Entwicklungsumgebung und Tools

Stellen Sie sich vor, Sie bauen ein komplexes Gebäude: Ohne die richtigen Werkzeuge, Pläne und eine koordinierte Zusammenarbeit im Team wäre das Projekt zum Scheitern verurteilt. Genauso verhält es sich in der Softwareentwicklung – die Wahl und der gezielte Einsatz von Entwicklungsumgebungen und Tools sind entscheidend für den Erfolg moderner Softwareprojekte.

In diesem Kapitel lernen Sie die wichtigsten Werkzeuge und Methoden kennen, die den Entwicklungsprozess effizient, sicher und nachvollziehbar machen. Dazu gehören Versionsverwaltungssysteme wie Git, Build- und Dependency-Management-Tools, Continuous Integration (CI) sowie bewährte Workflows für die Zusammenarbeit im Team. Sie erfahren, wie diese Tools ineinandergreifen, typische Fehler vermeiden helfen und die Qualität der Software nachhaltig sichern.

> 🔍 **Vertiefung:** Die richtige Kombination aus Tools und Prozessen ermöglicht nicht nur eine reibungslose Entwicklung, sondern auch eine schnelle Anpassung an neue Anforderungen und eine kontinuierliche Verbesserung der Software.

Quellen

- Git SCM Documentation
- Continuous Integration – Martin Fowler
- Apache Maven

Versionsverwaltung mit Git

Git ist ein **verteiltes Versionsverwaltungssystem (VCS – Version Control System)**, das zur Nachverfolgung von Änderungen an Dateien – insbesondere Quellcode – dient. Es wurde 2005 von **Linus Torvalds** (dem Entwickler des Linux-Kernels) entwickelt und ist heute der **Standard in der Softwareentwicklung**.

🔧 **Hauptzweck:**

- Nachverfolgung aller Änderungen an Dateien
- Zusammenarbeit mehrerer Entwickler:innen
- Wiederherstellen früherer Zustände
- Paralleles Arbeiten an verschiedenen Funktionen

Repositories

Das Repository stellt das Fundament der Versionsverwaltung dar. Ein **Repository** (kurz: *Repo*) ist das **zentrale Projektarchiv**, in dem Git alle Versionen und Metadaten eines Projekts speichert.

📦 **Arten von Repositories:**

Typ	Beschreibung	Beispiel
Lokales Repository	Auf dem eigenen Rechner gespeichert. Enthält vollständige Versionsgeschichte.	`.git`-Ordner im Projektverzeichnis
Remote Repository	Zentrale Version des Projekts auf einem Server (z. B. GitHub, GitLab).	`https://github.com/Schule/Projekt.git`

⚙️ Grundstruktur des Git-Repostroy

Ein Git-Repository besteht aus:

- **Arbeitsverzeichnis (Working Directory)** → enthält aktuelle Dateien
- **Staging Area (Index)** → Bereich, in dem Änderungen vorbereitet werden
- **Repository (.git)** → speichert alle Commits und Metadaten dauerhaft

```
Working Directory → Staging Area → Repository
```

💡 Beispiel:

```
git init                 # Neues lokales Repository erstellen
git remote add origin https://github.com/schule/projekt.git  # Remote-Repo
hinzufügen
```

🧱 `Commit` – Versionierung der Änderungen

Ein **Commit** ist eine **gespeicherte Momentaufnahme (Snapshot)** des Projekts zu einem bestimmten Zeitpunkt. Jeder Commit enthält:

- den Autor
- das Datum
- eine eindeutige **Commit-ID (Hash)**
- eine **Commit-Nachricht**
- die Änderungen (Delta) zum vorherigen Zustand

🔄 Commit-Workflow:

1. Änderungen im Arbeitsverzeichnis durchführen
2. Mit `git add` zur Staging Area hinzufügen
3. Mit `git commit` dauerhaft speichern

```
git add main.java
git commit -m "Implementiere Login-Funktion"
```

🧩 **Beispielhafter Commit-Verlauf:**

Commit-ID	Beschreibung	Datum	Autor
a1b2c3d	Projektinitialisierung	12.10.2025	Anna
d4e5f6g	Login-Funktion implementiert	13.10.2025	Max
h7i8j9k	Bugfix: Passwortprüfung korrigiert	14.10.2025	Max

> 💡 Jeder Commit ist ein „Speicherpunkt“ in der Projektgeschichte – man kann jederzeit darauf zurückspringen oder vergleichen.

🌿 `Branch` **– paralleler Entwicklungsstrang**

Ein **Branch (Zweig)** ist eine **unabhängige Entwicklungslinie** innerhalb des Repositories. Mit Branches kann man neue Funktionen entwickeln, ohne den Hauptcode (meist **main** oder **master**) zu gefährden.

⚙️ **Typischer Ablauf:**

1. Neuer Branch für eine Funktion erstellen
2. Änderungen durchführen
3. Branch testen
4. Änderungen wieder in den Hauptzweig **mergen**

```
git branch feature-login      # Neuen Branch erstellen
git checkout feature-login    # In den Branch wechseln
# ... Entwicklung ...
git checkout main             # Zurück zum Hauptbranch
git merge feature-login       # Änderungen übernehmen
```

📊 **Beispielhafte Struktur:**

```
main
 ├── feature-login
 ├── feature-database
 └── bugfix-session
```

> 💡 Branches erlauben **parallele Arbeit**, **Experimentieren** und **kontrollierte Integration** neuer Features.

🏷️ `Tag` **– markierte Version (z. B. Release)**

Ein **Tag** ist eine **feste Markierung** auf einem bestimmten Commit – oft für **wichtige Versionen** wie Releases (z. B. `v1.0.0`). Tags verändern sich **nicht** und dienen als **Referenzpunkte** in der Versionshistorie.

⚙️ **Beispiel:**

```
git tag -a v1.0.0 -m "Erste stabile Version"
git push origin v1.0.0
```

📦 **Verwendung:**

- Markierung von **Release-Ständen** (z. B. für Deployment oder Archivierung)
- Wiederherstellung einer bestimmten Version (`git checkout v1.0.0`)

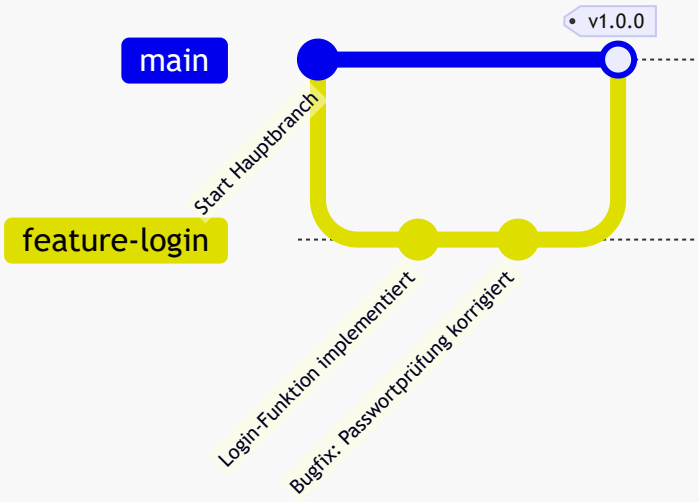
> 💡 Tags sind „Lesezeichen“ im Projektverlauf – sie helfen, wichtige Versionen leicht wiederzufinden.

🔗 **Zusammenspiel von Repository, Commits, Branches und Tags**

Im folgenden wird ein typischer Ablauf skizziert:

1. Entwickler erstellt Branch „feature-login“
2. Erstellt mehrere Commits (Änderungen)
3. Merge mit Hauptbranch
4. Version wird als `v1.0.0` getaggt

➡️ So entsteht eine **nachvollziehbare, reproduzierbare Entwicklungs-Historie**.



Das Diagramm zeigt, wie ein Feature-Branch entsteht, mehrere Commits gemacht werden, und nach dem Merge ein Tag für die Version gesetzt wird.

🧠 **Typische Fehler und Best Practices**

Typische Fehler	Bessere Praxis
Alles in einem Commit speichern	Kleine, logisch zusammenhängende Commits

Typische Fehler	Bessere Praxis
Ohne Branch direkt auf `main` entwickeln	Immer Feature-Branches verwenden
Leere oder unverständliche Commit-Messages	Klare, beschreibende Nachrichten
Keine Tags setzen	Wichtige Versionen taggen (`v1.0.0`, `v1.1.0`, ...)