



# VERSIONSKONTROLLSYSTEME IM SOFTWAREENGINEERING

GRUPPEN MITGLIEDER:

# INHALTSVERZEICHNIS

- Einleitung

- Vorstellung & Ziel der Präsentation
- Leitfragen vorstellen

- **Leitfrage 1: Vorteile verteilter vs. zentralisierter VCS**

- Zielsetzung von Versionskontrollsystmen
- Zentralisierte Systeme (CVCS)
- Verteilte Systeme (DVCS)
- Vergleich CVCS vs. DVCS
- Technische Darstellung (Git Graph)
- Zusammenfassung Leitfrage 1

# INHALTSVERZEICHNIS

- **Leitfrage 2: Branching-Strategien & Softwarequalität**
- Branching-Grundlagen
- Trunk-Based Development
- Feature Branching
- Forking Workflow
- Auswirkungen auf Softwarequalität
- Rollen-Spiel / Merge-Konflikt Demo
- Zusammenfassung Leitfrage 2
- **Fazit & Quellen**

## TEIL1: EINLEITUNG LEITFRAGE 1

Welche Vorteile bietet ein verteiltes Versionskontrollsystem im Vergleich zu einem zentralisierten Modell?

# ZIELSETZUNG VON VERSIONSKONTROLLSYSTEMEN

- Änderungen nachvollziehen
- Historie & alte Versionen wiederherstellen
- Paralleles Arbeiten im Team
- Dokumentation für spätere Bearbeitung
- Zitat: „Versionsverwaltung ist ein System, das Änderungen an Dateien über die Zeit protokolliert.“ [1]

# ZENTRALISIERTE SYSTEME

- Definition: Ein CVCS speichert das gesamte Projekt auf einem zentralen Server. Entwickler benötigen eine Verbindung, um Änderungen zu machen.
- Beispiele: Subversion (SVN), CVS
- Vorteile:
  - Klare zentrale Kontrolle
  - Einfache Rechteverwaltung
  - Geeignet für kleine Teams oder Projekte mit geringer Parallelität

# ZENTRALISIERTE SYSTEME

- Nachteile:
  - Serverausfall > gesamte Arbeit stoppt
  - Offline-Arbeiten nahezu unmöglich
  - Parallelität stark eingeschränkt, da alle Entwickler auf denselben Server zugreifen
- Arbeitsweise: Commits erfolgen nacheinander auf den Server, Änderungen werden linear gespeichert

# VERTEILTE SYSTEME

- Definition: DVCS speichert das komplette Repository lokal bei jedem Entwickler.
- Beispiele: Git, Mercurial
- Vorteile:
  - Offline-Arbeit möglich
  - Schnelle lokale Commits
  - Flexible Branching-Strategien
  - Hohe Ausfallsicherheit (jede Kopie ist ein Backup)

# VERTEILTE SYSTEME

- Nachteile:
  - Einarbeitung für neue Entwickler notwendig
  - Speicherbedarf bei sehr großen Projekten
- 
- Arbeitsweise: Jeder Entwickler kann unabhängig Änderungen commiten, testen, Branches anlegen und später in ein zentrales Repository pushen

# VERGLEICH ZENTRALISIERT (CVCS) VS. VERTEILTES SYSTEM (DVCS)

Merkmale	Zentralisiert (CVCS)	Verteiltes System (DVCS)
Speicherort	Server	Lokal bei jedem Entwickler
Offline-Funktionalität	Eingeschränkt	Voll möglich
Ausfallsicherheit	Gering	Hoch
Geschwindigkeit	Netzwerkabhängig	Lokal sehr schnell
Parallelität	Begrenzt	Sehr flexible

# TECHNISCHE DARSTELLUNG: GIT GRAPH

The image displays two screenshots illustrating Git commit graphs.

**Left Screenshot: GitKraken Desktop**

GitKraken Desktop interface showing a commit graph. The graph view on the left displays a timeline of commits across multiple branches. The main pane shows a list of commits with their messages, SHA, and author information. The commits are color-coded by branch: fix-e2e-tests (purple), debt/impro... (red), release/1... (blue), chore/string-i... (green), and debt/compose... (orange). The commit list includes:

- Adds support for mcp server installation
- Run unit tests on push to main branch because otherwise they oft...
- Unifies autolink enrichment keys for integrations in order to avoid ...
- Enforces code formatting check before running tests because it fr...
- Fixes e2e tests by supporting for type change file status in Git pars...
- Fix prettier
- Adds unit tests for the string interpolate function
- Extracts host app name lookup into reusable helper
- Expands command and integration deep link support
- Updates CHangelog (patch)
- Bumps to v17.4.1
- Updates CHangelog
- Adds expand/collapse all buttons to Composer
- Encodes/decodes bootstrap state to base64 - Avoids JSON encod...
- Adds styles for diff rendering performance
- Encodes/decodes bootstrap state to base64 - Avoids JSON encod...
- Updates eslint plugin to play with concurrency
- Fixes onboarding step tracker not being updated on prev click
- Adds new correlationId to output
- Adds correlation ID tracking to AI events
- Adds correlation ID tracking to AI events
- Adds correlation ID tracking to AI events

**Right Screenshot: GitLens in VS Code**

GitLens extension in VS Code showing a commit graph. The interface includes a sidebar for branches (All Branches) and a main pane for the 'vscode-gitlens' branch. The commit history is displayed with colored nodes corresponding to the branches shown in the GitKraken screenshot. The commits are:

- Adds gates and deeplink command support
- Updates storage key and splits automatic and
- Adds support for mcp server installation
- Run unit tests on push to main branch
- Unifies autolink enrichment keys for integrati...
- Enforces code formatting check before running
- Fixes e2e tests by supporting for type change
- Fix prettier
- Adds unit tests for the string interpolate functi...
- Extracts host app name lookup into reusable h...
- Expands command and integration deep link s...
- Updates CHangelog (patch)

# KONSEQUENZEN FÜR ARBEITSPROZESSE

- Parallelle Entwicklung möglich
- Lokale Commits & Tests beschleunigen Feedbackzyklen
- Unterschiedliche Merge-Methoden (Fast-Forward, Rebase, Merge Commit)
- Weniger Absprachen notwendig als bei CVCS

# ZUSAMMENFASSUNG LEITFRAGE 1

- DVCS: flexibel, offline, schnelle Commits, hohe Ausfallsicherheit
- CVCS: einfache Struktur, zentrale Kontrolle, Offline-Arbeit eingeschränkt
- DVCS ist vorteilhaft für Teams, parallele Arbeit und komplexe Projekte

## TEIL2: EINLEITUNG LEITFRAGE 2

- Wie beeinflussen unterschiedliche Branching-Strategien die Softwarequalität, insbesondere im Hinblick auf Fehlerhäufigkeit, Code-Stabilität und Merge-Konflikte?

# BRANCHING-GRUNDLAGEN

- Branching Strategien? Was ist das überhaupt?

Branch=eigenständiger Entwicklungszweig VCS ohne Einfluss am Hauptentwicklungsstand

Branching Strategien definieren damit die Regeln, wie man parallel an einem Branch arbeiten kann

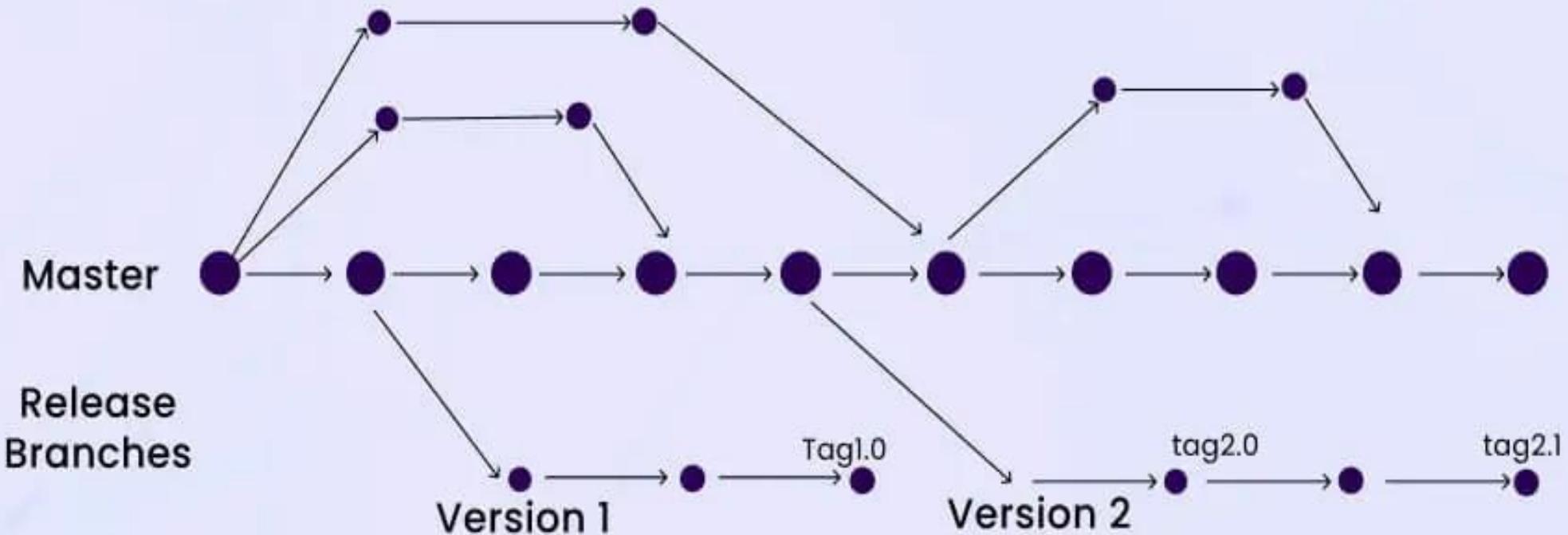
- Fehlerhäufigkeit, Code-Stabilität und Merge-Konflikte? Ich versteh nur Bahnhof!

->Fehlerhäufigkeit: wie oft treten Softwarefehler in einem bestimmten Zeitraum auf

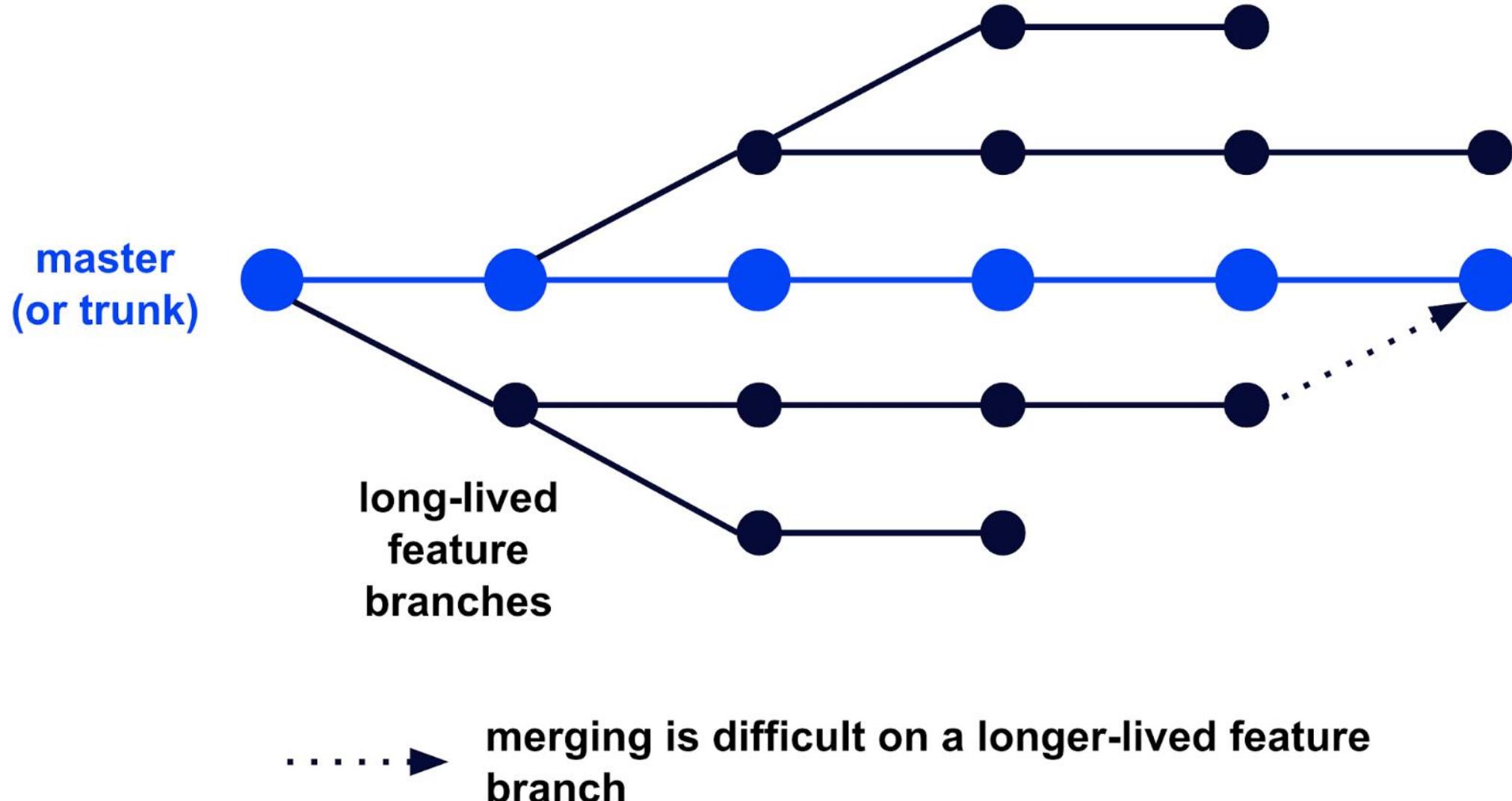
->Code-Stabilität: Wie verlässlich und vorhersehbar verhält sich Code bei Änderungen

->Merge-Konflikte: Änderungen aus unterschiedlichen Branches können nicht automatisch zusammengeführt werden

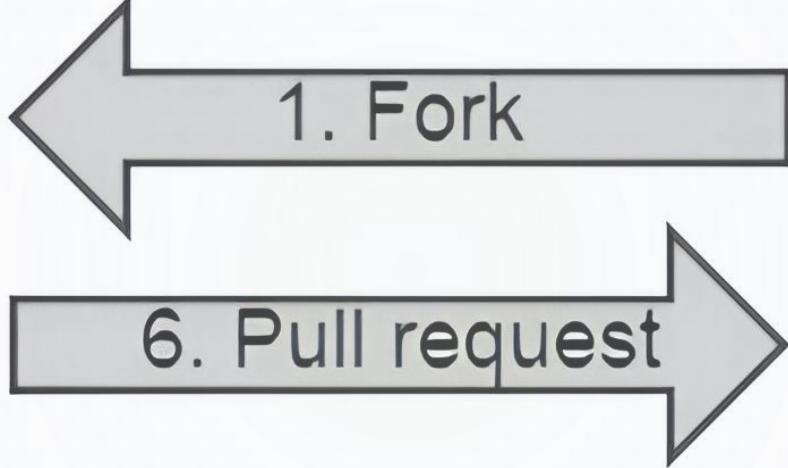
# Workflow of Trunk-based Development



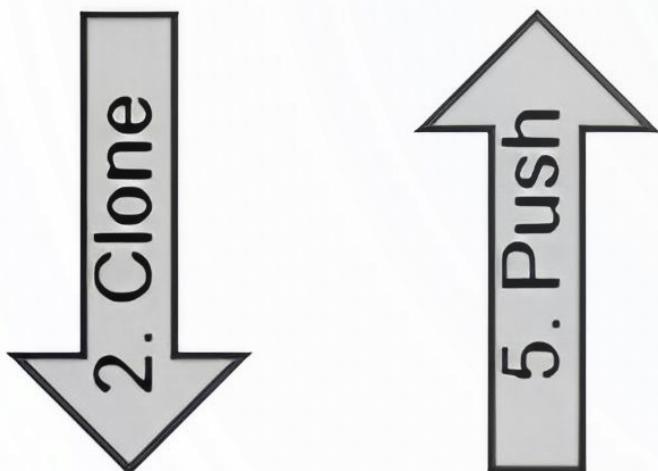
## ⌚ Feature-branched development



Your GitHub repo  
[github.com/you/coolgame](https://github.com/you/coolgame)



Joe's GitHub repo  
[github.com/Igelkaempfer/Versionsverwaltungs-system-Poster](https://github.com/Igelkaempfer/Versionsverwaltungs-system-Poster)



Your computer

3. Update a file

4. Commit

# FORKING WORKFLOW

(Open Source Konzept)

# AUSWIRKUNGEN AUF SOFTWAREQUALITÄT

(FEHLERHÄUFIGKEIT, CODE-STABILITÄT UND MERGE-KONFLIKTE)

## TRUNK-BASED DEVELOPMENT

Fehlerhäufigkeit: Hoch

Fehler früh sichtbar

Kleine Änderungen werden schneller getestet

Code-stabilität: Hoch

Kurze Branch-Lebensdauer und Automated CI Pipeline

Merge-konflikte: Niedrig

Branches, häufiges Rebasing und sofortiges Auflösen

## FEATURE BRANCHING DEVELOPMENT

Fehlerhäufigkeit: Mittel

Fehler oft erst beim Merge oder nach längerer Entwicklungszeit

Code-stabilität: Mittel

abhängig von Branch-Dauer und Testtiefe vor dem Merge

Merge-konflikte: Mittel

abhängig von Parallelentwicklung, Branch-Lebensdauer und Release Rhythmus

## FORKING WORKFLOW DEVELOPMENT

Fehlerhäufigkeit: Hoch

Integration erfolgt verzögert

Code-stabilität: Niedrig

häufig veraltete Codebasen, umfangreiche Divergenzen

Merge-konflikte: Hoch

seltene Merges, größere Code-Divergenz zwischen Teams oder Repositories

## ZUSAMMENFASSUNG LEITFRAGE 2

Trunk-Based Development am geeignetsten für kleine bis mittelgroße Teams, die eng zusammenarbeiten.

Feature Branching Development am geeignetsten mittelgroße Teams, die eng zusammenarbeiten.

Forking Workflow am geeignetsten für Open-Source-Projekte oder sehr große verteilte Teams, bei denen viele externe Mitwirkende ohne Schreibrechte am Hauptrepository arbeiten.