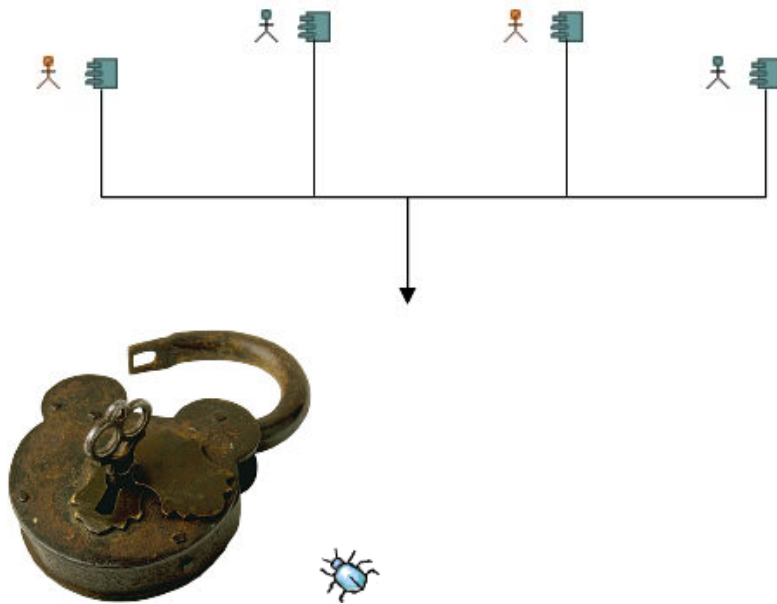


### Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.  
©2018 Beuth Hochschule für Technik Berlin

## SVN - Versions- und Fehlermanagement



### Versions- und Fehlermanagement

## Lernziele und Überblick

Softwareentwicklungsprojekte werden fast immer im Team erarbeitet.

Dabei muss (wie in den vorigen Lerneinheiten besprochen) der Prozess des Build- und Integrationsmanagements gehandhabt werden. Bekanntermaßen enthält Software (leider) fast immer Fehler oder es fehlen Features. Diese Fehler müssen verwaltet, gemanagt und dokumentiert werden und dürfen nicht im Alltagsgeschäft untergehen. In manchen Fällen ist es sogar notwendig sich auf einen früheren Stand der Software zurückzugehen. Auch ein Experimentieren mit dem Code kann es nötig machen, auf frühere Stände zurückzugreifen. Das essenzielle und heutzutage auch einfache Versionsmanagement ist eines der wichtigsten Werkzeuge der Softwaretechnik, um den Code in allen Varianten zu verwalten.



### Lernziele

Ziel dieser Lerneinheit ist es, die Konzepte des Versions- und Fehlermanagements zu verstehen und die beiden bekanntesten Systeme praxisnah verwenden zu können.



### Gliederung der Lerneinheit

- Grundlagen der Versionsverwaltung
- Anforderungen und Vorteile
- CVS / Beispiele unter Eclipse
- Subversion
- Fehlermanagement
- Bugzilla



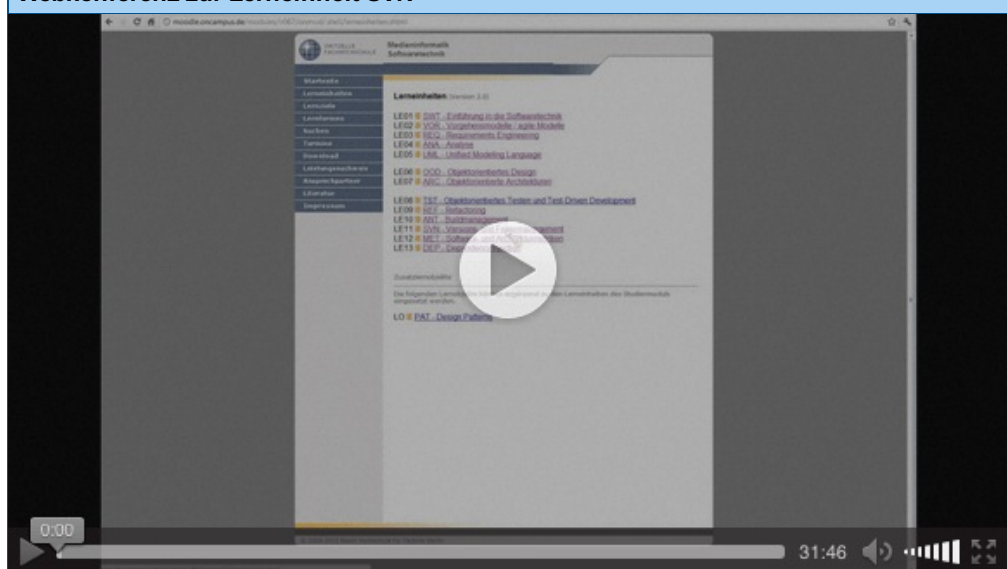
### Zeitbedarf und Umfang

Zum Durcharbeiten dieser Lerneinheit benötigen Sie ca. 120 Minuten. Für das Lösen der Übungsaufgaben werden ebenfalls 120 Minuten veranschlagt.



Film

### Webkonferenz zur Lerneinheit SVN



© Beuth Hochschule Berlin - Dauer: 31:45 Min. - Streaming Media 41.7 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

## 1 Einleitung

Ein zentraler Teil des Projektmanagements ist das Revisions- / Versionsmanagement- oder Konfigurationsmanagement. Dabei geht es darum, dass das gesamte Projekt in seinem Wachstum begleitet wird. Korrekte Auslieferungen und Codezustände müssen jederzeit sichergestellt werden können.



Definition

Wichtig dabei ist, dass das Versionsmanagement als Teil eines Gesamtzyklus zu sehen ist, der zudem oft als Continuous Integration bezeichnet wird. Dieses Vorgehen ist in Anlehnung an das extreme Programming zu sehen und bezeichnet mehrere Aufgaben, die von folgenden Tools unterstützt werden können:

- Quellen ziehen, sichern und verwalten: z. B. mit CVS
- Vorgänge automatisieren und in einen Buildprozess integrieren: z. B. mit Ant
- Codequalität mittels Tests sicherstellen : z. B. mit JUnit oder \*Unit
- Fehler- und Anforderungen verwalten : z. B. mit Bugzilla

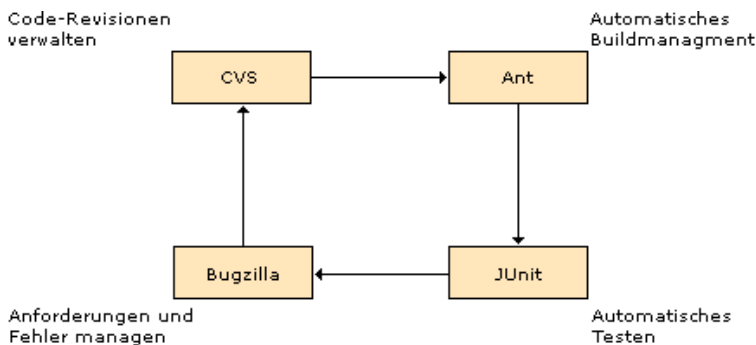


Abb.: Aufgaben und unterstützende Tools

Ziel des Continuous Integration ist es, lauffähigen Code jederzeit zur Verfügung stellen zu können. Idealerweise entspricht der aktuelle Build also immer dem Stand der Entwicklung. Dadurch werden mögliche Probleme bei der Integration des Systems sofort erkennbar.

Der Sourcecode selbst ist immer an einer zentralen Stelle gehalten und wird nicht nur manuell, sondern auch oft durch den Buildprozess von Werkzeugen wie Ant abgerufen und weiterverwendet (Compile, War, JUnit, etc.).

Der gesamte Buildprozess ist (siehe auch Lerneinheit ANT „Buildmanagement“) so automatisiert, dass lediglich ein Kommando („ant“) oder ein Klick den gesamten Code und Executables erstellt, ausliefert (deployt) und testet.



Hinweis

Ziel dieses Vorgehens im **Continuous Integration** ist, dass der Entwickler sich weniger als Einzelkämpfer fühlt, sondern sich immer als Teil des Teams sieht. Seine Entwicklungsarbeit gilt dann als abgenommen, wenn der gesamte Zyklus durchlaufen ist und das Produkt bzw. der Prototyp fehlerfrei arbeitet sowie erfolgreich integriert ist. Dies ist mit einem derartigen Buildzyklus viel einfacher zu erreichen.

Diese Integrationsbuilds sind deshalb so erfolgreich, weil sie mit einem Klick durchgeführt werden können und anschließend durch Tests zeigen, ob das Produkt erfolgreich arbeitet. Hier ist der Unterschied zu früheren Vorgehensweisen erkennbar. **Continuous Integration** spart in der Regel viel Geld und Zeit. Probleme können früher behoben, Prototypen integriert und ausreichend getestet werden.

## 1.1 Definition



Definition

### Versionsmanagement (engl. Revision Control)

Versionsmanagement bezeichnet ganz allgemein den Vorgang der Verwaltung von Sourcecode in verschiedenen Versionen durch ein Softwarewerkzeug.

Dabei geht es - neben der geordneten und zentralen Datenhaltung - auch um ein **Release-** oder **Revisionsmanagement**. Das heißt, bestimmte Entwicklungsstände oder Entwicklungslinien zu verfolgen und jederzeit geordnet ausliefern zu können.

Das Versionsmanagement kann seinerseits wieder ein Teil des **Konfigurationsmanagements** **sein**. Versionen sind sowohl „Konfigurationen“ des Projektes, als auch Buildinformationen wie Sprache, Zielumgebung (Windows, Linux, etc.) oder Application-Server. Allerdings kann es weitere Konfigurationen geben, die nicht Versionen sind (s. u.).

An dieser Stelle wird bereits klar, dass der Prozess des Versionsmanagements ohne Strategie und Hilfsmittel sehr schwierig ist. Die Verwaltung von Versionszweigen (*branches*) oder eine Fehlersuche ist ohne Zuhilfenahme von Tools meistens ein schwieriges Unterfangen.



Achtung

In vielen (gerade kleinen) Projekten wird ein Versionsmanagement „von Hand“ durchgeführt. Oft wird argumentiert, dass einfaches Kopieren von Verzeichnissen genauso einfach ist. Leider überträgt sich dieses Denken häufig auf größere Projekte und sorgt später für unglaubliche (negative!) Überraschungen.

Ziel dieser Lerneinheit ist es zu zeigen, dass das Aufsetzen von Versionskontrollumgebungen mittlerweile so einfach ist, dass der geringe zeitliche Mehraufwand fast immer investiert werden sollte. Das Arbeiten mit Versionskontrollsystemen wie CVS oder Subversion sollte danach sehr intuitiv sein und kaum noch Zeit kosten.

Versionsmanagementsysteme werden hier immer wieder im Kontext der Programmierung betrachtet. Immer häufiger sind Versionskontrollsysteme allerdings auch in ganz anderen Projekten zu finden. Dokumentationen, Bucherstellung, Grafikarbeiten sind nur einige Beispiele für den sinnvollen Einsatz eines zentralen Verwaltungswerkzeuges.



Definition

### Change Management

Unter Change Management versteht man daher die Summe folgender Systeme:

- Versionsmanagement: Versionen beispielsweise von Programmcode des Projektes
- Release Management: Verwaltung der Releases, also der Entwicklungslinien
- Konfigurationsmanagement: Verwaltung aller sonstigen Einstellungen, Dateien und Zustände, die für das Funktionieren des Systems relevant sind (z. B. Konfigurationen des Application Servers, Messaging Queues, Aktive Web-Services, etc.)

## 1.2 Anforderungen

Welche Anforderungen muss ein Versionskontrollsystem erfüllen:

- Es müssen ständig „Snapshots“ auf Versionsstände (Releases) vorgenommen werden können.
- Vom Entwickler freigegebener Sourcecode muss in das zentrale Repository (siehe nächste Seiten) übertragen werden.
- Es müssen die Verantwortungen und Kontrolle über den Sourcecode transparent sein.
- Es muss - gerade auch zu Dokumentationszwecken - nachvollzogen werden können, wer welche Änderung vorgenommen hat und welche „Features“ jetzt integriert sind.
- Das System muss Konflikte auflösen oder zumindest transparent machen, die in der Zusammenarbeit des Teams mit dem (gleichen) Code auftreten.
- Der Prozess muss leicht zu handhaben sein, z. B. durch das Eclipse Plug-In (Sync-Button).

## 2 Revision Control System (RCS)

### Zur Historie von Versionskontrollsystemen

Schon in den frühen 70er Jahren wurden Entwicklungsarbeiten in Versionen archiviert und verwaltet. In den Anfängen geschah dies durch einfaches Kopieren von Dateien und Ordern. Motiviert wurden „echte“ Versionskontrollsysteme auch besonders durch das UNIX-Kommando „**diff**“ und später **patch** (die sogenannten GNU diffutils). **diff** gibt die Differenz zweier Dateien aus und bezeichnet genau die Unterschiede. Mittels **patch** wird Quelltext anhand von Patches manipuliert. Ein Beispiel ist im nachstehendem Kasten zu finden.

Es ist klar, dass das Speichern der Differenzen von Dateien sehr viel mehr Platz spart, als das Speichern der Originaldateien in mehreren Zuständen. Mittlerweile verwenden gute Versionskontrollsysteme effiziente Binärformate, um die Änderungen (Deltas) am Sourcecode zu speichern.

Was jetzt noch fehlte, war ein Tool, welches das Bilden der Differenzen, die Archivierung, die Markierung und die Wiederherstellung von Dateien automatisiert. Hier gab es viele einfache Systeme und Experimente, die im Wesentlichen durch frühe Buildwerkzeuge wie „**make**“ motiviert waren. Von dort war der Weg nicht mehr weit zu einem „echten“ System, welches Versionen verwaltet.



Beispiel

### Beispielhafte Ausgabe von diff

```
1,2d0
< The Way that can be told of is not the eternal Way;
< The name that can be named is not the eternal name.
4c2,3
< The Named is the mother of all things.
---
> The named is the mother of all things.
>
11a11,13
> They both may be called deep and profound.
> Deeper and more profound,
> The door of all subtleties!
```

Das Revision Control System stammt ursprünglich aus deutscher Hand und wurde erst von **WALTER F. TICHY** (heute auch durch viele Studien über „Extreme Programming“ bekannt) während seiner Zeit an der Purdue University initiiert. Erst anschließend daran, ging das System in die Verwaltung des GNU-Projektes über([www](http://www.gnu.org) RCS bei GNU). Die offizielle Homepage [www](http://www.cs.purdue.edu) [www.cs.purdue.edu](http://www.cs.purdue.edu) befindet sich noch immer an der Purdue Universität.

RCS-Kommandos

Die zentralen Kommandos von RCS sind auch heute noch in allen Nachfolgesystemen zu finden. Sie zeigen auf, welche Tätigkeiten ein Entwickler mit welchen Kommandos bewerkstelligen kann:

- **rcsintro** Kurzeinführung in die Kommandos von RCS
- **ci** Einchecken einer Datei und Aufheben des Locks
- **co** Auschecken einer Datei in die lokale „Sandbox“ und Setzen des Locks
- **rcsdiff** Anzeige der Differenzen von Versionen
- **rlog** Anzeigen des Logs von Versionen, d. h. der Versionsdokumentation
- **rcsmerge** Zusammenführen von Versionen
- **rcsclean** Löscht alle Dateien, an denen nicht gearbeitet wurde
- **rcsfreeze** Einfrieren einer Konfiguration von Sourcen unter RCS

und weitere Kommandos wie beispielsweise **rcs**, **rcsfile**, etc.

SCCS

Das **CVS** (**C**oncurrent **V**ersion **S**ystem) wurde ursprünglich auf RCS aufgesetzt, dann aber eigenständig weiterentwickelt.

Abschließend sei noch **SCCS** erwähnt, das 1992 bei den Bell Labs entwickelt wurde und ebenso bedeutend ist wie RCS . Auch dieses System wurde initial auf Rechnern wie dem PDB 11 unter UNIX verfügbar gemacht.

SCCS wiederum wurde durch CSSC der GNU Foundation weiterentwickelt.

### 3 Versionsmanagement-Systeme

Auf dieser Seite wird nur ein kurzer Überblick der verfügbaren kommerziellen Systeme gegeben. Im Anschluss wird dann in die Methodik und Funktionsweise von Versionskontrollsystemen eingeführt. Als guter Softwaretechniker sollte man von diesen Systemen also lediglich einmal gehört haben.

Die meisten der hier genannten Systeme bieten ein umfangreiches Toolset zum Thema Versions- und Codemanagement an. Durch die vielfältigen Möglichkeiten werden eher größere Projekte adressiert.

- **Perforce**, [www](http://www.perforce.com) <http://www.perforce.com>  
Enthält zusätzlich zu Server und Client noch vielfältige Tools wie ein Reporting-System, einen komplett bedienbaren Web-Client, Merge-Tools die mit Änderungen an mehreren Orten umgehen können, Integration mit führenden Bug-Tracking Systemen und vieles mehr.
- **PVCS**, [www](http://www.pvcs.com) <http://www.pvcs.com>  
Auffallend bei PVCS ist die besondere Adressierung von Security und Performance. Weiterhin wird der Mainframe Bereich adressiert, Web Interfaces integriert, Vorkonfigurationsmöglichkeiten für viele Compiler sind gegeben, Unterstützung für Remote Builds, Aufgaben können übernommen werden, die auch Build-Tools vornehmen können, automatische Benachrichtigungen etc.
- Rational Clear Case, [www](http://www-306.ibm.com) [www-306.ibm.com](http://www-306.ibm.com) (sehr renommiert)
- StarTeam, [www](http://www.borland.com) [www.borland.com](http://www.borland.com)
- SurroundSCM, [www](http://www.seapine.com) [www.seapine.com](http://www.seapine.com)
- CMVC (Teil von IBM / db2 / WebSphere)

Viele der Systeme werden in den Ant Optional-Tasks oder anderen Buildsystemen mit eigenen Tasks unterstützt. Dies ermöglicht auch ein einfaches automatisches Lesen und Schreiben von Quellcode aus dem Buildprozess heraus.


Ein guter Vergleich zwischen Versionskontrollsystemen findet sich im Internet.

[www](http://better-scm.shlomifish.org) <http://better-scm.shlomifish.org>

## 4 Grundlegende Konzepte anhand von Concurrent Versions System (CVS)

CVS ist derzeit das am meisten verbreitete Versionskontrollsystem. Dies liegt besonders daran, dass es ein Open-Source Projekt und damit kostenlos erhältlich ist. Fast alle Communities wie die Apache Group, sourceforge.net oder java.net stellen dem Anwender CVS Schnittstellen zur Verfügung, um neue Projekte zu verwalten oder live in bestehende Projekte hineinschauen zu können.

Informationen über CVS sind beispielsweise über die beiden folgenden Links zu finden:


 <http://savannah.nongnu.org/projects/cvs>

 <http://www.nongnu.org/cvs>

### Historie

CVS wurde aus RCS weiterentwickelt und 1986 veröffentlicht. Heutzutage wird CVS immer noch von einer Reihe Entwicklern gepflegt und auf Savannah gehostet. In den 90ern wurde CVS auf Windows portiert und ist jetzt auch für Windows verfügbar.

 <http://www.cvsnt.com>

 <http://wincvs.en.uptodown.com>

Seit einigen Jahren sind Entwickler dazu übergegangen, CVS von Grund auf neu zu gestalten. Das neue System wurde Subversion genannt und löst CVS langsam ab, da es einige Limitierungen von CVS aufhebt.

### CVS Beschränkungen

Viele Entwickler hat es gestört, dass Dateien nicht einfach umbenannt werden können, da diese im üblichen Workflow zuerst gelöscht und dann neu „eingchecked“ werden müssen. Auch das Handling von Verzeichnissen in Bezug auf Umbenennung und Verschieben (was beim Package-Refactoring ja üblich ist) ist nicht möglich. Hier muss ebenfalls das Verzeichnis gelöscht und neu geschrieben werden.

Gegenüber RCS bietet CVS viele Vorteile wie beispielsweise ein echtes Client/Server System, Checkouts ohne Locking, Verfügbarkeit auf vielen Plattformen und vieles mehr.



## 4.1 Konzepte

Die grundlegenden Konzepte eines Versionskontrollsystems wie CVS sind:

- Der Entwickler entwickelt in seiner lokalen Sandbox. Der Code wird entweder initial neu „eingchecked“ oder für die Arbeit „ausgecheckt“ und lokal gehalten. Die nachfolgende Abbildung zeigt, dass dies lokal oder remote geschehen kann. Zentraler Kern ist daher bei allen Versionskontrollsystemen ein Repository, das in der Regel eine Serveranwendung ist.

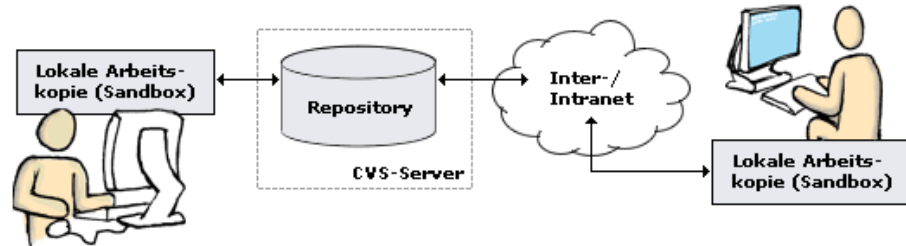


Abb.: Sandbox

- Das Einchecken von geändertem Source-Code wird mit einem Log-Eintrag versehen. Dieser ist für die Revisions-Dokumentation unumgänglich. Fast jedes Projekt bietet sogenannte Changelogs für neue Versionen an. Diese kann der Projektleiter leicht aus den Kommentaren zu Features oder Änderungen erstellen, die die Entwickler vorgenommen haben. Kommerzielle Produkte bieten genau dafür Reporting-Tools an.
- Jede Datei bekommt eine Revisionsnummer. Diese wird nach bestimmten Algorithmen automatisch erhöht und trägt dazu bei, Änderungen nachzuvollziehen und Versionsstände eindeutig zu benennen.
- Dateien können (müssen aber nicht) für die Bearbeitung gesperrt werden. Auf diesen interessanten Aspekt wird später noch genauer eingegangen.
- Versionskontrollsysteme müssen helfen Konflikte aufzulösen oder sie zumindest transparent zu machen (also z. B. im einfachsten Fall das Einchecken ablehnen).



Hinweis

### Einchecken / Auschecken

**Checkout** = „ich nehme die aktuelle Version für mich und ändere diese auf meinem Rechner. Das ist dann meine Arbeitsversion und diese sieht keiner“ (nach D. MARCOS).

**Checkin** = „meine Arbeitsversion wird eine offizielle Projektversion, diese stelle ich dadurch allen anderen zur Verfügung“ (nach D. MARCOS).

**2x Checkout** derselben Version führt nach Änderung und checkin zu Parallelversionen. Der Konfigmanager muss entscheiden, was soll an der Stelle getan werden: eine Version wegwerfen oder beide zusammenfügen = Merge.

## 4.2 Repositories

Der Zugriff auf das zentrale Server-Repository geschieht über folgende Expression:



Quellcode

### Repository-Zugriff

```
[[:method:]] [[[:user]][:password]@]hostname[:[:port]]]/path
```

Bei den meisten Projekten wird ein entsprechender String angegeben, z. B.

```
anoncvs@cvs.dev.java.net.
```

Die Zugriffsmethode ist meistens rsh oder ssh und bezeichnet, über welchen Kommunikationspfad der Server angesprochen wird.

Der Connection type ist in der Regel pserver. Alternative Modi wie kserver (Kerberos Sicherheitsarchitektur) oder gserver (GSSAPI) werden selten verwendet.

## 4.3 Kommandos



Definition

### Wichtige Begriffe

Da die meisten Begriffe bei der realen Anwendung englisch sind, werden diese im Folgenden hier auch englisch vorgestellt aber deutsch erklärt.

Begriff	Bedeutung
<b>Repository</b>	Das bereits erwähnte <u>Repository</u> ist der Ort, an dem die Dateien gespeichert werden. Dies ist in der Regel ein <u>Server</u> .
<b>Commit</b>	Ein commit ähnelt dem einer <u>Datenbanktransaktion</u> und bezeichnet einen check-in. In diesem Fall werden die Dateien des lokalen Sandboxverzeichnisses auf den Server übertragen und die Änderungen und Anmerkungen verwaltet.
<b>Check-Out</b>	In diesem Fall werden die angeforderten Dateien vom Repository in die lokale Sandbox des Entwicklers übertragen. Also quasi ein Export.
<b>Update</b>	Ein Update sorgt dafür, dass alle auf dem Server geänderten Dateien abgerufen und daher quasi lokal aufgefrischt werden.
<b>Merge</b>	Ein Merge bringt verschiedene <u>Versionen</u> zusammen. Beispielsweise den „Branch“ (Zweig) 1.3.12 mit der Version 1.4.2
<b>Revision</b>	Eine <u>Revision</u> ist ein Versionsstand innerhalb aller vorliegenden Versionen.
<b>Import</b>	Beim Import ist das Kopieren des gesamten Dateibaumes in den lokalen Workspace gemeint.
<b>Export</b>	Ein Export ähnelt einem kompletten <u>Check-Out</u> , wobei meist ein neues komplettes <u>Projekt</u> ohne Revisionen komplett zum Server übertragen wird.

Tab.: Kommandos

## Anlegen eines Repositories

Im Folgenden wird das praktische Einrichten eines Repositories gezeigt:



Quellcode

### Neues Repository anlegen

```
> cvs -d /usr/repository init
>ls /usr/repository/CVSRROOT
Emptydir      config      editinfo,v  modules,v  taginfo
checkoutlist  config,v    history     notify     taginfo,v
checkoutlist,v cvswrappers loginfo     notify,v   val-tags
commitinfo    cvswrappers,v loginfo,v   rcsinfo    verifysg
commitinfo,v  editinfo    modules     rcsinfo,v  verifysg,v
```

Das Repository wird hier mit dem Kommando `cvs` eingerichtet. Mit `-d` wird der Name des Repositorypfades angegeben. Das Verzeichnis `CVSRROOT` wird immer mit eingerichtet und enthält Verwaltungsinformationen.



Quellcode

### Daten importieren

```
> cd ~/noncvs_prj
> cvs import -m "Imported project" ourprj demo ver0-1
```

Hier wird in ein Verzeichnis gewechselt, welches noch nicht unter der Versionskontrolle steht. Danach wird das Kommando `cvs` aufgerufen, dann das lokale Projekt importiert und anschließend mit einem Namen versehen. `-m` spezifiziert eine globale Logging-Message, die auf alle Dateien angewendet wird. Die letzten Parameter bezeichnen das Vendor und Release Tag, die aber nicht so wichtig sind.



Quellcode

### Auschecken in ein Arbeitsverzeichnis

```
>cd ~/work
>cvs checkout outprj
cvs server: Updating
U ourprj/build.xml
U ourprj/README
```

Hier wird in das Verzeichnis gewechselt, in dem man arbeiten möchte. Danach wird das vorher importierte Projekt `outprj` wieder ausgecheckt und in das neue Verzeichnis eingefügt. Dies entspricht der lokalen Sandbox in der der Anwender arbeiten kann. In diesem Verzeichnis wird ein zusätzlicher Ordner angelegt der Metainformationen enthält. Für jede dort enthaltene Datei hat man volle Schreib- und Leserechte.

Natürlich ist es auch möglich mit `checkout` nur einzelne Dateien oder Verzeichnisse zu holen.



Quellcode

### Commit

```
> cd ~/work/ourprj
> cvs commit
cvs commit: Examining
```

Die vorher vorgenommenen Änderungen können nun von Hand übertragen werden. Dazu muss in das Verzeichnis gewechselt und der Commitbefehl abgesetzt werden. Mit diesen Aktionen werden alle Änderungen in das Repository übertragen. Scheitern können diese, wenn die Daten des Repositories nicht mehr aktuell sind.

Die Angabe von Log-Messages ist immer hilfreich, um Änderungen nachzuvollziehen und Anwendern neue Releaseinformationen zur Verfügung zu stellen:



Quellcode

### Log-Messages

```
> cvs commit -m "Neuen Testfall testRandomData() hinzugefügt."
```

Mit `cvs add A.java` kann man Dateien zum Repository hinzufügen und mit `cvs remove` wieder entfernen.

## 4.4 Keywords

Ein interessantes Konzept des Versionsmanagements ist es, in Texten Meta-Tags einbetten zu können. Dazu betrachten wir folgendes Beispiel.



Quellcode

### CVS Schlüsselwörter

```

/*****
* $Id: TestSortAlgorithm.java,v 1.2 2003/05/06 14:30:29 harry Exp$
* $RCSfile: TestSortAlgorithm.java,v $
* $Revision: 1.2 $
* $Date: 2003/05/06 14:30:29 $
* $Author: harry $
*****/
/*
* $Log: TestSortAlgorithm.java,v $
* Revision 1.2 2003/05/06 14:30:29 harry
* Neuen Testfall testRandomData() hinzugefügt.
*/

```

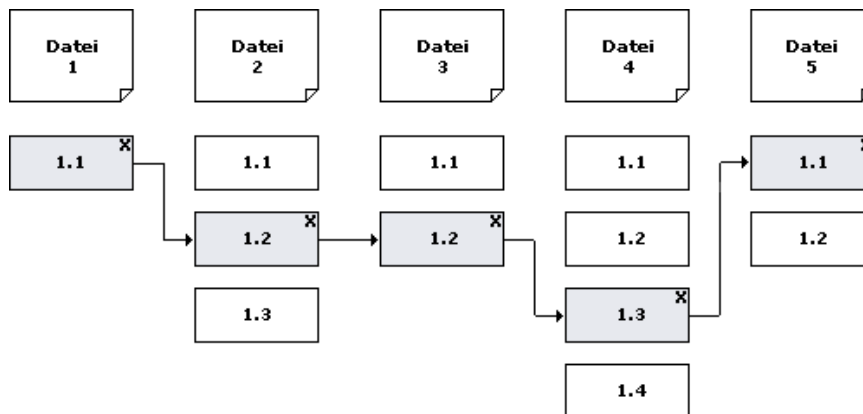
Automatische Updates  
in Dokumenten

Die hier angegebenen Schlüsselwörter wie `$Revision$` oder `$Date:$` werden als Meta-Tags bezeichnet, die das CVS-System beim Ein- und Auschecken lesen und deren Inhalte verändern können. Relativ oft ist es üblich die Meta-Tags in programmiersprachliche Kommentare einzubetten. Diese werden dann beispielsweise von `javadoc` verarbeitet und tragen dazu bei, immer über den aktuellen Stand des Quelltextes zu informieren. Ein manuelles Editieren ist nicht mehr nötig.

## 4.5 Branches und Tags

Aufgabe von CVS ist es, sich alle Versionen aller Dateien zu merken. Führt der Anwender ein Update durch, dann werden immer nur die neuesten Dateien übertragen. Ein Rückgriff auf verschiedene ältere Versionen ist mit dem gleichen Kommando nicht trivial möglich. Es stellt sich daher ziemlich schnell die Frage, wie man auf ältere Versionen des Programmes zugreifen kann.

Zu diesem Zweck gibt es symbolische Namen, die als Tags bezeichnet werden. Mit diesen kann quasi eine Momentaufnahme (wie ein Foto) des Arbeitsstandes gemacht werden.



Das obenstehende Beispiel zeigt, dass fünf verschiedene Dateien existieren und ein spezieller Bereich dieser Dateien markiert wurde.



Definition

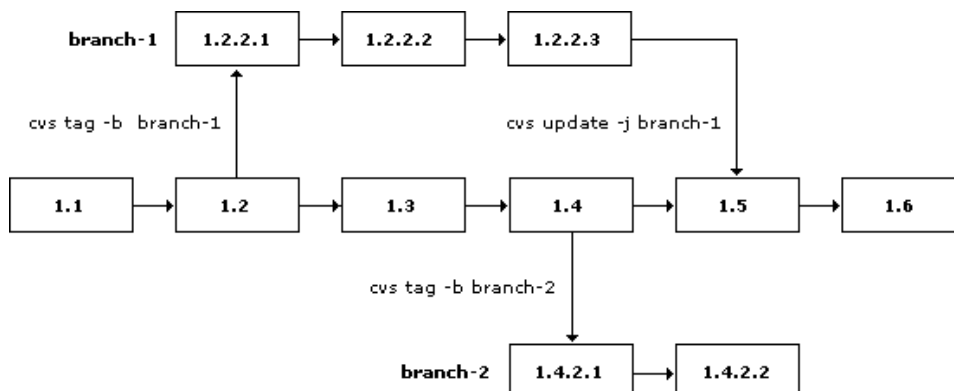
### Revision vs. Release

Bei einer einzelnen Datei spricht man von einer Version oder einer Revision. Geht es dagegen wie in obigem Beispiel um eine Menge von Dateien, die ausgezeichnet wurden, dann spricht man von einem Release.

Normalerweise kann einfach ein *Tag* vergeben werden. So können z. B. mit `cvs tag before-midnight` alle Dateien markiert werden.

Branches

Möchte man nun einen alten Versionsstand (anders) weiterentwickeln, müssen Branches gesetzt werden. Dies geschieht mittels `cvs tag -b branch-1`.



Obiges Beispiel zeigt eine Branchabspaltung inklusive Rückführung.

Abspaltung

Um einen aktuellen Revisionsstand auf einen bestimmten Branch zurückzusetzen, muss ein Update durchgeführt werden:

```
cvs update -r branch-1
```

Liegt noch keine Sandbox (lokale Arbeitskopie) vor, so kann man mit

```
cvs rtag -b branch-1
```

einen Branch aus dem Repository abspalten.

Zusammen-  
führung

Abschließend soll noch erwähnt werden, dass es, wie oben dargestellt, möglich ist, einen Zweig wieder in den Hauptbranch zu integrieren. Dies geschieht mit dem Aufruf

```
cvs update -j branch-1
```

Dabei müssen natürlich alle auftretenden Konflikte beseitigt und mit commit wieder ins Repository übertragen werden.

### 4.6 Vorteile von CVS



Hinweis

Was sind nun die Vorteile von CVS insbesondere bei der Verwendung von CVS unter IDEs wie Eclipse, wo dies standardmäßig integriert ist?

- Mehrere Versionen des Programmes können leicht verwaltet werden.
- Im Fehlerfall kann einfach auf eine frühere Version zurückgesprungen werden. Gleichzeitig ist damit die Integrität der Dateien sichergestellt, da alle zu einem Versionsstand passen.
- Die Dokumentation der Änderungen kann gleichzeitig für einen offiziellen Changelog und für die offizielle Dokumentation herangezogen werden.
- Ein aggressives Ändern von Code in agilen Projektumgebungen wird durch CVS unterstützt und erfolgt quasi auf einem „sicheren Boden“.

### Locking Strategien

Unter Locking versteht man das Sperren von Dateien in der Versionskontrolle.

Pessimistic Locking /  
Lock-Modify-Unlock

Fast alle Versionskontrollsysteme unterstützen diese Arbeitsstrategie, bei der die Datei, an der man selbst arbeitet, gesperrt wird. Es erscheint als die einfachste und sicherste Variante Konflikte zu vermeiden. Es hat sich allerdings gezeigt, dass effizientes Arbeiten dadurch nicht besser wird. In vielen Fällen werden Sperren vergessen und führen damit zu Problemen. Ein disjunktes Arbeiten an verschiedenen Codestellen einer Datei ist so ebenfalls ausgeschlossen.

Optimistic Locking /  
Copy-Modify-Merge

Dagegen wird fast immer die Strategie des **Optimistic Locking** angewendet, die ein gleichzeitiges Arbeiten an Dateien erlaubt, ja sogar fast dazu motiviert!

Es hat sich gezeigt, dass Entwickler in den seltensten Fällen an gleichen Codestellen arbeiten. Gleichzeitig ist es in IDEs mittlerweile trivial (ein Mausklick) ein einchecken durchzuführen. Die Wahrscheinlichkeit für das Auftreten von größeren Konflikten wird dadurch stark minimiert.

Auftretende Konflikte lassen sich in der Regel sehr einfach beseitigen, da die IDE gute Differenzdarstellung liefern kann (siehe nächster Abschnitt).

## 4.7 IDE Integration

Das Einrichten einer Verbindung zum CVS-Server ist in allen relevanten IDEs trivial und wird im Folgenden kurz anhand von Eclipse demonstriert:

In der CVS Repository Exploring Perspektive kann mit der rechten Maustaste auf „New | Repository Location“ eine neue Verbindung eingerichtet werden:

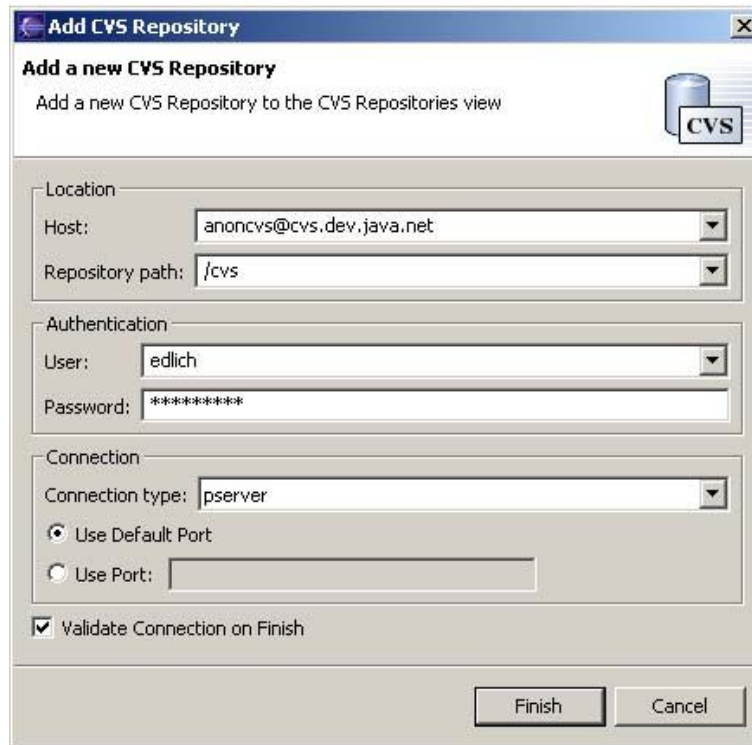


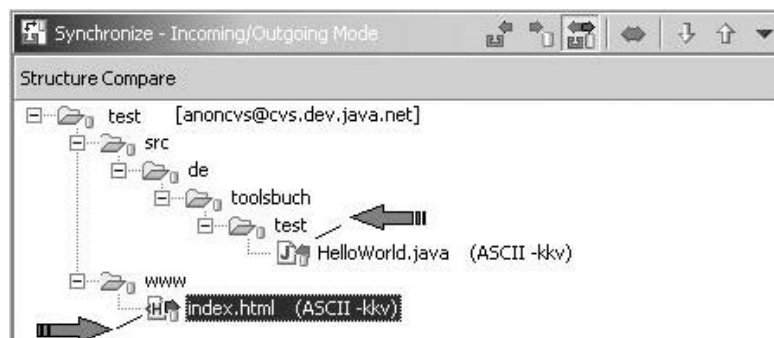
Abb.: Neue Verbindung einrichten

Die Parameter wie host, user, Repository-Pfad und der Verbindungstyp werden hier eingegeben. Wichtig ist, dass die Validierung der Verbindung getestet wird (wie auch unter Subversion). Dadurch soll vermieden werden, dass die Verbindung nicht nur als bloßes Symbol dort liegt, sondern auch gleich korrekt verbunden ist.

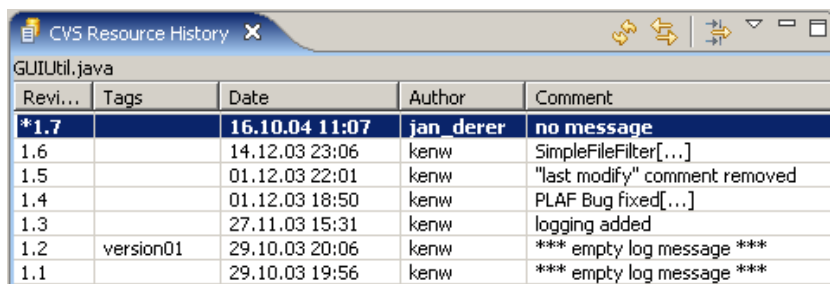
Danach kann beispielsweise der HEAD-Zweig geöffnet und das passende Projekt herausgesucht werden. Dieses kann dann ausgecheckt und mit einem neuen oder bestehenden Projekt verbunden werden. Es kann auch ein lokales Projekt der IDE genommen und mit dem geöffneten Repository verbunden werden.

Synchronize

Bei Änderungen von Dateien können diese mit „Team | Synchronize with Repository“ synchronisiert werden. Neben dem Synchronize-View können Änderungen in einem Diff-viewer angezeigt werden.



Um sich einen Überblick über die Revisionshistorie einzelner Dateien zu verschaffen, kann mit „Team | Show Resource History“ eine Übersicht geöffnet werden:



Revi...	Tags	Date	Author	Comment
*1.7		16.10.04 11:07	jan_derer	no message
1.6		14.12.03 23:06	kenw	SimpleFileFilter[...]
1.5		01.12.03 22:01	kenw	"last modify" comment removed
1.4		01.12.03 18:50	kenw	PLAF Bug fixed[...]
1.3		27.11.03 15:31	kenw	logging added
1.2	version01	29.10.03 20:06	kenw	*** empty log message ***
1.1		29.10.03 19:56	kenw	*** empty log message ***

## 4.8 Konfliktlösung

Jede Synchronisation mit dem Repository kann mögliche Konflikte anzeigen, die im optimistic locking mode mit anderen Entwicklern aufgetreten sind:

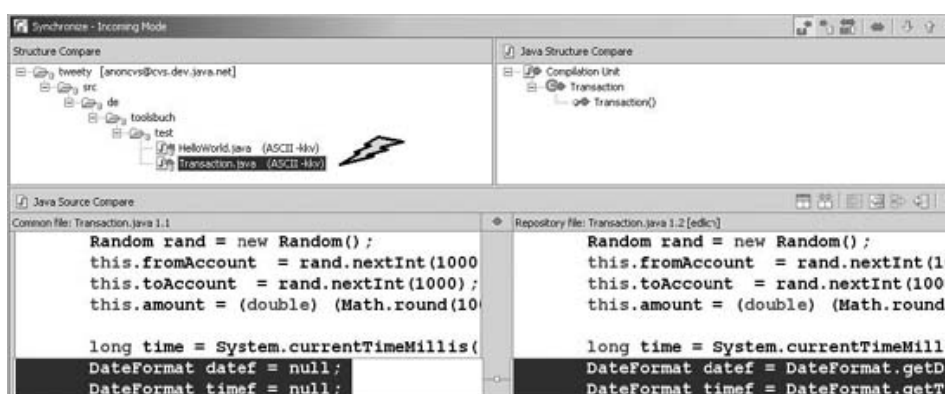


Abb.: Konflikte im optimistic locking mode

Von Vorteil ist, dass direkt in dem Fenster editiert werden kann, welches die Differenzen zeigt. Wahlweise kann auch entschieden werden, welche Version letztendlich bevorzugt werden soll, wenn beispielsweise eine Version nur ein Test oder Irrtum war. Der Editor für den Konfliktmodus zeigt immer an, welche Version „Ingoing“ oder „Outgoing“ ist und bietet auch die Operation „Override and Commit“ an.



## 5 Subversion

Subversion wurde ins Leben gerufen, um alle Schwächen von CVS zu beheben und damit CVS quasi abzulösen. Die Entwicklung von Subversion wurde von CollabNet initiiert. Subversion wird auf [www.subversion.tigris.org](http://www.subversion.tigris.org) gehostet und von CollabNet betreut. CollabNet stellt lediglich Entwickler, kontrolliert aber nicht direkt den Weg von SubVersion. Man kann daher getrost von einem unabhängigen System sprechen. Firmen können dieses System ohne Bedenken einsetzen. Der Erfolg von Subversion liegt wahrscheinlich darin begründet, dass es von bekannten (CVS)-Entwicklern begleitet wird, es schnell in einer relativ stabilen Version verfügbar war und es fast keine Open-Source Alternative gibt.



Erfahrung mit den modernsten Versionskontrollsystemen kann nicht angelesen, sondern nur praktisch gesammelt werden. Daher wird auch hier nur ein minimaler Anteil an Konzepten, Features und Handhabung vermittelt. Wichtigster Teil dieses Kapitels ist jedoch die Aufgabe am Ende. Bei dieser Aufgabe soll versucht werden, selbst einen svn-Server und einen Client (z. B. subclipse) zu installieren und anschließend damit zu arbeiten.

Freies Buch

Wer sich für die Interna von Subversion interessiert, dem sei das frei downloadbare Buch von COLLINS-SUSSMAN, FITZPATRICK und PILATO verwiesen, das hier als PDF vorliegt.

<http://svnbook.red-bean.com>

### 5.1 Konzepte

Viele der wichtigsten Änderungen von Subversion zu CVS sind Effizienzsteigerungen im Protokoll. Die für den Einsteiger wichtigsten vier Neuerungen sind:

- SVN fühlt sich zunächst an wie CVS und ist erst einmal ein Superset von CVS. Die meisten Zugriffsfunktionen sind ähnlich oder gleich.
- Der Zugriff auf das Repository / den Server kann über mehrere Wege geschehen:
  - A) über ein Web-Protokoll
  - B) über einen Client, der die Subversion Bibliothek implementiert
  - C) lokal

Wie die Abbildung auf der nächsten Seite zeigt, kann das *http* basierte WebDAV/DeltaV Protokoll als Erweiterung von Apache genutzt werden, um Quelltexte zu editieren.

- Das Umbenennen, Verschieben und Löschen von Dateien und Verzeichnissen ist komplett überarbeitet worden, sodass sich diese nun einfacher handhaben lassen. Dies ist eine wesentliche **Voraussetzung für ein flexibleres Refactoring!** Ohne dieses Feature ginge beim Package-Refactoring (merken, remove und neues insert) die Historie verloren.
- Subversion bietet einen einfachen Standalone Server. Dies macht das System auch zum Kennenlernen oder für kleine Projekte zum idealen Versionskontrollsystem.

Für Subversion existieren neben dem Web-Client auch Clients auf Commandshell-Basis, für Eclipse ([www.subclipse.org](http://www.subclipse.org)), Clients, die sich in den Explorer integrieren ([www.tortoissvn.org](http://www.tortoissvn.org)) und viele mehr.

## 5.2 Architektur

Der Subversion Server kann seine Dateien in einer Open-Source Datenbank (z. B. [www.BerkeleyDB](#)) oder in einem proprietären Dateiformat speichern.

Auf das Repository kann entweder lokal, über den Server oder über die Apache-Extension zugegriffen werden. Verschiedene Clients können mit Hilfe der Bibliothek, die Subversion zur Verfügung stellt, implementiert werden.

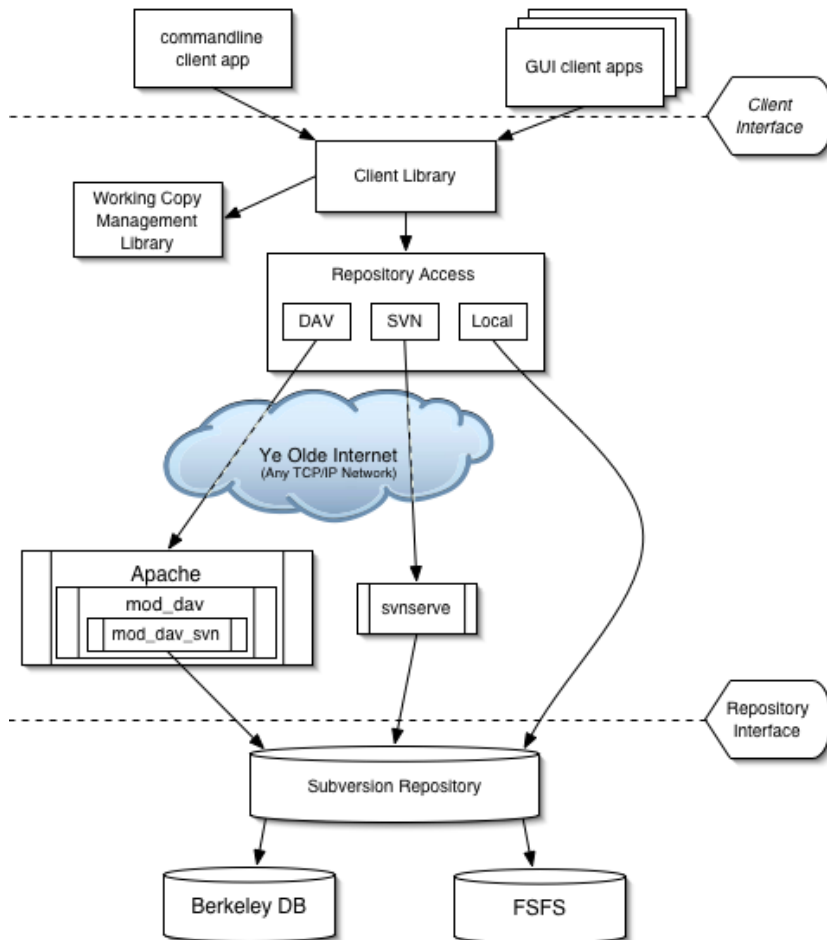


Abb.: Version Control with Subversion

Quelle: Sussmann, Fitzpatrick, Pilato „Version Control with Subversion“, O'Reilly

Die Client-Bibliothek ist ebenfalls in Java erhältlich: [www.JavaSVN](#). Das bedeutet, jeder kann auch unter Java Clients schreiben, die dann als Swing/SWT-Anwendung, Plugin oder Webanwendungen laufen. Normalerweise liegen in den Distributionen die C++ Quellen dann als dll oder in Versionen für UNIX vor.

### 5.3 Der Subversion Server

Der Subversion Server und alle Administrationsprogramme liegen für Windows als Zip- oder als Installationsdatei vor. Letzteres ist relativ unsinnig, da Subversion über die installierten Menüeinträge nicht gestartet werden kann. Für andere Betriebssysteme (RedHat, Debian, Suse, FreeBSD, OpenBDS, Solaris, MacOS) liegen die Pakete in den entsprechenden Formaten vor.

Das Werkzeug `svnadmin` erstellt ein leeres Repository:

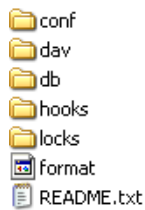
Erzeuge  
Repository

#### Create Repository

```
$ svnadmin create /path/to/repos
```

Unter Windows kann dort irgendein Pfad genommen werden wie z. B. `C:/SVNREPOS42`

Dadurch wird die folgende Verzeichnisstruktur erstellt:



Ein initialer Import von Daten kann mit dem `svn` Kommando ausgeführt werden:

Import

#### Import Data

```
$ svn import /tmp/project file:///path/to/repos -m "initial import"
```

Alle weiteren Befehle, wie der nachstehende, sieht man sich am besten mit `svn help` an.

#### Arbeiten mit svn

```
$ svn checkout file:///path/to/repos/trunk project
```

Serverstart

Der Server selbst kann, falls er remote angesprochen werden soll, mit `svnserve` gestartet werden. Ein Beispiel

#### Server starten

```
$ svnserve -d --listen-port 6642
```

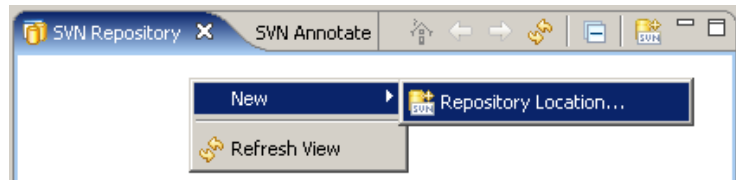
## 5.4 IDE Integration

Für Subversion existieren mittlerweile für fast alle renommierten IDEs Plugins.

Im Folgenden wird ein schneller Blick auf Subclipse geworfen, da die Installation und die Bedienung einfach ist.

Subclipse kann per Eclipse Update [www http://subclipse.tigris.org/update](http://subclipse.tigris.org/update) als Feature installiert werden. Auf der Subclipse Seite befindet sich eine ausführliche Installationsbeschreibung, die auch für Eclipse 3.\* gilt.

Nach der Installation von Subclipse kann das vom Server erstellte Repository geöffnet werden:



Fehlerträchtig ist immer die Angabe der URL die folgende Formen annehmen kann:

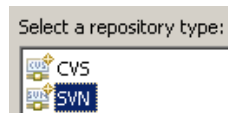
Schema:	Zugriffsmethode
file://	für Zugriff auf ein lokales Repository
http://	Zugriff auf Apache mittels WebDAV
svn://	Zugriff mittels svn-Protokoll an den svn-Server

Daneben existieren natürlich auch noch die Secure-Varianten. In den remote Varianten ist dann als Pfad `/host[:port]/path` anzugeben. Bei der Angabe der Repository location können Username und Passwort weggelassen werden (remote Anwendungen fragen in dem Fall wo Daten benötigt werden, selbst danach).

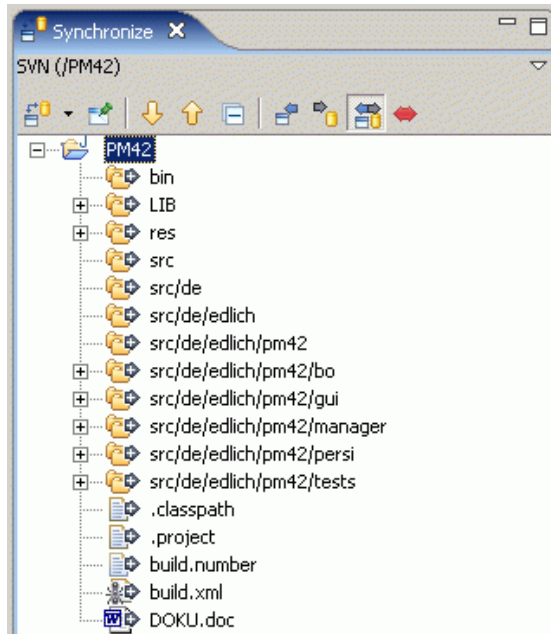
Bei einer lokalen Anmeldung in nachstehender Form ist gar kein Starten des Subversion-Servers mehr nötig!



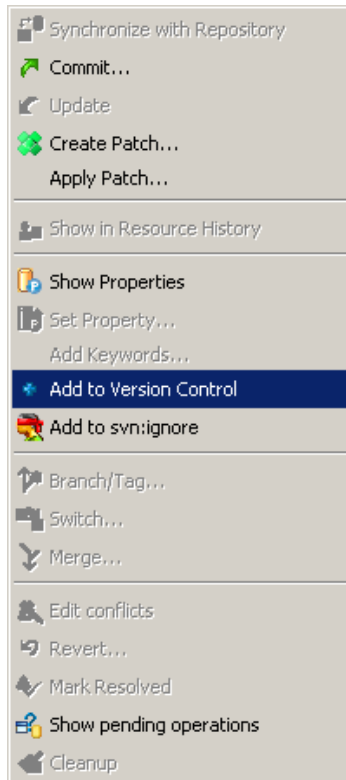
Wurde das Repository erfolgreich geöffnet, kann ein Java Projekt per rechter Maustaste auf „TEAM | Share Project“ mit dem Repository verknüpft werden.



Danach öffnet sich der Repository-View, in der offene Aktionen oder Konflikte aufgezeigt werden.



Abschließend können ganze Projekte oder Packages unter die Kontrolle von Subversion gestellt und nach Belieben committed oder ignoriert werden.



## 6 Verteilte Versionskontrolle mit GIT und Mercurial

Dieses Kapitel wurde mit freundlicher Unterstützung von Frau Anja Wegner zusammengestellt.  
(Editor: Stefan Edlich)

Die bisher vorgestellten Versionskontrollsysteme haben alle eine zentrale Struktur mit zwei Arten von Datenbeständen: ein zentrales Repository zur Bereitstellung aller Revisionen und eine lokale Arbeitskopie für den Anwender.

### Zentrale Versionskontrolle

Die Historie wird ausschließlich zentral vorgehalten. Alle Operationen, die auf Daten des Repositorys zurückgreifen oder Daten in das Repository überführen sollen, können nicht lokal durchgeführt werden. Die lokalen Arbeitsbereiche sind dazu auf eine Verbindung mit dem zentralen Repository angewiesen.

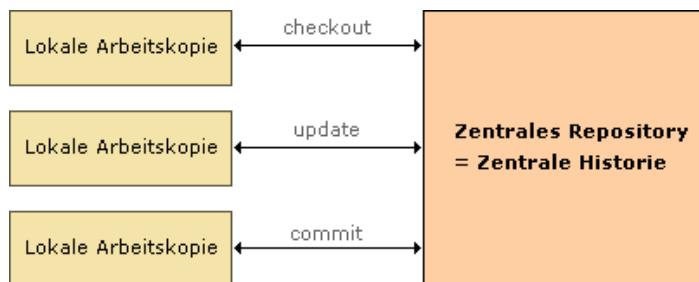


Abb.: Zentrale Versionskontrolle

### Verteilte Versionskontrolle

Bei verteilter Versionskontrolle besitzt jeder Anwender sein eigenes lokales Repository, in das er Änderungen einpflegt. Die einzelnen individuellen Repositories können über bestimmte Operationen (z. B. push, pull) Informationen austauschen [ [Dat10](#) ] .

Die lokalen Repositories ermöglichen weitgehend unabhängiges Arbeiten. Eine gemeinsame Projektplattform kann über ein Remote Repository realisiert werden.

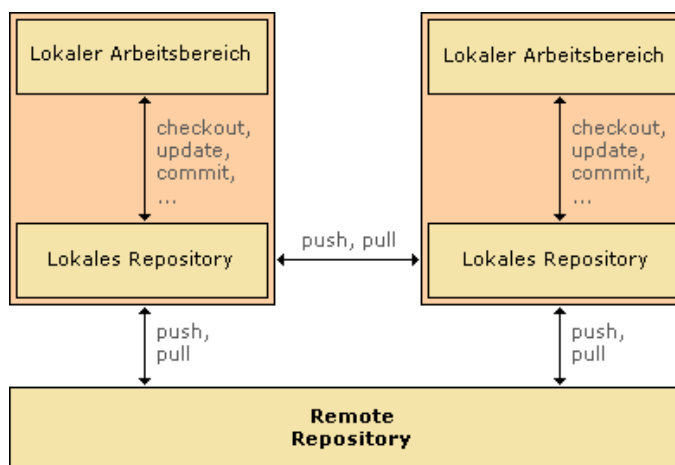


Abb.: Verteilte Versionskontrolle


## 6.1 Funktionsweise und Operationen der verteilten Versionskontrolle

Basis der verteilten Versionskontrolle ist das „verteilte Repository“. Jeder Nutzer hat sein eigenes lokales Repository – komplett mit Historie und eigenen Entwicklungszweigen (Branches).

Diese lokalen Repositories können weitgehend unabhängig arbeiten. Operationen wie z. B.: Hinzufügen von Dateien in das Repository, Verändern von Dateien, Protokollieren von Änderungen (committing), Durchsuchen der Dateihistorie, Anlegen von alternativen Entwicklungszweigen (branching) und Zusammenführen verschiedener Branches (merging) finden im lokalen Bereich statt. Für diese Operationen ist kein Austausch mit einem zentralen Server oder einem anderen Repository nötig.

Ohne äußere Vorgaben sind alle Repositories gleichberechtigt. Es besteht jedoch auch die Möglichkeit, ein „zentrales“ remote Repository als gemeinsame Projektplattform einzurichten und dieses bei Bedarf hierarchisch aufzuwerten. Grundlegende Funktionen für den Austausch von Informationen zwischen unterschiedlichen lokalen Repositories oder auch zwischen einem lokalen Repository und einem zentralen Serverarchiv sind

- **push** (Transfer von lokalen Informationen zu einem externen, nicht lokalen Repository) und
- **pull** (Kopieren externer, nicht lokaler Informationen in das eigene lokale Repository).

Als Key-Feature liefern Systeme zur Unterstützung der verteilten Versionskontrolle die Möglichkeit einer eindeutigen Beschreibung der eigenen Historie, üblicherweise mittels SHA1 Hashes zur eindeutigen Datenidentifikation.  [ Goo10 ]

## 6.2 Vor- und Nachteile der verteilten Versionskontrolle

Verteilte Versionskontrolle bringt vor allem zwei entscheidende Vorteile: sie ermöglicht unabhängiges und schnelles Arbeiten. Die meisten Operationen können lokal ausgeführt werden und laufen damit deutlich schneller ab als bei zentralen Versionskontrollsystemen. Der Entwickler ist nicht auf eine Netzwerkverbindung oder den Zugang zu einem zentralen Server angewiesen, um effektiv arbeiten zu können. Er hat in seiner lokalen Sandbox alle nötigen Informationen und operativen Möglichkeiten um unabhängig arbeiten zu können. Ebenso ist er beim Anlegen und Weiterentwickeln verschiedener Branches in seinem lokalen Bereich völlig frei.

Darüber hinaus ist verteilte Versionskontrolle außerordentlich demokratisch und kommunikativ. Eine Entwicklung kann verschiedene Wege, d. h. verschiedene lokale Repositories, durchlaufen, die sich alle auf der gleichen „hierarchischen“ Stufe bewegen. Gleichmaßen besteht die Möglichkeit, ein Repository hierarchisch höher einzustufen und damit klassisch zentral zu arbeiten.

Als aufwendiger wird bewertet, dass die bei der zentralen Versionskontrolle transparent zu handhabenden Revisionsnummern durch global eindeutige Hashwerte (z. B. SHA 1) ersetzt werden müssen. Durch deren Verteilung auf unterschiedliche Systeme kommt es zu einem erhöhten Aufwand bei der Verwaltung der Revisionen. Jeder Entwickler muss die Infrastruktur des „verteilten Netzes“ kennen.

### Werkzeuge zur verteilten Versionskontrolle

Werkzeuge zur Unterstützung der verteilten Versionskontrolle sind z. B. Git, Mercurial, Monotone (alle Open Source), Clear Case (Proprietär), Bitkeeper (Proprietär). Das System Bazaar fährt hingegen einen hybriden Ansatz mit oder ohne zentralem Server.

Die Versionskontrolle mit Git und Mercurial wird im Folgenden vertieft untersucht.

## 6.3 Versionskontrolle mit Mercurial

Mercurial ist ein plattformunabhängiges Werkzeug zur Unterstützung der verteilten Versionskontrolle mit Fokus auf Einfachheit und Schnelligkeit.

Mercurial steht unter [www.mercurial-scm.org/](http://www.mercurial-scm.org/) als Open-Source zum Download zur Verfügung. Es wird primär über die Kommandozeile gesteuert, alle Befehle beginnen mit `hg`, dem Elementsymbol für Quecksilber (engl. Mercury).

Graphische Frontends  
und IDE-Integration


Entwicklungsumgebungen wie Netbeans oder Eclipse unterstützen Mercurial direkt bzw. über zusätzliche Installation eines Plug-Ins.

Mercurial selbst liefert nur die Bedienung über die Kommandozeile. Allerdings stehen mit TortoiseHG für Microsoft Windows und mit MachG und Murky für Mac OS X grafische Frontends zur Verfügung, die eine Bedienung ohne Kommandozeile ermöglichen.



### 6.3.1 Mercurial – grundlegende Konzepte

Mit Mercurial läuft Versionskontrolle konsequent verteilt: Jeder Entwickler besitzt eine lokale Kopie eines „Basis – Repositories“ die alle Dateien des Projektes einschließlich der Dateihistorie umfasst. Die so erzeugten lokalen Repositories sind absolut selbständig und unabhängig.

Für Mercurial ist die Historie unantastbar. Ein Mercurial-Repository ist im Sinne einer stetig wachsenden Sammlung unveränderlicher Objekte aufgebaut.  [ Goo10 ] Jedes Repository enthält ein geschütztes Verzeichnis namens `.hg`. Darin befindet sich das echte Repository, die Historie des Projektes.

#### Working Directories


Daneben existieren weitere Verzeichnisse und Dateien, die **working directories**. Diese sind frei veränderbar und zeigen einen Ausschnitt des Projektes zu einem bestimmten Punkt in der Historie auf.  [ Sul09 ]

Die Verbindung zwischen den working directories und dem Repository liefert ein Commit, der entsprechende Veränderungen im working directory in das Repository übernimmt (Kommando `hg commit`).

Mercurials Historie baut auf folgenden grundlegenden Konzepten auf:

- Changeset
- Changelog
- Manifest und
- Filelog

#### Changeset


Ein Changeset ist eine Sammlung von Changes an Dateien im Repository. Es enthält alle aufgezeichneten lokalen Modifikationen, die zu einer neuen Revision des Repositories geführt haben. Jede Bestätigung einer Veränderung (`hg commit`) erzeugt ein neues Changeset. Das working directory kann auf jedes erzeugte Changeset (zurück-)gesetzt werden. In Mercurial hat jedes Changeset entweder null, ein oder zwei Eltern-Changesets.  [ Mer10a ]

#### Changelog


Alle Changesets eines Repositories sind im Changelog enthalten. Für jede Dateirevision wird aufgezeichnet, wer die Veränderungen bestätigt hat, ggf. mit weiteren Changeset-bezogenen Informationen, und welche die Revision des zugehörigen Manifests ist.

 [ Sul09 ]

#### Manifest

Das Manifest ist eine Datei, die den Inhalt des Repositories zu einem bestimmten Changeset beschreibt. Es enthält eine Liste der zu diesem Zeitpunkt aktuellen Dateinamen und Dateirevisionen  [ Mer10b ] .

#### Filelog

Ein Filelog enthält die Historie einer bestimmten Datei in Form von Metadaten. Dabei hält das Filelog zwei Arten von Informationen bereit: die Revisionsdaten und einen Index, um Dateirevisionen schnell und effizient wiederfinden zu können  [ Sul09 ]

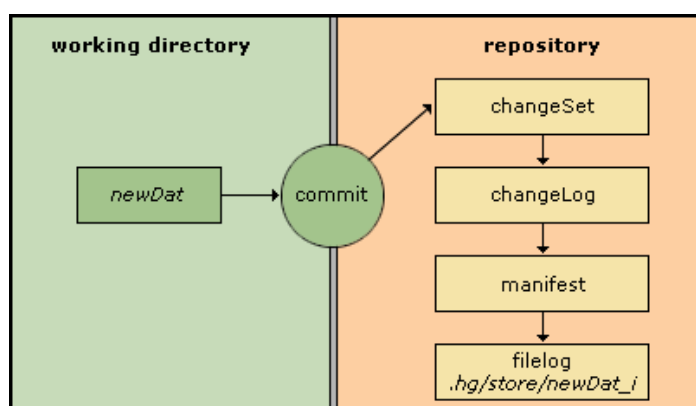


Abb.: Grundkonzepte Mercurial

Mercurial hat ein einziges Datenformat zur Speicherung von Dateirevisionen: Das Revlog. Jede Revision wird entweder voll-ständig komprimiert oder nur als Delta gegenüber der vorigen Revision im gleichen Revlog gespeichert. Eine vollständige Datei wird immer dann gespeichert, wenn zum Wiederherstellen der Datei zu viele Delta-Daten gelesen werden müssen [Hei06].

### 6.3.2 Mercurial – Basisfunktionen, Kommandos und Operationen

Um ein Verzeichnis unter die Kontrolle von Mercurial zu stellen, muss dieses zunächst importiert werden, d. h. als lokales Arbeitsverzeichnis angelegt werden. Dies kann entweder über das Kommando `hg init` oder über `hg clone` erfolgen.

Anschließend kann man Dateien für die Verwaltung im Repository anmelden, bearbeiten und entsprechende Veränderungen committen. Wichtig ist, dass jede Veränderung erst mit einem entsprechenden Commit in das Repository übernommen wird.

Die nachfolgende Übersicht zeigt eine Zusammenfassung der Basiskommandos in Mercurial.

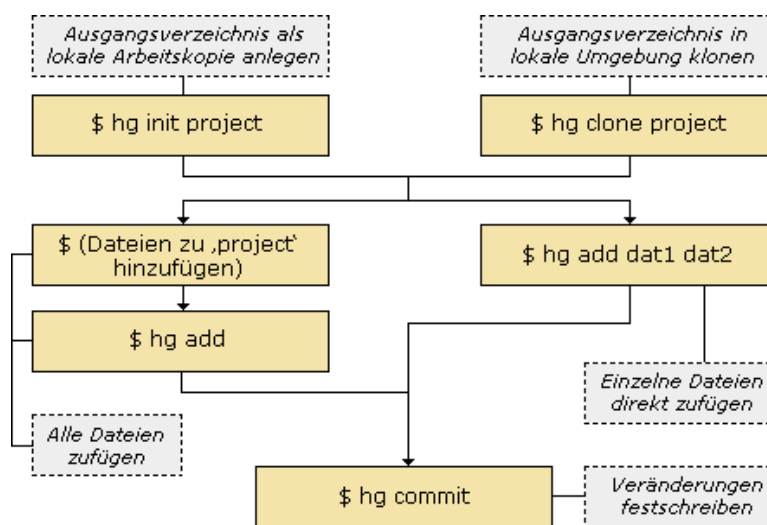


Abb.: Basiskommandos von Mercurial: Projekt anlegen / initialisieren

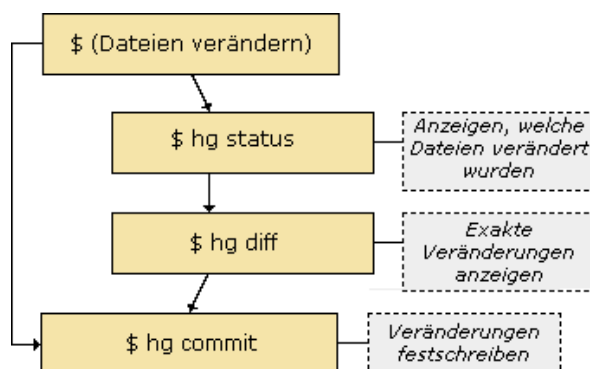


Abb.: Basiskommandos von Mercurial: Dateien verändern

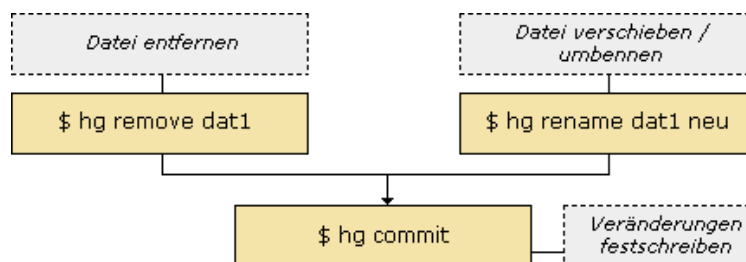


Abb.: Basiskommandos von Mercurial: Dateien entfernen oder verschieben



Abb.: Basiskommandos von Mercurial: Projekthistorie anzeigen

### 6.3.3 Branching und Merging

Über das Kommando `hg update [Revisionsnummer]` besteht die Möglichkeit, frühere Versionen (Changesets) aufzurufen, zu bearbeiten und Veränderungen zu committen. Anschließend können die Veränderungen mit der aktuellen Version zusammengeführt und eventuelle Konflikte identifiziert und aufgelöst werden.

Beispiel  [ [Mer10c](#) ]


```
$ hg update 3
$ (Datei verändern)
$ hg commit
$ hg merge // mit der letzten Version zusammenführen
$ hg resolve -list // Konflikte auflisten
$ hg resolve conflicting_file // Konflikte auflösen
$ hg resolve --mark conflicting_file // als gelöst markieren
$ hg commit
```

Mercurial kann Konflikte, die beim Mergen auftreten nicht eigenständig lösen. Die Auflösung von Konflikten erfolgt per Hand oder mit Unterstützung fremder Software (z. B. `kdiff3`).


### 6.3.4 Austausch mit externen Repositories (push/ pull)

Mercurial bietet die Möglichkeit, mittels push- und pull-Operation Informationen mit anderen Repositories auszutauschen.


Die mittels `hg pull` eingeholten Informationen werden zunächst separat angelegt, d. h. die Pull-Operation tastet das working directory nicht an. Über `hg update` können die über `hg pull` gezogenen Veränderungen in das working directory übernommen werden.

Gleichmaßen verändert auch das Kommando `hg push` nicht eigenständig das working directory im empfangenden Repositories, allerdings ist auch update-Kommando seitens des pushenden Repositories ausgeschlossen, dies kann nur innerhalb des empfangenden Repositories ausgelöst werden  [ Sul09 ]

<pre>\$ hg tip changeset: 4:2278160e78d4 tag: tip user: Bryan O'Sullivan &lt;bos@serpentine.com&gt; date: Sat Aug 16 22:16:53 2008 +0200 summary: Trim comments.</pre>	Aktuelle Version anzeigen
<pre>\$ hg pull ../my-hello pulling from ../my-hello searching for changes adding changesets adding manifests adding file changes added 1 changesets with 1 changes to 1 files (run 'hg update' to get a working copy)</pre>	Informationen holen
<pre>\$ hg tip changeset: 5:b6fed4f21233 tag: tip user: Bryan O'Sullivan &lt;bos@serpentine.com&gt; date: Tue May 05 06:55:53 2009 +0000 summary: Added an extra line of output</pre>	Aktuelle Version anzeigen

Abb.: Pull-Operation mit Mercurial  [ Sul09 ]

<pre>\$ hg push ../hello-push pushing to ../hello-push searching for changes adding changesets adding manifests adding file changes added 1 changesets with 1 changes to 1 files</pre>	Informationen abgeben
--	-----------------------


Abb.: Push-Operation mit Mercurial  [ Sul09 ]

Über `hg incoming` bzw. `hg outgoing` besteht die Möglichkeit, sich die Informationen anzuzeigen, die über `hg pull` bzw. `hg push` übertragen werden würden.


## 6.4 Versionskontrolle mit GIT

Git ist eine freie Software zur Unterstützung der verteilten Versionskontrolle, die im Zuge der Entwicklung des Linux-Kernels von Linus Torvalds entwickelt wurde.

Git sieht seinen Fokus auf Schnelligkeit, Effizienz und praktische Tauglichkeit in großen Entwicklungsprojekten. Es wurde primär für die Anwendung auf UNIX-basierten Systemen, wie z. B. Linux, Solaris, Mac OS X entwickelt. Für Microsoft Windows muss auf die Hilfe der Cygwin-Umgebung oder auf Msysgit zurück gegriffen werden.

Git steht unter  <http://git-scm.com/> zum Download zur Verfügung.

### Anwendung und IDE-Integration

Eine ausführliche Liste der Tools, Frontends und Interfaces für Git liefert  <https://git.wiki.kernel.org/>.

Für Git sind zahlreiche grafische Frontends verfügbar, z. B. Gitk, GitGUI, TortoiseGit (Windows) und GitX (Mac OS X). Selbstverständlich sind auch verschiedenste IDE-Integrationen möglich, z. B. Eclipse mit eGit oder Netbeans mit nbGit.

### 6.4.1 Git – Grundlegende Konzepte

Als Werkzeug zur Unterstützung der verteilten Versionskontrolle bietet auch Git die grundlegenden Funktionen an: Jeder Entwickler verfügt über ein unabhängiges lokales Repository, d. h. eine vollständige Kopie der Projektdateien und der Projekthistorie selbst.

Das Git-Repository ist eine Datenbank mit allen notwendigen Informationen zur Aufbewahrung und Verwaltung der Dateien und der Historie. Die Daten des Repositories werden in der Wurzel des Arbeitsverzeichnisses in einem Verzeichnis namens `.git` gespeichert. Innerhalb des Repositories unterstützt Git hauptsächlich zwei Datenstrukturen: den Objektspeicher und den Index.

Der Kern des Git-Repositories befindet sich im Objektspeicher. Er enthält alle Datendateien im Original, alle Lognachrichten, Autorennformationen und weitere Informationen zum Wiederherstellen von Verzweigungen oder Versionen. Dabei gibt es vier Arten von Objekten:

- Blob
- Tree
- Commit
- Tag

Blobs

Blobs (Binary Large Object) enthalten die Daten der Dateien, jedoch keine Metadaten. Ein Baum (Tree) bündelt einen oder mehrere Blob-Objekte in eine Verzeichnisstruktur.

Commits

Commits enthalten die Metadaten jeder Änderung am Repository und verweisen auf das entsprechende Baumobjekt, das für den Zeitpunkt der Ausführung des Commits einen vollständigen Snapshot des Repositories liefert. Mit Ausnahme des Root-Commits besitzt jedes Commit ein oder mehrere Elterncommits.

Tags

Tags weisen einem bestimmten Objekt einen für Nutzer lesbaren Namen zu.

Index

Der Index ist eine dynamische, temporäre Binärdatei, die als Arbeitsbereich (*staging area*) zwischen *working directory* und Repository genutzt werden kann. Er umfasst eine bestimmte Version der Gesamtstruktur eines Projektes – entweder darstellbar durch einen Commit oder einen Baum zu einem beliebigen Zeitpunkt des Projektes oder aber als Zustand, der aktiv und aktuell entwickelt wird. Damit ermöglicht der Index eine Trennung zwischen Entwicklungsschritten über verschiedenste Git-Kommandos und dem Bestätigen dieser Entwicklungen über ein Commit. Git erlaubt es, den Index in definierten Schritten zu verändern

 [ Loe10 ] .

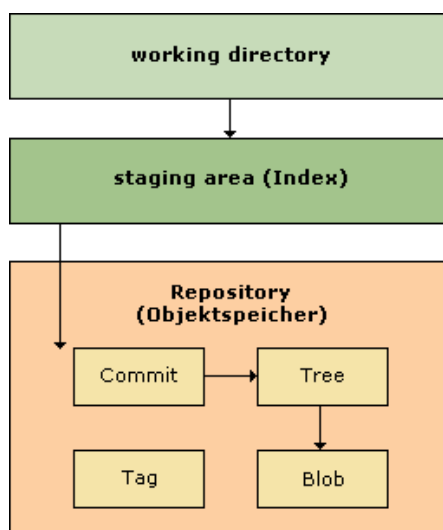


Abb.: Grundkonzepte von Git

## 6.4.2 Basisfunktionen, Kommandos und Operationen

Nachfolgende Übersicht zeigt die Basiskommandos in Git, welche nahezu identisch zu Mercurial sind. Die Unterschiede sind rot markiert.

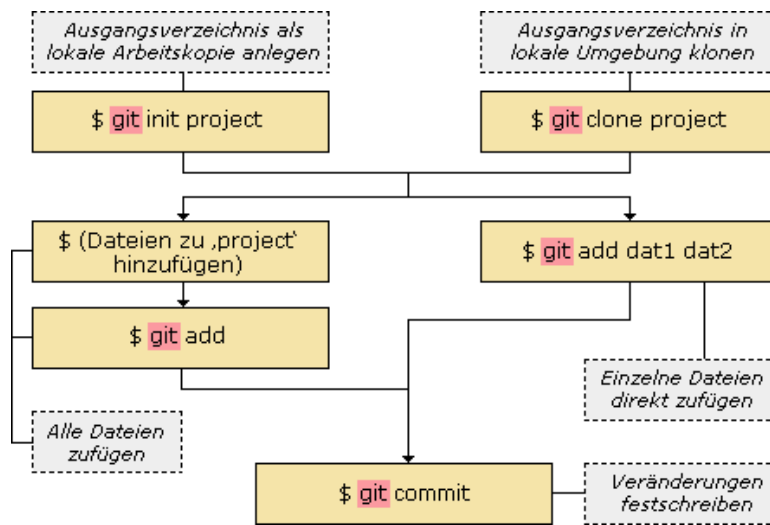


Abb.: Basiskommandos Git:  
Projekt anlegen / initialisieren

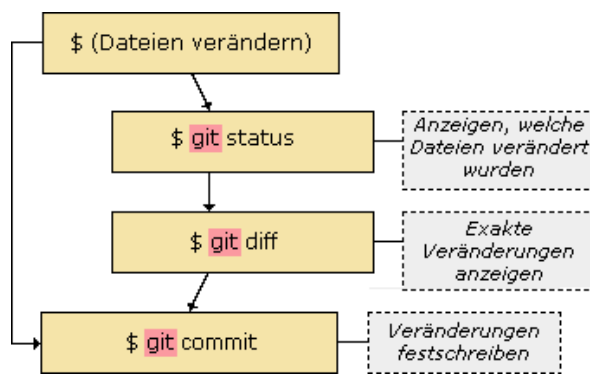


Abb.: Basiskommandos Git:  
Dateien verändern

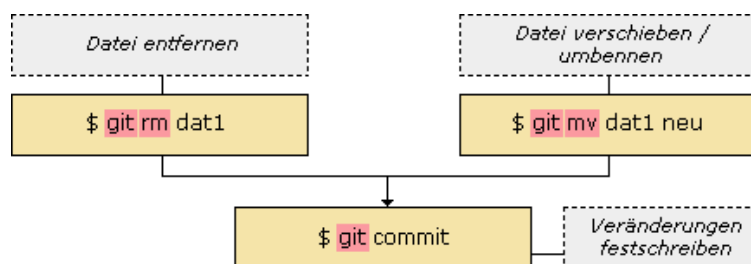


Abb.: Basiskommandos Git:  
Dateien entfernen oder verschieben

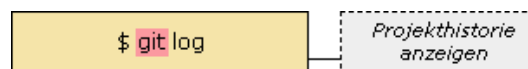


Abb.: Basiskommandos Git:  
Projekthistorie anzeigen

### 6.4.3 Branching und Merging

Grundlegendes Kommando für die Arbeit mit verschiedenen Entwicklungszweigen (Branches) ist das Kommando `$ git branch`.

```
$ git branch //listet verfügbare Zweige auf
$ git branch (branchname) //erzeugt neuen Zweig
$ git checkout -b (branchname) //erzeugt neuen Zweig und wechselt dorthin
$ git branch -d (branchname) //löscht einen Zweig
```

Für das Zusammenführen verschiedener Zweige bietet Git das Kommando `$ git merge [branchname]`. Dabei kontrolliert Git eigenständig, ob beim Zusammenführen der Zweige Konflikte entstehen und listet diese gegebenenfalls auf:

```
$ git merge fix_readme
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

Abb.:  
Merge-Konflikte Git

### 6.4.4 Austausch mit externen Repositories (push/ pull)

Git bietet folgende Kommandos, um Informationen zwischen einem lokalen Repository und einem remote Repository auszutauschen:

`$ git fetch` führt eine Synchronisation der verschiedenen Repositories durch, in dem nicht lokal vorhandene Daten vom remote Repository geladen werden und in *remote Branches* abgelegt werden. Mit *remote Branches* können grundsätzlich alle Operationen durchgeführt werden, die auch lokale erzeugte Branches bieten, einzige Ausnahme ist ein Checkout.

`$ git pull` verbindet einen fetch sofort mit einem merge zwischen dem *remote Branch* und dem aktuell ausgeführten lokalen Branch.

Das Kommando `$ git push` ermöglicht in entgegengesetzter Richtung das Update eines remote Repositories mit lokalen Veränderungen.

Git bietet zudem Vereinfachungen für die Arbeit mit verschiedenen remote Repositories: Das Kommando `$ git remote` liefert eine Liste aller verzeichneten remote Repositories einschließlich deren URL, `$ git remote add` ermöglicht das Hinzufügen neuer, `$ git remote rm` das Entfernen existierender remote Repositories.

 [ GH10 ]

### 6.4.5 Zeitreise durch Tagging

In der Softwareentwicklung ist es normal, dass eine Anwendung ständig Änderungen unterworfen ist. Sei es durch Behebung von Fehlern, Hinzufügen von Funktionen oder Strukturverbesserungen in Form von Refactoring. Kein Programm ist auf Anhieb perfekt, falls es so etwas wie ein perfektes Programm überhaupt gibt. Um den Überblick über Änderungen zu behalten, macht es Sinn mit Versionsnummern zu arbeiten. Möchte ich zu einer bestimmten Version springen, geht das mit Git ganz leicht über `$ git checkout <commit-hash>`.

Problem

Das Problem ist: Woher weiß ich, welcher git-commit zu welcher Versionsnummer gehört?

Eine Möglichkeit ist über `$ git log` in den Commit-Messages nach einem Hinweis zu suchen. Voraussetzung dafür ist natürlich, dass beim Committen daran gedacht wurde, einen entsprechenden Vermerk mit in die Nachricht zu packen. Das ist aber nicht das einzige Problem. Man stelle sich vor, zwischen zwei Versionen liegt eine unüberschaubare Anzahl an Commits. Welchen Zeitaufwand würde es bedeuten, diese alle zu durchsuchen!



Abhilfe schafft Tagging. Git bietet zwei Arten von Tags, die sich im Inhalt ihrer Informationen unterscheiden. Mit `$ git tag -a <tag-name> -m <tag-message>` speichert man eine Referenz mit dem Namen von `<tagname>` auf den letzten Commit. Zusätzlich werden die eingegebene Nachricht, der Name des Taggers und das Datum abgespeichert. Möchte man diese Informationen nicht mitspeichern, reicht es `$ git tag <tagname>` zu benutzen. Beispiel:  
`$ git tag -a v0.1 -m "My version 0.1"`.

Über `$ git tag` kann man nun eine Liste aller Verfügbaren Tags anzeigen lassen. Mehr Informationen zu einem Tag erhält man über `$ git show <tag-name>`.

Ein Checkout ist nun einfach über `$ git checkout <tag-name>` möglich.



Achtung

**Wichtig:** Tags werden nicht automatisch über `$ git push` auf einen Remote übertragen. Um einen lokalen Tag zu teilen, muss man ihn explizit mit `$ git push <remote-name> <tag-name>` pushen. Alternativ kann man auch `$ git tag push <remote-name> --tags` benutzen, um alle lokalen Tags zu pushen.



Hinweis

Tags müssen sich nicht einzig und alleine auf bestimmte Versionen beschränken. Jeder Commit, der später schnell auffindbar sein soll, hat einen Tag verdient.

Weitere Informationen zum Thema Tagging:

<https://git-scm.com/book/en/v2/Git-Basics-Tagging>.


## 6.5 GIT und Mercurial im Vergleich

Grob betrachtet weisen Git und Mercurial weitgehende Übereinstimmungen auf: Beide Systeme haben einen in weiten Teilen identischen Kommandosatz und weisen eine ähnliche grundlegende Logik im Programmablauf und -aufbau auf.

Nachfolgend sind die wichtigsten Bereiche der Anwendung eines Versionskontrollsystems vergleichend dargestellt:

### Systemphilosophie

Sowohl Git als auch Mercurial setzen die verteilte Versionskontrolle konsequent um. In beiden Systemen besitzen die lokalen Repositories weitgehende Unabhängigkeit – ein Austausch zwischen unterschiedlichen Repositories wird unterstützt. Beide Systeme ermöglichen flexibel die Unterstützung verschiedener Projektstrukturen: von der vollständig verteilten Entwicklung über ausschließlich gleichrangige lokale Repositories bis hin zu einer klassisch zentralisierten Entwicklung über Einbindung und Austausch mit einem (ggf. hierarchisch übergeordneten) remote Repository.


Mercurial legt seinen Fokus dabei weitgehend auf Einfachheit und leichte Erlernbarkeit des Systems, der Fokus für Git liegt in Schnelligkeit und Effizienz – vor allem hinsichtlich des Erstellens und Zusammenfügens verschiedener Entwicklungszweige. Git steht daher im Ruf, eine flachere Lernkurve als Mercurial zu besitzen.  [ Goo10 ]

### Plattformabhängigkeit

Git ist im Grunde ein Linux-Produkt, das zwar z. B. mit Msysgit auch für Windows einsetzbar ist, aber dennoch Linux-orientiert bleibt. Mercurial hingegen liefert grundlegend Plattformunabhängigkeit.


### Lokale Repositories und Projekthistorie

Ein Git-User verfügt über mehr Macht innerhalb seines Repositories als ein Mercurial-User. Mit Git ist es möglich, einzelne Bestandteile eines Entwicklungszweigs im Repository zu entfernen, d. h. bestimmte Revisionsdaten zu entfernen (z. B. über `$ git reset`).

Mercurial's Historie ist als ständig anwachsende Sammlung von Objekten aufgebaut und benötigt immer alle Revisionsdaten bis hin zum initialen Commit.  [ Goo10 ] Um Entwicklungszweige in sich zu dezimieren sind daher mit Mercurial einige Umwege (z. B. über differenziertes Klonen) nötig.

Darüber hinaus bietet Git über die staging area (Index) eine Art Zwischenschicht zwischen working directory und Repository. Innerhalb der staging area besteht die Möglichkeit, den nächsten Commit genau vorzubereiten. Dabei können bei Bedarf nur einzelne Dateien in die staging area übernommen und committed werden, andere modifizierte Dateien im Arbeitsverzeichnis sind von einem solchen Commit nicht betroffen.

### Branching und Merging

Git lässt eine unbegrenzte Anzahl an Eltern-Commits zu und ermöglicht es damit, einen Merge zwischen n Branches durchzuführen. Mercurial begrenzt die Anzahl der Eltern-Commits auf zwei, so dass für einen Merge zwischen mehr als zwei Entwicklungszweigen mehrere Zwischenschritte erforderlich sind.  [ Goo10 ]

## Zusammenfassung

Sowohl Git als auch Mercurial bieten effektive Unterstützung für die verteilte Versionskontrolle. Beide Systeme bieten in weiten Teilen gleichartige Abläufe und Anwendungsmöglichkeiten. Welches der beiden Tools den besseren Support liefert, kann bestenfalls nutzergruppenspezifisch bewertet werden.

Git bietet über die staging area einige Möglichkeiten mehr, sich in der Entwicklung eines Projektes auszuprobieren. Andererseits bleibt Mercurial in seiner Anwendungslogik dafür klarer und schlanker.

Insgesamt bleibt festzuhalten, dass sich Anhänger der verteilten Versionskontrolle mit Sicherheit in beiden Systemen wiederfinden und grundlegende Features vorfinden werden.

## 6.6 Literatur zum Thema GIT und Mercurial

### [ Dat10 ]

DATAKOM Buchverlag GmbH (Hrsg.): VCS (version control system) – ITwissen.info.

Quelle:  <http://www.itwissen.info/...> (Abruf 12.11.2010)


### [ Goo10 ]

GOOGLE Inc. (Hrsg.): DVCS Analysis, Analysis of Git and Mercurial – Google Project Hosting.

Quelle:  <http://code.google.com/...> (Abruf 31.12.2010)

### [ Wik10 ]

WIKIMEDIA Foundation Inc. Inc. (Hrsg.): Mercurial.


Quelle:  <http://de.wikipedia.org/...> (Abruf 31.12.2010)

### [ Sul09 ]

**O` SULLIVAN, BRYAN** (2009): Mercurial – The Definitive Guide. 1. Aufl. Sebastopol: O`Reilly Media, Juni 2009 – ISBN 978-0596800673

### [ Mer10a ]

MERCURIAL Community: Mercurial – Changeset.

Quelle:  <https://www.mercurial-scm.org/...>

### [ Mer10b ]

MERCURIAL Community: Mercurial – Manifest.

Quelle:  <https://www.mercurial-scm.org/...>

### [ Mer10c ]

MERCURIAL Community: Mercurial – Guide.

Quelle:  <https://www.mercurial-scm.org/...>

### [ Hei06 ]

**HEIN, THOMAS ARENDSSEN** (2006): INTEVATION GmbH (Hrsg.): Mercurial Distributed SCM – Die verteilte Alternative zu CVS.

Quelle:  <http://intevation.net/~thomas/...> (Abruf 31.12.2010)

### [ GC10 ]

GIT Community: Git – About Git. Quelle:  <http://git-scm.com/about> (Abruf 17.12.2010)

### [ Loe10 ]

**LOELIGER, JON** (2010): Versionskontrolle mit Git. deutsche Ausgabe, Köln: O`Reilly Verlag GmbH & Co. KG, ISBN 978-3897219458

### [ GH10 ]

GITHUB Inc. (Hrsg.): Git Reference. Quelle:  <http://gitref.org> (Abruf 31.12.2010)

## 7 Fehlermanagement



In mittleren oder großen Softwaresystemen ist das Management und die Weiterverfolgung von Fehlern unerlässlich. Man spricht daher auch oft vom Defect-, Bug-, Errortracking oder -Management.

Die meisten Systeme gehen jedoch in das Projektmanagement über und werden mittels integrierter Tools und Features, die benötigt werden, bearbeitet. Das Fehlermanagement ist dann ein SubFeature dieser Tools. Features, wie die Verwaltung fehlender Funktionalität (Request for Enhancements (RFEs), auch Issues oder Tasks genannt), sind ein Teil dieser Projektmanagement Software und auch immer ein Teil reiner Bug-Tracking Systeme wie Bugzilla.

In kleineren Projekten kann es jedoch völlig ausreichend sein, Fehler, oder das gesamte Projekt, mit allgemeineren Systemen, beispielsweise Wikis, zu erfassen.

### LITERATUR

 [ BE03 ]

- [5] **MARTIN BACKSCHAT, STEFAN EDLICH,**  
2003 J2EE-Entwicklung mit Open-Source-Tools,  
Spektrum Verlag (Elsevier), ISBN 3-8274-1446-6

Dieses Buch enthält im Kapitel 5.3 eine ausführliche Einführung in das Bugtracking, die insbesondere auch auf integrierte und fortschrittliche Lösungen wie Maven eingeht.

## 7.1 Anforderungen

Wichtig für die Beschreibung von Fehlern sind die folgenden Informationen:

- **Textbeschreibung**  
Eine textuelle Beschreibung des Fehlers mit Angaben, wie dieser reproduziert werden kann.
- **Priorität**  
Die Beurteilung wie gravierend ein Fehler ist, wird in der Regel durch einen Projektleiter o.ä. vorgenommen. Eine gewisse Anzahl relevanter Bugs kann ein Update der Software rechtfertigen.
- **Konfiguration**  
Angabe, unter welchen exakten Soft- und Hardwarebedingungen der Bug entdeckt wurde.
- **Systemteil**  
In welcher Komponente wurde der Bug verursacht.
- **Ergänzende Informationen**  
Hier werden häufig Screenshots oder Logfiles abgelegt, wie sie meistens auch vom Hotline-Support für ein Softwareprodukt angefordert werden.




Das reine Erfassen von Bugs ist in der Regel nicht ausreichend. Benötigt werden Systeme, die das Ganze managen und die folgenden Informationen zur Verfügung stellen.

- **Bug-Zuordnung**  
Bugs müssen Entwicklern oder Abteilungen zugeordnet werden. Durch ersteres muss auch verhindert werden, dass Bugs von zwei Entwicklern parallel gefixt werden.
- **Bug-Zustand**  
Bug Zustände sind in der Regel ASSIGNED, RESOLVED, VERIFIED oder CLOSED. Der Bug selbst durchläuft üblicherweise einen Lebenszyklus, der in der Regel aus FIXED, INVALID, WONTFIX, DUPLICATE und WORKSFORME (kann nicht reproduziert werden) besteht. In seltenen Fällen gibt es auch LATER oder REMIND.
- **Bug-Historie**  
Alle vergangenen Aktionen an einem Bug müssen aufgezeichnet werden. Fehler können erhebliche Kosten verursachen, weshalb nachträglich eine sachgerechte (wirtschaftliche) Analyse immer möglich sein sollte.
- **Paralleles Arbeiten und Berechtigungen**  
Bugs müssen von beliebig vielen Personen auch parallel erfasst werden können. Berechtigungen stellen sicher, dass Bugs nur dann als behoben freigegeben werden, wenn er von der Qualitätssicherung getestet und abgenommen wurde.
- **Abfrage und Auswertung**  
Eine beliebige Suche oder Abfrageformulierung über Bugs oder Module ist wichtig. Dazu gehört auch ggf. die Generierung von Reports, die das Ganze auch graphisch aufbereiten können.

## 7.2 Werkzeuge

Es folgt eine kurze Übersicht über die wichtigsten Werkzeuge für das Fehlermanagement. Einen guten Überblick über die Leistungsfähigkeit derartiger Systeme kann man sich verschaffen, indem man unter [www.Sourceforge](#) stöbert oder vielleicht dort sogar selbst ein ernsthaftes Projekt einrichtet.

Produkt	Fähigkeiten
Perforce Jobs	In das gesamte Perforce-System eingebettetes defect-tracking system.
<a href="#">www.Bugzilla</a> 	Ein Open-Source Werkzeug für die Erfassung und Zuordnung von Fehlern in einer Gruppe von Entwicklern. Entwickler können Bugzilla verwenden, um ToDo-Listen zu führen, Punkte zu priorisieren, zu scheitern und <u>Abhängigkeiten</u> zu verfolgen.
<a href="#">www.DevTrack</a>	Leistungsfähiges Defect und Projekt Tracking System.
<a href="#">www.ExtraView</a>	Kommerzielles Webbasierendes System. Unterstützt <u>Workflows</u> Projekte und Produktlinien. Rollenbasiertes System mit API Support.
<a href="#">www.Jira</a>	Ein J2EE basiertes System für das <u>Projektmanagement</u> mit Fokus auf zu erreichende Tasks. Unterstützt Workflows, Tracking von Dokumenten und Versionen, Suche und Filter, E-Mail, RSS, Excel, <u>XML</u> und Web-Services. Die kommerzielle Standard-Version kostet ca. 1200 \$.
<a href="#">www.Scarab</a>	Scarab tritt an, um Bugzilla zu ersetzen. Es bietet eine breite Palette an Features wie: Bug-Management, Suchen, Reports, Benachrichtigungen, Workflowunterstützung, Tracking von Abhängigkeiten, Mehrsprachigkeit, Im-/Export, Look- and Feel Anpassbarkeit.
<a href="#">www.Soffront TRACKWeb</a>	Ein flexibles Aufgaben- und Projektmanagementsystem (inkl. Zeit und Kosten). Als Web-Client verfügbar. Unterstützt Reporting, E-Mail, SSL, Visual-Source-Safe, Testfälle, Anhänge, <u>Aktivitäten</u> und Taskplanung etc.
<a href="#">www.SourceForge Enterprise</a>	Der Host der wohl bekanntesten <a href="#">www.Community</a> für Entwicklungsprojekte bietet seine Software auch kommerziell an. Es enthält eine Vielzahl an Features, die ein gemeinsames Entwickeln ermöglichen und wird von Entwicklern aller Welt geschätzt (derzeit über 105.000 Projekte und über 1,1 Mio. registrierte Entwickler!).
<a href="#">www.TeamTrack</a>	Ein <u>Workflow</u> orientiertes Defect- und Issue-Tracking System der Firma Serena.
<a href="#">www.Visual Intercept</a>	Microsoft orientierte Defect-Tracking Lösung, die Visual Studio, Word, Excel und Power-Point integriert. Unterstützt Email Notifications, Prozess-Kontrolle, Security, Report-Generierung,

Tab.: Werkzeuge

Quelle: Auszüge aus [www.http://perforce.com](#)

### 7.3 Bugzilla



Bugzilla wurde im Rahmen von mozilla.org entwickelt und ist heute das Bugtracking-System mit den meisten Installationen weltweit. Viele Open-Source Projekte setzen auf Bugzilla, da ein Web-interface für eine breite Zugänglichkeit sorgt.

Bugzilla ist in Perl entwickelt und setzt auf mySQL auf. Empfohlen wird der Apache Webserver.

Wichtige Key-Features sind:

- **E-Mail-Benachrichtigung:** An dem Bug beteiligte Personen können per E-Mail informiert werden. Interessant ist ein Feature, bei dem Bugzilla informiert, wenn ein Bug zu lange ignoriert wird.
- **Milestone-Management**
- **Mail Interface:** Mit E-Mails lassen sich Änderungen am Zustand von Bugs vornehmen. Dies erlaubt auch eine Anbindung an CVS.
- **Security und Vertraulichkeit**
- **Gute Query und Reporting Möglichkeiten**



Beispiel

#### Fehlerreport aus Bugzilla

Das nachfolgende Bild zeigt einen Fehlerreport aus Bugzilla für Mozilla

Search for bugs

Summary: contains all of the words/terms

Product: FoodReplicator, MyOwnBadSelf, WorldControl

Component: Cursort, comp2, EconomicControl, PoliticalReckStabbing, SatSprinkler

Version: 1.8 unspecified

Status: UNCONFIRMED, NEW, ASSIGNED, REOPENED, RESOLVED, VERIFIED, CLOSED

Resolution: FIXED, INVALID, WON'TFIX, LATER, REMIND, DUPLICATE, WORKSFORME

Priority: P1, P2, P3, P4, P5

OS: All, Windows 3.1, Windows 95, Windows 98, Windows ME, Windows 2000, Windows NT

Email and Numbering: Any of, bug owner, reporter, CC list member, assignee

Bug Changes: Only bugs changed in the last days, Only bugs when any of the fields (bug owner, assignee, accessible, bug file, CC) were changed between and View

Sort results by: Bug Number


Advanced Querying Using Boolean Charts: bug

New | Cancel | Find bug# | Reports | New issues | Log in

Ein Bug unter Bugzilla enthält die folgenden Elemente:

- Product and Component
- Status and Resolution
- Assigned To
- URL, Summary
- Status Whiteboard
- Keywords
- Platform and OS
- Version
- Priority
- Severity
- Target
- Reporter
- CC list
- Attachments
- Dependencies
- Votes and Additional Comments

## Zusammenfassung

- Versions- und Fehlermanagement sind Teil des **Continuous Integration** Zyklus und unabdingbare Werkzeuge der Softwaretechnik.
  - Erste Systeme waren RCS und SCCS, die unter UNIX versuchten, einen automatisierten Management-Layer über „diff“ zu legen.
  - Ein Entwickeln mit CVS unter einer komfortablen IDE ermöglicht ein **mutiges Entwickeln** auf sicherem Boden ohne Hindernisse. Fast alle relevanten Projekte (z. B.  sf.net) setzen daher auf Tools wie CVS oder Subversion auf.
  - Systeme wie CVS oder Subversion ermöglichen **Optimistic Locking**.
  - Subversion ermöglicht u.a. mit verbesserter move- und delete-Struktur ein **transparentes Refactoring** auch über Packages hinweg.
  - Das Aufsetzen eines Subversion Servers - sei es für das lokale Arbeiten unter Windows oder für ein remote Zugriff auf einen Server - ist **schnell**, also in einer halben Stunde **erledigt** und kann viele Tage Arbeit ersparen.
  - Ein **Bugtracking** in Projekten ist mit freien Tools wie Bugzilla oder Scarab für Projekte ab mittlerer Größe wichtig und **einfach** realisierbar. Für kleine Projekte bieten sich Wikis an, die teilweise mit einem Kopierbefehl (z. B. unter Tomcat) realisiert werden können.
-



## Wissensüberprüfung



Installieren

### Übung SVN-01

#### CVS-Client - Installation und Zugriff

Installieren Sie sich einen CVS-Client wie bspw. WinCVS und verwenden Sie CVS, um auf Repositories zuzugreifen die Sie interessieren!

Versuchen Sie ein komplettes Projekt herunterzuladen und dieses zu Builden.

Gute Startpunkte / Projekte sind z. B.

[www.java.net](http://www.java.net)

[www.apache.org](http://www.apache.org)

[www.Jakarta](http://www.jakarta.org) dort z. B. [www.Commons](http://www.commons.org))

[www.Sourceforge](http://www.sourceforge.net)

Bearbeitungszeit: 20 Minuten



Planen und Entwickeln

### Übung SVN-02

#### Server Installation

Bitte setzen Sie für Ihr Softwareprojekt einen Versionskontroll-Server - wie Subversion, Mercurial oder Git - auf.

Entwicklen Sie mutig!

Bearbeitungszeit: 60 Minuten



Testen

### Übung SVN-03

#### Experimente mit dem Server

- Importieren Sie ein bestehendes Projekt in Ihren Server!
- Erzeugen Sie mit Ihrem Kollegen einen **gleichzeitigen Commit** auf dem Server, damit Sie mit dem Konfliktmanager die verschiedenen Editierungsstellen sehen und bearbeiten können!
- Versuchen Sie testweise, einen **Branch** abzuzweigen und diesen etwas am Leben zu erhalten. Ungefährlich wäre auch, einen Branch abzuzweigen und etwas zu integrieren, dass Sie sowieso gerne im Hauptzweig (**trunk**) hätten und dann später dieses Feature wieder in den Hauptzweig zu integrieren, indem Sie den Branch wieder zusammenführen.

Bearbeitungszeit: 40 Minuten