

Verteilte Systeme und Komponenten

Versionskontrollsysteme

Source Code Management (SCM)

Version Control System (VCS)

Roland Gisler



Inhalt

- Grundlagen – was ist ein Versionskontrollsystem (VCS)?
- Wie arbeitet man mit einem VCS?
- Was gibt es für unterschiedliche Konzepte?
- Verschiedene Beispiele / Produkte
- Verschiedene Benutzerschnittstellen
- Konkret: Arbeit mit git und HSLU GitLab
- Zusammenfassung

Lernziele

- Sie kennen die Aufgaben eines Versionskontrollsystems und können grundlegend damit arbeiten.
- Sie kennen die verschiedenen Konzepte und Arten von Versionskontrollsystemen.
- Sie können mit verschiedenen (Client-)Werkzeugen von Versionskontrollsystemen alleine und im Team arbeiten.

Grundlagen

Ziel und Zweck von VCS

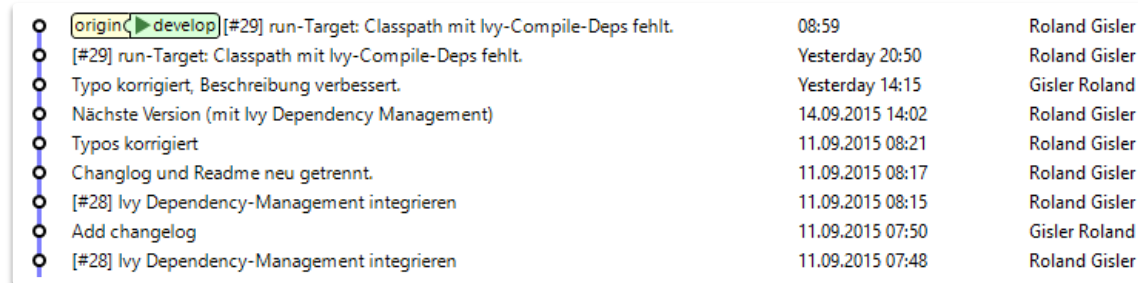
- Versionskontrollsysteme werden vorwiegend (aber nicht nur) in der Softwareentwicklung genutzt.
 - Alternativ als SCM (source control management) bezeichnet.
- Sie sind für eine eher «technisch» orientierte Nutzung konzipiert.
- Das Hauptziel ist:

Die Abfolge aller Bearbeitungsschritte an Artefakten benutzerbezogen und detailliert zeitlich nachvollziehen zu können.

- Vereinfacht formuliert:
«**Wer** hat **wann** und aus welchem **Grund** in welcher **Datei** welche **Änderungen** vorgenommen?»

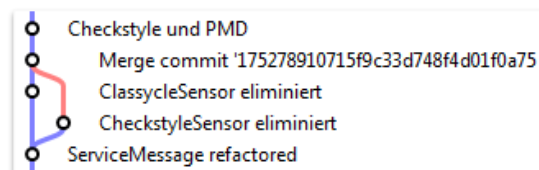
VCS - Grundlagen

- System, welches die zeitliche Entwicklung von Artefakten festhält und jederzeit einen Rückgriff auf alte Änderungsstände erlaubt.
 - so genannte Revisionen (Überarbeitungsstände).



○ origin → develop	[#29] run-Target: Classpath mit Ivy-Compile-Deps fehlt.	08:59	Roland Gisler
○	[#29] run-Target: Classpath mit Ivy-Compile-Deps fehlt.	Yesterday 20:50	Roland Gisler
○	Typo korrigiert, Beschreibung verbessert.	Yesterday 14:15	Gisler Roland
○	Nächste Version (mit Ivy Dependency Management)	14.09.2015 14:02	Roland Gisler
○	Typos korrigiert	11.09.2015 08:21	Roland Gisler
○	Changelog und Readme neu getrennt.	11.09.2015 08:17	Roland Gisler
○	[#28] Ivy Dependency-Management integrieren	11.09.2015 08:15	Roland Gisler
○	Add changelog	11.09.2015 07:50	Gisler Roland
○	[#28] Ivy Dependency-Management integrieren	11.09.2015 07:48	Roland Gisler

- Ist für konkurrenzierende Zugriffe und Modifikationen ausgelegt.
 - Teamarbeit auf gemeinsamen Quellen.
 - Automatisches Merging bei Konflikten (soweit möglich).
 - Entweder zentrale Datenhaltung oder auch verteilt!
 - **Kein** Ersatz für fehlende Koordination!



Abgrenzung zu Filesharing-Diensten

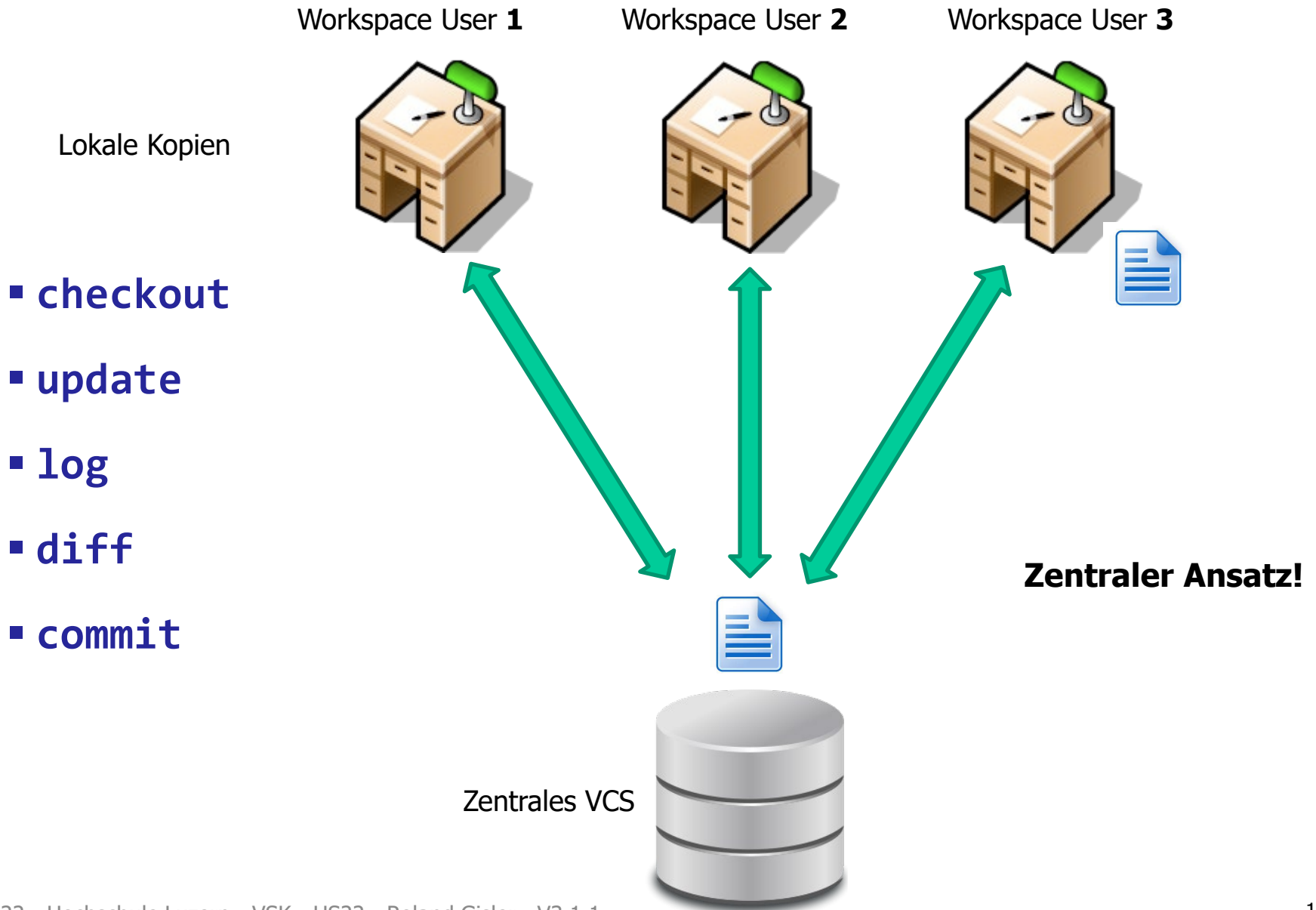
- Synchronisieren Dateien (inkl. Verzeichnisstrukturen) zwischen verschiedenen Rechnern und/oder der Cloud.
 - Typisch automatisch und im «Hintergrund».
- Artefakte werden bei **jeder** Veränderung bzw. bei jedem Speichern **sofort** übertragen.
 - Jedoch keine Garantie, kann auch verzögert erfolgen.
- Erzeugen teilweise auch Revisionen (pro Artefakte).
 - Meist stark beschränkte Anzahl an Revisionen.
- Beispiele: Dropbox, OneDrive, SWITCHdrive, Tresorit etc.

VCS – Abgrenzung und Fokus

- Versionskontrollsysteme sind **weder** Backupsystem noch Filesharing-Dienst (OneDrive etc.) und auch kein DMS.
- Bewusster Umgang: **Sie** als Entwickler*in **bestimmen, wann** eine **neue Version festgeschrieben** wird!
- Versionskontrollsystem haben einen anderen Fokus:
Nachvollziehbarkeit von Änderungen
 - Workflows und Koordination in (verteilten) Teams.
- Änderungen werden als sogenannte «Changesets» innerhalb einer Transaktion gespeichert.
 - **1..n** Dateiartefakte, die von einem konsistenten Zustand **z_1** zum nächsten konsistenten Zustand **z_2** führen.

Arbeit mit einem VCS

Grundlegende Arbeit mit einem VCS (Zentral)



Grundlegende Arbeit mit einem VCS

- **checkout** - von einem Projekt eine lokale Arbeitskopie erstellen.
 - Auf dieser Kopie wird dann gearbeitet.
 - **update** - Änderungen Dritter in lokaler Arbeitskopie aktualisieren.
 - Periodisch oder nach Bedarf, aber unbedingt vor einem Commit.
 - **log** - Bearbeitungsgeschichte der Artefakte ansehen.
 - Wer hat wann und warum welche Artefakte geändert?
 - **diff** - verschiedene Revisionen miteinander vergleichen.
 - Fremde (oder auch eigene) Änderungen nachvollziehen.
 - **commit** - Artefakte in das Repository schreiben (auch: **checkin**).
 - Lokale Veränderungen in das Repository zurückschreiben.
 - Veränderungen werden für Dritte bei einem Update sichtbar.
- ➔ Commit **immer** mit **Kommentar*** und ggf. einer Issue-Number!

Simple lokale Versionsverwaltung mit git

- Wenn Sie **git** installiert haben, funktioniert das auch rein **lokal!**

git init

Erzeugt ein neues Repo.

git add .

Bezieht (neue) Dateien mit ein.

git commit -a -m "Message"

Schreibt geänderte fest (Changeset).

git log

Zeigt die Geschichte des Repos an.

git status

Zeigt den lokalen Änderungsstatus an.

➔ Natürlich ist das mit einem grafischen Client alles viel einfacher.

Demo / Screenshot's

- git mit einem lokalen Repository einsetzen:

EP_11_SC01_GitLokal.mp4

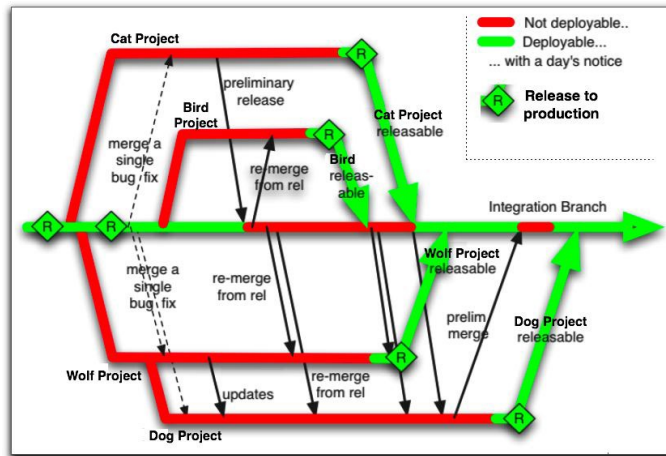


Tagging – markieren von Revisionsständen

- Markieren eines bestimmten Revisionsstandes mit einem Namen.
 - Eine textuelle, einfach identifizierbare Marke oder Version.
→ erleichtert die spätere Selektion dieses Standes.
 - Beispiele: **1.4.3** **1.6.1** **1.5.2beta**
- Sehr nützlich bei einem **Release** eines Produktes.
 - Als Markierung von Meilensteinen, Testversionen und/oder Auslieferungsständen, für einfachere Selektion.
- Tagging wird von den Systemen sehr unterschiedlich realisiert:
 - Nur eine Markierung in jeder Datei. (z.B. bei cvs)
 - Kopie aller Artefakte in ein anderes Verzeichnis. (z.B. bei Subversion)
 - Identifikation eines Änderungsstandes des gesamten Dateisystems. (vereinfacht formuliert, z.B. bei git, hg)

Branching

- Voneinander getrennte Entwicklungszweige, lokal oder zentral, für:
 - Prototypen, Tests und/oder Experimente.
 - Bugfixing bereits geschlossener Versionen.
 - Professionelle (Änderungs-)Workflows (z.B. [GitFlow](#) etc.)



- Wenn es sich nicht um «Wegwerf»-Entwicklungen handelt ist später ein Merging möglich/notwendig.
 - Läuft idealerweise (halb-)automatisch ab.
 - Kann aber auch sehr aufwändig (manuell) werden.

Was verwaltet man in einem VCS?

- Ausschliesslich **Quell**-Artefakte einchecken!
 - Sourcen, Konfigurationen, ggf. Dokumentation
 - Beispiele für Java: `*.java`, `*.properties` etc.
- Aber **NIE** Artefakte die generiert bzw. erzeugt werden können
 - Beispiele: `*.class`-Dateien, erzeugte HTML-Reports etc.
 - ➔ Konkret: `./target/**` **NIE** einchecken!
- In der Regel bieten die VCS-Systeme Hilfen an, um ausgewählte Verzeichnisse/Dateien automatisch zu ignorieren.
 - Beispiel für git: `.gitignore`-Datei mit Filtereinträgen.
 - ➔ In unseren HSLU-Projekten bereits vorbereitet.



Konzeptionelle Unterschiede

Was gibt es für Unterschiede?

- VCSs werden in banalen Vergleichen häufig «in einen Topf» geworfen, zumal die Befehle zum Teil identisch lauten.
 - Gefährliche Vereinfachung, weil zwar die Ziele identisch, aber die Lösungswege grundverschieden sind.
- Beispiele für Konzeptunterschiede:
 - **Zentrale** oder **verteilte** Systeme.
 - **Optimistische** und **pessimistische** Lockverfahren.
 - Versionierung auf der Basis einer **Datei**, der **Verzeichnisstruktur** (FS) oder der **Änderung** (changeset).
 - **Transaktions**unterstützung (vorhanden oder nicht).
 - Verschiedene **Zugriffsprotokolle** und Sicherheitsmechanismen.
 - **Integration** in Webserver (vorhanden oder nicht).

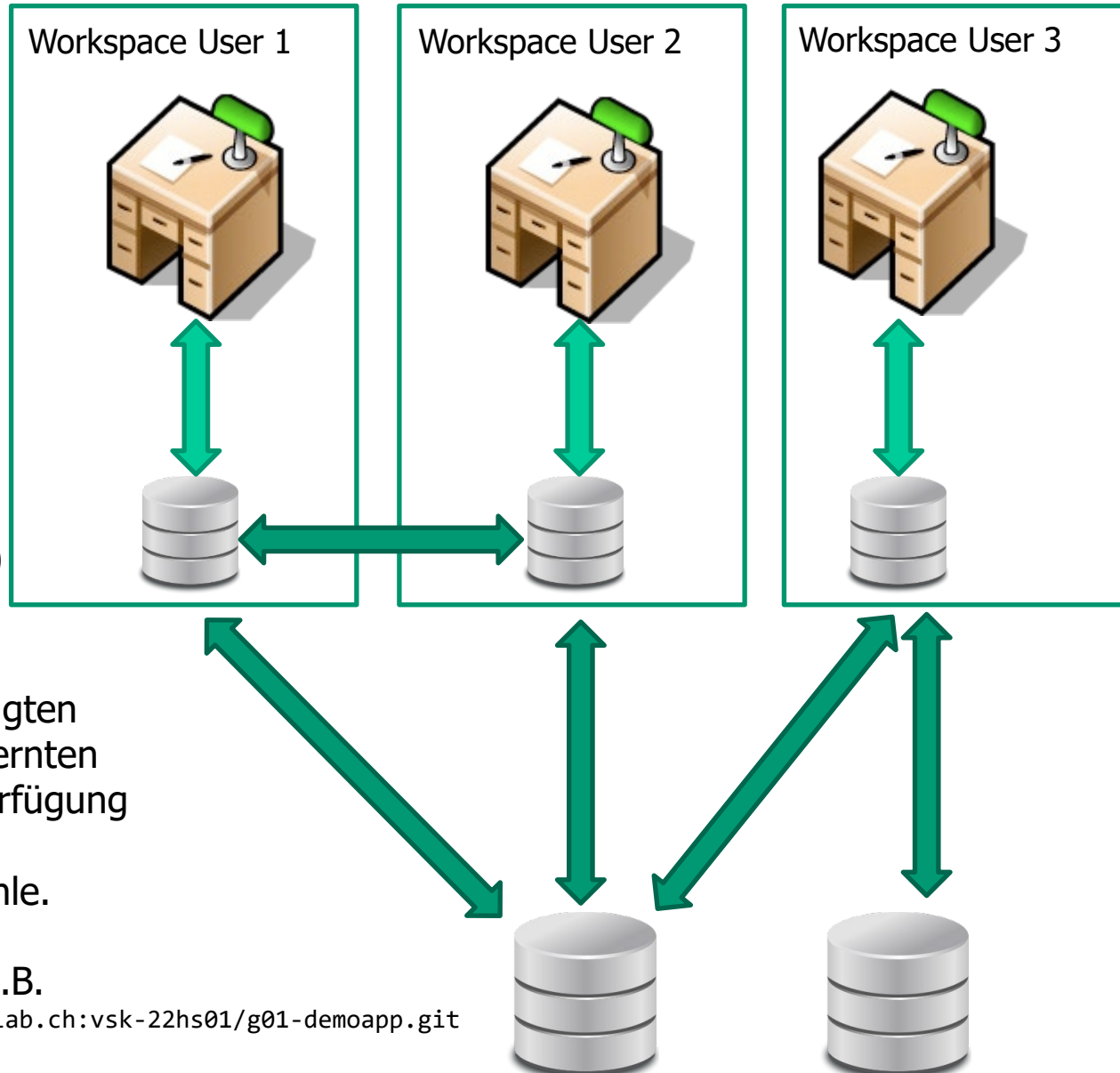
Verteiltes VCS - Konzept

Die «klassischen» Befehle wirken sich **nur** auf das lokale Repository aus!

Lokale Repos
(.git-Verzeichnis)

Um die lokal erzeugten Revisionen in entfernten Repositories zu Verfügung zu stellen gibt es **zusätzliche** Befehle.

Entfernte Repos, z.B.
`git@gitlab.enterpriselab.ch:vsk-22hs01/g01-demoapp.git`



- checkout
- update
- log
- diff
- commit

- clone
- fetch
- pull
- push

Verteilte Versionsverwaltung mit git

- Wenn wir verteilt bzw. mit einem zentralen, gemeinsamen Repository arbeiten, müssen wir dieses **zusätzlich** Synchronisieren!

git clone <url>

Lokales klonen (kopieren) einen entfernten Repos und Workspace einrichten (einmalig, Initialisierung).

git pull

Änderungen vom entfernten Repo lokal nachführen und in Workspace mergen (fetch/merge).

git push

Lokale Änderungen (im Repo) in entferntes Repo übertragen.

Demo / Screencast's

- git mit einem zentralen Repository einsetzen:
EP_11_SC02_GitRemote.mp4



Drei konkrete Produkte

CVS – nicht mehr zeitgemäss



- (Ur-)altes Ur-Versionskontrollsystem
 - Sehr robust, Verbreitung stark abgenommen, gilt als «veraltet».
 - Clients und Integrationen nehmen stark ab.
 - Basiert auf einem zentralen Server.
- Vorteile
 - Sehr stabil, nur noch sehr wenig Fehler.
 - Einfache Anwendung, gut überschaubar.
 - Repository-Konzept strukturell vorgegeben.
- Nachteile
 - Nur dateibasierend (Verzeichnisstruktur **nicht** versioniert).
 - Unterscheidung zwischen Text- und Binärdateien.
 - Ablage von Binärdateien sehr ineffizient (Platzintensiv).
 - **Keine Transaktionen!**

- Wurde offiziell als CVS-Nachfolger eingeführt.
 - Popularität sinkt derzeit sehr stark.
 - Aber noch immer gute und breite Unterstützung.
 - Basiert auf einem zentralen Server.
- Vorteile
 - **Transaktionsorientiert** (Commit in Transaktion, LUW).
 - Versioniert die ganze **Verzeichnisstruktur**.
 - Optimierte/effiziente Speicherung und Übertragung.
 - Repositorystruktur frei wählbar (für Experten flexibler).
 - Integration in / mit Webserver möglich.
- Nachteile
 - Repositorystruktur frei wählbar (für Anfänger schwieriger).
 - Branching und Tagging technisch eigentlich Kopien/Links.

git – sehr Populäres, zeitgemässes System



- Verteiltes System, wird u.a. für Linux-Codeverwaltung verwendet.
 - Entwickelt von Linus Torvalds, sehr flexibles Konzept.
 - Ausgelegt für massives, billiges Branching.
 - Operationen möglichst 'billig' und schnell, weil primär **lokal**.
- Vorteile
 - Verteiltes System, beliebig viele Server / Repos möglich.
 - Sehr flexibel, auch rein **lokal** (genial!) einsetzbar.
 - Skaliert (Funktionsumfang, Verteilung, Grösse).
 - Meist mit Integration in/mit zusätzlichen Web-Applikationen.
- Nachteile
 - Erfordert bei verteiltem Einsatz ein solides Konzept, das organisiert und verstanden werden muss.
 - Für Einsteiger schwieriger, weil sehr mächtig / viele Funktionen.

Benutzerschnittstellen (Clients)

Verschiedene Benutzerschnittstellen

Im wesentlichen gibt es drei verschiedene Varianten:

- Kommandozeile in der Shell.
 - Für einfache Befehle (z.B. `git clone`) sehr effizient.
 - Sehr effizient (wenn man es beherrscht), aber auch kompliziert.
 - Spezialisierte VCS-GUI-Clients.
 - In der Regel etwas einfacher zu Bedienen.
 - Unterstützen das jeweilige VCS optimal.
 - Qual der (Aus-)Wahl! (siehe Quellen und Links)
 - Integrierte VCS-Clients, z.B. in der Entwicklungsumgebung.
 - Eigentlich sehr bequem (alles in einer Applikation).
 - Aber je nach Integration etwas verwirrend.
- ➔ Nutzen Sie individuell die Ihnen zusagende Schnittstelle!

HSLU - GitLab

GitLab Enterprise auf EnterpriseLab – 15.3.x

The image shows two overlapping screenshots of the GitLab interface. The background screenshot is the 'Help' page for GitLab Enterprise Edition 15.3.1-ee. It contains introductory text about GitLab, links to documentation, and an 'Overview' section. The foreground screenshot is a project page for 'oop_maven_template' on the 'gitlab.enterpriselab.ch' instance. It shows project details, a commit history table, and a list of files.

GitLab Enterprise Edition 15.3.1-ee

GitLab is open source software to collaborate on code.
Manage git repositories with fine-grained access controls that keep your code secure.
Perform code reviews and enhance collaboration with merge requests.
Each project can also have an issue tracker and a wiki.
Used by more than 100,000 organizations, GitLab is the most popular solution to manage git repositories on-premises.
Read more about GitLab at about.gitlab.com.

[Check the current instance configuration](#)

Visit docs.gitlab.com for the latest version of this help information with enhanced navigation, discoverability, and readability.

GitLab Docs

Welcome to [GitLab](#) documentation.
Here you can access the complete documentation for GitLab, the single application for the [entire DevOps lifecycle](#).

Overview

No matter how you use GitLab, we have documentation for you.

Project: oop_maven_template

Template für einfache Java-Projekte mit Maven-Build.

SSH: `git@gitlab.enterpriselab.ch:oop/o`

Files (236 KB) | Commits (48) | Branches (2) | Tags (8) | Readme | LICENSE | CI configuration | [Add Changelog](#) | [Add Contribution guide](#)

develop | oop_maven_template /

#9 Hamcrest-Exclusion bei JUnit entfernt.
Roland Gisler authored a week ago

Name	Last commit	Last update
config	Vorbereitung für Release 1.5 (18FS)	a week ago
src	Formatierung	a week ago
.classpath	Eclipse-Projektkonfig ergänzt	a year ago
.gitignore	Initial Commit	a year ago
.gitlab-ci.yml	CI auf GitLab aktiviert	9 months ago
.project	Eclipse-Projektkonfig ergänzt	a year ago
LICENSE.txt	Vorbereitung für Release 1.5 (18FS)	a week ago
README.md	Vorbereitung für Release 1.5 (18FS)	a week ago

■ <https://gitlab.enterpriselab.ch/vsk-22hs01>

GitLab – Funktionalität

- GitLab stellt auf der Basis von git-Repositories eine komplette Codehosting-Plattform zur Verfügung.
 - Codeverwaltung in Projekten und Gruppen.
 - Planung von Milestones, Issue-Tracking, Taskboard.
 - Einfache Website (Markup-Readme) und Wiki.
 - Direktes bearbeiten von Artefakten über Web-GUI.
 - Merge-Requests, Pull-Requests, CI-Dienste etc.
- Vergleichbar mit bekannten Diensten wie GitHub, BitBucket, Sourceforge etc.
- Hinweis zum Einsatz in VSK: Wir nutzen GitLab für die **Planung, Controlling und die Codeverwaltung**. Die CI-Dienste werden durch eine zusätzliche Infrastruktur angeboten.

GitLab – Zugriff, Protokolle und Clients

- Zentrale Git-Repositories mit Web-GUI, hosted im EnterpriseLab
 - Läuft in der DMZ, somit direkt (ohne VPN) erreichbar.
- URL: <https://gitlab.enterpriselab.ch/>
 - Login mit ELAB-Account! → <https://eportal.enterpriselab.ch/>
- Zugriff auf das Repository
 - **https**: Einfach, Passwort wird ggf. in jedem Client hinterlegt.
 - **ssh**: Elegant, Public-/Private-Key Infrastruktur notwendig.
- Direktzugriff auf die Git-Repositories mittels:
 - NetBeans / Eclipse - EGit-Client (vorinstalliertes Plugin).
 - SourceTree - Für Windows und Mac.
 - Git in Konsole/Shell - Für Puristen, sehr effizient.
 - **SmartGit** - Sehr guter Client (Multiplattform), **Empfehlung!**

Wichtige Hinweise

Wichtige Empfehlungen für die Arbeit mit VCS/git



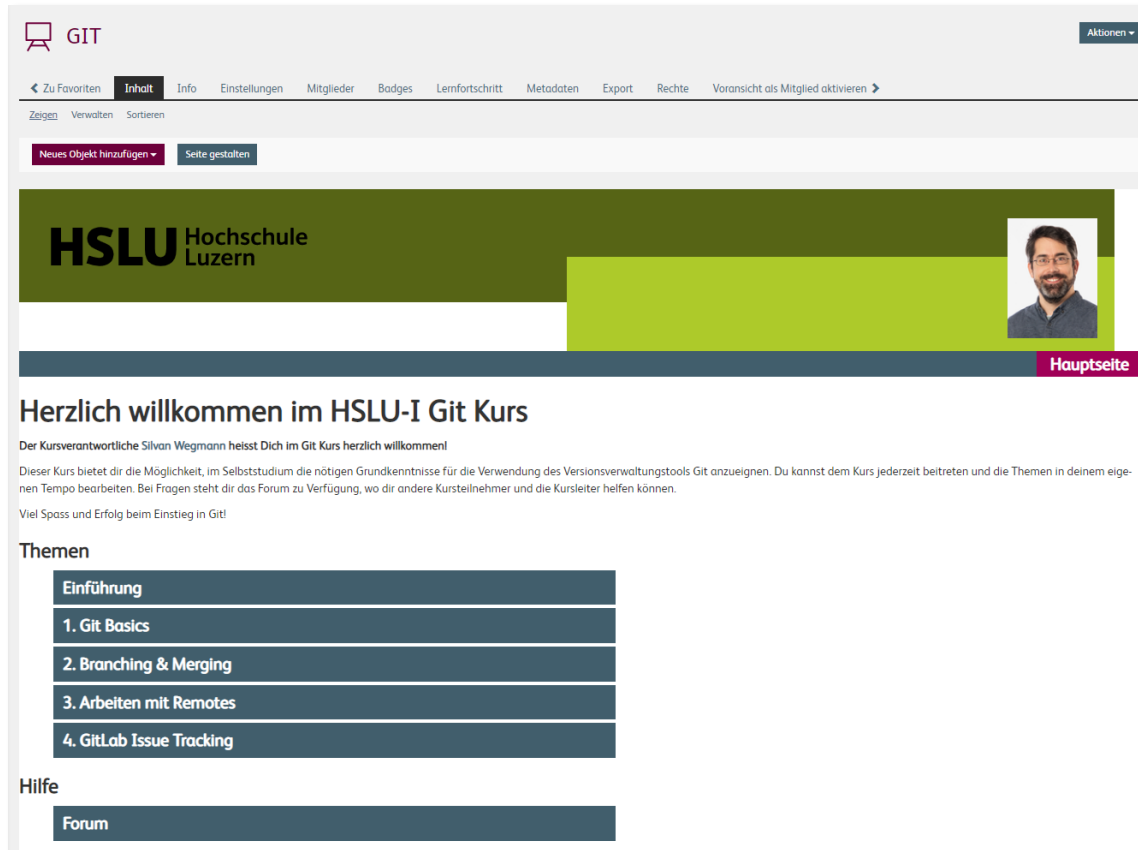
- Vor Arbeitsbeginn und vor jedem **commit/push** ein **pull** machen
 - Aktuellsten Stand für Weiterarbeit übernehmen.
 - evt. parallele Änderungen prüfen und einarbeiten (merge).
- **commit immer** mit einem **aussagekräftigen** Kommentar!
 - Woran bzw. warum hat man etwas geändert?
 - Wenn vorhanden Link auf Issue!! (#nn, z.B.: #18)
 - Zeitnah (sofort!) ein **push** in das zentrale Repo!
- Generierte, automatisch erstellbare Artefakte **NIE** einchecken
 - **.gitignore**-Datei (pro Verzeichnis) mit Einträgen was ignoriert werden soll, ist bereits vorkonfiguriert, ggf. ergänzen!
- Wer noch keine Erfahrung hat: *Üben, üben, üben!*
 - Am Besten zuerst Lokal, dann gemeinsam in einem Projekt.



Für Selbststudium: git-Kurs auf ILIAS

- Im Sommer wurde ein ILIAS-Kurs zum Thema **git** erstellt:

https://elearning.hslu.ch/ilias/goto.php?target=crs_5207233&client_id=hslu



The screenshot shows the ILIAS course interface for 'HSLU-I Git Kurs'. At the top, there's a navigation bar with 'GIT' and 'Aktionen'. Below it, a menu includes 'Zu Favoriten', 'Inhalt', 'Info', 'Einstellungen', 'Mitglieder', 'Badges', 'Lernfortschritt', 'Metadaten', 'Export', 'Rechte', and 'Voransicht als Mitglied aktivieren'. A secondary bar has 'Zeigen', 'Verwalten', and 'Sortieren'. The main header area features the 'HSLU Hochschule Luzern' logo, a green decorative bar, and a profile picture of a man. A 'Hauptseite' button is visible. The main content area starts with the title 'Herzlich willkommen im HSLU-I Git Kurs' and a welcome message from Silvan Wegmann. It describes the course's purpose for self-study in Git version control. Below this, a 'Themen' section lists five topics: 'Einführung', '1. Git Basics', '2. Branching & Merging', '3. Arbeiten mit Remotes', and '4. GitLab Issue Tracking'. A 'Hilfe' section at the bottom contains a 'Forum' button.

- Geben Sie uns gerne Feedback zum Kurs! Danke.

Zusammenfassung

- Versionskontrollsysteme bewusst einsetzen und nutzen.
 - Erfahrungen sammeln in der Anwendung.
- Minimale Basis:
 - Lokal: **checkout** und **commit**
 - Verteilt/Zentral: **clone**, **pull** (**fetch/merge**) und **push**
- Fortgeschrittene Nutzung:
 - Taggen – markieren von Revisionsständen (Baseline).
 - Branchen – getrennte Entwicklungszweige.
- **Koordination** zwischen den Teammitgliedern **sinnvoll!**
 - Konflikte können passieren, sind aber **nicht** das Ziel!
- Commits **immer** mit sinnvollem, aussagekräftigem Kommentar, welcher auf eine Issue-Nummer verweist (wenn vorhanden)!

Quellen und Links

- git - <http://git-scm.com/> - für alle Plattformen.
- Git Clients
 - SmartGit - <http://www.syntevo.com/smartgit/> (Empfehlung!)
 - SourceTree - <https://www.sourcetreeapp.com/> (Atlassian)
 - TortoiseGit - <https://tortoisegit.org>
- IDE's
 - Alle relevanten IDE's verfügen über eine git-Integration.
 - Aber: Sehr unterschiedliche Umsetzungen...
- Git Hosting
 - HSLU GitLab – <https://gitlab.enterpriselab.ch/>
 - GitHub - <https://github.com/> (nur öffentliche Repos, Gratis)
 - BitBucket - <https://bitbucket.org/> (auch private Repos, Gratis)

Fragen?