Teil 2.2: Konfigurationsmanagement

Literatur

- The source code control system by Marc Rochkind (http://ieeexplore.ieee.org/document/6312866/)
- RCS – A System for Version Control by Walter Tichy (http://www.gnu.org/software/rcs/tichy-paper.pdf)
- The Subversion-book by Ben Collins-Sussman et al. (http://svnbook.red-bean.com)
- Pro Git by Scott Chacon and Ben Straub (https://git-scm.com/book/en/v2)

Agenda

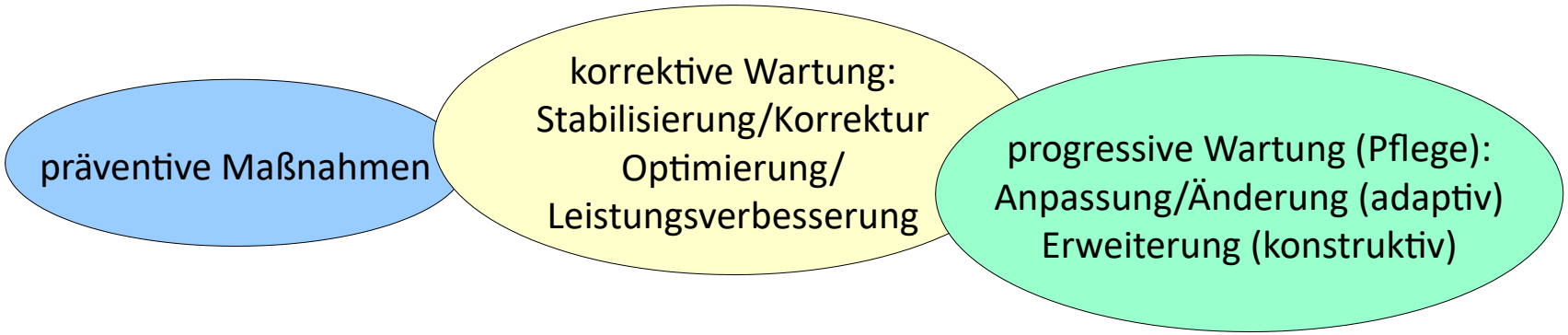
- Source Code Management
- Build-Automatisierung & Dependency Management
- Release-Strategien

Ziele

- Entwicklung eines Verständnis vom Objekt *Programmcode* sowie ähnlicher Artefakte
- Zusammenhang von Anforderungen und Quellcode
- Verständnis für die Herausforderungen beim Warten von Software

Software-Wartung

- In der Wartungs- & Pflegephase lassen sich durchzuführende Aktivitäten in
- folgende Gruppen einteilen:



- Wartungsanforderungen bzgl. Dringlichkeit:
  - sofort (operativ): schwerwiegende Softwarefehler, die die weitere Nutzung in Frage stellen
  - kurzfristig: Code-Korrekturen
  - mittelfristig: Systemerweiterungen (Upgrades)
  - langfristig: Restrukturierung (System Redesign)

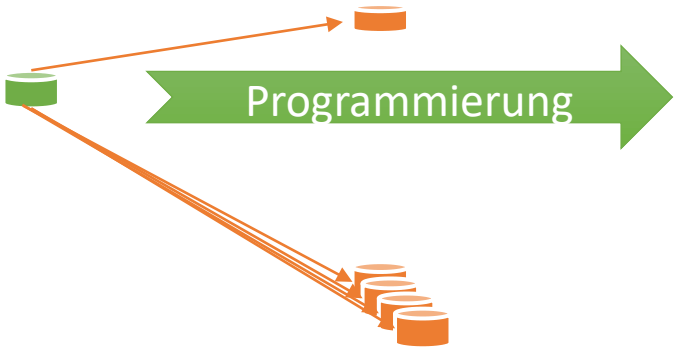
Um was geht es?

Der Lebenszyklus von Quellcode

Am Anfang gibt es eine Version des Quellcodes

Um was geht es?

Der Lebenszyklus von Quellcode

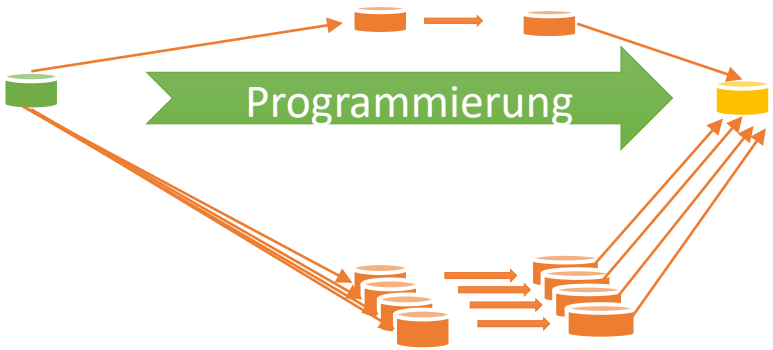


Während der Programmierung ändert jeder Programmierer einen Teil des Systems.

Dazu wird die gemeinsame Code-Basis meistens mehrfach kopiert.

Um was geht es?

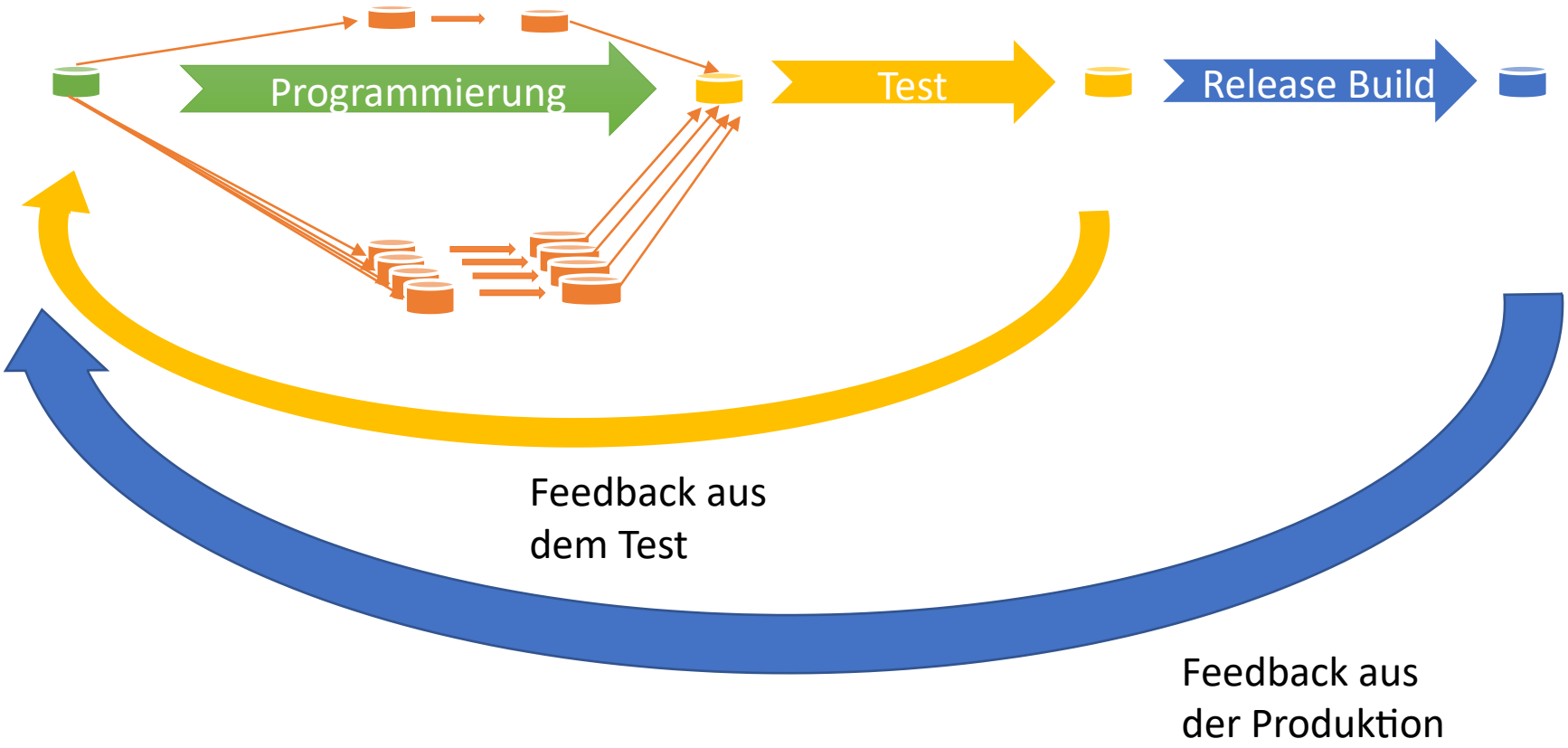
Der Lebenszyklus von Quellcode



Nach einer gewissen Zeit spielen alle ihre Änderungen zurück.

Um was geht es?

Der Lebenszyklus von Quellcode



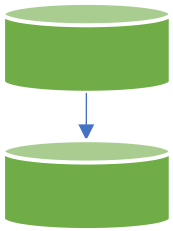
Versionsverwaltung

- **Definition: Der Versionsverwaltung obliegt die persistente Verwaltung der Komponenten.**
- Identifizierung von Versionen: Komponentenname, Versionsnummer, Variante
  - Komponentennamen können Pfadnamen, Surrogates oder URLs sein
- Festlegung aller zu verwaltender Bestandteile, die zu einer Komponente gehören, wie Analyse-Spezifikation, Entwurfsspezifikation, Code der Komponente, Testspezifikation, Dokumentation, …
- Bestimmen der Namenskonventionen und Bezeichnung der Relationen zwischen den Komponenten
- Abspeichern verschiedener Versionen einer Komponente einschließlich ihrer Wiederauffindungspfade
- Bereitstellen beliebiger abgelegter Versionen
- Dokumentieren von Änderungen
- Festlegen und Kontrollieren von Zugriffsrechten
- Verwalten des Komponenten-Repositories

Wie viele Versionen waren das?

Initiale Version

Wie viele Versionen waren das?



Initiale Version

Änderungen von Entwickler 1

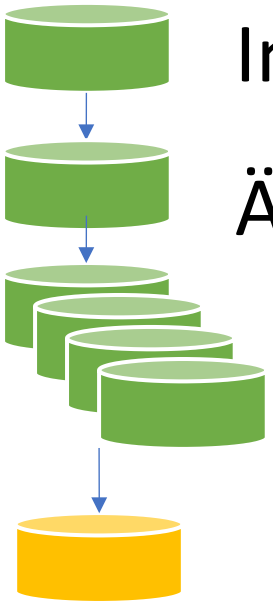
Wie viele Versionen waren das?

Initiale Version

Änderungen von Entwickler 1

Änderungen von Entwickler 2-5

Wie viele Versionen waren das?



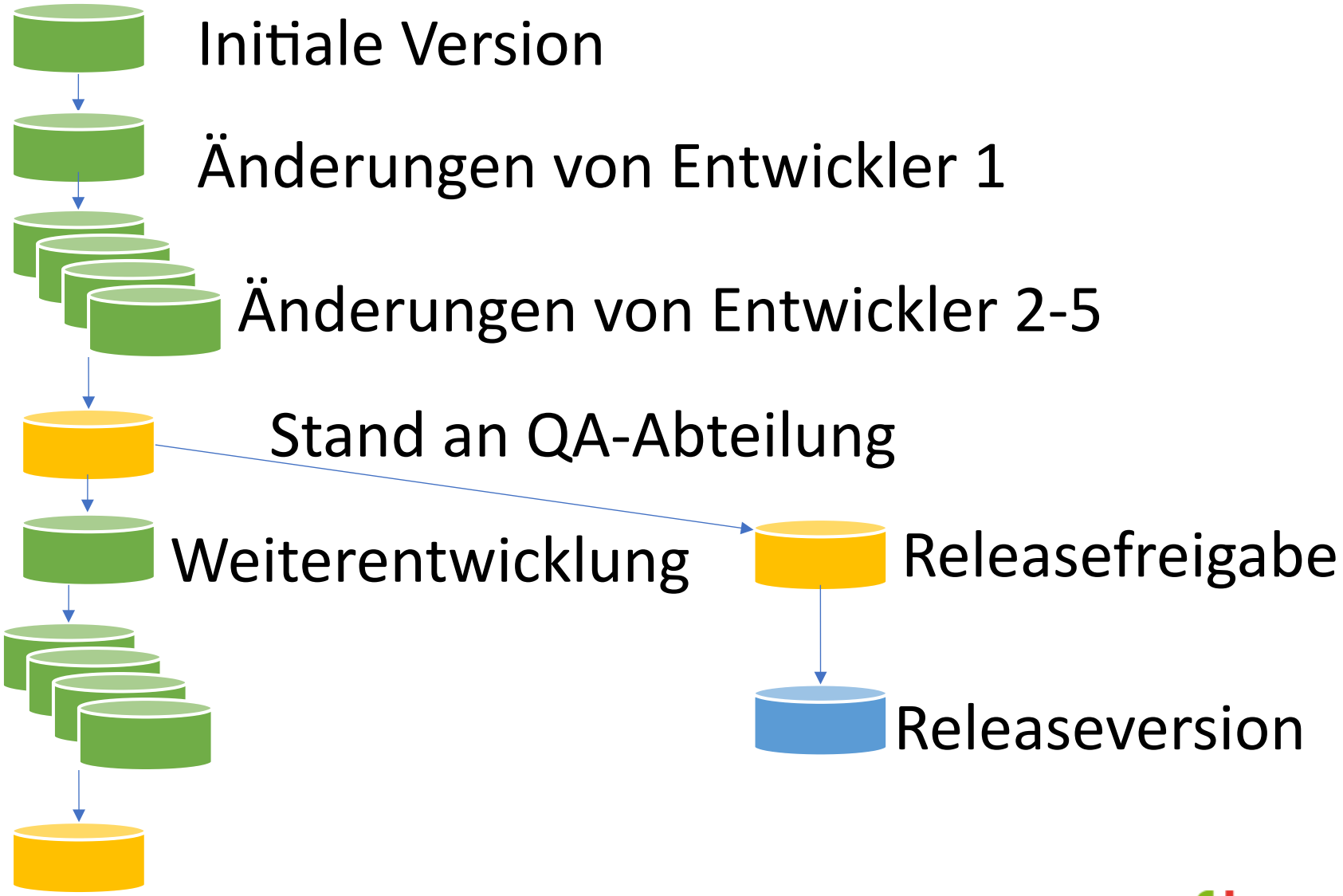
Initiale Version

Änderungen von Entwickler 1

Änderungen von Entwickler 2-5

Stand an QA-Abteilung

Wie viele Versionen waren das?



The Source Code Control System (SCCS)

Marc J. Rochkind (1975, Bell Laboratories) :

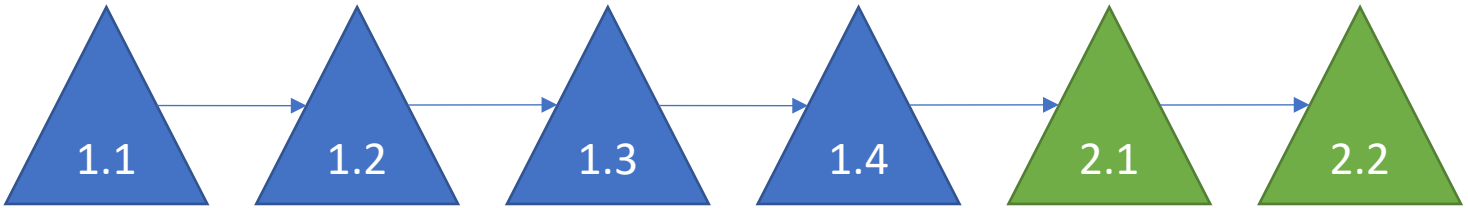
- Speicherverwaltung: Vermeidung von Duplikaten
- Änderungen werden nicht übertragen
- Chronologie von Änderungen
- Abbildung von Kunden auf Versionen

Orginal Paper:
Rochkind, Marc J. „The Source Code Control System.“ *IEEE Transactions on Software Engineering* 4 (1975): 364-370.

The Source Code Control System (SCCS)

Speicherverwaltung:

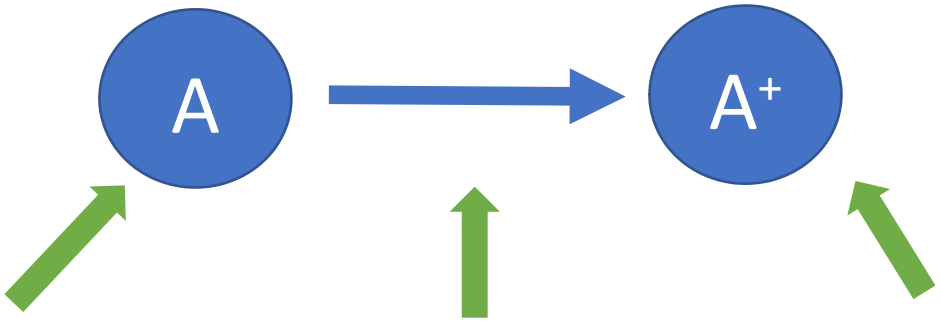
- Alle Versionen eines Moduls sind in einer Datei erfasst.
- Änderungen werden als *Delta* gespeichert.
- Benennung legt fest, welche *Deltas* zu welchem *Release* gehören.



Features von SCCS

- Optionale Deltas → Erlauben das Benennen einer Bedingung, um dieses Delta an- und abzuwählen.
- Including & Excluding anderer Deltas → bestimmte Änderungen können erzwungen bzw. abgelehnt werden.
- Versionsangabe für Abhängigkeiten
- Module locking & kennwortgeschützte Änderungen
- Dokumentation von Änderungen und Commit-Messages

Commit



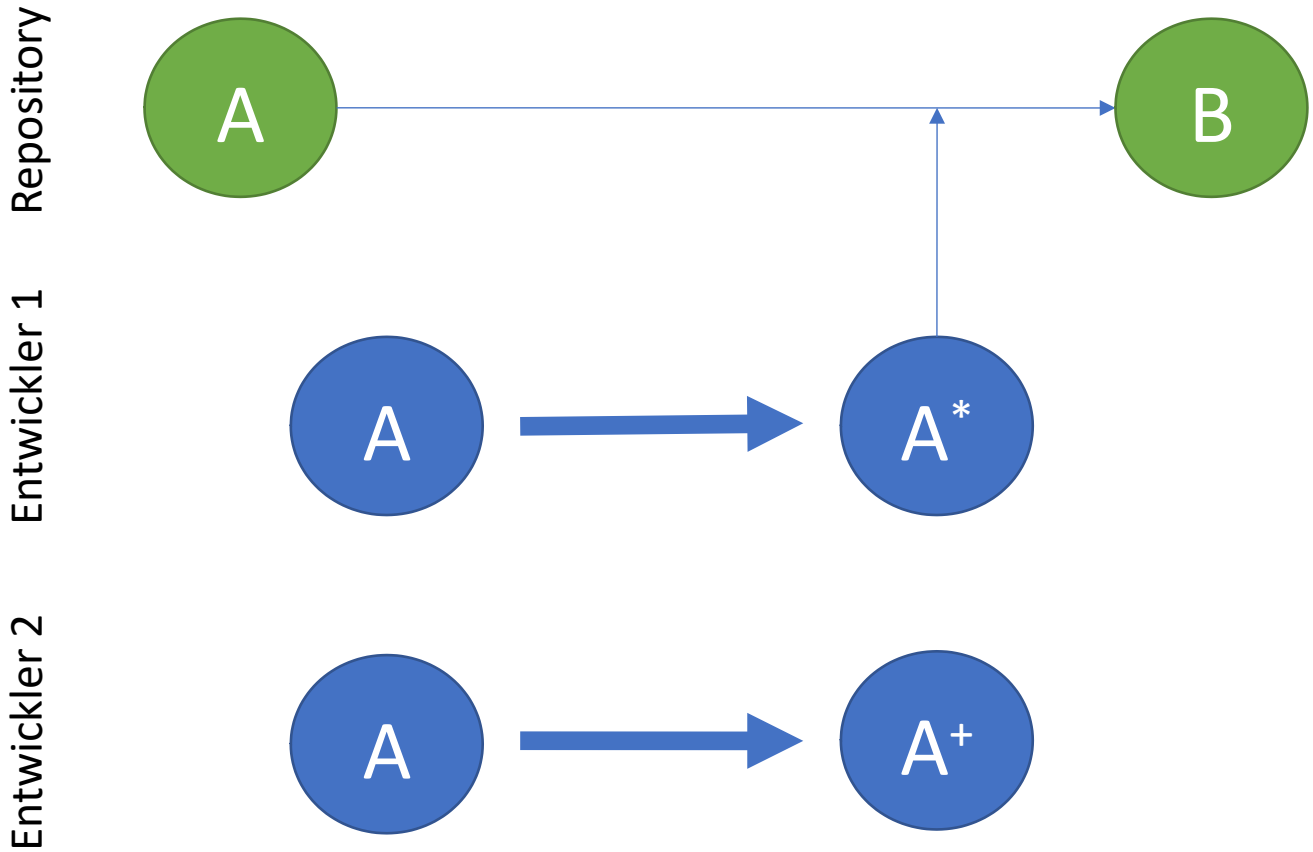
Ursprüngliche Version

Änderungsphase

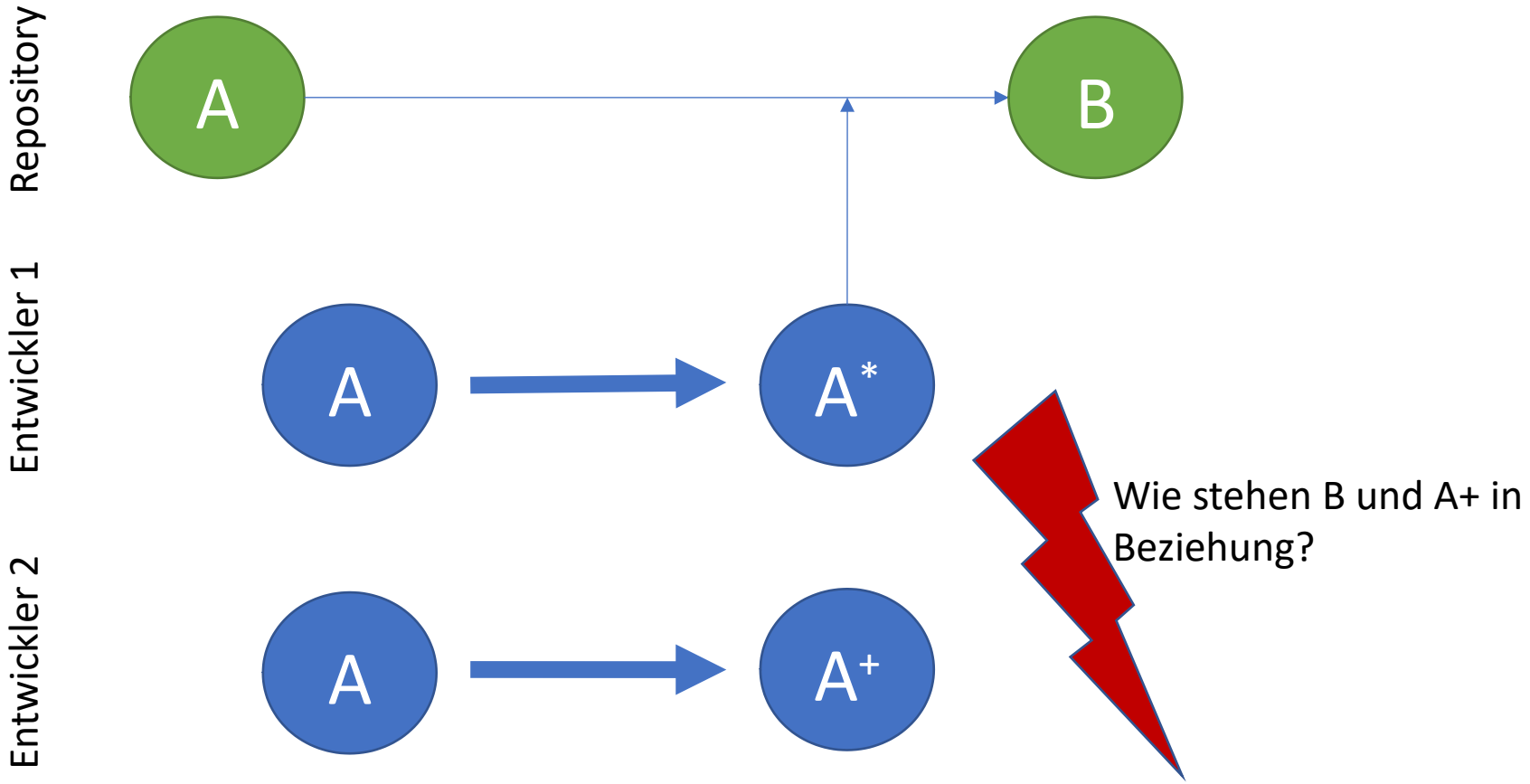
Abschluss der Änderungsphase → Commit

Ein Commit beschreibt eine Menge von Änderungen. Diese Beschreibung erhält mindestens ein Datum, den Autor, eine neue Versionsnummer und eine optionale Begründung.

Parallele Bearbeitung

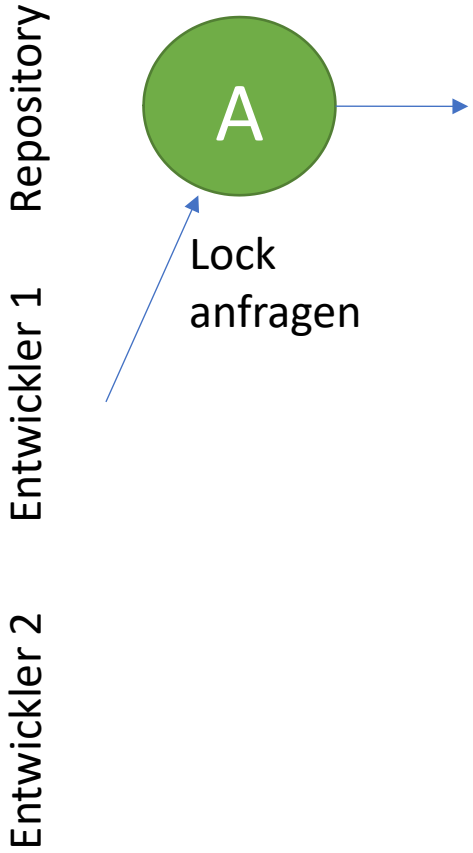


Parallele Bearbeitung



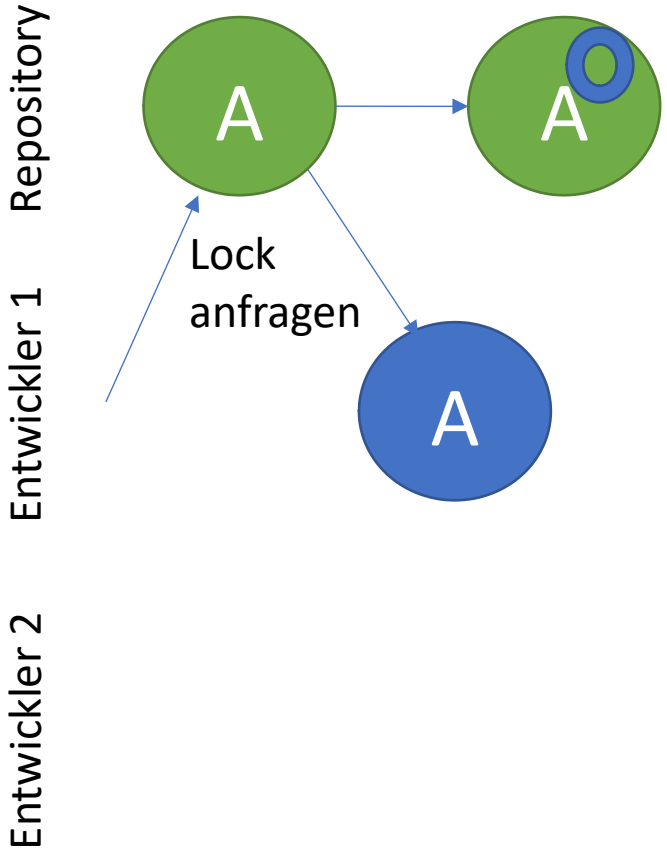
Mögliche Lösungen

Lock einführen → exklusives Bearbeitungsrecht



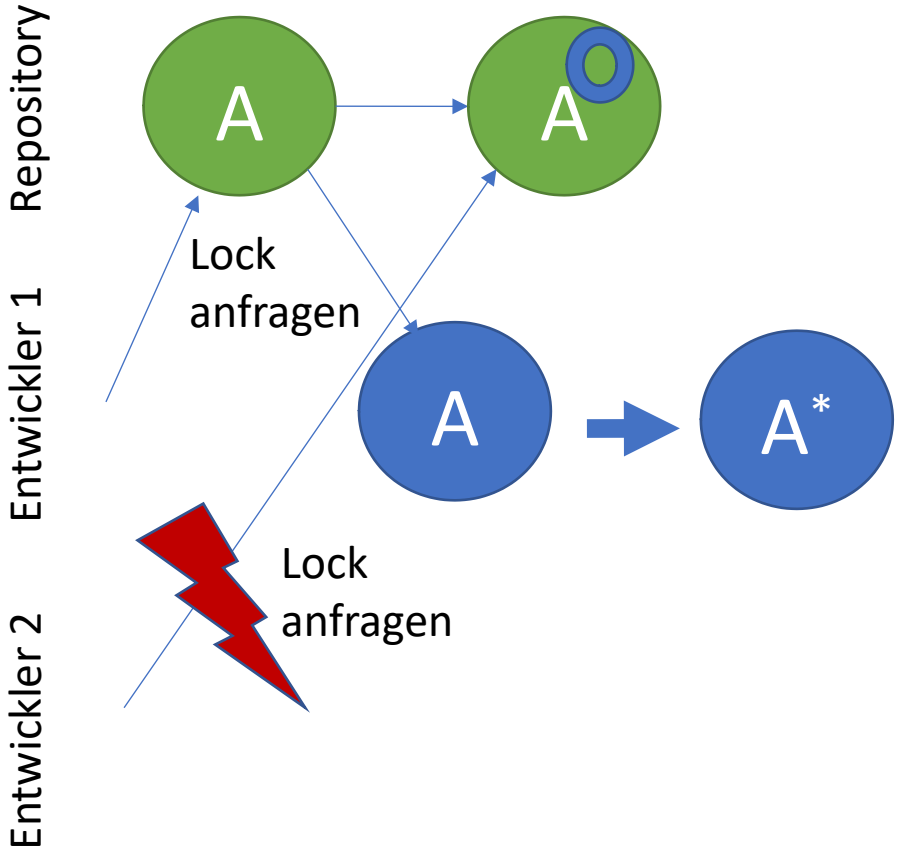
Mögliche Lösungen

Lock einführen → exklusives Bearbeitungsrecht



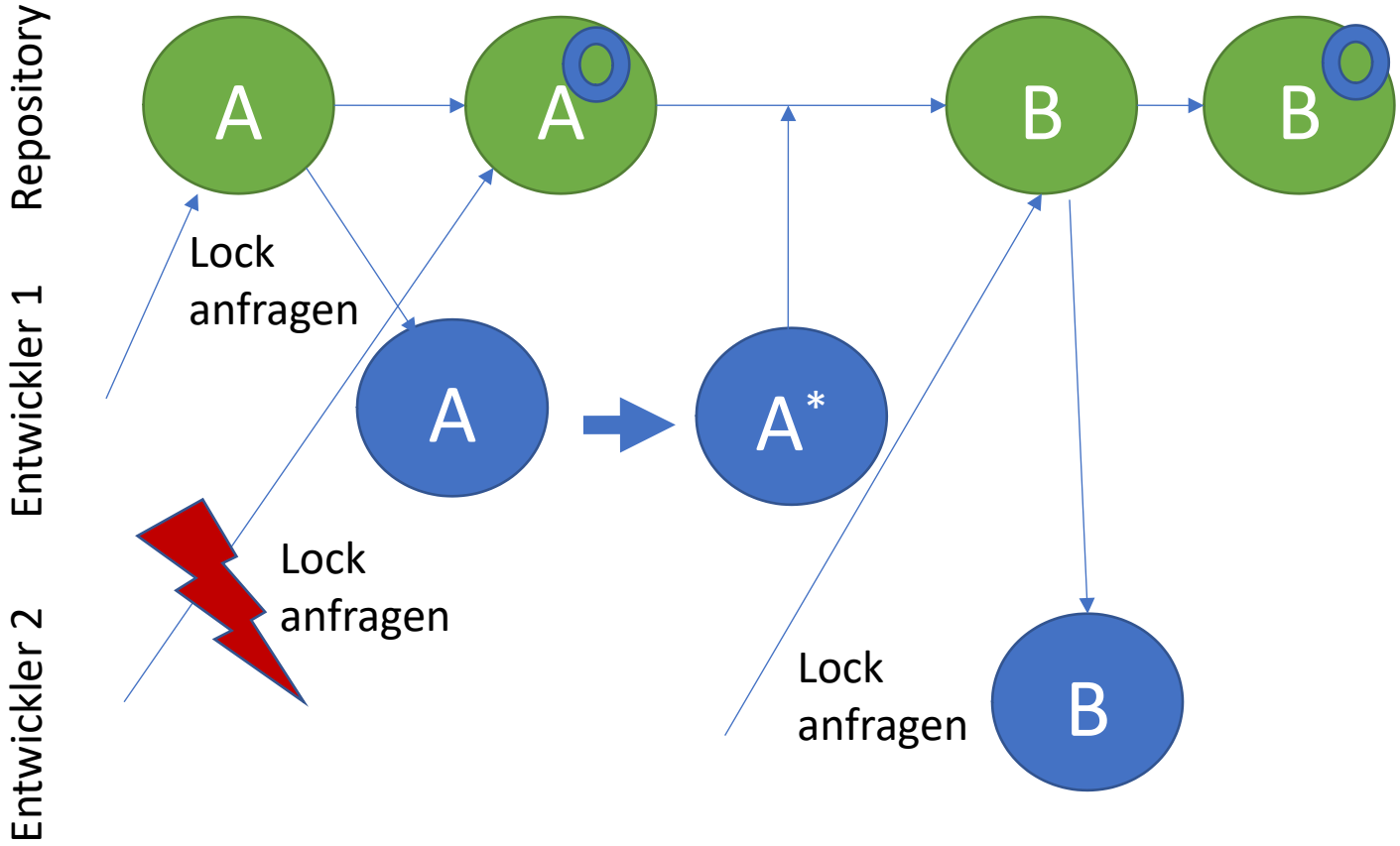
Mögliche Lösungen

Lock einführen → exklusives Bearbeitungsrecht



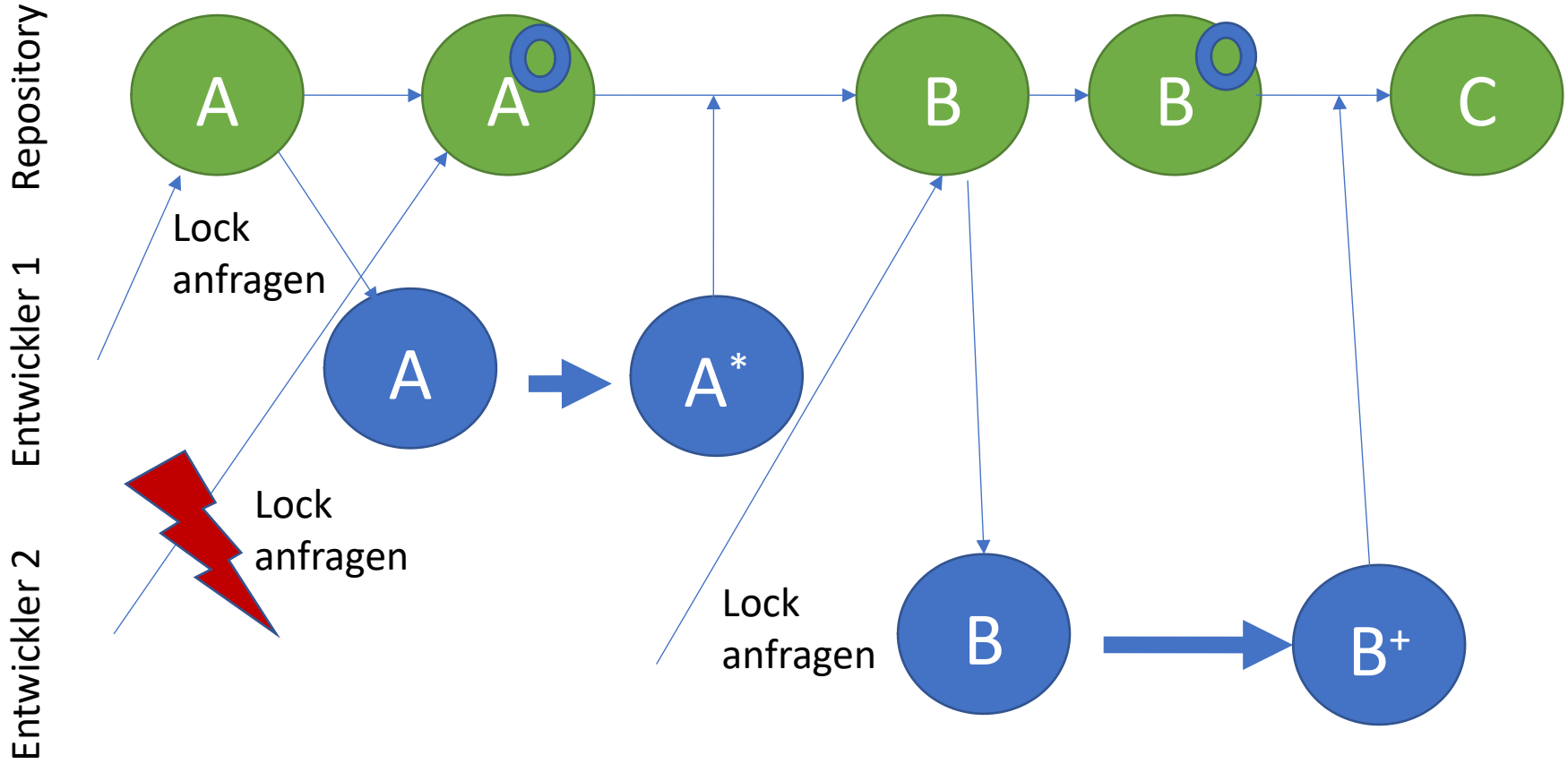
Mögliche Lösungen

Lock einführen → exklusives Bearbeitungsrecht



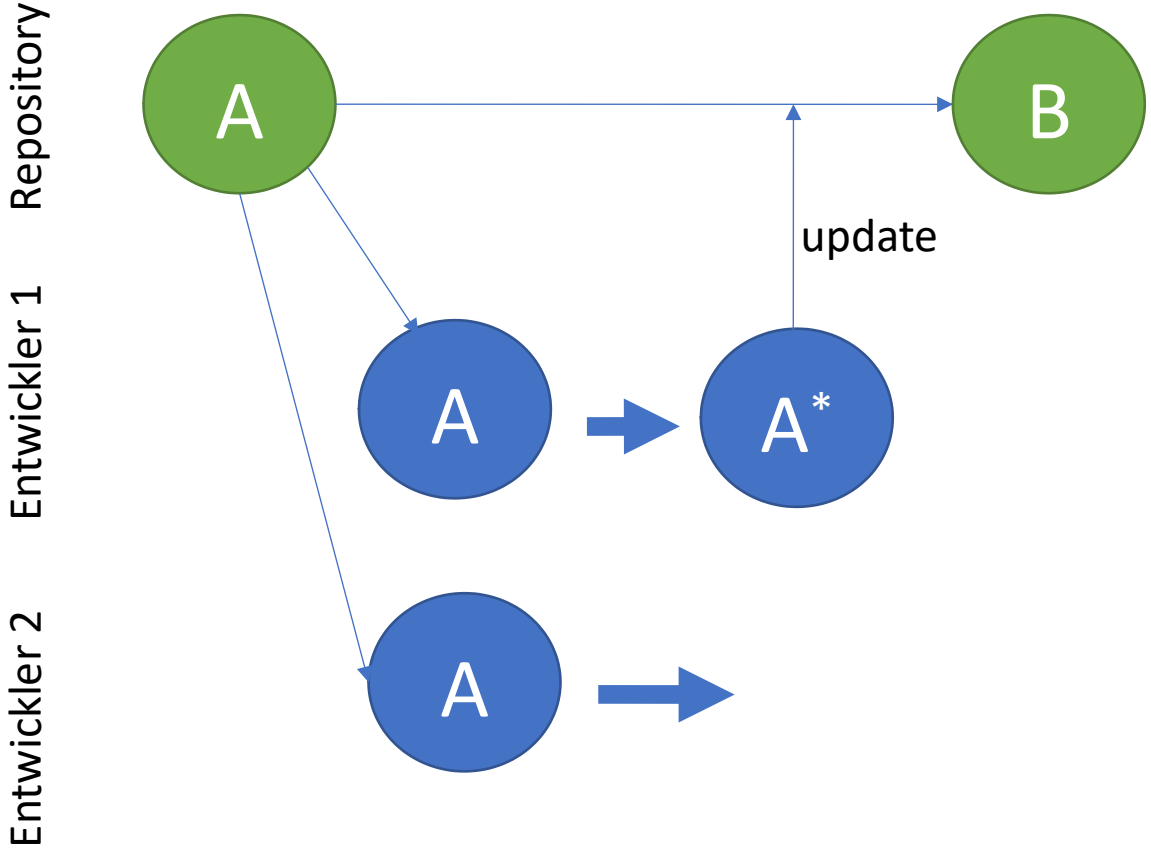
Mögliche Lösungen

Lock einführen → exklusives Bearbeitungsrecht



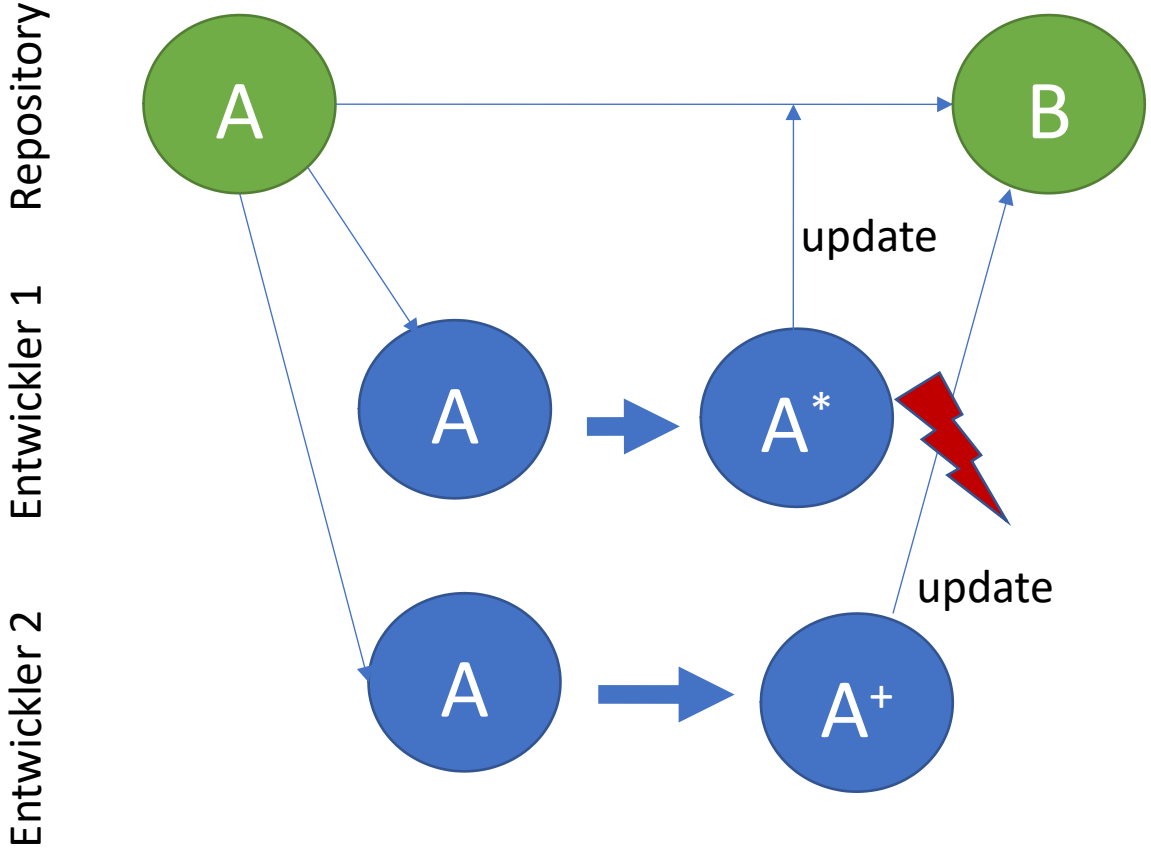
Mögliche Lösungen

Merge



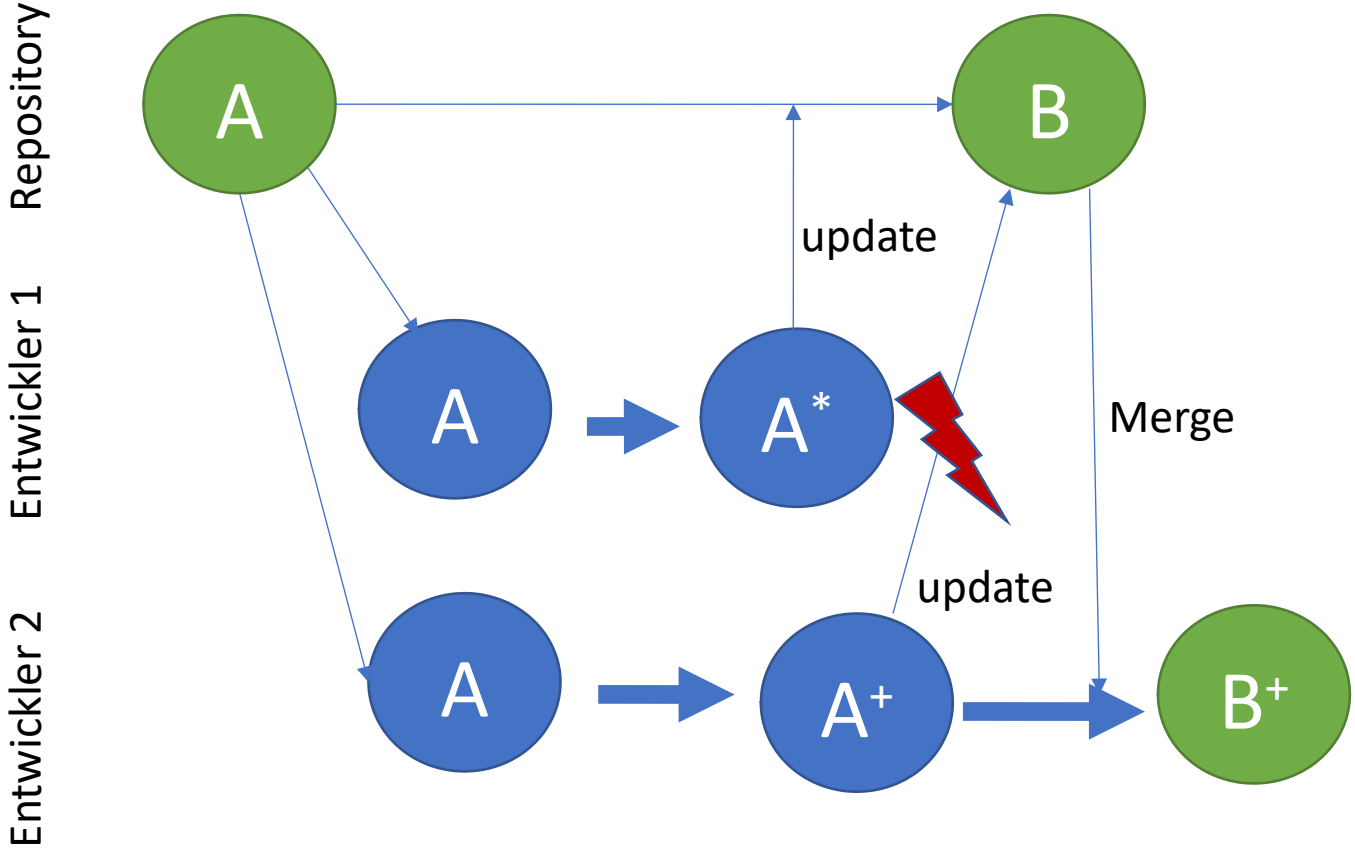
Mögliche Lösungen

Merge

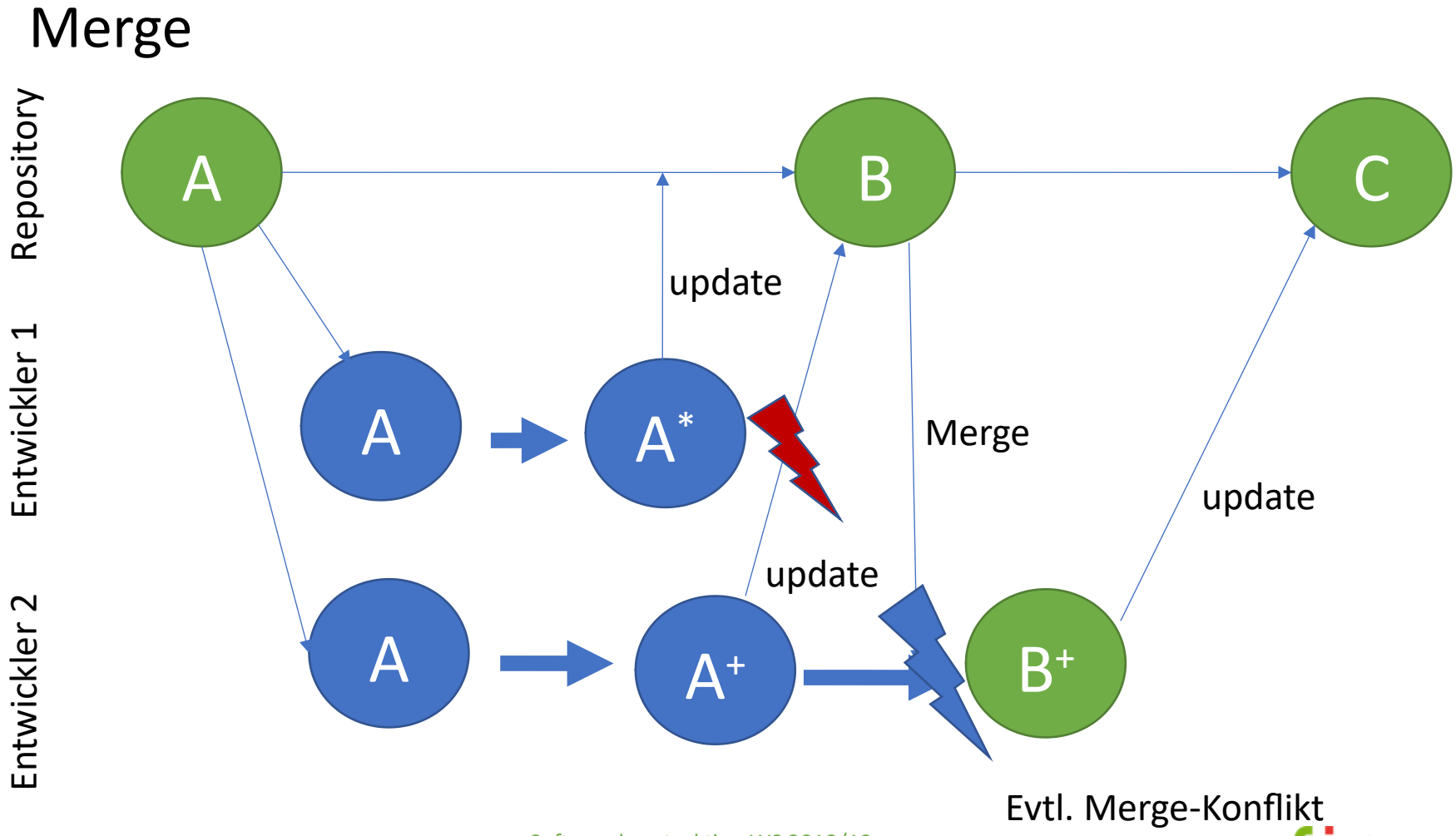


Mögliche Lösungen

Merge

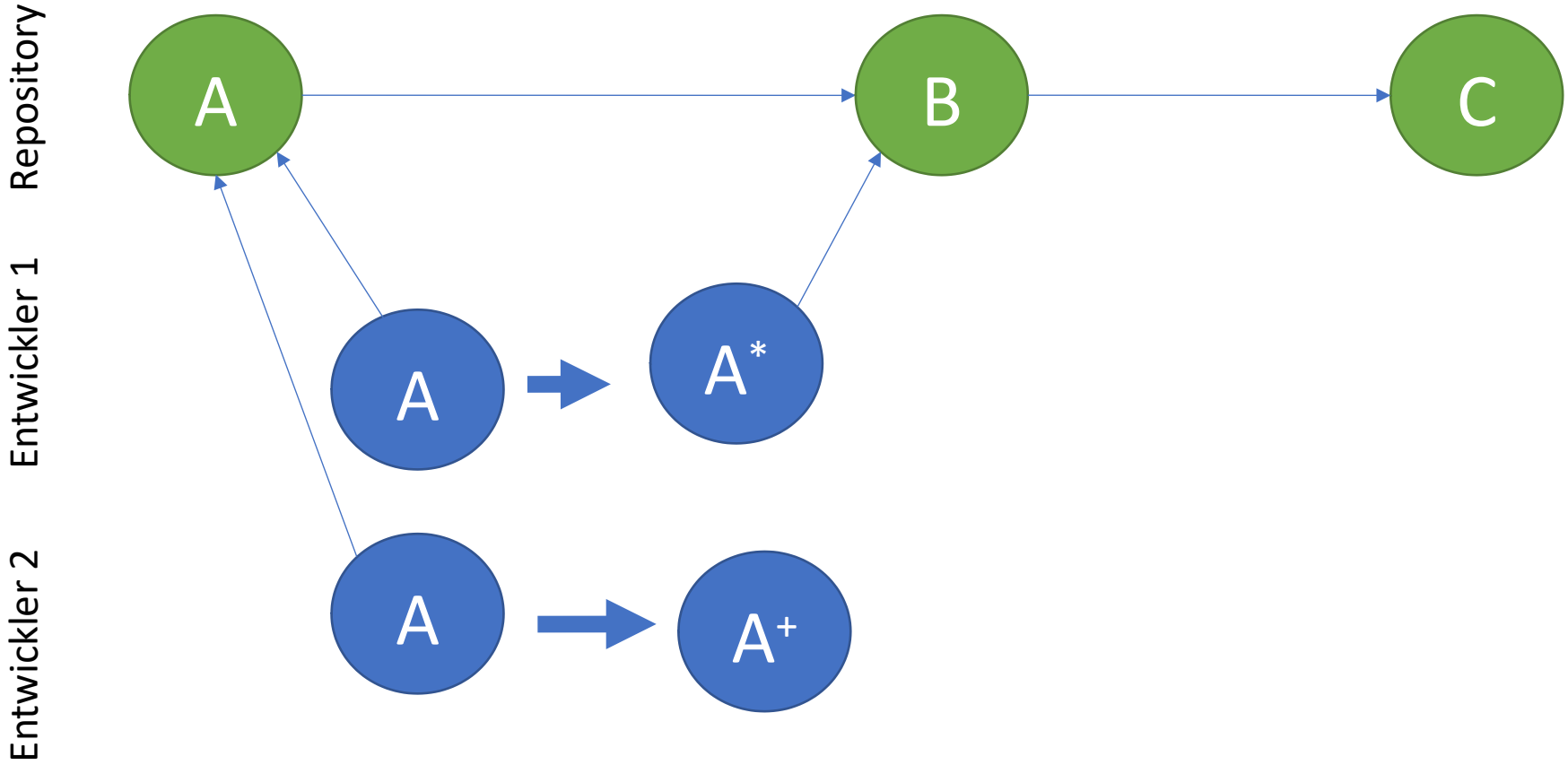


Mögliche Lösungen

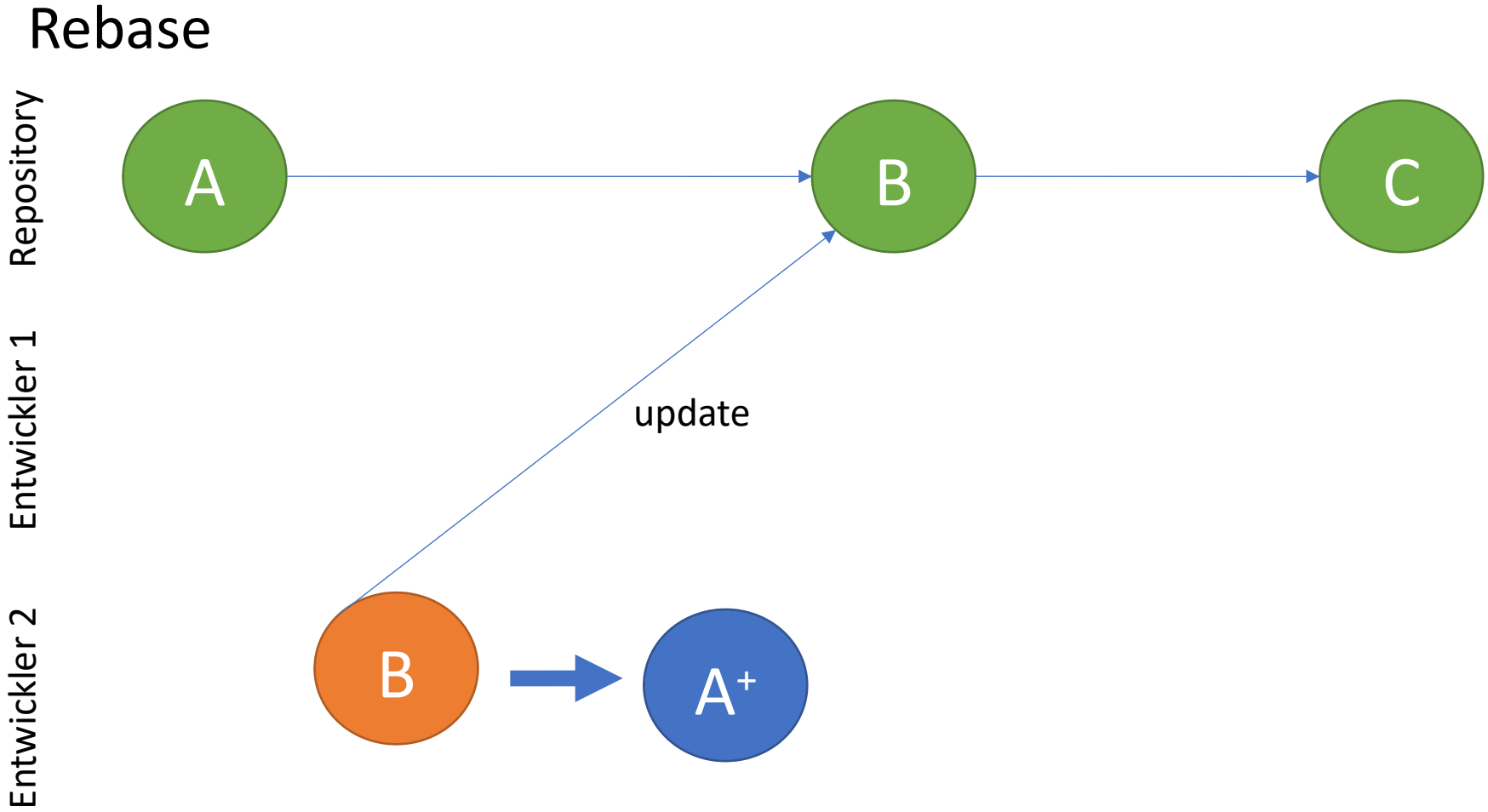


Mögliche Lösungen

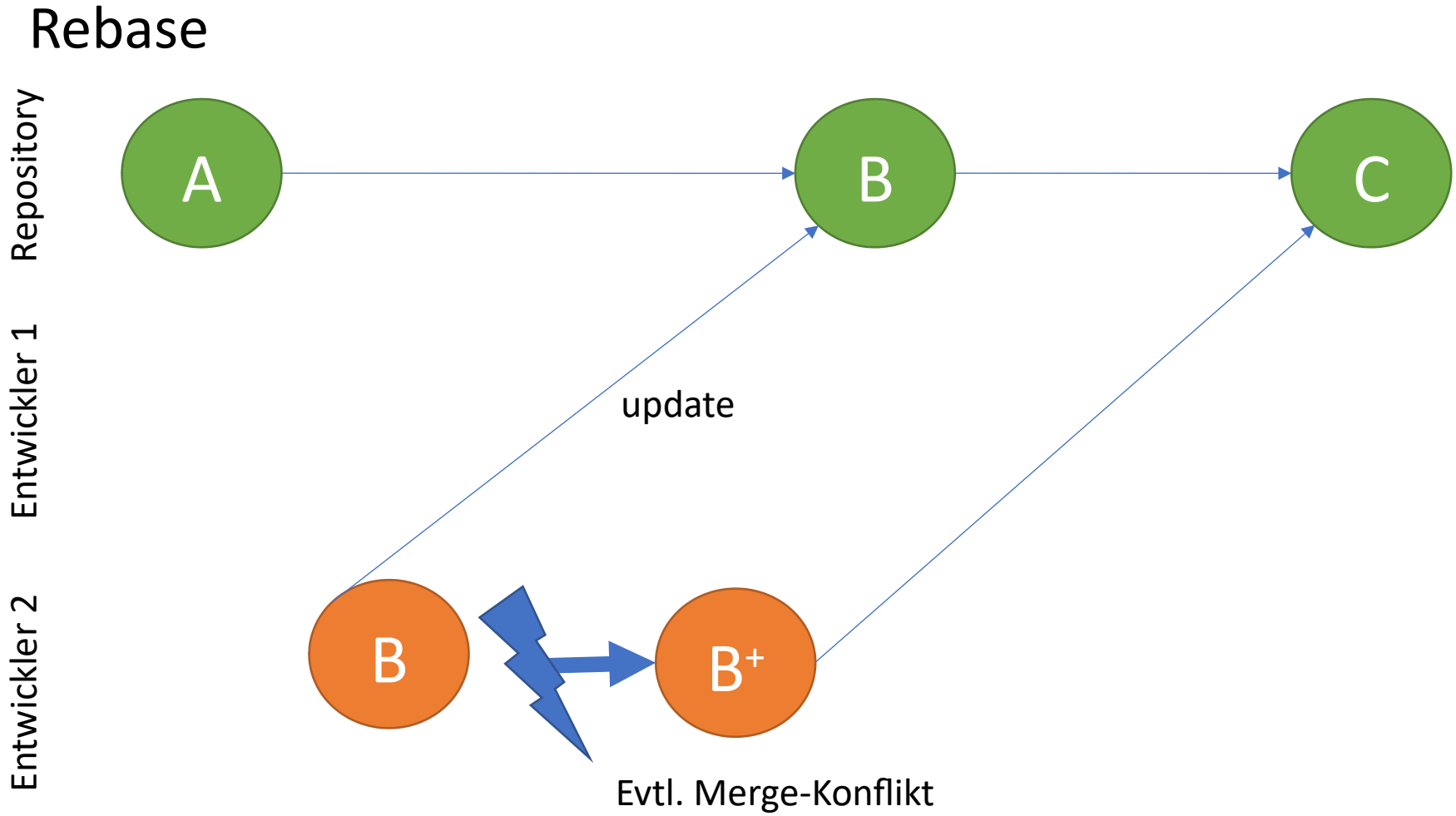
Rebase



Mögliche Lösungen



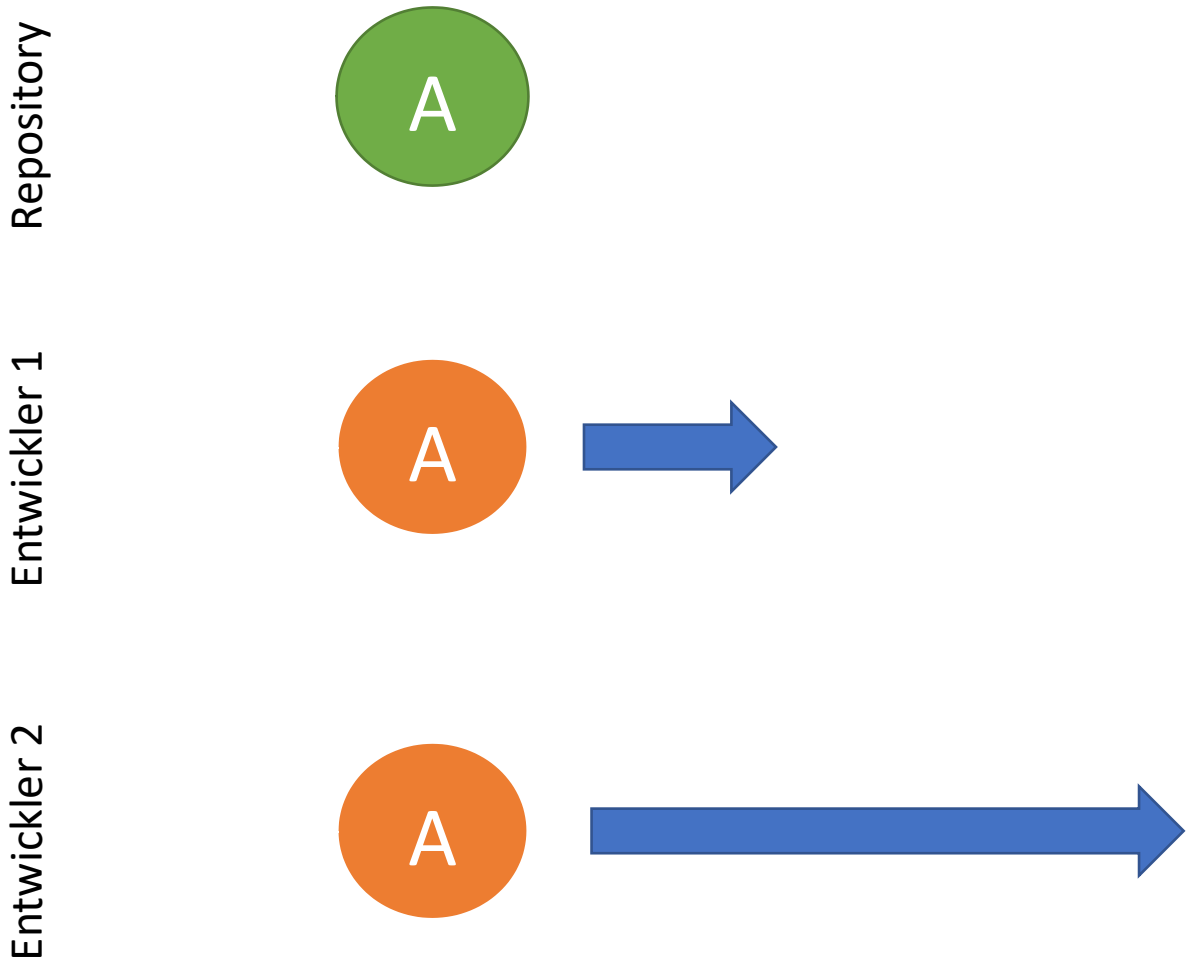
Mögliche Lösungen



Zentrale SCM-Tools

Master-Kopie

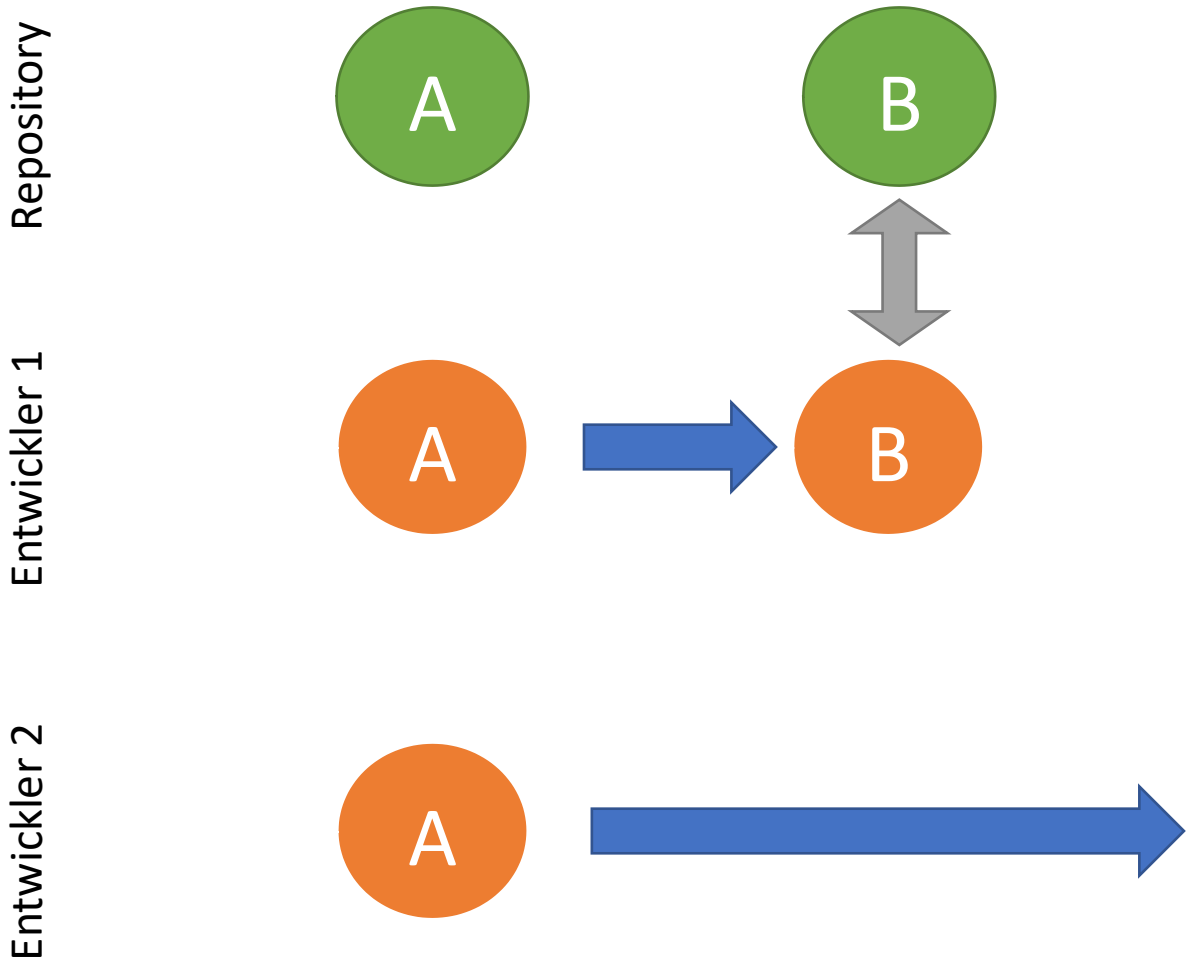
Arbeitskopie



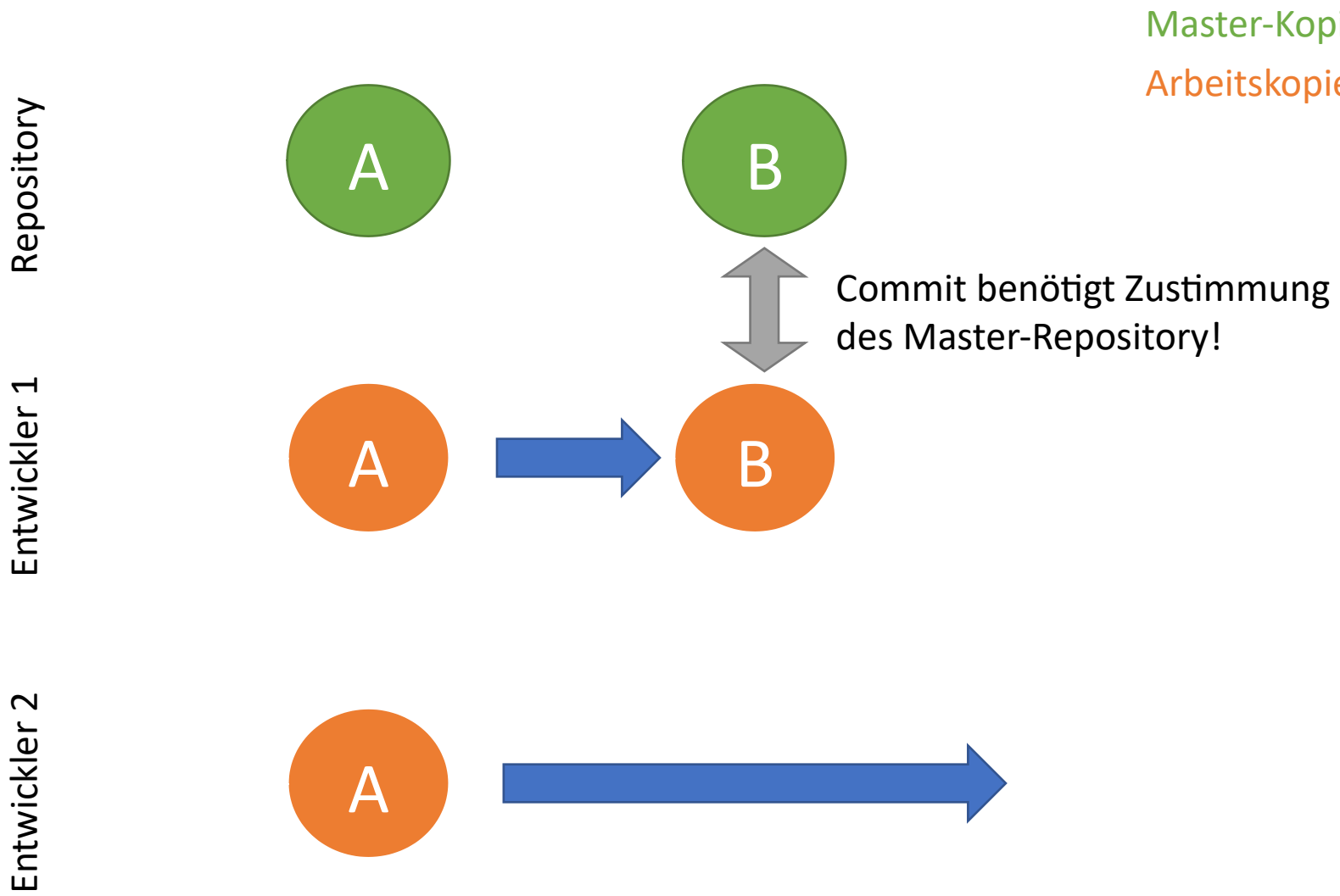
Zentrale SCM-Tools

Master-Kopie

Arbeitskopie



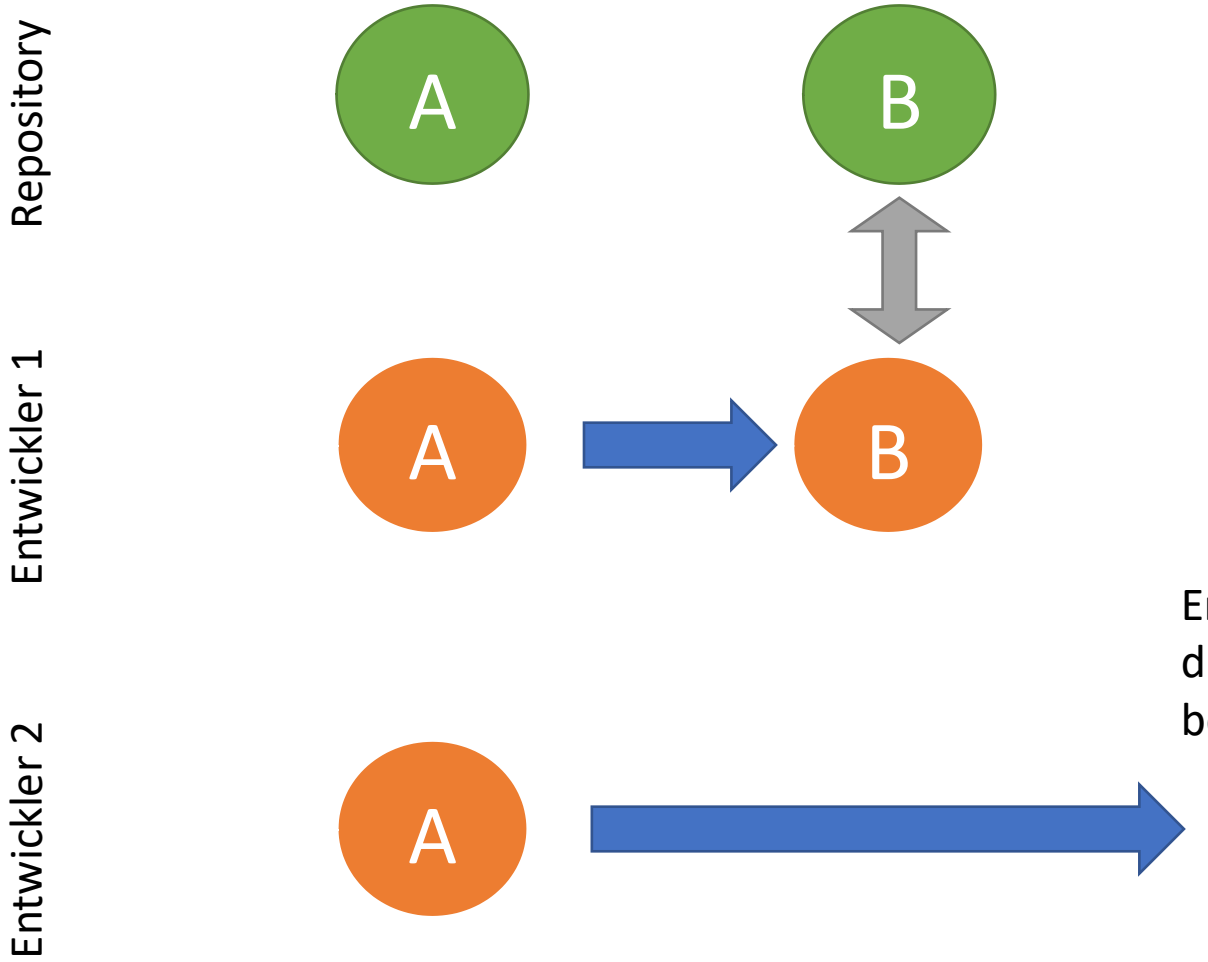
Zentrale SCM-Tools



Zentrale SCM-Tools

Master-Kopie

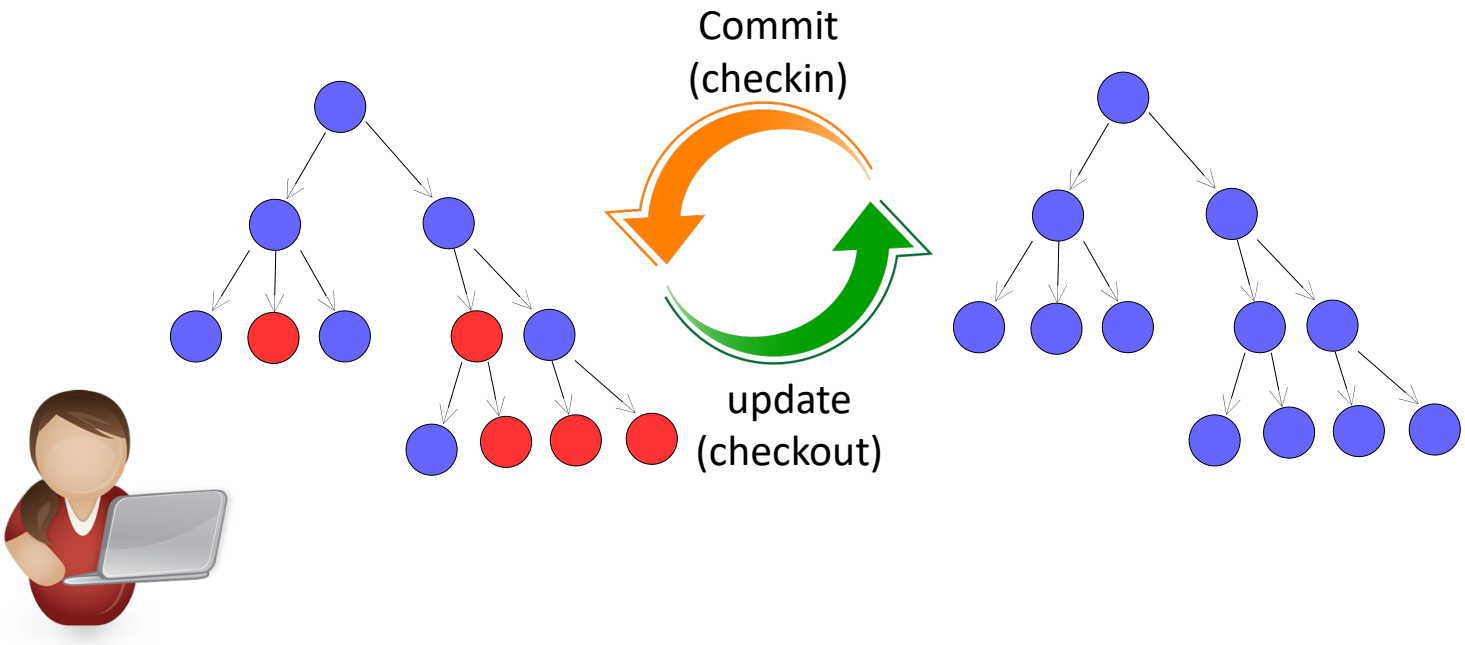
Arbeitskopie



Entwickler 2 muss Commit B in die Arbeitskopie integrieren, bevor er commiten kann.

Einfache Sichten

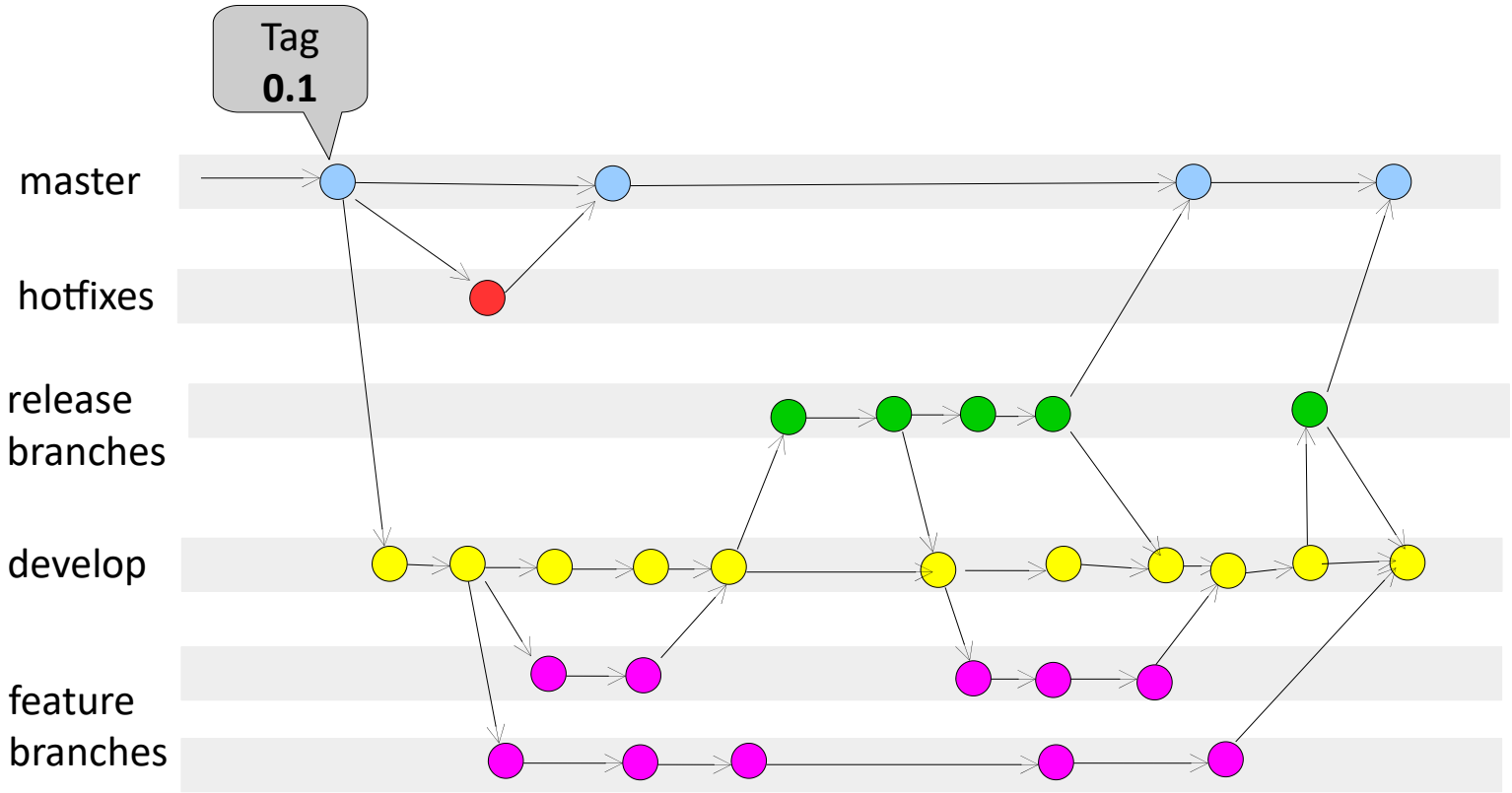
- Nur Tiefe von 1 (CVS und SVN)



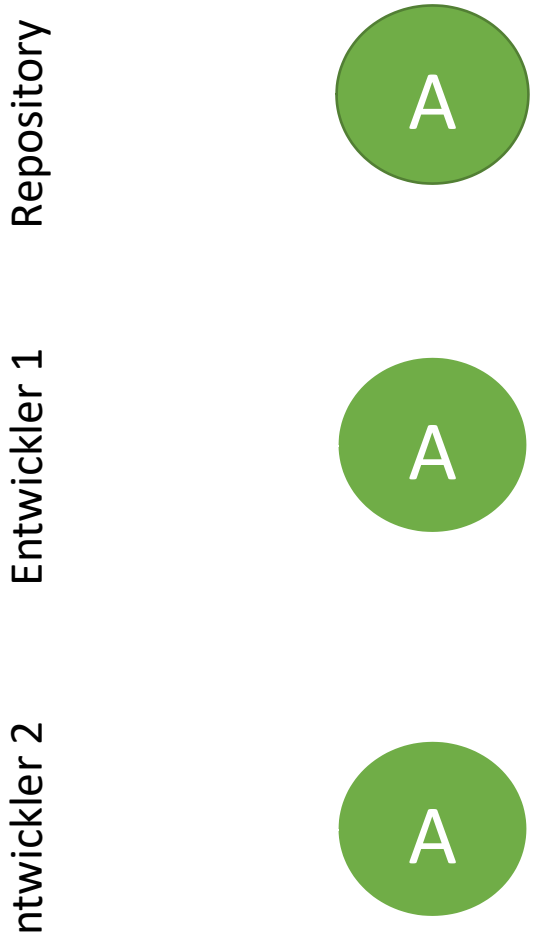
GIT

- Ein zentrales Repository ist keine Pflicht, sondern beliebig “transportierbar”
- Statt dessen GIT Workflows, verschiedene Arten, wie Sichten zusammenarbeiten
- GIT speichert keine Deltas, sondern Vollversionen
- GIT kennt Features, die benannten Branches entsprechen

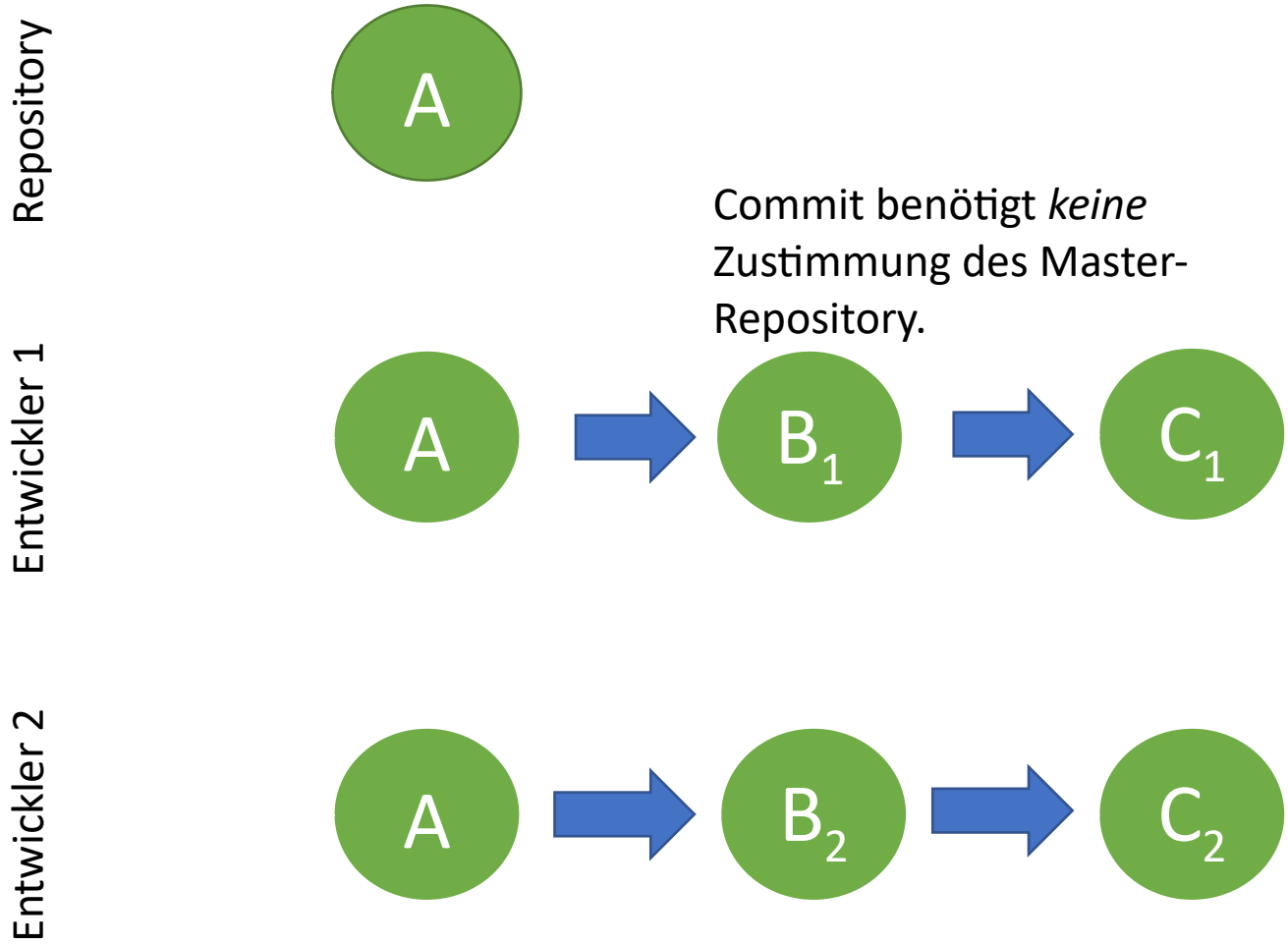
GIT-Workflow



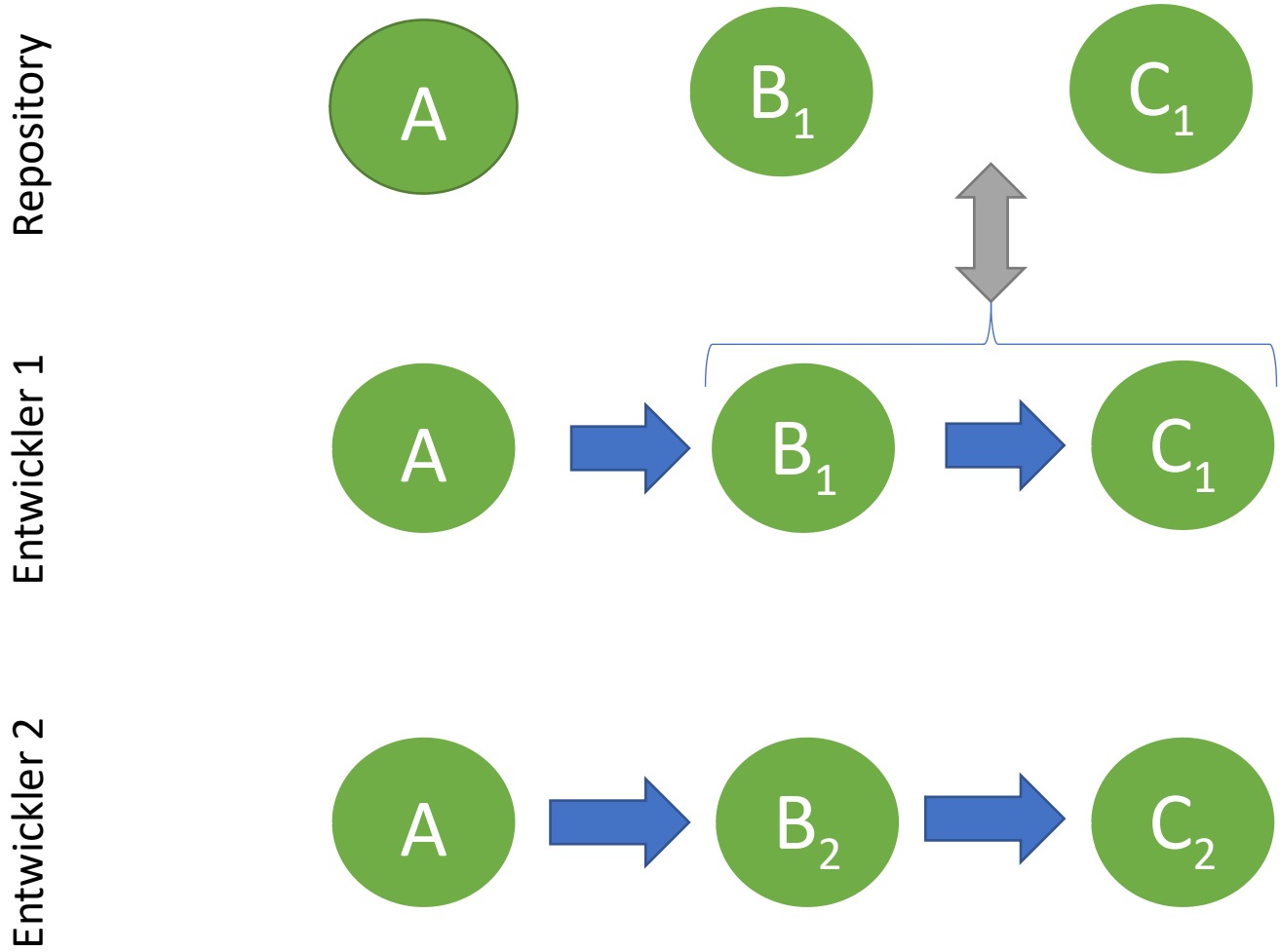
Dezentrale SCM-Tools



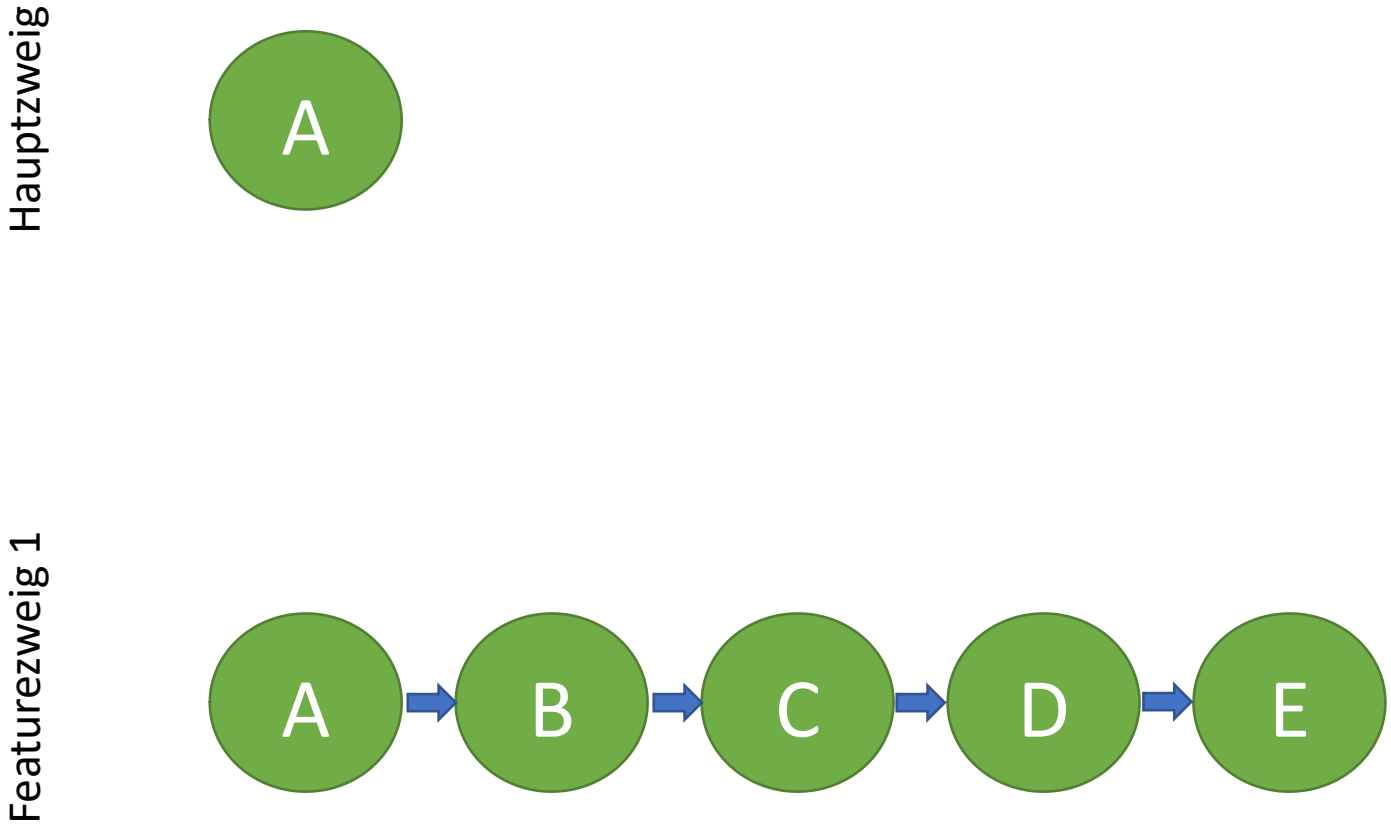
Dezentrale SCM-Tools



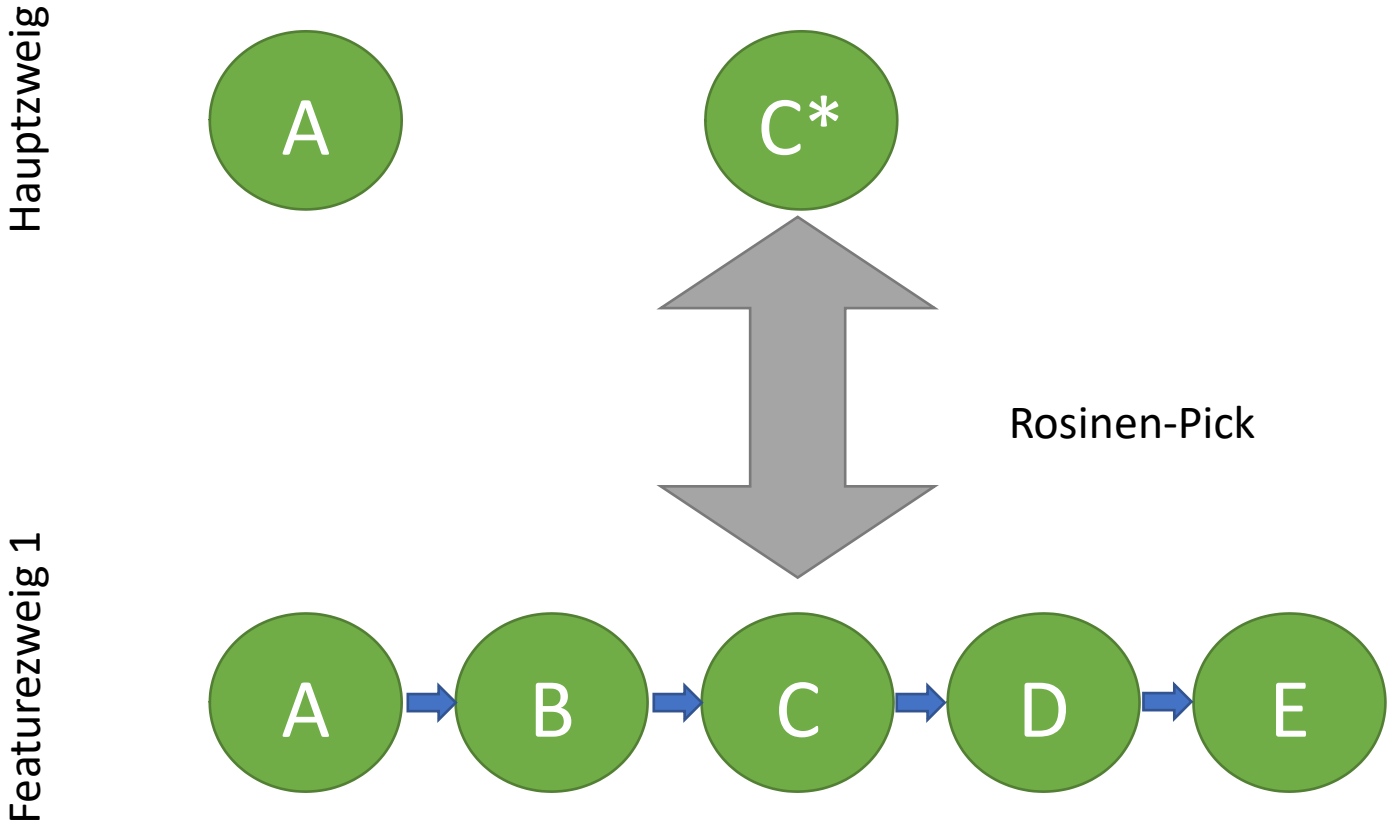
Dezentrale SCM-Tools



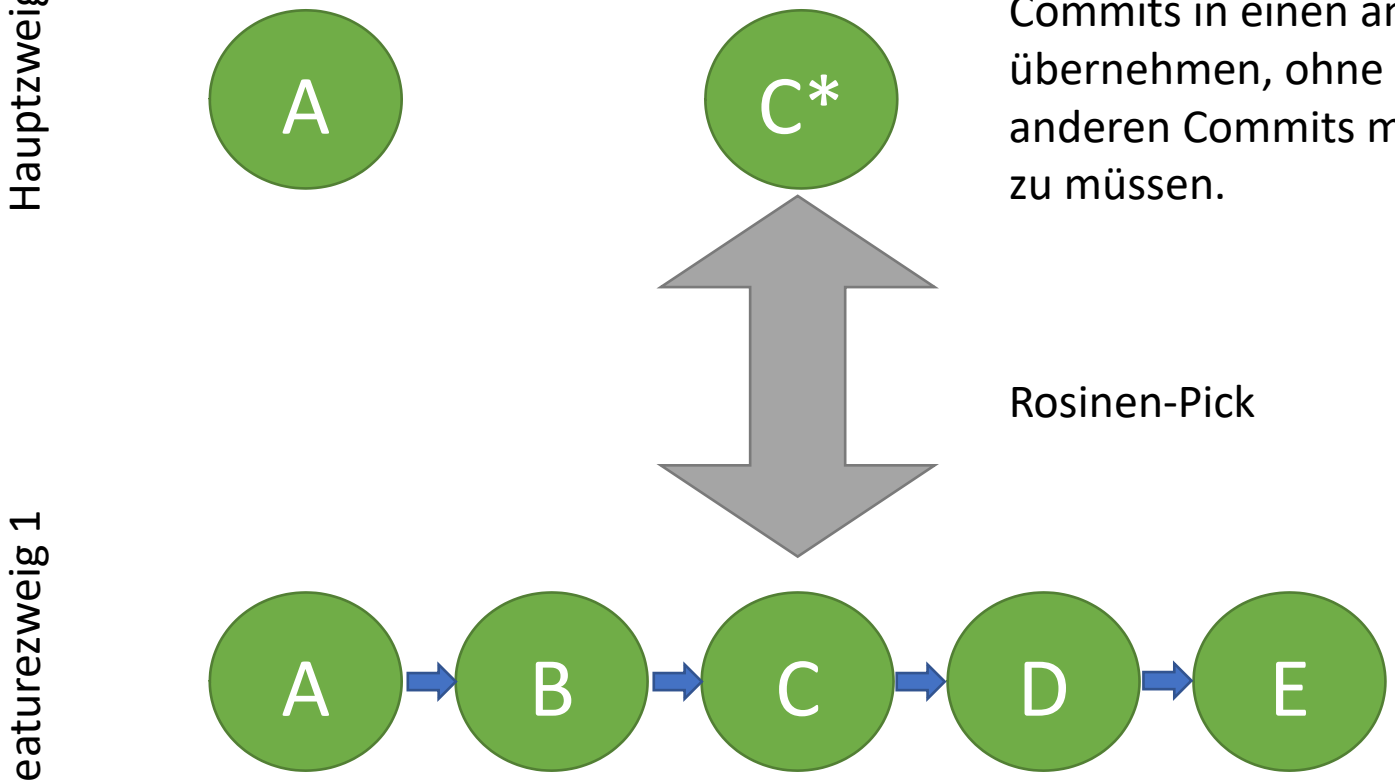
Rosinenpicken



Rosinenpicken



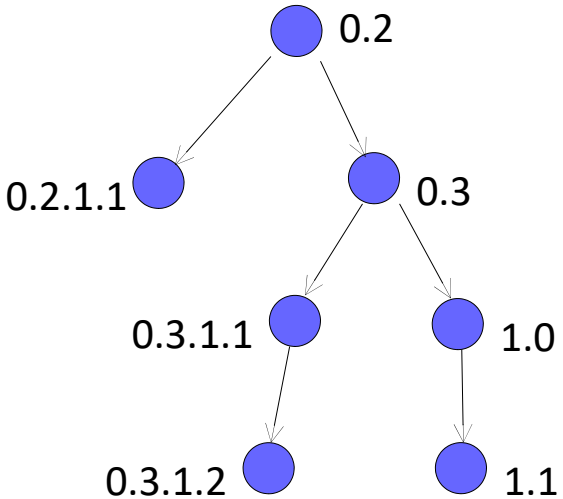
Rosinenpicken

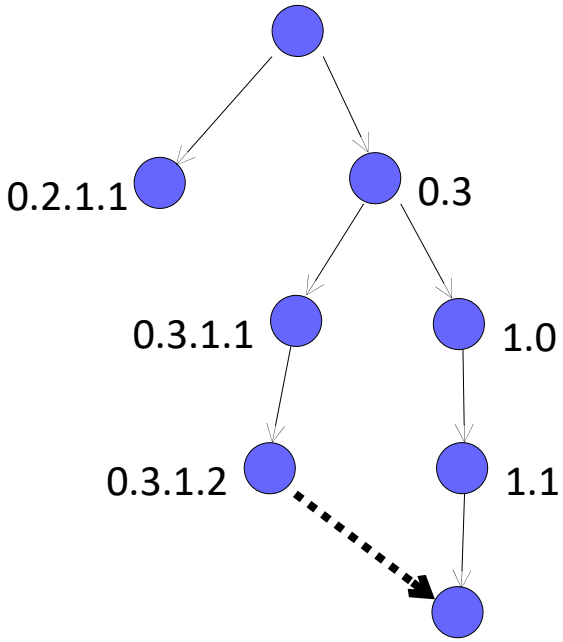


Rosinenpicken erlaubt es, einzelne Commits in einen anderen Zweig zu übernehmen, ohne deswegen alle anderen Commits mit übernehmen zu müssen.

Versionsdimension

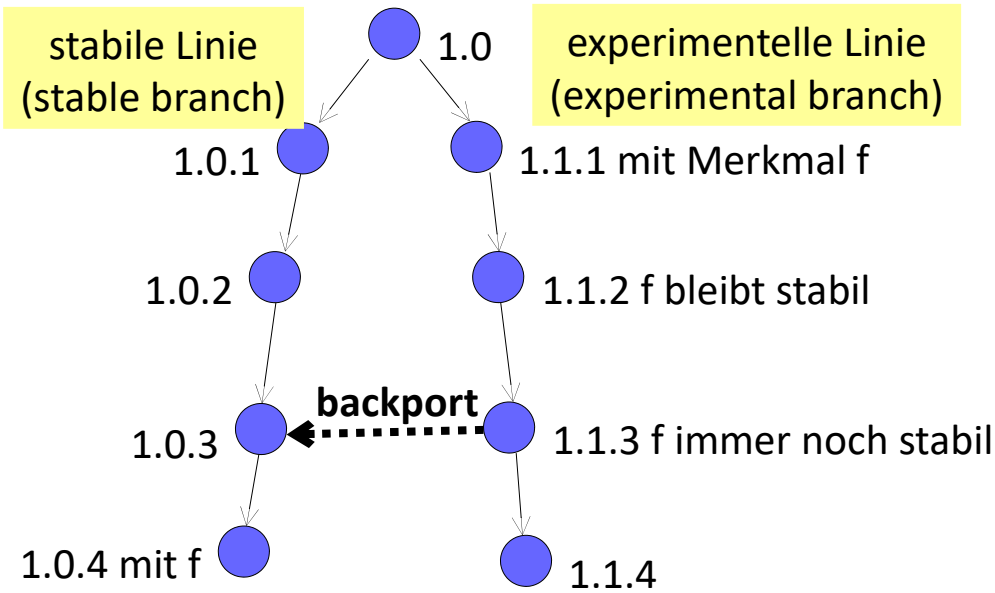
- Verzweigungsbaum unbestimmter Tiefe und Breite
- Jede Modifikation generiert ein Kind oder einen Zweig (branch)
- Verschmelzen ist möglich (branch merge)





Backporting

- Stabile Versionen erhalten oft eine „gerade Linie“ von Versionsnummern, experimentelle eine ungerade Linie
- Erweist sich ein Feature auf einer experimentellen Linie als stabil, dann es in die stabile Linie **rückintegriert werden (backporting)**
- Vorteil: stabile Linien erlauben es, kommerzielle Nutzer zu befriedigen, Backporting frischt alte stabile Versionen auf



Konfigurationsmanagement (KM)

- Gegenstand des KM ist das gesamte Softwaresystem mit Komponenten und Artefakten (Produkt- und Artefaktstruktur)
  - **Spezifikation:** Daten und Requirements (Anforderungen)
  - **Entwurf:** formale und informale Dokumente
  - **Programme:** Code-Teile, Datenbeschreibungen, Prozeduren
  - **Testfälle und -umgebung:** Dokumente für Testdaten, Testumgebung
  - **Integrationskonzept:** alle Dokumente für Integration und Einführung (auch Benutzerdokumentation)
- Verwaltung der Komponenten in der Produkt- (Artefakt-) Bibliothek

Begriffe des KM

- **Konfigurationselement:** Eindeutig benanntes Objekt im Zusammenhang mit dem Projekt
  webui, *user_documentation, …*
- **Variante**: Untergliederungskoordinate für Änderungsstände, Modifikationen, Revisionen *innerhalb* einer Version der Komponente (z.B. für eine Umgebung)
  acmecorp, offline, exportcrypto
- **Version:** Instanz / Entwicklungsstand eines Elements; Editierung erhöht die Versionsnummer. Identifikation: Name + Versionsnummer (+ Variante)
  webui-1.4.2-acmecorp, user_documentation-19

Begriffe des KM

- **Konfiguration:** benannte und formal freigegebene Menge von Komponenten für eine „Abarbeitung“ (Bearbeitung, Build etc.)
  server_bundle = webui + user_documentation
- **Baseline:** Sammlung fester, versionierter Komponenten
  webui-1.4.2 + jquery-3.3.1 + user_documentation-19 + …
- **Codeline:** Sammlung versionierter Softwarekomponenten und ihrer internen Abhängigkeiten
  webui-1.4.2 + util-2.3 + …
- **Mainline:** Eine Folge von Baselines, die die „Geschichte“ des Systems beschreiben
- **Release**: eine für die Nutzung freigegebene Version des Systems; üblicherweise versioniert
  server_bundle-2018.11.01 = webui-1.4.2 + …

Begriffe des KM

- **Checkout**: Lesen der Version *n* für Editierung
  $ svn checkout
- **Checkin**: Schreiben der Version *n+1* nach Editierung
  $ svn commit
- **Verzweigen**: Erzeugung einer neuen Codeline aus einer Version einer bestehenden Codeline
  $ svn copy […]/trunk […]/branches/jdoe/bug-481516
- **Zusammenführen:** Erzeugung einer Codeline aus zwei Versionen bestehender Codelines
  $ svn merge […]/branches/jdoe/bug-481516
- **Systemerstellung:** Erstellen einer ausführbaren Systemversion aus einer Baseline
  $ make -j16

Agenda

- Source Code Management
- Build-Automatisierung & Dependency Management
- Release-Strategien

Release-Management

- **Ziel**: Erzeugung und Bereitstellung von neuen Softwareversionen für die Öffentlichkeit
- Formate
  - Source Code + Buildskripte + Instruktionen
  - Pakete von ausführbaren Dateien für spezifische Plattformen
  - Portable Formate: SAAS, Java Web Start
  - Patches und Updates: automatisch oder manuell
- Inhalt
  - Quellcode und/oder Binärdateien, Datendateien, Installationsskripte, Bibliotheken, Nutzer- und/oder Entwicklerdokumentation, Feedbackprogramme, etc.

Konfigurationsmanagementplan

- Standards
  - IEEE 828 (SCM Plans), ANSI-IEEE 1042 (SCM), ITIL, …
- Bestandteile eines Plans
  - **Was** wird verwaltet? (Kriterien für Komponenten)
  - **Wer** ist für welche Aktivitäten verantwortlich? (Rollen und Aufgaben)
  - **Wie** macht man es? (Prozess für Änderungsanfragen, Aufgabenverteilung, Überwachung, Testen, Release, …)
  - **Welche** Ressourcen und welche **Kapazitäten** werden benötigt? (Werkzeuge, Personen, Geld, …)
  - Wie wir die **Einhaltung** des Plans überwacht (Verantwortliche, Planänderungen, …)?

Erstellungsprozess

- Build-Prozess: Automatische Generierung eines fertigen Anwendungsprogramms
- Code-Kompilierung, Linken etc.
- Tools:
  - GNU Make, Cmake, automake, Ant, Maven, Gradle, MSBuild, Bazel, ...
- Eigener Prozess für *sehr* komplexe Entwicklungsprojekte

Buildskripte

- **Was** soll **gebaut** werden?
- **Welche Tests** sollen ausgeführt werden?
- **Welche Abhängigkeiten** in welchen Versionen werden benötigt?
- **Welche Werkzeuge** in welchen Versionen werden genutzt?
- **Wie** kann man das Programm **starten**?
- **Welche Version** des Projektes baut das Skript?

Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>de.tudortmund.ffi.ls14.swk</groupId>
   <artifactId>madn-server</artifactId>
   <packaging>war</packaging>
   <version>0.0.1-SNAPSHOT</version>
   <name>madn-server Maven Webapp</name>
   <url>http://maven.apache.org</url>
   <!-- und so weiter… -->
```

Dependencies mit Maven


```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.3.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.3.7.Final</version>
    </dependency>
</dependencies>
```

Maven-Repositories

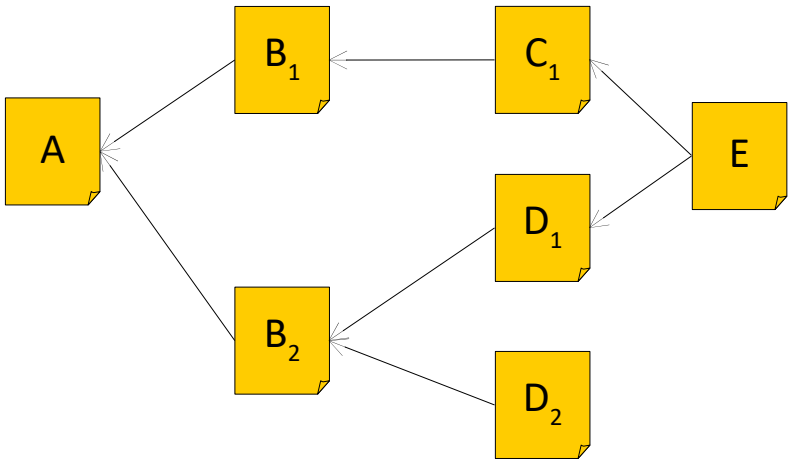
https://central.maven.org/maven2/

- Verzeichnis von Java-*Artefakten*
- Gehostest von Sonatype
- Analysieren Jars, verkaufen Beratung …

Erstellungsstrategien

- Wir betrachten Tools wie GNU Make:
  - Rekursives Bauen
  - Globale Abhängigkeiten der Module als gerichteter kreisfreier Graph (DAG)
  - O(*n*) Updates notwendig (*n*=Anzahl der Module im System)

- Beispiel: $ make A
- Rekursive Tiefensuche ergibt Buildreihenfolge:
  <E, C_1, B_1, D_1, D_2, B_2, A>



Algorithmus: Rekursives Erstellen

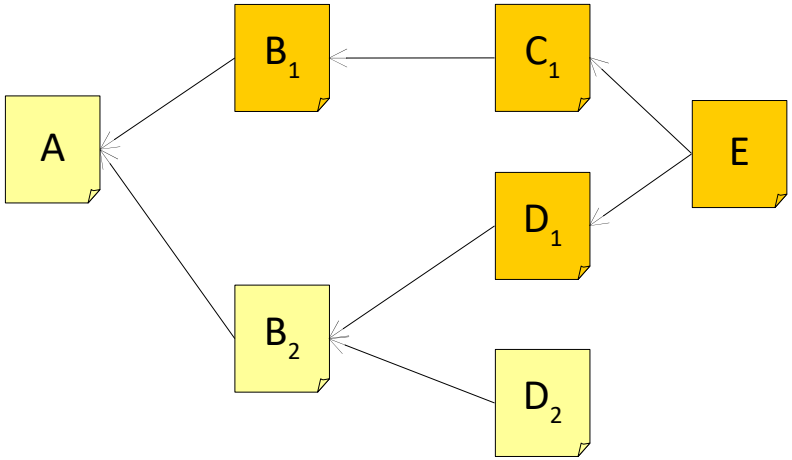
- Pseudo-Algorithmus für rekursives Erstellen:
  1) **Erzeuge DAG:** Erzeuge globalen DAG aus Modulabhängigkeiten
  2) **Update:** Durchlaufe DAG, prüfe, welche Module veraltet (out-of-date) sind und aktualisiere diese

Algorithmus: Rekursives Erstellen


```
• update_file(f)
    need_update = false
    foreach dependency d {
      if(update_file(d) == UPDATED ||
      timestamp(d) newer than timestamp(f))
      need_update = true
    }
    if(need_update) {
      perform command to update f
      return UPDATED
    }
  return PASS
```

Nachteile der rekursiven Erstellung

- Bei jedem Erstellen wird der gesamte DAG durchlaufen und Module erstellt, …
- …auch wenn nur eine Datei (z.B. D_2) geändert wurde.
- O(*n*) Updates notwendig



Automatisierte Builds

- Automatisierung der unterschiedlichen Aufgaben eines Softwareentwicklers:
  - Übersetzen des Quellcodes in Objektcode
  - Verpacken des Objektcodes (ELF, PE, JAR, WAR, etc.)
  - Ausführung von Tests
  - Bereitstellung des Produkts auf Produktionssystemen
  - Erstellung von Dokumentation und Freigabevermerken (release notes)
  - u.v.m.

Build-Management

- Build-Management ist ein Prozess, der die einzelnen **Komponenten** einer Softwareanwendung in ein **auslieferbares Softwareprodukt zusammenfügt**.
- Probleme:
  - Compliance mit Regularien (z.B. Sarbanes-Oxley Act)
  - Verteilte Entwicklungsteams
  - Focus auf Wiederverwendbarkeit
  - Outsourcing von Entwicklungsleistungen
  - Marktdruck

Agenda

- Source Code Management
- Build-Automatisierung & Dependency Management
- Release-Strategien

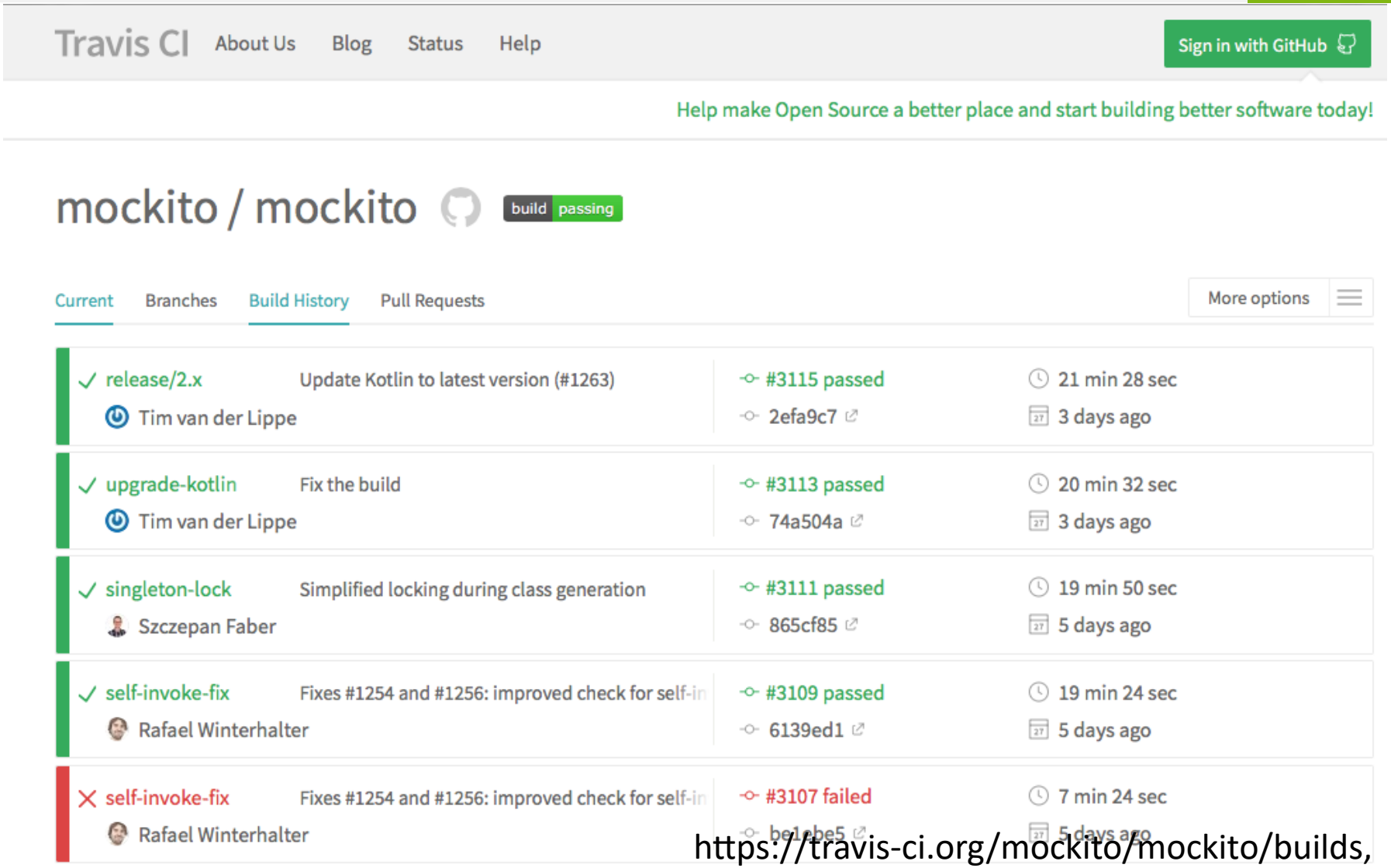
Continuous Integration

- Ständiges Zusammenfügen von Komponenten zu einer Anwendung.
- Ziel: Steigerung der Softwarequalität
- Üblicherweise werden neben dem Bau des Gesamtsystems auch
  - automatisierte Tests durchgeführt und
  - Qualitätsmetriken berechnet.

Continuous Integration: Prinzpien

- Gemeinsame Codebasis
- Automatisierte Übersetzung
- Kontinuierliche Test-Entwicklung
- Häufige Integration
- Integration in den Hauptentwicklungszweig
- Kurze Testzyklen
- Gespiegelte Produktionsumgebung
- Einfacher Zugriff
- Automatisiertes Reporting
- Automatisierte Verteilung

z.B. Travis CI



https://travis-ci.org/mockito/mockito/builds, Zugriff 27.11.2017, 22:30

z.B. Travis CI

Most popular mocking framework for Java

https://github.com/mockito/mockito, _Zugriff 27.11.2017, 22:30

z.B. Jenkins

Jenkins

Jenkins

Suchen anmelden

Jenkins

ENABLE AUTO REFRESH

Benutzer

Build-Verlauf

Build Warteschlange

recovery
recovery
android
android
android
android
android
android
android
android
android
android
android
android
android
android
android
android
android

All

S	W	Name ↓	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer
		android	44 Minuten (cm_manta-userdebug)	1 Stunde 14 Minuten (cm_jewel-userdebug)	30 Minuten
		android-build-all-lunches	2 Monate 24 Tage (#146)	4 Monate 16 Tage (#109)	17 Sekunden
		cm-build-all	8 Stunden 5 Minuten (#122)	Unbekannt	2 Minuten 1 Sekunde
		cm_daily_build_cycle	8 Stunden 5 Minuten (#235)	6 Monate 22 Tage (#25)	0.48 Sekunden
		recovery	31 Minuten (4f1cac2a9bac78321ed06c7b00360f34)	6 Minuten 32 Sekunden (9c5440be96d4d1eee566a4862337b4e0)	3 Minuten 20 Sekunden
		submission-test	3 Stunden 31 Minuten (gerrit-test-34890)	15 Stunden (gerrit-test-34681)	16 Minuten

Symbol: S M L

Legende RSS Alle Builds RSS Nur Fehlschläge RSS Nur jeweils letzter Build

Build-Umgebungs-Automatisierung

- Docker (https://www.docker.com)
- Vagrant (https://www.vagrantup.com)
- Chef (https://www.chef.io/chef/)

Was ist das Ziel?

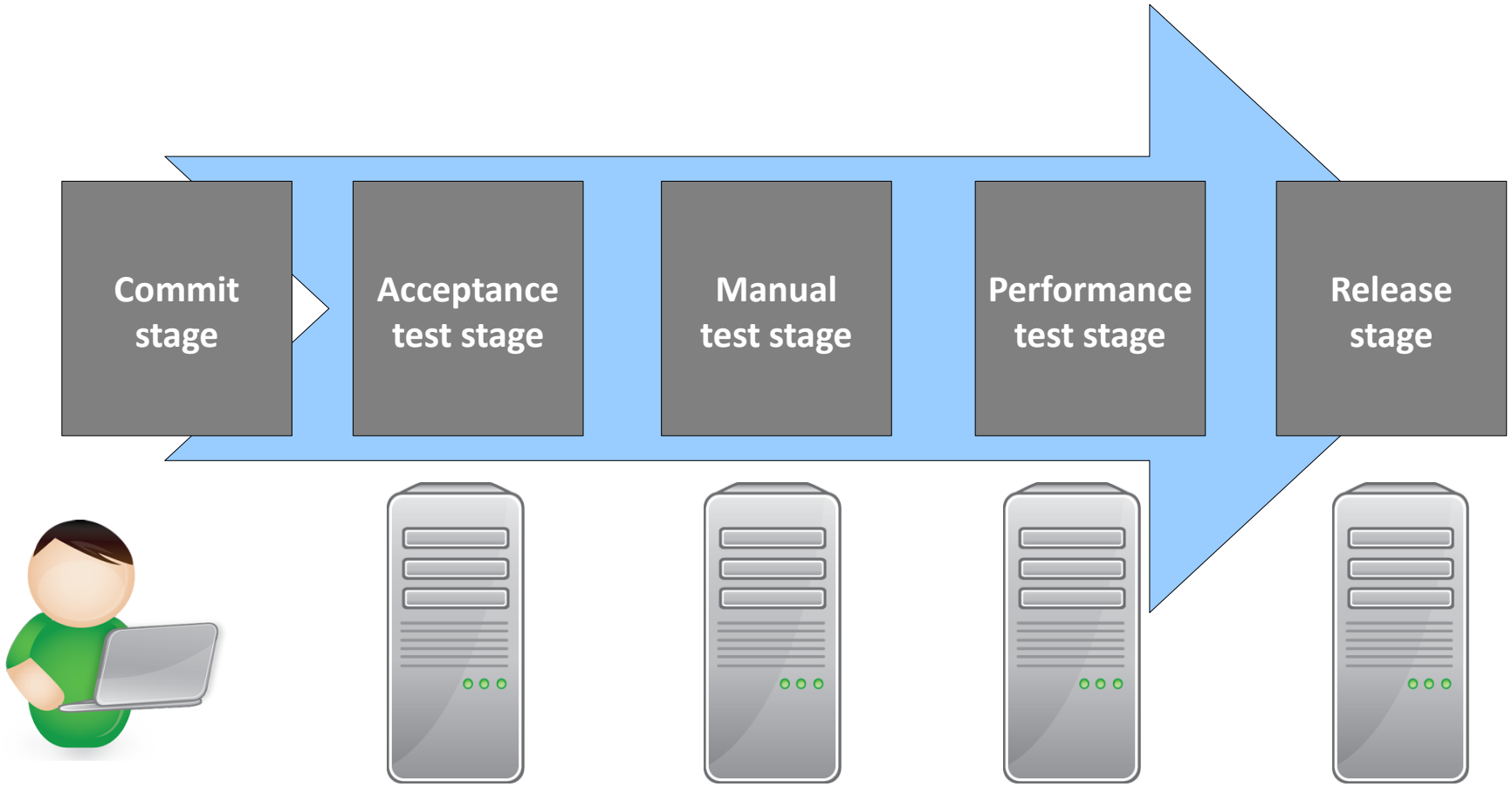
- Kontrolle über Ausführungsumgebung
- Vorinstallieren von Software
- Automatisierung von Installationsroutinen
- Wissenstransfer

Continuous Delivery

- Häufigste Fehler beim Release („Release Antipatterns“):
- **Manuelle** Software Deployments
- die Software wird erst **nach ihrer Fertigstellung** in einer der **Produktivumgebung** ähnlichen Testumgebung **getestet**
- **manuelle** Konfiguration der Produktivumgebung

Continuous Delivery = Continuous Integration + Release

Deployment-Pipeline

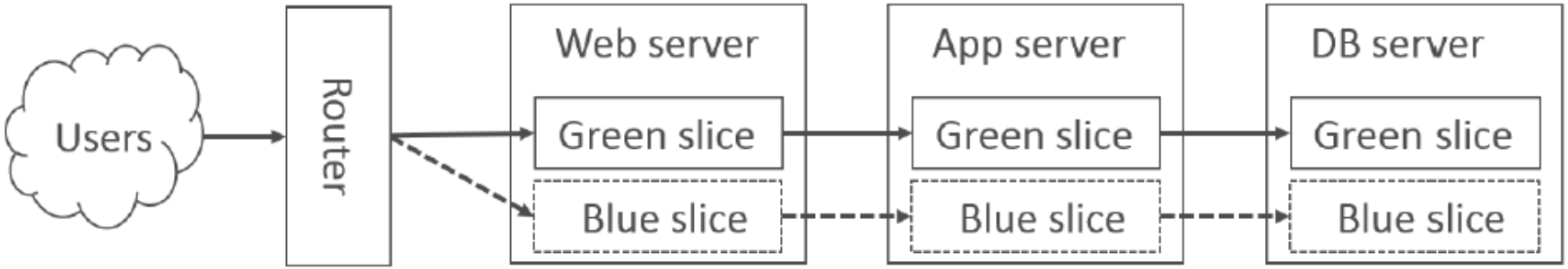


Deployment-Pipeline

- **Commit Stage:** Code wird kompiliert.
  - Durchführung von Unit Tests und Code Analyse
  - Packen der Software in ein Format (ELF, PE, JAR, WAR, …)
- **Acceptance Test Stage:** Test von User Stories.
  - Test von Anwendungsfälle und funktionalen Eigenschaften.
- **Manual Test Stage:** explorative Tests
- **Performance Test Stage:** Testen von nichtfunktionalen Eigenschaften
  - Performanz, Skalierbarkeit, Durchsatz, Verfügbarkeit…
- **Release Stage:** Software wird auf Produktivsystem installiert oder als installierbare Software gepackt

Innerhalb der Release Stage

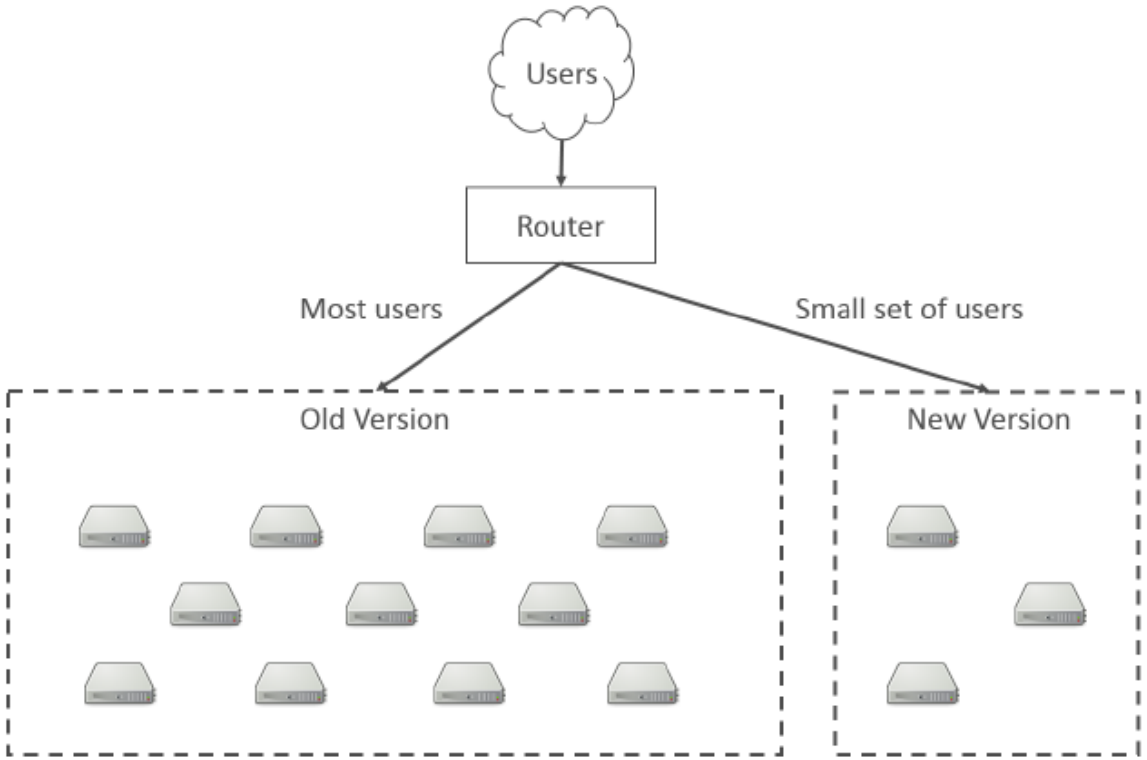
- **Blue-Green Deployments:** zwei identische Versionen der Produktivumgebung



- Bei Problemen: umschalten auf vorhergehendes Deployment
- **Continious Deployment** (Webapplikationen): Aktivieren und deaktivieren von Features (configs) über Flags

Innerhalb der Release Stage

- **Kanarienvogel-Release** (canary release): Release auf Teilmenge der Produktivsysteme



Zusammenfassung

- Softwarekonfigurationsmanagement ist zentraler Bestandteil der Softwareentwicklung
- Versionsmanagement verwaltet Konfigurationen
- Build Management steuert die Erstellung von Produkten
- Release Management verwaltet Erzeugung und Bereitstellung von neuen Softwareversionen für die Öffentlichkeit
- Continuous Delivery setzt Idee von Continuous Integration fort