Arbeiten im Team

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Allgemeine Qualifikationen

Ein Software-Entwickler benötigt folgende allgemeine Qualifikationen:

Abstraktionsvermögen. Nur mit Hilfe der Abstraktion können komplexe Systeme bewältigt werden.

Kommunikationsfähigkeit. Gute sprachliche Ausdrucksweise und Präsentation ist wichtig. Für die Dokumentation benötigt man eine gute Schriftform.

Teamfähigkeit. Mitarbeiter sollen Teamgeist besitzen und konstruktiv und kooperativ zum Teamergebnis beitragen.

Allgemeine Qualifikationen (2)

Wille zum lebenslangen Lernen. Das Wissen der Software-Technik verdoppelt sich alle vier Jahre.

Intellektuelle Flexibilität und Mobilität. Auch das gesamte Umfeld der Software-Technik ändert sich permanent (Beispiel: Internet, e-commerce, Mobile Anwendungen).

Kreativität. In der Software-Technik gibt es (noch?) kein breites Erfahrungspotential, aus dem man unbesehen schöpfen könnte.

Allgemeine Qualifikationen (3)

Hohe Belastbarkeit. Mitarbeiter müssen „stressverträglich“ sein.

Umfrage (1989):

- 85% aller EDV-Fachkräfte machen Überstunden
- 33% regelmäßig
- 21% sogar mehr als 5 Überstunden pro Woche
- 45% machen Überstunden ohne Freizeit- oder Bezahlungs-Ausgleich

Englisch lesen und sprechen. Publikationen, technische Dokumente und Produkte werden in der Regel zunächst in Englisch geschrieben.

Viele Produkte und Dokumente gibt es nur in Englisch.







Das Zerrbild des Programmierers

The popular image of a programmer is of a lone individual working far into the night peering into a terminal or poring over reams of paper covered with arcane symbols.

Tatsächlich jedoch hoher Anteil (ca. 50%) an Gruppenarbeit und Kommunikation.

The lack of encouragement and the unattractive image may explain why even the most mathematically able college women are more likely to major in the humanities than in a discipline like computer science.

Was motiviert Menschen bei der Arbeit?

Wir unterscheiden drei *Motivationstypen*:

- Aufgabenbezogener Typ
- Selbstbezogener Typ
- Interaktionsbezogener Typ







Aufgabenbezogener Typ

- motiviert durch
  intellektuelle Herausforderung der Arbeit
- Selbstcharakterisierung als unabhängig, einfallsreich, zurückhaltend, introvertiert, energisch, wetteifernd, selbständig
- Mehrheit der Softwareentwickler ist aufgabenbezogen







Selbstbezogener Typ

- motiviert durch *persönlichen Erfolg*
  (gemessen in Geld oder Statussymbolen)
- Selbstcharakterisierung als eklig, lästig, hartnäckig, dogmatisch, introvertiert, eifersüchtig, draufgängerisch, wetteifernd
- Ziellerreichung vor allem im Management







Interaktionsbezogener Typ

- motiviert durch *Zusammenarbeit*
- Selbstcharakterisierung als friedfertig, hilfsbereit, rücksichtsvoll, besonnen, geringes Autonomie- und Statusbedürfnis
- durch Anwender-orientierte Projekte angezogen
- Frauen häufiger interaktionsbezogen (Gründe unklar!)







Gruppenzusammensetzung

Der Gruppenerfolg ist abhängig von der *Zusammensetzung:*

Gleichartig motivierte Gruppen erreichen ihr Ziel nur bei interaktionsbezogenen Mitgliedern.

Folgerung:

- Gruppen sollten Mitglieder *aller Motivationskategorien* enthalten
- Der Gruppenleiter sollte *aufgabenbezogen* sein.

Vielfalt steigert die Kreativität!

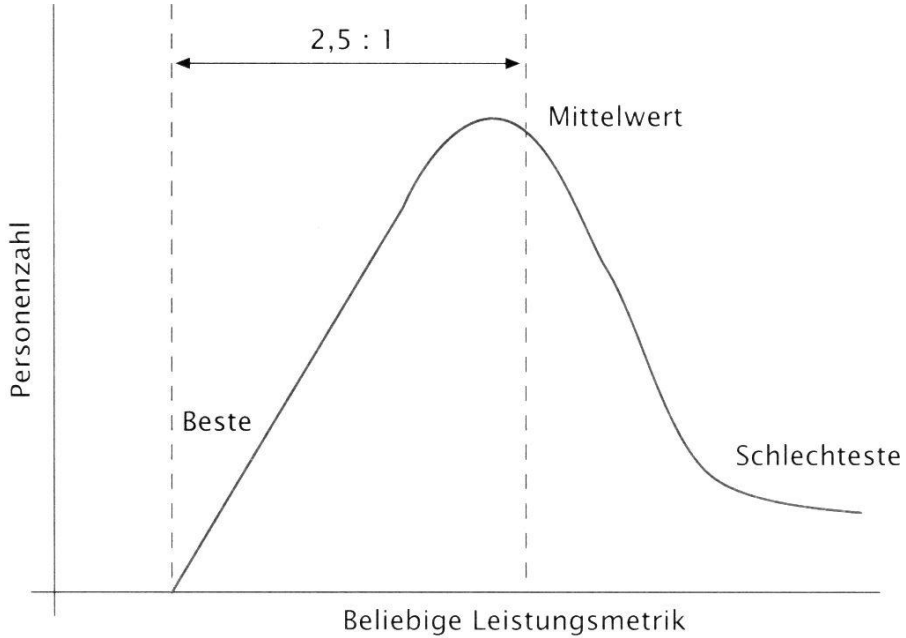






Produktivität fördern

Große Produktivitätsunterschiede zwischen Software-Entwicklern:



Grundregeln Produktivität

- Die besten Mitarbeiter sind 10× besser als die schlechtesten
- Die besten Mitarbeiter sind 2,5× besser als der Durchschnitt
- Die überdurchschnittlichen Mitarbeiter übertreffen die unterdurchschnittlichen Mitarbeiter im Verhältnis 2:1.

Diese Regeln gelten für fast jede beliebige Leistungsmetrik (Zeit, Fehler...).







Relevante Faktoren

Ein Mitarbeiter ist um so produktiver

- je ruhiger sein Arbeitsplatz (= je weniger er gestört wird)
- je besser die Privatsphäre ist
- je größer der Arbeitsplatz ist.

Übersicht Umfrageergebnisse:

	Arbeitsplatzfaktoren	bestes Viertel der Teilnehmer	schlechtestes Viertel der Teilnehmer
1	Wieviel Arbeitsplatz steht Ihnen zur Verfügung?	7m^2	4,1 m^2
2	Ist es annehmbar ruhig?	57% ja	29% ja
3	Ist Ihre Privatsphäre gewahrt?	62% ja	19% ja
4	Können Sie Ihr Telefon abstellen?	52% ja	10% ja
5	Können Sie Ihr Telefon umleiten?	76% ja	19% ja
6	Werden Sie oft von anderen Personen grundlos gestört?	38% ja	76% ja

Hintergrund

Um produktive Ingenieurarbeit zu erledigen, muss man „in Fahrt“ *(in flow)* sein:

- „in Fahrt“: Zustand tiefer, fast meditativer Versunkenheit
- Man fühlt eine leichte Euphorie und verliert das Zeitgefühl

Für diesen Zustand benötigt man ca. 15 Minuten voller Konzentration

In diesem Zustand ist man besonders anfällig für Störungen

Ein Telefonanruf und die 15 Minuten Eintauchphase beginnen von vorne!







Konsequenzen

- Abstellbare (oder gar keine) Telefone – keine Lautsprecherdurchsagen
- Einzelbüros (mit verschließbarer Tür) – keine „Cubicles“
- Niedriger Geräuschpegel – keine Großraumbüros

Musik kann zwar einen Geräuschpegel übertönen, blockiert jedoch die rechte Gehirnhälfte, die für Kreativität zuständig ist

Teambildung fördern

Die folgenden Faktoren fördern die Teambildung:

Team zu Erfolgen verhelfen. Teams brauchen ein gemeinsames *konkretes* Ziel. Keine abstrakten Firmenziele *(mission statements)*, sondern messbare, prüfbare Ziele. Die Arbeit sollte so aufgeteilt werden, dass genügend oft Erfolgserlebnisse da sind.

Elite-Team. Mitarbeiter brauchen das Gefühl, *einzigartig* zu sein. Egal, worin sich die Einzigartigkeit ausdrückt, sie ist Grundlage für die Identität des Teams.







Teambildung fördern (2)

Qualitätskult. „Nur das Beste ist gut genug für uns“. Jedes Team braucht eine *Herausforderung.* Mittelmäßige Aufgaben nehmen den Ehrgeiz, eine herausragende Leistung zu bringen. Niemand ist stolz, an einem „Schundprojekt“ zu arbeiten.

Kein Overengineering. Vergolden Sie keine Funktionen, nur weil es dem Team Spaß macht. Perfektionieren Sie das Notwendige.

Vielfalt. Größere Erfolgschancen, wenn Team vielfältig zusammengesetzt ist (z.B. Männer und Frauen, Endanwender und Entwickler. . .)

Persistenz. „Never change a winning team“.







Teambildung fördern (3)

Vertrauen statt Kontrolle. Der Manager soll sich beschränken, *Strategien* vorzugeben, sich aber nicht in die Taktik einmischen. Der Manager muss dem Team *Vertrauen* entgegenbringen; das Team muss die Möglichkeit haben, autonom sein Vorgehen zu wählen.

Bürokratie vermeiden. Im richtigen Umfang ist Dokumentation notwendig. Aber: Es darf nicht der Eindruck entstehen, das Management sei nur auf „Planerfüllung“ aus. Das Team muss spüren, dass auch das Management an sein Ziel glaubt.

Räumliche Nähe. Räumliche Trennung behindert das Gefühl der Zusammengehörigkeit.







Teambildung fördern (4)

Ein Team pro Nase. Eingeschworene Teams entstehen nur, wenn die Mitglieder den größten Teil ihrer Zeit darin verbringen. Mitgliedschaft in mehreren Teams erschwert die Teambildung – und die Effizienz.

Echte Termine. Das Management darf nur Termine vorgeben, die auch einzuhalten sind. Alles andere zerstört Glaubwürdigkeit und Vertrauensbasis.







Teamorientierte Manager...

- erkennen Kompetenz bei Mitarbeitern an.
- übertragen gewisses Maß an Freiheit und Verantwortung an ihre Mitarbeiter.
- gewähren Vertrauensvorschuss.
- lassen Teams sich selbst bilden oder räumen Mitspracherecht bei der Zusammensetzung ein.
- räumen administrative und organisatorische Hürden aus dem Weg.
- lassen Teams zeitweise völlig autonom arbeiten.
- „verbannen“ Teams zeitweise in völlige Isolation (Hotel, Ferienhaus).

Teamorientierte Mitarbeiter. . .

- sind positiv zur Teamarbeit eingestellt.
- sind Kritik- und Konflikttolerant.
- erkennen und respektieren die fachliche Qualifikation und persönliche Integrität anderer.
- verhalten sich partnerschaftlich.
- können widersprüchliche und voneinander abweichende Informationen verarbeiten.
- sind bereit, sich voll im Team zu engagieren.
- sind mit sich selbst zufrieden.







Merkmale eines eingeschworenen Teams

- Niedrige Fluktuationsrate
- Ausgeprägtes Identifikationsbewusstsein
- Freude an der Arbeit
- Bewusstsein einer Elitemannschaft



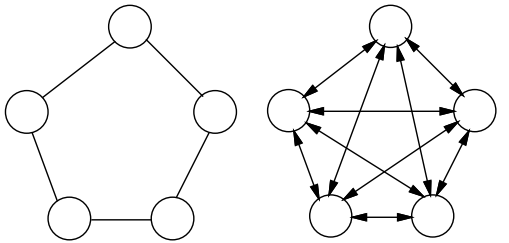




Egoless Programming

egoless programming – demokratische, dezentralisierte Organisation:

- Alle Mitglieder haben gleiche Befugnisse
- Teamleitung variiert mit Kompetenz
- Zielfindung durch Konsens
- *Code exchange*: Programme sind „Allgemeingut“ und werden zur Fehlersuche und Begutachtung ausgetauscht (vergl. *extreme programming*)



(a) Organisationsstruktur (b) Kommunikationsstruktur

Egoless Programming (2)

Vorteile

- geeignet für schwere Aufgaben
- einheitlicher Programmier- und Dokumentationsstil
- unempfindlich gegen Personalwechsel
- hohe Arbeitszufriedenheit

Nachteile

- Kommunikationsoverhead
- ineffizient bei Standardaufgaben
- häufig Terminprobleme
- hohe Risikotoleranz kann zum Misserfolg führen
- neue Ideen können unterdrückt werden

Kreativität fördern

Viele Aktivitäten der Software-Entwicklung erfordern ein hohes Maß an Kreativität – insbesondere der *Entwurf* und die *Fehlerbeseitigung.*

Kreativität ist die Fähigkeit, Wissen und Erfahrung aus verschiedenen Bereichen zu neuen Lösungen und Ideen zu verschmelzen, wobei verfestigte Denkmuster überwunden werden.







Klassisches Brainstorming

Brainstorming ist eine spezielle Form einer Gruppensitzung, mit dem Ziel, Ideen und Gedanken einer Gruppe frei fließen zu lassen und sie zu neuem zu kombinieren.







Brainstorming – Regeln

1. *Freies und ungehemmtes Aussprechen von Gedanken.* Auch sinnlos erscheinende und phantastische Einfälle sind erwünscht, da sie andere Teilnehmer inspirieren können.

   Alle Vorschläge an Tafel oder Flipchart schreiben.

2. Die gemachten Vorschläge sind als *Anregungen* aufzunehmen und assoziativ weiterzuentwickeln.

   Voraussetzung: Zuhören und inhaltlich offen sein.

3. *Kritik und Bewertung* sind während der Sitzung verboten.

   Keine *Killerphrasen* wie „Das haben wir noch nie gemacht“, „Das hat noch keiner geschafft“.

4. *Quantität* geht vor Qualität; Vernunft und Logik sind nicht gefragt.

Brainstorming – Voraussetzungen

- Erfahrener Moderator
- Disziplinierte Teilnehmer
- 4–7 Teilnehmer
- Maximale Dauer: 30 Minuten

Alternative: *Brainwriting* (Ideen werden auf Karten geschrieben, die zum Nachbarn weitergereicht werden)

Weniger spontan, aber geringeres Risiko, dass rhetorisch begabte Teilnehmer dominieren.







Effiziente Besprechungen

Viele Besprechungen kommen nur schlecht zum Ergebnis; sie gleichen mehr

Laberrunden: alle reden durcheinander

Selbstfindungssitzungen (ohne Ergebnis, „gut, dass wir darüber geredet haben“) oder

Haifischbecken (alle hacken aufeinander ein).







Tips für effiziente Besprechungen

Nur dann tagen, wenn es keine Alternative gibt. Gibt es nichts zu besprechen, muss man auch keine Zeit darauf verwenden.

Moderator bestimmen. Vor jeder Sitzung sollte ein Moderator bestimmt werden, der sich um Raum, Einladungen, Tagesordnung kümmert.

Der Moderator muss nicht der Gruppenleiter sein.

Pünktlich anfangen. Nicht warten, bis alle da sind – sonst gehen die ersten wieder und müssen später eingesammelt werden.

Störungen vermeiden. Die Sitzung sollte nicht gestört werden (auch nicht durch Zuspätkommende). Telefone und Handys ausschalten.

Tips für effiziente Besprechungen (2)

Tagesordnung. Vor jeder Sitzung muss klar sein, worüber geredet werden soll – damit sich die Teilnehmer vorbereiten können.

Eine generische Tagesordnung sieht so aus:

1. Protokoll der letzten Sitzung
2. Stand der Dinge
3. Ziele
4. Umsetzung
5. Nächste Schritte
6. Verschiedenes

Der Moderator stellt die Tagesordnung zu Beginn der Sitzung vor. Sie kann auch während der Sitzung geändert werden.

Tips für effiziente Besprechungen (3)

Stand der Dinge. Eine Sitzung beginnt gewöhnlich mit einer Zusammenfassung über den Stand der Dinge (z.B. die Ziele der letzten Sitzung und deren Umsetzung)

Ziele setzen. Die Ablaufstruktur einer Sitzung muss *zielorientiert* sein. (Für reine Informationen benötigt man keine Sitzung; an der Vergangenheit kann man nichts mehr ändern.)

Nach dem Stand der Dinge soll man deshalb *Ziele festlegen und erkennen* – möglichst spezifisch, messbar und auf ein konkretes Datum bezogen („Am 01.03. erwartet unser Kunde eine Präsentation“)







Tips für effiziente Besprechungen (4)

Problembearbeitung mit Methode. Jedes Problem (= jedes Ziel) lässt sich wie folgt behandeln:

1. Was ist das konkrete Problem?
2. Was sind die Ursachen für das Problem?
3. Was sind mögliche Lösungen?
4. Was ist die beste Lösung für das Problem?

Umsetzung planen. Wie kann man die beste Lösung erreichen? Hier ist Diskussion gefragt. Der Moderator achtet darauf, dass zielgerichtet diskutiert wird – also die geplanten Aktivitäten auch zu den Zielen passen.

Themen, die später auf der Tagesordnung stehen, kommen auch erst später dran.







Tips für effiziente Besprechungen (6)

Diskussionsregeln.

Die wichtigsten Regeln für gutes Diskutieren:

- Bis zum Schluss zuhören und andere aussprechen lassen
- Wortbeiträge nur mit vorheriger Wortmeldung
- Der Moderator erteilt und entzieht das Wort
- Einhaltung der Zeitvorgaben, z. B. Pausen
- Wir bleiben immer beim Thema
- Neue Themen und Gedanken werden notiert
- Fragen sind jederzeit zugelassen
- Fasse Dich kurz

Dies verlangt viel Disziplin, steigert aber die Effizienz.







Tips für effiziente Besprechungen (7)

Zusammenfassen. Treffen während der Diskussion verschiedene Ansichten aufeinander, ist es hilfreich und effizient, die Standpunkte *zusammenzufassen*.

Rechtzeitiges Zusammenfassen ist Aufgabe des *Moderators*.

Es ist aber auch eine gute Übung für die *Kontrahenten*, vor den eigenen Argumenten die des Vorredners zusammenzufassen. *(und ggf. neu zu interpretieren :-)*







Tips für effiziente Besprechungen (8)

Nächste Schritte. Hier werden wiederum *konkrete, messbare* Aktivitäten entschieden, die später (z.B. in der nächsten Sitzung) geprüft werden können.

Ergebnisse niederschreiben. Vergessen Sie nicht, die Ergebnisse (= Stand der Dinge und nächste Schritte) in einem *Protokoll* festzuhalten.

Dies ist gewöhnlich Aufgabe des Protokollführers (der auch identisch mit dem Moderator sein kann).

Der Moderator sorgt dafür, dass alle Teilnehmer das Protokoll erhalten (um ggf. später Einspruch einzulegen).

Zusammenfassung Teamarbeit

- Gut gestaltete Arbeitsplätze fördern die Produktivität
- Es gibt zahlreiche Faktoren, die die *Teambildung* fördern (Messbare Erfolge, Qualitätskult, . . .)
- Teams sollten möglichst divers zusammengesetzt sein (Motivation, Hintergrund)
- Brainstorming fördert die Kreativität
- Besprechungen sollen *effizient* gestaltet werden







Technische Aspekte der Teamarbeit

Neben den *organisatorischen* Aspekten gibt es auch *technische* Hilfsmittel, die die Teamarbeit unterstützen.

Wir betrachten

- Konfigurationsmanagement
- Problem Tracking







Konfigurationsmanagement

Ein großes Software-Produkt besteht aus

- Tausenden von Komponenten,
- die von Hunderten oder gar Tausenden Personen entwickelt und gewartet werden,
- die oftmals auch noch auf viele Orte verteilt sind,

und an all diesen Komponenten werden von all diesen Personen *Änderungen* vorgenommen.

Die Aufgabe, solche Änderungen zu organisieren und zu kontrollieren, heißt *Software-Konfigurationsmanagement*.







Konfigurationsmanagement: Ursprung

Konfigurationsmanagement: ursprünglich von der US-Raumfahrtindustrie in den 50er Jahren eingeführt

Problem zu dieser Zeit: Raumfahrzeuge unterlagen während ihrer Entwicklung zahlreichen undokumentierten *Änderungen.*

Erschwerend: Raumfahrzeuge wurden im Test normalerweise *vernichtet*

Folge: Nach einem erfolgreichen Test waren Hersteller nicht in der Lage waren, eine Serienfertigung aufzunehmen oder auch nur einen Nachbau durchzuführen.

Konfigurationsmanagement soll solchen *Informationsverlust* verhindern.







Ziele des Konfigurationsmanagements

Rekonstruktion: *Konfigurationen* müssen zu jedem Zeitpunkt wiederhergestellt werden können; eine Konfiguration ist dabei eine Menge von Software-Komponenten in bestimmten Versionen.

Koordination: Es muß sichergestellt werden, daß Änderungen von Entwicklern nicht versehentlich verlorengehen. Dies bedeutet insbesondere das Auflösen von *Konflikten.*

Identifikation: Es muß stets möglich sein, einzelne Versionen und Komponenten eindeutig zu identifizieren, um die jeweils angewandten Änderungen erkennen zu können.







Software-Systeme verwalten mit CVS

CVS = Concurrent Versions System

Weitverbreitetes Werkzeug zum Konfigurationsmanagement

Standard-Werkzeug in der Open-Source-Szene

SourceForge: unterhält CVS-Archive für >25.000 Projekte







CVS-Archive

CVS unterhält ein *Archiv,* in dem die *Versionen* eines Systems aufbewahrt werden.

Versionen werden *platzsparend* gespeichert, und zwar als Urfassung + n Änderungen (nach Datum sortiert).

Will man etwa auf die Fassung vom 1. Februar (1. Juni) zugreifen, nimmt CVS die Urfassung und wendet darauf sämtliche Änderungen bis zum 1. Februar (1. Juni) an.

Auf diese Weise kann der Zustand zu jedem Zeitpunkt wiederhergestellt werden.

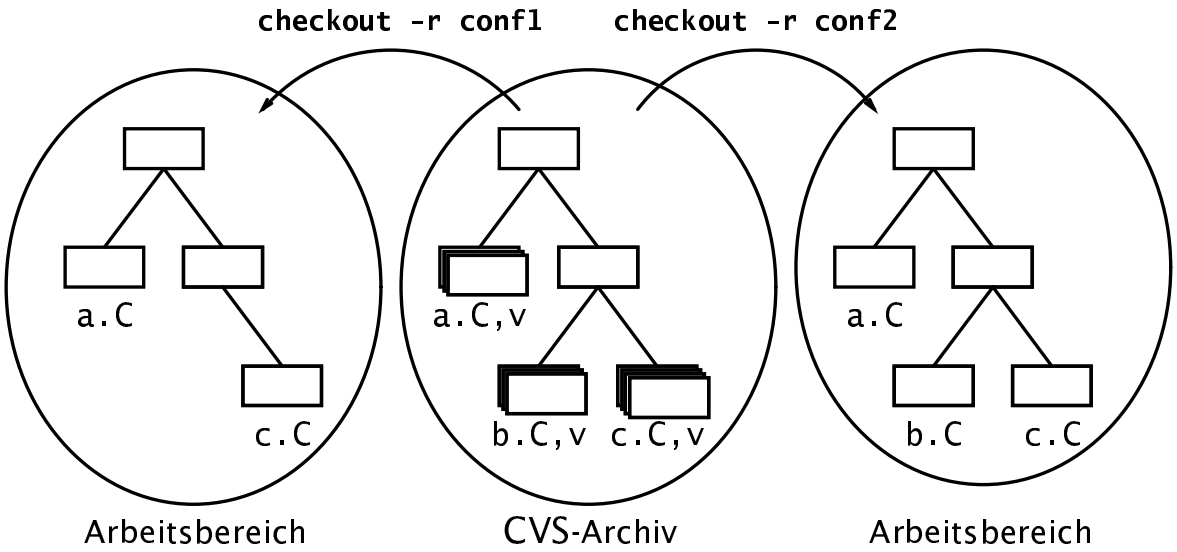






Projekt einrichten: Checkout

Mit einem *check out* wird die letzte Revision in einen *Arbeitsbereich* kopiert:



Projekt einrichten: Checkout (2)

Beispiel: Anke holt sich die aktuelle Fassung des Duden-Projekts:

```
anke$ cvs checkout duden
cvs checkout: Updating duden
U duden/Makefile
U duden/duden.h
U duden/grammatik.C
U duden/woerter.C
anke$ _
```

Mit der Option -D kann Anke beliebige frühere Zustände wiederherstellen:

```
anke$ cvs checkout -D "2001-11-29 14:30" duden
```







Änderungen propagieren: Commit

Anke hat die Datei duden.h geändert und möchte diese Änderung öffentlich machen (und *begründen*):

```
anke$ cvs commit duden.h
(Aufruf des Texteditors)
```

oder

```
anke$ cvs commit -m "Neu: NUMMERIERUNG" duden.h
Checking in duden.h;
/usr/share/CVS/duden/duden.h,v  <--  duden.h
new revision: 1.2; previous revision: 1.1
done
anke$ _
```

Noch bequemer: *Aufruf aus Programmierumgebung*
(z.B. `M-x cvs-update` in Emacs)







Änderungen propagieren: Commit (2)

Wird nichts angegeben, werden Änderungen im gesamten Verzeichnis propagiert:

```
anke$ cvs commit -m "Neue Deklination"
cvs commit: Examining .
cvs commit: Committing .
Checking in grammatik.C;
/usr/share/CVS/duden/grammatik.C,v <-- grammatik.C
new revision: 1.2; previous revision: 1.1
done
Checking in woerter.C;
/usr/share/CVS/duden/woerter.C,v <-- woerter.C
new revision: 1.4; previous revision: 1.3
done
anke$ _
```







Änderungen nachvollziehen

Mit `cvs log` kann man die Entstehungsgeschichte einer Datei verfolgen:

```
$ cvs log ddd.C
revision 1.632
date: 2001/07/31 16:10:05; author: zeller; state: Exp;lines: +3 -3
Fix: use XawInitializeWidgetSet() to setup Xaw converters;
     Xaw3d has no XawInitializeDefaultConverters().
----------------------------
revision 1.631
date: 2001/07/31 16:00:09; author: zeller; state: Exp;lines: +10 -0
Fix: don't have Xaw converters override our own
----------------------------
revision 1.630
date: 2001/07/30 22:18:27; author: zeller; state: Exp;lines: +18 -6
New: Options are automatically saved upon exiting DDD (can be turned off)
----------------------------
revision 1.629
date: 2001/07/30 21:10:59; author: zeller; state: Exp;lines: +12 -5
New: The Tip of the Day comes with an option to turn it off.
...
```

Änderungen nachvollziehen (2)

cvs diff zeigt die Unterschiede zwischen zwei Revisionen:

```
$ cvs diff -r1.631 -r1.632 ddd.C
RCS file: /cvsroot/ddd/ddd/ddd/ddd.C,v
retrieving revision 1.631
retrieving revision 1.632
diff -r1.631 -r1.632
2140,2142c2140,2142
<     // Register Xaw Converters.  This is done before installing our
<     // own converters.
<     XawInitializeDefaultConverters();
---
>     // Initialize Xaw widget set, registering the Xaw Converters.
>     // This is done before installing our own converters.
>     XawInitializeWidgetSet();
```







Konfigurationen benennen

Will Anke die aktuelle Konfiguration *benennen*, (z.B. weil sie an einen Kunden ausgeliefert wird), benutzt sie `tag`:

```
anke$ cvs tag duden1-1
cvs tag: Tagging .
T Makefile
T duden.h
T grammatik.C
T woerter.C
anke$ _
```

Die aktuelle Konfiguration kann nun jederzeit unter diesem Namen *rekonstruiert* werden:

```
anke$ cvs checkout -r duden1-1 duden
```







Konfigurationen benennen (2)

Alternativen zur Rekonstruktion:

- *Der Kunde soll gefälligst die neueste Software benutzen!*
- *Ich weiß auch nicht, warum es gestern noch lief...*







Versionen identifizieren

Bei jedem `commit` werden automatisch fortlaufende *Versionsnummern* vergeben, mit denen man später die Konfiguration wiederherstellen kann.

Diese Versionsnummern lassen sich auch in die Dokumente aufnehmen – und zwar mit Hilfe von *Schlüsselwörtern.*

In CVS sind Schlüsselwörter Bezeichner, die in „`$...$`“ eingeschlossen sind; sie werden beim `checkout` automatisch expandiert bzw. beim `update` auf den neuesten Stand gebracht.

Das Schlüsselwort `$Id$` expandiert z.B. zu einem Standard-Dateikopf:

```
$Id: cvsdriver.c,v 1.3 2001/11/21 18:38:08 zeller Exp $
```

Versionen identifizieren (2)

Die Schlüsselwörter lassen sich auch in den Programmtext übernehmen – z.B. als

```
static char version_string[] =
"$Id$";
```

Dieser Code wird beim nächsten checkout zu

```
static char version_string[] =
"$Id: cvsdriver.c,v 1.3 2001/11/21 18:38:08 zeller Exp $
```

expandiert.

Alternative zur Identifikation: *Raten!*







Dateien hinzufügen

Zu den Änderungen, die Entwickler vornehmen, gehören auch das *Hinzufügen* von neuen Dateien.

```
anke$ cvs add ortho.C
cvs add: scheduling file ‘ortho.C’ for addition
cvs add: use ’cvs commit’ to add this file
    permanently
anke$ _
```







Dateien hinzufügen (2)

Beim nächsten `commit` wird `ortho.C` ins CVS-Archiv aufgenommen.

```
anke$ cvs commit -m "Neu: ortho.C - Orthographie"
cvs commit: Examining .
cvs commit: Committing .
RCS file: /usr/share/CVS/duden/ortho.C,v
done
Checking in ortho.C;
/usr/share/CVS/duden/ortho.C,v  <--  ortho.C
initial revision: 1.1
done
anke$ _
```







Dateien löschen

Analog zu cvs add löscht cvs remove Dateien aus dem System:

```
anke$ rm ortho.C
anke$ cvs remove ortho.C
cvs remove: scheduling ‘ortho2.C’ for removal
cvs remove: use ’cvs commit’
to remove this file permanently
anke$ cvs commit -m "ortho.C geloescht"
cvs commit: Examining .
cvs commit: Committing .
Removing ortho.C;
/usr/share/CVS/duden/ortho.C,v  <--  ortho.C
new revision: delete; previous revision: 1.1
done
anke$ _
```







Arbeit beenden

Wird das Arbeitsverzeichnis nicht mehr benötigt, kann es mit `cvs release` gelöscht werden.

```
anke$ cd ..
anke$ cvs release -d duden
You have [0] altered files in this repository.
Are you sure you want to release
directory ‘duden’: yes
anke$ _
```

`cvs release` prüft dabei, ob auch tatsächlich keine offenen Änderungen ausstehen.







Paralleles Arbeiten

Problem: Entwickler arbeiten gleichzeitig auf derselben Datei (bzw. einer Kopie in ihrem Arbeitsbereich).

Wessen Änderungen werden übernommen und wie?

Werkzeuge zum Konfigurationsmanagement kennen zwei Mechanismen:

- Sperren (pessimistisch)
- Integration (optimistisch)







Sperren

Ist in CVS eine Datei (oder ein System) mit `cvs watch` markiert worden, unterhält CVS auf dieser Datei eine *Sperre*:

- Nach einem Checkout ist die Datei nur zum Lesen geöffnet
- Um sie zum Schreiben zu öffnen, muß der Benutzer explizit `cvs edit` eingeben:

  ```
  anke$ cvs edit duden.h
  ```

- Will Petra nun die Datei ebenfalls bearbeiten,

  ```
  petra$ cvs edit duden.h
  ```

  so werden Anke und sie automatisch benachrichtigt – so können sie sich über das weitere Vorgehen verständigen.







Sperren (2)

Andere Konfigurationssysteme als CVS sind da weit restriktiver:

In RCS (*revision control system*) etwa ist es nicht möglich, eine Datei zu bearbeiten, die bereits von einem anderen bearbeitet wird.

Solche restriktiven Sperren führen zu Problemen bei

- *Hot spots* – Teilen des Systems, die häufig bearbeitet werden
- *Langen Checkouts* – Sperren, die über lange Zeit bestehen







Integration

Die Alternative zu Sperren heißt *Integration:*

Bei jedem `commit` prüft CVS, ob die geänderten Dateien im CVS-Archiv unverändert sind:

- Wenn ja, werden die Änderungen übernommen
- Wenn nein, muß der Benutzer zuvor die neuen Änderungen in seinen Arbeitsbereich *integrieren*.







Integration (2)

`cvs update` überträgt Änderungen aus dem CVS-Archiv in das Arbeitsverzeichnis. Seien

- D die ursprüngliche Fassung der Datei,
- D'_1 die neue Fassung im CVS-Archiv und
- D'_2 die neue Fassung im Arbeitsbereich.

Folgende Fälle können beim `update` auftreten:

1. $D = D'_1 = D'_2$ $\Rightarrow$ Nichts geschieht.
2. $D = D'_1 \neq D'_2$ $\Rightarrow$ D'_2 wird ins CVS-Archiv übernommen.
3. $D = D'_2 \neq D'_1$ $\Rightarrow$ D'_1 wird in den Arbeitsbereich übernommen.
4. $D \neq D'_1 \neq D'_2$ $\Rightarrow$ Die Änderungen werden integriert.

Integration (3)

Ist eine Datei sowohl im CVS-Archiv als auch im Arbeitsbereich geändert worden, so müssen die Änderungen *in dieser Datei* integriert werden.

CVS geht dabei so vor:

- Sind die Änderungen (= Einfügen, Ändern, oder Löschen von Zeilen) mehr als 5 Zeilen voneinander entfernt, werden beide angewandt.
- Ansonsten tritt ein *Konflikt* auf: Beide Änderungen werden (durch Sonderzeichen gekennzeichnet) Teil der Datei.







Ein Konflikt

Beispiel: Anke und Petra haben beide dieselbe Zeile in `duden.h` geändert; Anke hat ihre Änderungen bereits wieder eingespielt.

Nach Petras `update` markiert CVS den Konflikt:

```
<<<<<<< duden.h
Ankes neuer Text
=======
Petras neuer Text
>>>>>>> 1.1.2.1
```

Vor dem nächsten `commit` muß Petra diesen Konflikt auflösen.

Alternative zur Koordination: *Händisches Umkopieren!*







Arbeitsweise mit CVS

1. `cvs checkout`, um eine Kopie des Projekts anzulegen
2. `cvs update`, um aktuelle Änderungen zu übertragen
3. `cvs commit`, um eigene Änderungen zu veröffentlichen
4. `cvs release`, um die eigene Kopie zu löschen.

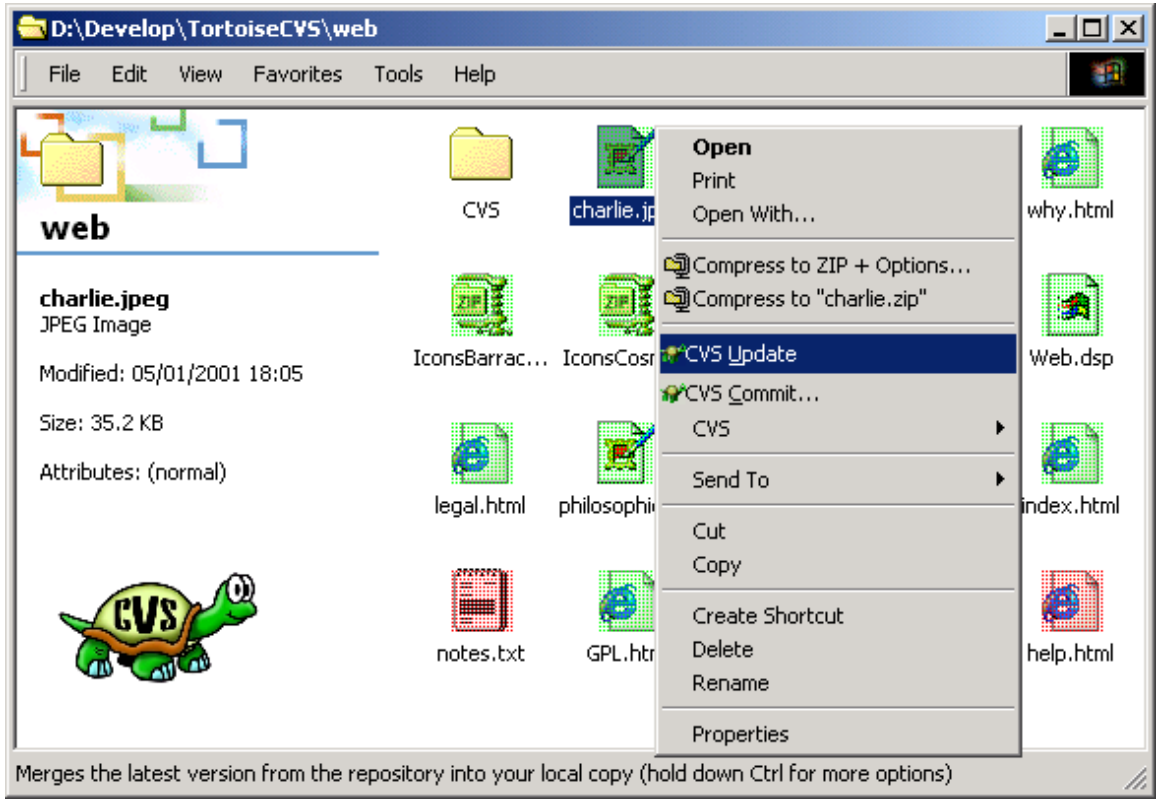
In der Praxis: update / commit / update / commit . . .



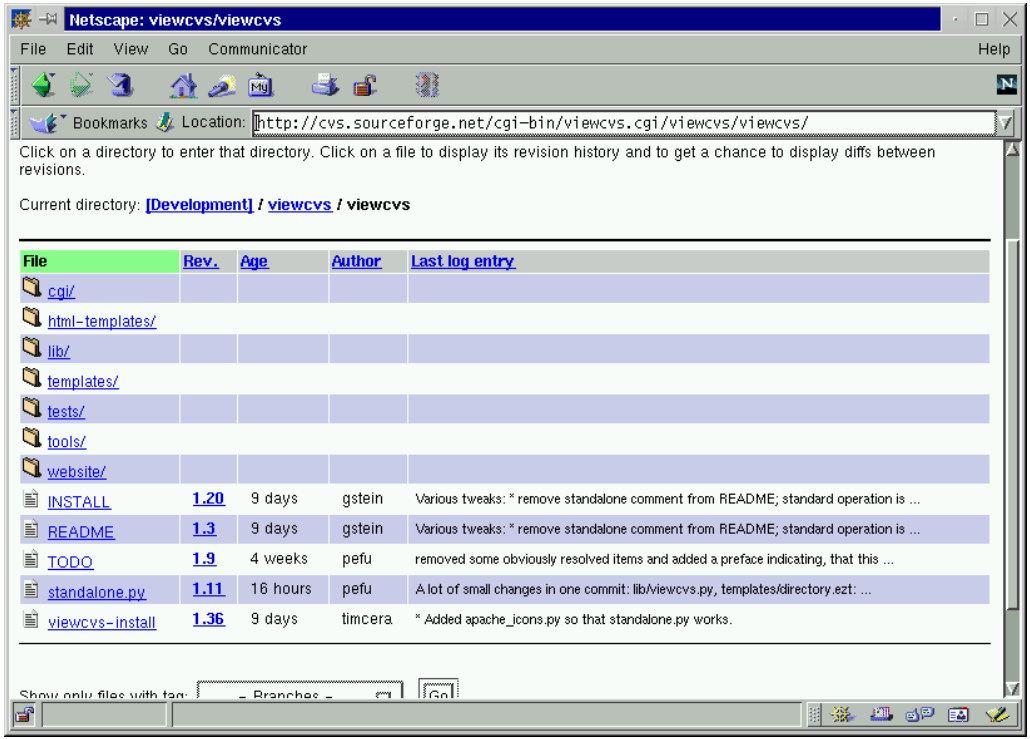




TortoiseCVS



ViewCVS


Netscape: viewcvs/viewcvs
File Edit View Go Communicator Help
Bookmarks Location: http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/viewcvs/viewcvs/
Click on a directory to enter that directory. Click on a file to display its revision history and to get a chance to display diffs between revisions.
Current directory: [Development] / viewcvs / viewcvs
File Rev. Age Author Last log entry
cgi/
html-templates/
lib/
templates/
tests/
tools/
website/
INSTALL 1.20 9 days gstein Various tweaks: * remove standalone comment from README; standard operation is ...
README 1.3 9 days gstein Various tweaks: * remove standalone comment from README; standard operation is ...
TODO 1.9 4 weeks pefu removed some obviously resolved items and added a preface indicating, that this ...
standalone.py 1.11 16 hours pefu A lot of small changes in one commit: lib/viewcvs.py, templates/directory.ezt: ...
viewcvs-install 1.36 9 days timcera * Added apache_icons.py so that standalone.py works.
Show only files with tag: - Branches - Go

Zusammenfassung CVS

- In das CVS-Archiv können Revisionen hinein- und wieder herauskopiert werden (*commit* bzw. *update*).
- Jede Änderung wird vom Entwickler *begründet*; das daraus gebildete *Änderungsprotokoll* dokumentiert die Entstehungsgeschichte einer Komponente.
- In einem CVS-Archiv werden nur die *Unterschiede* zwischen Revisionen gespeichert, was viel Platz spart.
- Das CVS-System ermöglicht die Versionierung von kompletten *Dateibäumen*, indem CVS auch das *Anlegen* und *Löschen* von Dateien berücksichtigt.
- CVS realisiert eine *optimistische Kooperationsstrategie*, die gleichzeitige Änderungen mehrerer Entwickler integriert.







Problem Tracking

Ein Anwender (oder ein Entwickler) hat ein Problem. Wie kann der Entwickler das Problem reproduzieren, um es zu beheben?

Lösung – *Alle relevanten Informationen* über das Problem erheben.







Was ist relevant?

Variante 1—Nicht genug Information:

```
From: user@inter.net
To: support@vendor.com

Your program crashed.  Just wanted to let you know.

X.
```







Was ist relevant? (2)

Variante 2—Zuviel Information:

```
From: another_user@inter.net
To: support@vendor.com

I experienced the following problem: [...]
With the enclosed dump, you can reproduce it easily.

Sincerely,

Y.
```

[Attachment: ISO image of user's hard disk, 80 GB]







Was ist relevant? (3)

Variante 3—Irrelevante Information:

```
From: yet_another_user@inter.net
To: support@vendor.com

Your program doesn't work properly.  It cannot
connect to the internet. Could this be because
I installed a new desktop background color
pattern? Or do I have to set up this modem thing?

Sincerely,

Z.
```







Was ist relevant? (4)

Lösung: Explizite *Richtlinien* aufsetzen, was erhoben wird.
Etwa:

- *Produkt-Version*
- *Einsatzumgebung* (z.B. Betriebssystem)
- *Problem-Geschichte*
- Beschreibung des *erwarteten* Verhaltens
- Beschreibung des *beobachteten* Verhaltens (typischerweise mit Verweis auf das Pflichtenheft)
- Wünschenswert: *Testfall*, der das Problem automatisch reproduziert

Diese Informationen enden in einem *Problembericht* (problem report, bug report)







Probleme verfolgen

Ist ein Problembericht erhoben, muss er gespeichert werden.

Mögliche Speicherorte:

Ein ‘PROBLEMS.txt’-Dokument. Einfach zu installieren, skaliert aber nicht.

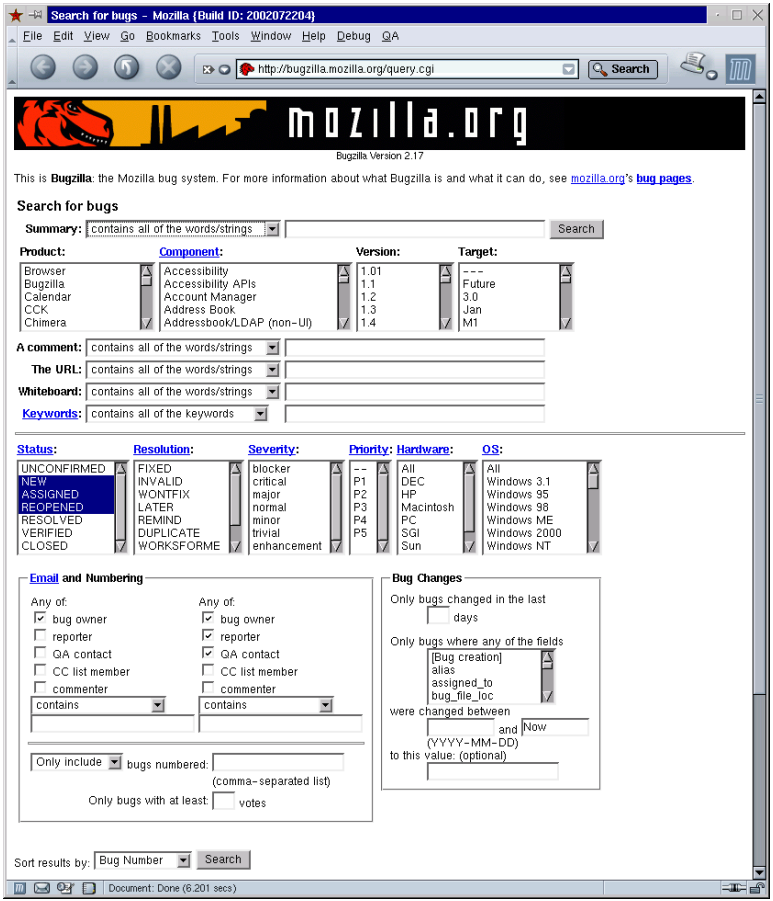
Eine *Problem-Datenbank*. Speichert alle Problemberichte.







Die Bugzilla Problem-Datenbank



Problem-Identifikation

Jedes Problem hat einen eindeutigen *Bezeichner* (auch bekannt als *PR number* oder *CR number* – von PR = problem report, CR = change request).

Entwickler können sich darauf beziehen in

- E-mails
- Änderungsmeldungen
- Status-Berichten







Schwere des Problems

Blocker Blocks development and/or testing work. This highest level of severity is also known as *Showstopper.*

Critical Crashes, loss of data, severe memory leak.

Major Major loss of function.

Normal This is the “standard” problem.

Minor Minor loss of function, or other problem where an easy workaround is present.

Trivial Cosmetic problem like misspelled words or misaligned text.

Enhancement Request for enhancement.

Problem-Status und Auflösung

FIXED This problem has been fixed and tested.

INVALID The problem described is not a problem.

WONTFIX The problem described is a problem which will never be fixed.

LATER The problem will be fixed in a later version.

REMIND Like LATER, but might still be fixed earlier.

DUPLICATE The problem is a duplicate of an existing problem.

WORKSFORME All attempts at reproducing this problem were futile. If more information appears later, the problem will be re-assigned.

Problembezogene Teamarbeit

Grundidee: *Das Team arbeitet so lange, bis das letzte Problem gelöst ist.*

Das Projekt beginnt mit dem Ursprungsproblem „Das Produkt ist nicht da.“; man teilt dieses Problem auf und weist Aufgaben zu.

Probleme können nicht nur im Produkt auftreten, sondern auch in allen Dokumenten – also von Beginn an.

Der Status *aller Probleme* wird über die Problem-Datenbank verwaltet (und ist jederzeit einsehbar).

Ist das letzte Problem gelöst, ist die Arbeit beendet :–)

Und nun: Viel Spass bei der Teamarbeit!