure kubernetes is at the latest version] ****************************
changed: [192.168.100.100]
changed:: [192.168.100.101]
TASK: [ensure kube-apiserver is active] ***************************************
changed:: [192.168.100.100]
changed:: [192.168.100.101]
PLAY RECAP ********************************************************************
192.168.100.100            : ok=4   changed=3   unreachable=0   failed=0
192.168.100.101            : ok=4   changed=3   unreachable=0   failed=0
```

Quelle: [6]

IBM Rational ClearCase

- 1992 *Atria* Software für *Unix*, jetzt auch *Windows*, *Linux*
- Integriertes kommerzielles Werkzeug für *Versions*-, *Build*- und *Release*-Management
  - zentrale Versionskontrolle
- 2 Modi für Anfänger und Experten
  - *Base ClearCase*
  - *Unified Change Management* (UCM), Teil des RUP
- weg vom *Repository*-Ansatz (*Snapshot*-Sicht)
  - jetzt: dynamische Sicht
- Zugriff auf *Versioned Object Base* (VOB) durch ein sogenanntes *Multi Version File System* (MVFS)
  - kontinuierlichen Integration gut unterstützt
- Terminologie nicht intuitiv und verwirrend [1]

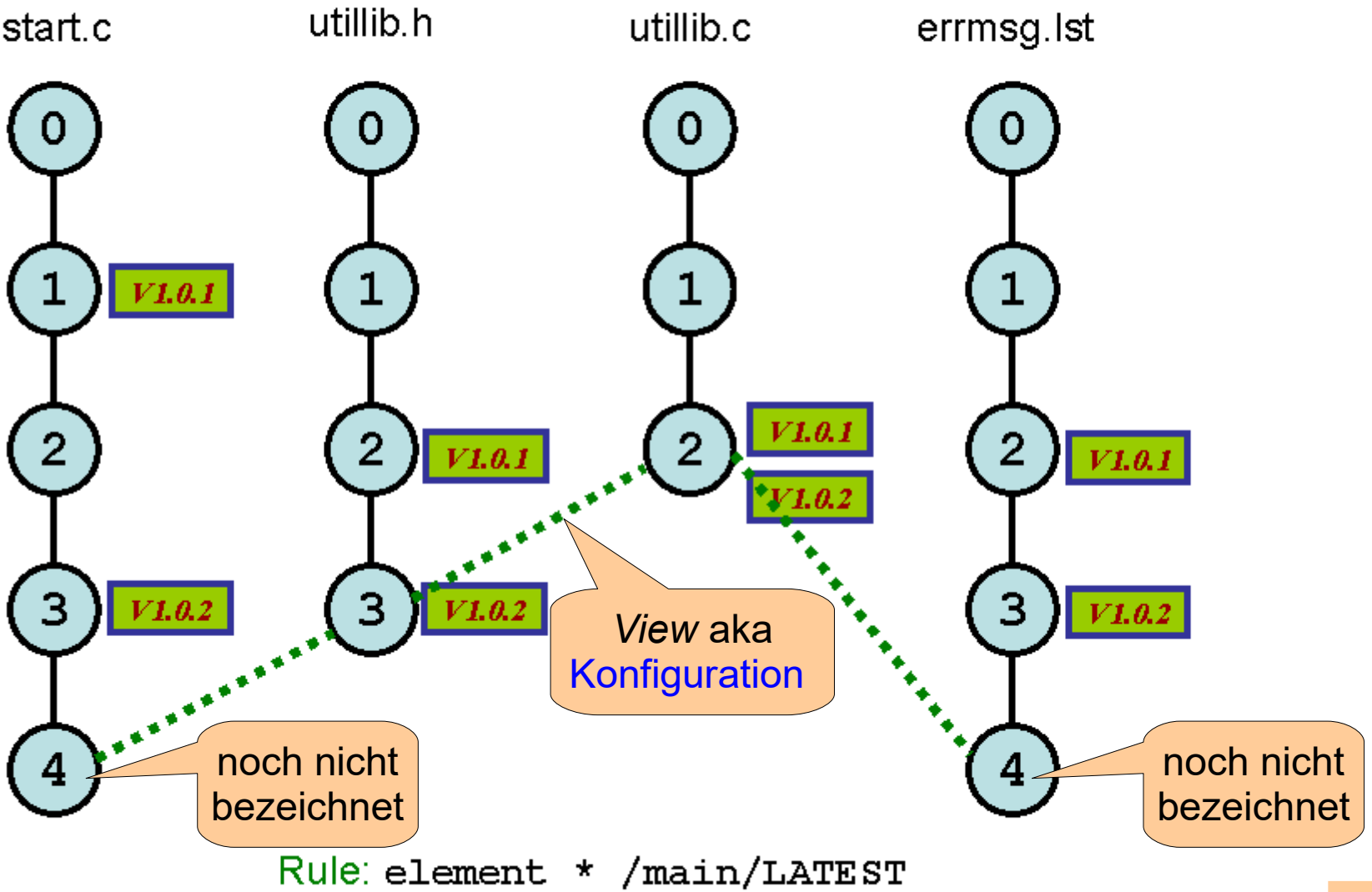
ClearCase Branches



Quelle: [1]

ClearCase View LATEST

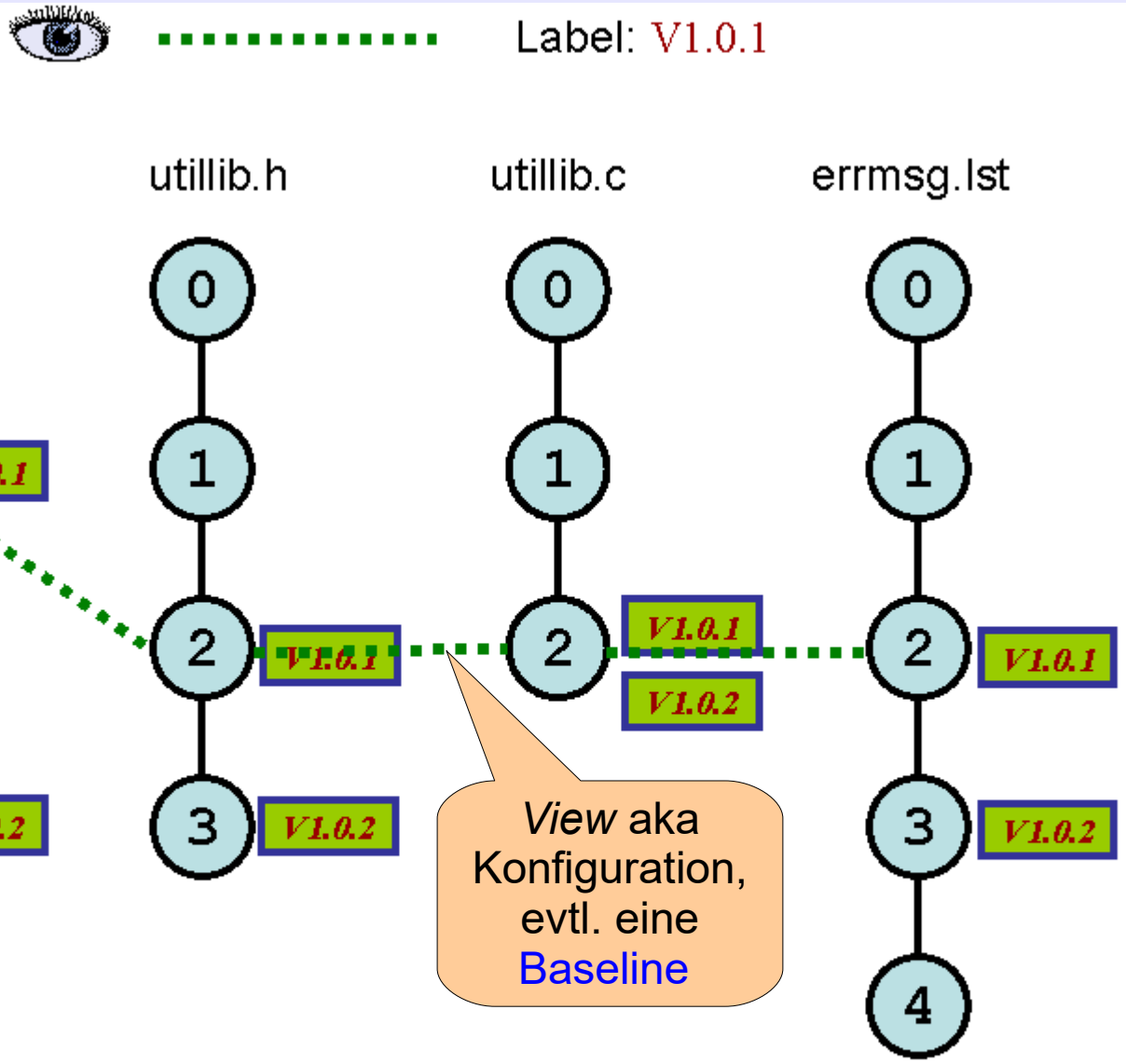
LATEST



Rule: `element * /main/LATEST`

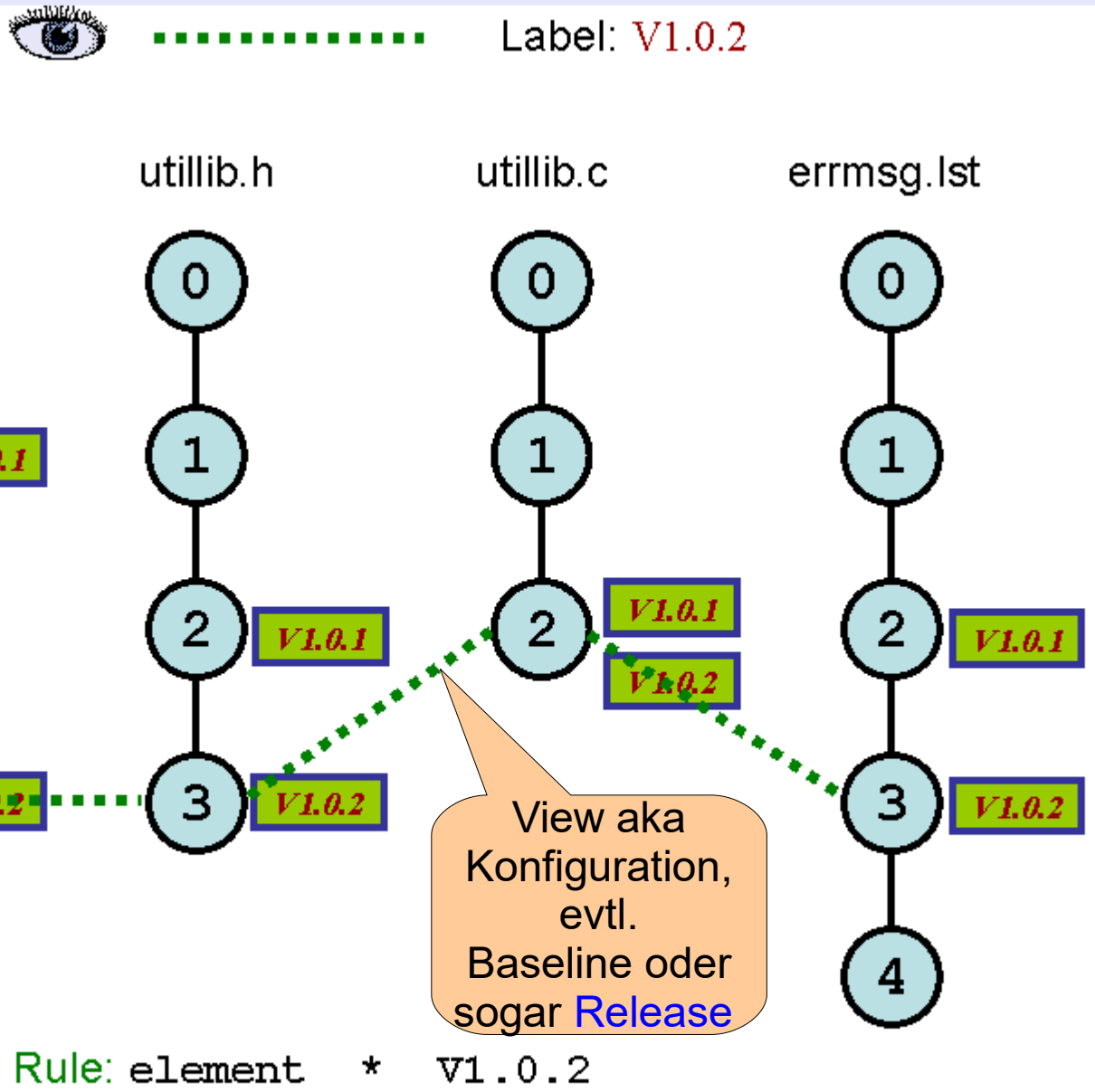
Quelle: [1]

ClearCase View V1.0.1



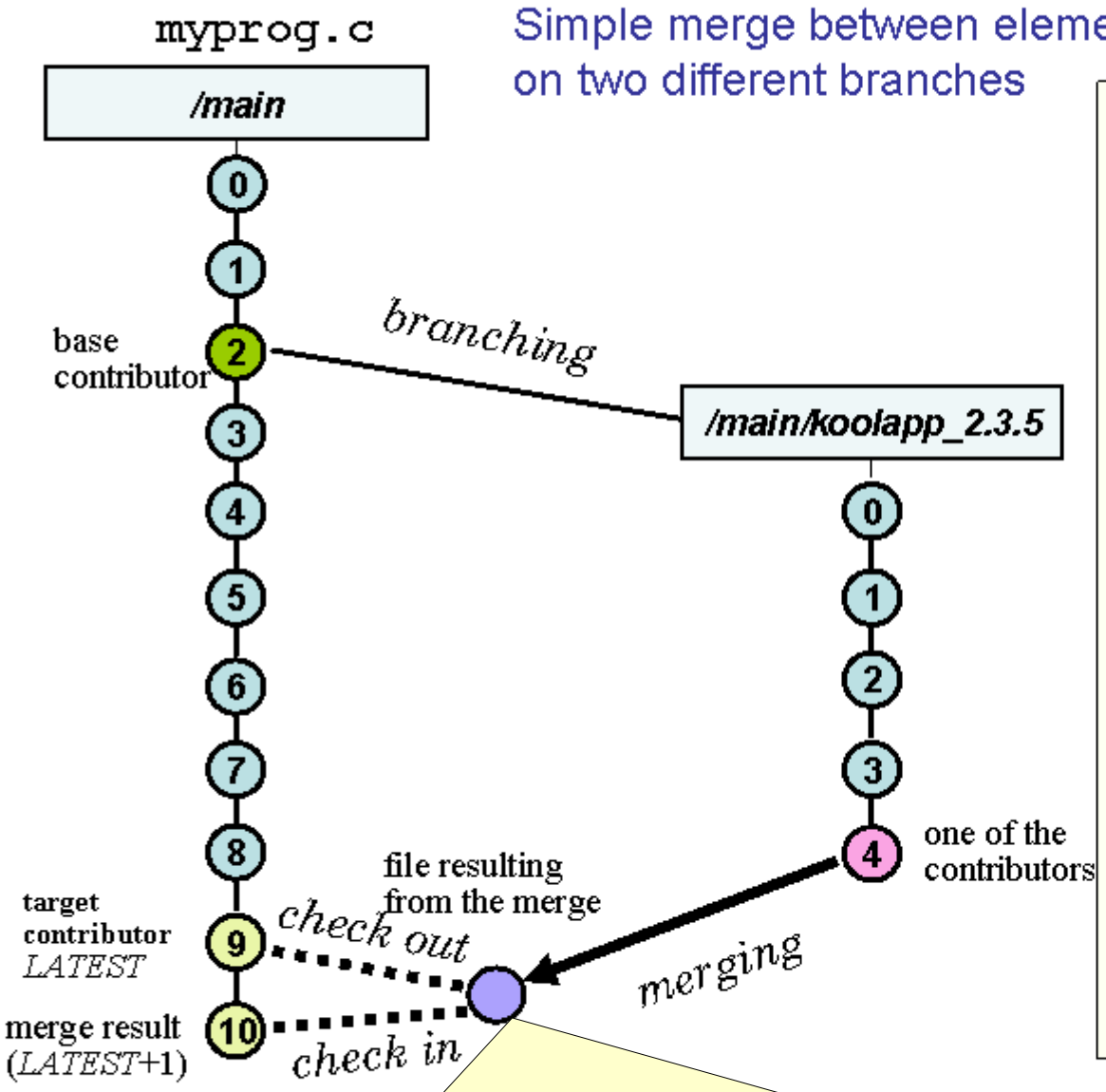
Quelle: [1]

ClearCase View V1.0.2



Quelle: [1]

ClearCase Merging

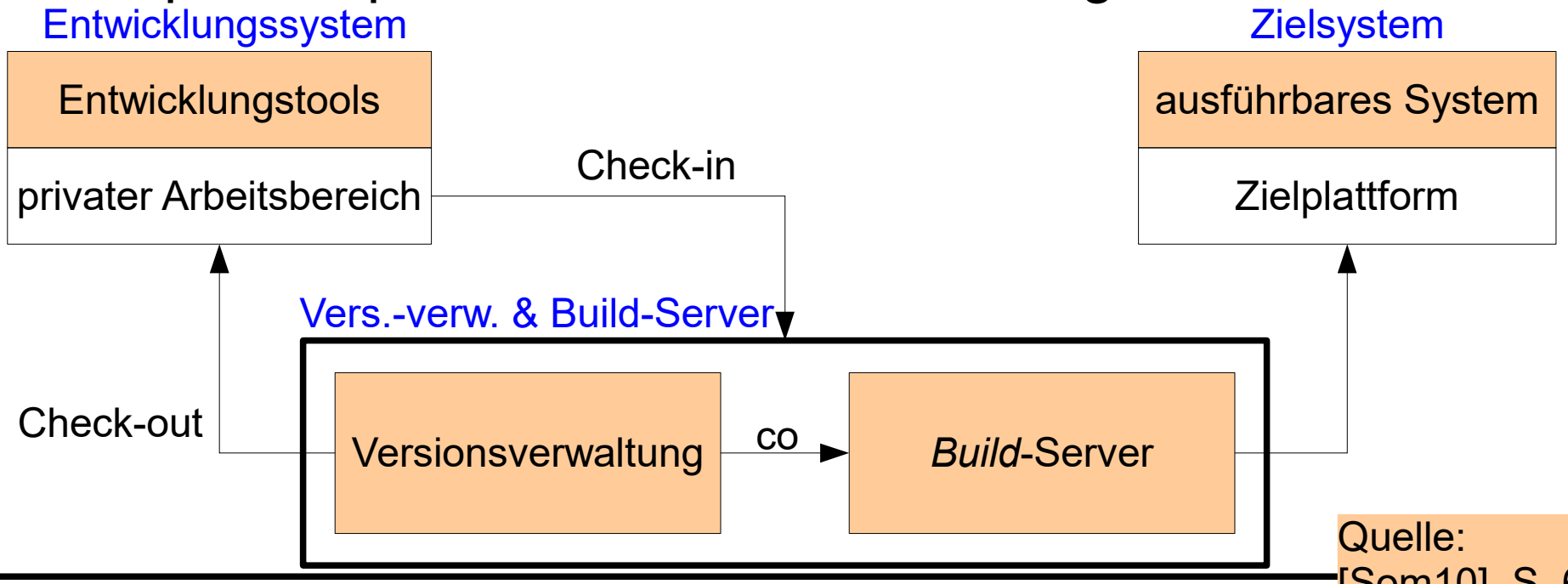


Quelle: [1]

-ähnlich zu UNIX-Kommandos *diff* und *patch*
-nicht nur zeilenweise, Nachbarzeilen (Kontext) mit berücksichtigt
-Konflikte liefern Fragen, aber auch ohne Fragen falsche Resultate mgl.

System Building

- Kompilieren und Linken von Komponenten, Daten und Bibliotheken zu einem ausführbaren System
- enge Zusammenarbeit mit Versionsverwaltung sinnvoll
  - Aus-checken entsprechender Konfigurations-Items
- kompliziert, potenziell 3 Plattformen möglich



Quelle: [Som10], S. 694

Ablauf eines *Build*-Prozesses

1. Quellelemente ermitteln
2. Produkt aus den Quellelementen erstellen
   → Aufrufe von Compiler und Linker,
   ggf. Generatoren (modellgetrieben)
3. Produkt prüfen
   → automatisierte Modultests mit *Mock*-Objekten
   (noch kein Integrationstest möglich)

   Minimalforderung an jeden *Build*-Prozess
4. Produkt ausliefern
   → Erstellen eines Auslieferungspakets (z. B. EAR-/WAR-Dateien)
   ggf. Installation und Integrationstests
5. Ergebnisse dokumentieren
   → Schritte des *Build* in Log-Datei dokumentieren
   ggf. Konfiguration anlegen sowie Benachrichtigung über E-Mail und Projekt-Homepage

Quelle: [4], S. 46

3 Arten von *Builds*

- Entwickler-*Build*
  - im privaten Arbeitsbereich
  - schnell und einfach, manches teilweise oder übersprungen
  - oft in IDEs (z.B. *Eclipse*) integriert, im 5-Minutentakt möglich
- Integrations-*Build*
  - Frage nach Verträglichkeit der Änderungen
  - automatisches oder manuelles Testen
  - für alle parallel aktiven Entwicklungspfade
  - agile Methoden: alle 30 min, plangetrieben: täglich
- *Release-Build*
  - Zweck ist die Auslieferung an Kunden
  - manchmal explizite Werkzeugunterstützung (*Maven*)
  - manchmal nur Umdeklarierung eines Integrations-*Builds* (*Ant*)

Button in IDE

auf der Kommandozeile, imperativ, regelbasiert oder deklarativ

Quelle: [4], S. 47f.

Signaturen

- Zweck: Zuordnung von Objekt-Code zu Quellcode, um unnötige Kompilierläufe zu sparen

  Ansätze

- Zeitstempel (*Timestamp*, TS)
  - sekundengenauer Zeitpunkt des Schreibens in Dateisystem
  - Regel: Falls TS(Quelle) > TS(Ziel), dann Neukompilierung.
- Checksummen der Quell-Elemente
  - Checksumme (CS) einer Datei mit Quellcode wird unmittelbar vor Kompilierung berechnet, möglichst schnell und änderungssensitiv
  - Markierung (*Tagging*) des Quellcodes und des zugehörigen Ziels mit dieser Checksumme
  - Regel: Falls NOT ∃ Ziel: *Tag*(Quelle)==*Tag*(Ziel), dann Neukompilierung.
  - Vorteil: parallele Kompilierung verschiedener Versionen möglich

Quelle: [Som10], S. 696

Zusammenfassung

- KM als Mgmt. eines sich entwickelnden SW-Systems
  - Versions-, *Release*-, Änderungs- und *Build*-Management
- Versionsmanagement
  - pessimistisch mit *Locks*, optimistisch mit (automatischem) *Merge*
  - Drei-Wege-*Merge* von 2 Dateien durch gemeinsamen Vorfahren
  - lokal, zentral, verteilt; entscheidende Vorteile bei verteiltem Ansatz
- *System Building*
  - bis zu 3 verschiedene Plattformen, 3 Arten von *Builds*
  - automatisierte Modultests als Minimalforderung
  - Automatisierung in IDE und auf Kommandozeile
  - Signaturen, um unnötiges Kompilieren zu vermeiden
- Trend
  - agile Praktiken wie kontinuierliche Integration durch verteilte Versionskontrolle (z. B. *Git*), Mehrversionen-Dateisystem (*ClearCase*) oder automatisierte *Builds* immer besser unterstützt

Literatur

[1] Jan Labanowski: „ClearCase lingo demystified“, 2009
http://www.ccl.net/cca/software/UNIX/clearcase/index.shtml, Download am 4.5.2014

[2] Lars Vogel: „Git Tutorial“, Version 5.6, 2014
http://www.vogella.com/tutorials/Git/article.html,
Download am 4/5/2014

[3] Pablo: „Merge recursive strategy“, The Plastic SCM blog, 2011,
http://codicesoftware.blogspot.com/2011/09/merge-recursive-strategy.html
Download am 5/5/2014

[4] Gunther Popp: „Konfigurationsmanagement mit Subversion, Maven und Redmine“, dpunkt-Verlag, 4. Auflage, 2013

[5] Carl Fredrik Malmsten: „Evolution of Version Control Systems“, Bachelor of Applied Information Technology Thesis, University of Gothenborg, 2010

[6] Thorsten Scherf: „Konfigurationsmanagement mit Ansible - Zentral gesteuert“, ADMIN-Magazin, Online-Artikel, 26.01.2015, Download am 01.03.2018,
http://www.admin-magazin.de/Online-Artikel/Konfigurationsmanagement-mit-Ansible/(language)/ger-DE

[7] Tom Preston-Werner: „Semantic Versioning 2.0.0“, Download am 10.02.2020,
https://semver.org/lang/de/