Technische Hochschule
Brandenburg
University of
Applied Sciences
Fachbereich
Wirtschaft

Versionsverwaltung im Softwareengineering

Projektbetreuer*innen:
Michael Höding, Ralf Teusner

1 Unser Projektstudium

Das Projektstudium in Versionskontrollsysteme helfen dabei, Veränderungen im Code zu erkennen, frühere Versionen zurückzuholen und gleichzeitige Arbeiten übersichtlich zu organisieren. Dadurch wird ersichtlich, wer was geändert hat, Fehler lassen sich einfacher finden und das Projekt bleibt auch später noch gut bearbeitbar. Vor allem bei gemeinsamer Arbeit stellen sie sicher, dass alle Schritte dokumentiert sind, sodass niemand ratlos vor den Anpassungen steht.
Dieses Grundprinzip wird im Pro Buch prägnant beschrieben: „Versionsverwaltung ist ein System, das die Änderungen an einer oder einer Reihe von Dateien über die Zeit hinweg protokolliert, sodass man später auf eine bestimmte Version zurückgreifen kann." Damit wird deutlich, dass VCS nicht nur Speicherwerkzeuge sind, sondern ein wesentliches Element für Qualitätssicherung und effiziente Zusammenarbeit.

2 Zentralisierte Systeme

Zentrale Systeme – zum Beispiel Subversion – arbeiten mit einem einzigen Haupt-Speicherort, zu dem jeder Entwickler eine Verbindung braucht.

2.1 Vorteile und Nachteile

Vorteile	Nachteile
Zentrale Kontrolle	Serverausfall stoppt Entwicklung
Klare Rechteverwaltung	Kaum Offline-Arbeit möglich
Gut für kleine Teams	Starke Netzwerkabhängigkeit
Einfache Struktur	Wenig flexibel für parallele Arbeit

3 Verteilte Systeme

DVCS speichern das komplette Repository lokal bei jedem Nutzer. „In einem DVCS . . . erhält jeder Anwender nicht einfach nur den jeweils letzten Zustand des Projektes von einem Server: Er erhält stattdessen eine vollständige Kopie des Repositories." Dadurch entstehen flinke Prozesse, zuverlässige Systeme und team über greifende Arbeitsweisen.

3.1 Vorteile von verteilten Systemen

Vorteil	Nachteil
Geschwindigkeit	Erhöhter Aufwand beim Erlernen des Systems
Ausfallsicherheit	Struktur kann bei großen Projekten unübersichtlich werden
Kollaboration	Konflikte beim Zusammenführen treten häufiger auf
Offline-Funktionalität	Für neue Nutzer oft weniger intuitiv
Branching& Merging	Verschiedene Remote-Konzepte können zu Verwirrung führen

4 Konsequenzen für Arbeitsprozesse

Zentrale Systeme brauchen häufiger Absprachen als schrittweise Prozesse. DVCS machen es möglich, gleichzeitig Zweige zu nutzen, Änderungen lokal abzulegen oder unterschiedliche Methoden zur Zusammenführung einzusetzen.

5 Technische Unterschiede

Zentralisierte Systeme speichern Bearbeitungen hintereinander – also als einfache Liste. Im Gegensatz dazu greift Git auf einen gerichteten, nichtzyklischen Graphen zurück, weil er gleichzeitige Entwicklungsstränge effizienter darstellen kann. Zweige wiegen kaum etwas, weil sie lediglich auf eine Version verweisen. Durch diesen Aufbau funktioniert die getrennte Bearbeitung besser – gleichzeitig bleibt die Integration später einfach. Die Unterlagen erläutern das anschaulich. „Branching ist ein zentraler Bestandteil von Git, der besonders einfach und schnell funktioniert." Diese Bauweise erlaubt anpassbare Abläufe, rasche lokale Schritte oder eine klare Trennung unterschiedlicher Entwicklungslinien. Branching ist ein zentraler Bestandteil von Git, der besonders einfach und schnell funktioniert.

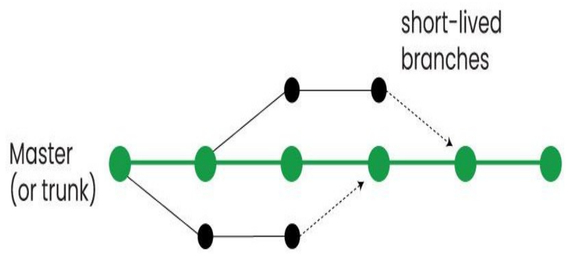
6 Architekturvergleich

Vorteil	Nachteil
Geschwindigkeit	Erhöhter Aufwand beim Erlernen des Systems
Ausfallsicherheit	Struktur kann bei großen Projekten unübersichtlich werden
Kollaboration	Konflikte beim Zusammenführen treten häufiger auf
Offline-Funktionalität	Für neue Nutzer oft weniger intuitiv
Branching& Merging	Verschiedene Remote-Konzepte können zu Verwirrung führen

7 GIT und Branching-Strategien

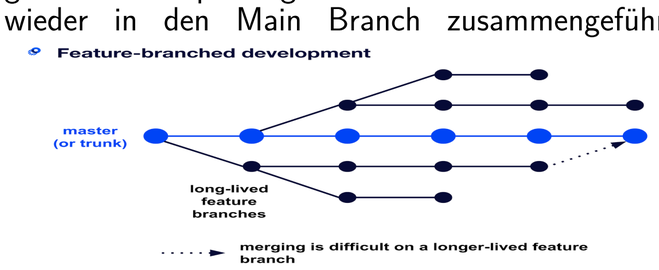
7.1 Trunk-Based Development

Trunk-Based Development ist ein agiler Ansatz, bei dem alle Entwickler kontinuierlich in einen gemeinsamen Hauptzweig („Trunk") integrieren. Änderungen werden in kleinen, häufigen Commits vorgenommen, wodurch Merge-Konflikte reduziert und die Software stets in einem lauffähigen Zustand gehalten wird. Feature-Flags ermöglichen das schrittweise Aktivieren neuer Funktionen. Dieser Ansatz fördert schnelle Feedbackzyklen, höhere Codequalität und eine stabilere Release-Pipeline.



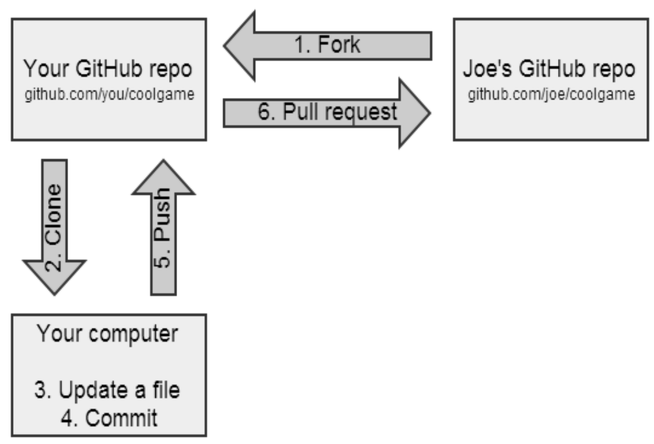
7.2 Feature Branching

Feature Branching ist eine gängige Strategie in der Versionskontrolle (z.B. Git), bei der Entwickler für jede neue Funktion, Fehlerbehebung oder jedes Experiment einen separaten Branch von der Hauptentwicklungslinie erstellen. Dadurch wird die Isolation des Codes gewährleistet. Die Arbeit kann unabhängig und ohne Risiko, die stabile Hauptversion zu beeinträchtigen, durchgeführt werden. Erst nach Fertigstellung und erfolgreicher Überprüfung wird der Feature Branch wieder in den Main Branch zusammengeführt.



7.3 Forking Workflow

Der Forking-Workflow unterscheidet sich fundamental von anderen Arbeitsweisen, da er keine Schreibrechte für alle Entwickler auf einem zentralen Repository voraussetzt. Stattdessen erstellt jeder Beteiligte eine serverseitige Kopie (Fork) des Projekts. Dieses Modell hat sich als De-facto-Standard in der Open-Source-Entwicklung etabliert, da es eine sichere Zusammenarbeit ohne zentrale Verwaltung von Zugriffsrechten ermöglicht. Dabei fungieren eine oder mehrere vertrauenswürdige Personen (Maintainer) als „Gatekeeper". Nur sie besitzen die Berechtigung, Änderungen in das offizielle Haupttrepository zu integrieren, wodurch eine klare Hierarchie und Qualitätssicherung gewährleistet wird.



8 Wie beeinflussen unterschiedliche Branching-Strategien die Softwarequalität?

	Trunk-Based Development	Feature Branching	Forking Workflow
Fehlerhäufigkeit	Niedriger, da kontinuierliche Integration Fehler früh sichtbar macht und kleine Änderungen schneller getestet werden	Mittel, da Fehler oft erst beim Merge oder nach längerer Entwicklungszeit erkannt werden	Höher, da Integration oft verzögert erfolgt und viele Änderungen gebündelt werden
Code-Stabilität	Hoch, da kontinuierliche Builds, Tests und kurze Branch-Lebensdauer Stabilität erzwingen	Mittel, abhängig von Branch-Dauer und Testtiefe vor dem Merge	Niedriger, häufig veraltete Codebasen, umfangreiche Divergenzen
Merge-Konflikte	Gering, da kurze Branches, häufiges Rebased und sofortiges Auflösen	Mittel, abhängig von Parallelentwicklung, Branch-Lebensdauer und Release-Rhythmus	Hoch, da seltene Merges, größere Code-Divergenz zwischen Teams oder Repositories

Dieses Poster ist im Rahmen der Lehrveranstaltung *Projektstudium - Wissenschaftliches Arbeiten* im Sommersemester 2022 entstanden.

Ihr Kontakt: Prof. Dr. Michael Höding
TH Brandenburg, Postfach 2132
D-14737 Brandenburg, Germany
E-Mail: hoeding@th-brandenburg.de

Referenzen ->