

Testdokumentation

Seminararbeit von Florian Ruh

im Fach „Softwaretechnik“

des Masterstudiengangs „Informatik/Mobile Systeme“
an der Hochschule Harz, Wernigerode

Sommersemester 2005

vorgelegt bei

Prof. Dr. A. Schmietendorf und Prof. Dr. F. Stolzenburg

Zusammenfassung

Systematische Softwaretests im Rahmen der Qualitätssicherung benötigen eine strukturierte Testdokumentation. Diese Dokumentation spezifiziert die korrekte Planung und das genaue Vorgehen beim Testen. Das Vorbereiten der Testumgebung und die Abnahmekriterien in einer Testvorschrift werden genauso festgehalten wie die Testprotokolle und die Fehler- und Abschlussberichte.

Heutzutage spielt das Konfigurationsmanagement eine wichtige Rolle – gerade bei der Versionierung des Quellcodes, der Fehlerverfolgung und des Änderungsprozesses. Hier hinein muss auch die Versionierung und Änderung der Projektdokumentation einfließen, wobei speziell die Testdokumentation Ursache für Änderungen an anderen Dokumentationen sein kann und

Diese Seminararbeit entsteht im Rahmen der Vorlesung „Software-technik“ (Vertiefung) im Masterstudiengang „Informatik/Mobile Systeme“ der Hochschule Harz, Wernigerode, im Juni 2005. Sie bietet den Studenten eine kurze Einführung in das Thema der Testdokumentation und des Konfigurationsmanagements mit einigen Beispieldokumenten aus den zitierten Quellen und Beispielapplikationen aus der „Open-Source“-Gemeinde.

Es seien hier besonders die Quellen von Balzert ([Bal98]) und von Schirmacher ([Sch01]) hervorgehoben, in denen mögliche Vorgehensweisen und Strukturierungen zur Testdokumentation ausführlich erläutert werden.

Inhaltsverzeichnis

1 Testdokumentation	4
1.1 Ziele	4
1.2 Struktur und Aufbau	4
1.2.1 Testplan	4
1.2.2 Testbericht	7
1.2.3 Abschlussbericht	7
1.2.4 Normen	9
1.2.5 Ablage der Dokumentation	10
2 Konfigurationsmanagement	11
2.1 Überblick	11
2.2 Versionen und Varianten	12
2.3 Beispiel „CVS“	14

1 Testdokumentation

1.1 Ziele

Testdokumentation hält nicht nur die Ergebnisse aus der Testphase der Softwareentwicklung fest, sondern regelt auch die Planung und das strukturierte Vorgehen *vor* dem eigentlichen Testen.

Neben den systematisch durchgeführten Tests tragen auch geeignete Dokumentationsregeln dazu bei, die Qualitätssicherung eines Softwareprojekts möglichst effizient zu gestalten. Der *Testprozess* per se besteht aus mindestens drei Schritten:

- Testplanung,
- Testdurchführung und
- Testkontrolle.

Nach der Empfehlung von Balzert in [Bal98] decken die Dokumentation für die ersten beiden Schritte

- der Testplan und die Testvorschrift und
- der Testbericht ab.

Der Abschlussbericht dokumentiert schließlich die Testkontrolle.

1.2 Struktur und Aufbau

1.2.1 Testplan

Der *Testplan* beschreibt alles Notwendige, damit die Vorbereitung und die Durchführung der Tests gesteuert und kontrolliert werden können (vgl. [Bal98], S. 547ff.). Als ein Teil des (übergeordneten) Projektplans enthält er die zeitliche und personelle Einplanung des Testens. Er legt fest, welcher Projektmitarbeiter zu welchem Zeitpunkt welche Testaktivitäten auszuführen hat.

Dabei kann auch angegeben werden, um welche(n) Testtyp(en) es sich handelt (z. B. Einzel-, Integrations-, System- oder Abnahmetest). Darüberhinaus enthält sie die Testvorschrift.

Die *Testvorschrift* legt die Art und den Zweck der Testdurchführung fest und enthält somit auch die ausgewählten Testfälle. Das Dokument berücksichtigt die Arbeiten, die nötig sind, die erforderliche Ausgangssituation mit bestimmten Vorarbeiten und bestimmten Ressourcen zu erreichen. Aus dieser Testumgebung heraus legt die Testvorschrift fest, wie die Testfälle auszuführen sind und in welcher Reihenfolge dies geschehen soll. Dazu kann sie die in ihr enthaltenen Testabschnitt ordnen.

Ein *Testabschnitt* gruppiert alle Testfälle, die mit einer gemeinsamen Testumgebung ausgeführt werden können. Eine Testumgebung bezeichnet hier das Umfeld und die Vorbedingungen, die für den Testfall eingerichtet bzw. erfüllt werden müssen. Die dazu notwendigen Daten wie Zweck und Referenz zur Spezifikation, die getesteten Software-Einheiten, die Vorbereitungsarbeiten werden durch die Aufräumarbeiten und die Testergebnisse aus diesem Abschnitt komplettiert. In einer sinnvollen Reihenfolge ordnet die *Testsequenz* die Fälle innerhalb eines Testabschnitts. Diese können im Rahmen eines Einsatzumfeldes zweckmäßig aufgereiht werden. Jeder Testfall kann dabei die Vorbedingungen für den nachfolgenden schaffen. Soll z. B. in einem Fall das Löschen eines Datensatzes überprüft werden, könnte im vorhergehenden Fall kontrolliert werden, ob dieser Datensatz gemäß den Softwareanforderungen korrekt angelegt wird.

Ein *Testfall* bezeichnet hierbei die unterste, atomare Testeinheit. Er enthält die (notwendigen) Eingabedaten und die Ausführungsanweisung sowie die zu erwartende Ausgabe (Soll-Ausgabe). Optional kann Platz für die Ist-Ausgabe und den Befund gelassen werden, sobald entschieden ist, die Testvorschrift zu einem Testprotokoll zu erweitern.

Es ist durchaus üblich, die Testvorschrift so zu gestalten, dass sie später als Testprotokoll genutzt werden kann. Dies bedeutet, dass die Testfälle einer Sequenz am besten tabellarisch angeordnet werden und es Möglichkeiten gibt, das Testergebnis und den Befund zu notieren sowie weitere relevante Angaben zu machen (z. B. Datum und Unterschrift).

Abb. 1 auf der nächsten Seite zeigt ein mögliches Inhaltsverzeichnis einer Testvorschrift.

1 Einleitung
1.1 Zweck des Tests
1.2 Testumfang
1.3 Referenzierte Unterlagen
2 Testumgebung
2.1 Überblick
2.2 Test-Software und -Hardware
2.3 Testdaten, Testdatenbanken
2.4 Personalbedarf
3 Abnahmekriterien
3.1 Kriterien für Erfolg und Abbruch
3.2 Kriterien für Unterbrechungen
3.3 Voraussetzungen für Wiederaufnahme
4 Testabschnitt 1
4.1 Einleitung
4.1.1 Zweck, Referenz zur Spezifikation
4.1.2 Getestete Software-Einheiten
4.1.3 Vorbereitungsarbeiten für den Testabschnitt
4.1.4 Aufräumarbeiten nach dem Testabschnitt
4.2 Testsequenz 1-1
4.2.1 Testfall 1-1-1
Eingabe, Anweisung, Soll-Ausgabe, Raum für Ist-Ausgabe und Befund
4.2.2 Testfall 1-1-2
4.3 Testsequenz 1-2
...
4.n Ergebnisse des Abschnitts 1
5 Testabschnitt 2
...

Abbildung 1: Inhaltsverzeichnis einer Testvorschrift

1.2.2 Testbericht

Der *Testbericht* entsteht während der Durchführung der Tests. Er enthält:

- die Testzusammenfassung,
- das Testprotokoll
- eine Liste aller Problemmeldungen und
- eine Liste der Software-Einheiten.

Ist die Testvorschrift nicht derart gestaltet, dass sie die Testergebnisse aufnehmen kann, wird das *Testprotokoll* ein eigenständiges Dokument.

Jeder Fehler, der innerhalb der Testdurchführung gefunden wurde, wird in der Liste der Problemmeldungen eingetragen. Heutzutage kann dieses Fehlermanagement auch ein sog. „Bug-Tracking“-System übernehmen. Bekanntestes Open-Source-Beispiel ist „Bugzilla“ von der Mozilla Organisation (s. <http://www.bugzilla.org/>).

Die Software-Einheiten, die vom Test betroffen sind, werden mit ihrer eindeutigen Kennzeichnung in der Liste der Software-Einheiten aufgeführt. Dies erleichtert es, Tests am gleichen Testobjekt zu wiederholen, falls in vorherigen Untersuchungen Fehler aufgetreten sind.

Ein Beispiel einer Testzusammenfassung ist in Abb. 2 auf der nächsten Seite gegeben.

1.2.3 Abschlussbericht

Die Zielgruppe des Abschlussberichts ist das Projektmanagement (vgl. [Sch01]). In diesem Dokument wird das Ergebnis der Testaktivitäten festgehalten. Es werden alle erfolgreichen Tests notiert, aber auch gelöste wo sie ungelöste Probleme schildert. Die Testaktivitäten werden nach ihrer „Gründlichkeit“ abgeschätzt und bzgl. ihrer Testkriterien bewertet. Es erfolgt eine Risikoabschätzung zu den Fehlern, die noch auftreten könnten, oder den Problemen, die noch ungelöst sind. Es wird ferner festgehalten, welche Ressourcen und in welchem Maße sie während des Testprozesses verbraucht wurden. Darüber hinaus wird spezifiziert, wer für die Genehmigung des Berichts verantwortlich ist. Für die einzelnen Personen wird Platz für deren Unterschrift bereitgestellt.

Testzusammenfassung		
Test-Nr.:	Arbeitspaket-Nr.:	
Testbeginn (Datum und Zeit):		
Testende (Datum und Zeit):		
Testdauer:		
Gegenstand und Zweck des Tests		
Projekt/Produkt:	Release-Nr.:	
Geliefert von:		
<input type="checkbox"/> Einzeltest	<input type="checkbox"/> Systemtest	
<input type="checkbox"/> Integrationstest	<input type="checkbox"/> Abnahmetest	
Testvorschrift		
Nummer/Version:	Titel:	
Empfehlung		
<input type="checkbox"/> akzeptieren (keine Wiederholung des Tests)	<input type="checkbox"/> wie es ist <input type="checkbox"/> nur kleine Fehler	
<input type="checkbox"/> nicht akzeptieren (Wiederholung des Tests)	<input type="checkbox"/> einige Funktionsfehler <input type="checkbox"/> einige fatale Fehler	
<input type="checkbox"/> Test nicht beendet		
Zusammenfassung		
Anlagen		
<input type="checkbox"/> Liste der getesteten Software-Einheiten		
<input type="checkbox"/> Liste der Problemmeldungen		
<input type="checkbox"/> andere:		
Testteam		
Name	Datum	Unterschrift
(Leiter)		

Abbildung 2: Beispiel einer Testzusammenfassung

1.2.4 Normen

Für die Testdokumentation gibt es mehrere ANSI¹/IEEE²-Normen. Diese auf Software für kritische Anwendungen zielenden Normen sind recht umfangreich und teilweise kompliziert. Folgende Normen beziehen sich auf Testdokumentation:

- ANSI/IEEE Std 829-1983
Software Test Documentation
- ANSI/IEEE Std 1008-1987
Standard for Software Unit Testing
- ANSI/IEEE Std 1012-1986
Standard Verification and Validation Plans

Bei kleineren bis mittleren Softwareprojekten kann man die normierten Dokumente zusammenfassen, wie es hier dargestellt ist. Für größere, evtl. internationale Projekte empfiehlt sich immer aufgrund des erhöhten Kommunikationsaufwands auf die Standards von ANSI/IEEE zurückzugreifen und die Dokumente strikt zu trennen. Den Zusammenhang zwischen den hier angeführten und den von den Normen geforderten Dokumenten verdeutlicht folgende Tabelle.

ANSI/IEEE-Dokument	Hier angeführtes Dokument
Test Plan	Testplan (abstrakte)
Test Design Specification	Testvorschrift (konkrete Planung)
Test Case Specification	Testvorschrift (bzgl. Testfälle)
Test Procedures Specification	Testvorschrift (bzgl. Durchführung)
Test Item Transmittal Report	Liste der Software-Einheiten
Test Log	Testprotokoll
Test Incident Report	Liste der Problemmeldungen
Test Summary Report	Testzusammenfassung

¹ANSI: *American National Standard Institute*, U.S.-amerik. Institut zur Normung in der Industrie

²IEEE: *Institute of Electrical and Electronics Engineers*, internat. Standardisierungsgremium

1.2.5 Ablage der Dokumentation

Die Dokumentation sollte generell zusammen mit den anderen Projektdaten an einem gemeinsamen Ort gespeichert werden. Hierbei empfiehlt es sich, auf entsprechende Konfigurationsmanagementsysteme (KMS) zurückzugreifen. Diese bieten u. a. eine Verwaltung sowohl von Quell- und Binärcode als auch von Projektdokumentation an.

In einigen Fällen jedoch sollte die Ablage der (allgemeinen) Projektdokumentation anderen geografisch entfernt arbeitenden Teams unabhängig von dem unternehmenseigenen KMS zugänglich sein. Dabei kommt dann ein unabhängiges Dokumentenmanagementsystem in Frage, welches Projekt- und gerade auch Testdokumentation im Sinne einer gemeinsamen Schnittstelle anbietet.

2 Konfigurationsmanagement

2.1 Überblick

Das Paradigma des *Konfigurationsmanagements* stammt ursprünglich aus der amerikanischen Raumfahrtindustrie der 50er Jahre des letzten Jahrhunderts. Raumfahrzeuge wurden damals während ihrer Entwicklung häufig modifiziert, ohne dass man diese Änderungen ausreichend (bisweilen auch überhaupt) dokumentierte. Üblicherweise wurden die Fahrzeuge während des Testens zerstört, so dass die Hersteller im Erfolgsfall keinen Nachbau anfertigen konnten, weil die Änderungen am Prototyp unwiderrufflich verloren und nicht nachvollziehbar waren. Diesen Informationsverlust sollte das Konfigurationsmanagement (KM) kompensieren. (vgl. [Bal98], S. 234ff.)

Auch in der heutigen Zeit spielt das KM eine große Rolle. Bei jedem Softwareprojekt treten in der Entwicklung Probleme in mehr oder weniger großem Umfang auf.

So erlebt das Projektteam bei häufigen Änderungen an der Software ein großes Durcheinander in der Fehlerbehebung. Bereits behobene Fehler treten wieder auf und es ist zudem nicht nachvollziehbar, wer wann warum welche Änderungen vorgenommen hat. Das Software-KM kann diese Unklarheiten mit Hilfe einer „History“, d. h. einer Aufzeichnung aller Modifikationsvorgänge und deren Metadaten (Datum, Verantwortlicher, Änderungsobjekt, Grund usw.), beseitigen. Das KM hilft darüberhinaus die Änderungswünsche mit den erfolgten Änderungen zu kombinieren und den Änderungsprozess zu überwachen. Zusätzlich kann es konsistente Entwicklungsstände und Konfigurationen herbeiführen. Dies geschieht durch automatische Auswahl der Version und Konfiguration und ist z. B. dann überaus nützlich, falls die letzten Änderungen fehlerhaft sind und zurückgezogen werden müssen.

Eine *Konfiguration* bezeichnet hierbei eine benannte und formal freigegebene Menge von Software-Elementen, die mit ihrer bestimmten Versionsnummer im Produktlebenszyklus in ihrer Wirkungsweise und ihren Schnittstellen aufeinander abgestimmt sind und somit eine gemeinsame Aufgabe erfüllen sollen ([Bal98], S. 234f) – dazu gehört übrigens auch die Dokumentation!

Software-Elemente können übrigens sowohl im Quellcode (Quellelement) als auch im Objektcode (abgeleitetes Element) vorliegen. Zu jeder Konfiguration gehört ein *Konfigurations-Identifikationsdokument* (KID), in welcher die Software-Elemente, die zu ihr gehören, eindeutig beschrieben werden.

Dieses Dokument nennt man auch „Konfigurationshierarchie“ oder „Elementstrukturplan“. Auch nicht auszuliefernde Elemente, z. B. Hilfsmittel und Werkzeuge zur Erstellung (Compiler), werden in der Regel aufgeführt und a. a. O. archiviert.

Für jede Änderung in der späteren Wartungsphase ergibt sich eine neue Konfiguration und somit ein neues KID. Dies erlaubt das Zurückgehen in eine konsistente frühere Konfiguration, falls die neue sich als fehlerhaft herausstellt.

Auch während dem eigentlichen Entwicklungsprozess werden Konfigurationen aus den Zwischenergebnissen erstellt. Diese Konfigurationen werden auch *baselines* genannt.

2.2 Versionen und Varianten

Eine Version kennzeichnet den Status eines Software-Elements. Mehrere Versionen spiegeln somit den zeitlichen Verlauf der Entwicklung an diesem einen Element wider. Durch sie kann der Änderungsprozess detailliert nachvollzogen werden. Die Versionsnummer besteht i. a. aus zwei Nummernblöcken, die mit einem Punkt getrennt sind. Die erste Nummer ist die *Release-Nummer*, die zweite die *Level-Nummer*, wobei es durchaus vorkommen kann, dass ein Projekt sogar vier oder noch mehr Blöcke erfordert. Dabei wird dann in der Wichtigkeit der Änderungen unterschieden. Je wichtiger, umfangreicher und bedeutender die Änderungen sind, desto eher wird eine vordere Versionsnummer erhöht. In Unternehmen ist es aber üblich, dies in interner Projektdokumentation zu den Vorgehensweisen bei der Versionierung festzulegen.

Als Beispiel sei eine Anwendung in der Version 1.0 aufgeführt, die sich in der Weiterentwicklung befindet. Es wurde ein Tippfehler im User Interface entdeckt und behoben. Die Levelnummer wird daraufhin um eins erhöht, da dies keine gravierende Änderung bedeutet. Die aktuelle Version ist also nun 1.1. Direkt danach erfolgt eine größere Änderung an der Implementation des Sicherheitskonzepts. Die Versionsnummer wird im Anschluss auf 2.0 gesetzt, da diese Änderung sehr umfangreich und gravierend ist und somit eine Erhöhung der Release-Nummer rechtfertigt.

Versionen werden mit verschiedenen Modellen verwaltet. Das verbreitetste ist das *Checkin/Checkout-Modell*. Dabei werden die Software-Elemente in Archiven (Repositorys) an zentraler Stelle gesammelt.

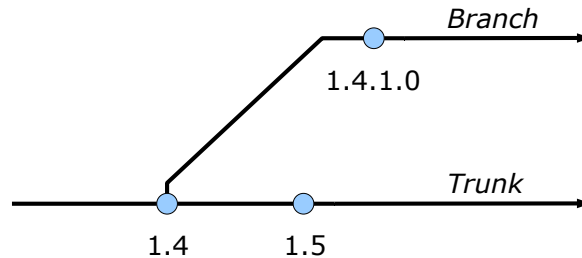


Abbildung 3: Versionen und Variante

Der Entwickler kann eine Kopie des Elements mit einer Checkout-Operation aus dem Archiv holen und damit arbeiten. Das System reserviert diese Kopie für den Ausbucher. Bei gleichzeitigem Zugriff auf dasselbe Element wird entweder eine Fehlermeldung ausgegeben oder ein neuer Entwicklungsast abgespalten. In jedem Fall sorgt dieser Mechanismus dafür, dass Änderungen sich nicht gegenseitig überschreiben und nicht unbeabsichtigt parallel entwickelt wird.

Die Checkin-Operation fügt dem Repository die neue Version hinzu und notiert die dazugehörigen Metadaten. Ist ein Element erst einmal eingebucht, kann es nicht mehr geändert werden. Jede Überarbeitung mündet nach einem Checkout/Checkin-Zyklus in einem neuen Dokument.

Die sog. Deltatechnik, die zumeist bei Checkin/Checkout-Modellen angewandt wird, speichert statt kompletter Kopien nur die Unterschiede (Deltas) einer Version zu ihrem zeitlichen Vorgänger. Die Technik bleibt vom Benutzer jedoch unbemerkt.

Diese Versionssequenzen können durch *Varianten* erweitert werden. Grundsätzlich werden als Varianten die Ausprägungen paralleler Entwicklungslinien verstanden. Es gibt aber auch weitere Definitionen, die eine einheitliche, umfassende leider nicht zulassen. U. a. können Variationen auch auf unterschiedliche Hardware- und Systemsoftware-Konstellationen zugeschnitten sein oder durch verschiedene Implementierungen bei gleichbleibender Funktionalität erzwungen werden.

Abb. 3 zeigt ein Beispiel für einen Versionsstamm (Trunk), eine Variante in einem neuen, parallellaufenden Zweig (Branch) und die unterschiedlichen

Versionsnummern. Hier wurde von der Version 1.4 aus eine Variante mit der Nummer 1.4.1.0 entwickelt. Dabei sind die letzten beiden Nummernblöcke wie eigenständige Versionsnummern zu behandeln, d. h. die Variantenummer setzt sich aus Release- und Level-Nummer sowie *Branch- und Sequence-Nummer* zusammen. Die Variante 1.4.1.0 kann zeitlich nach der Version 1.5 erstellt worden sein.

Diese Art der Bezeichnung von Versionen gilt natürlich nicht nur für Quell- und Objektcode sondern natürlich auch für Testdokumentation.

2.3 Beispiel „CVS“

Das Concurrent Versions System (CVS) ist eine Open-Source-Anwendung und sei hier als Beispiel für ein einfaches KMS genannt (s. [Col03]). Entstanden ist es 1986, als ein Benutzer seine UNIX-Shell-Skripte in einer Internet-Newsgroup veröffentlichte. Drei Jahre später nahmen sich dann andere UNIX-Programmierer den Quellen an und entwickeln seitdem das Projekt ständig weiter.

CVS beherrscht die Versionierung von ASCII- und Binärdateien. Dabei können die Dateien einzelne Versionsnummern bekommen, es können Varianten in einem gesonderten Entwicklungszweig behandelt werden und es erlaubt die Benutzung von sog. Tags zur Markierung und Definition von einzelnen Konfigurationen.

CVS wird zumeist im Multi-User-Betrieb angewandt. Dabei greifen mehrere Clients (teils zugleich) auf ein gemeinsames Repository, dem Projekt-Heimatverzeichnis auf dem CVS-Server zu. Der Server regelt dann per Zugriffskontrolle die Benutzung des Repositories und überwacht evtl. gleichzeitig auftretende Änderungen am Quelltext bzw. Checkins neuer Revisionen einer Datei.

Es gibt mehrere Möglichkeiten CVS zu benutzen. Die trivialste ist die über den Konsolenzugriff auf den Server. Mittlerweile gibt es aber schon Front-Ends für den entfernten Zugriff auf den CVS-Server. So ist z. B. WinCVS eine GUI für Windows-Clients (vgl. Abb. 4 auf der nächsten Seite). WinCVS erlaubt das komfortable Verwalten von Repositories und den dort enthaltenen Dokumenten.

Natürlich gibt es in größeren Unternehmen nicht nur kommerzielle Produkte

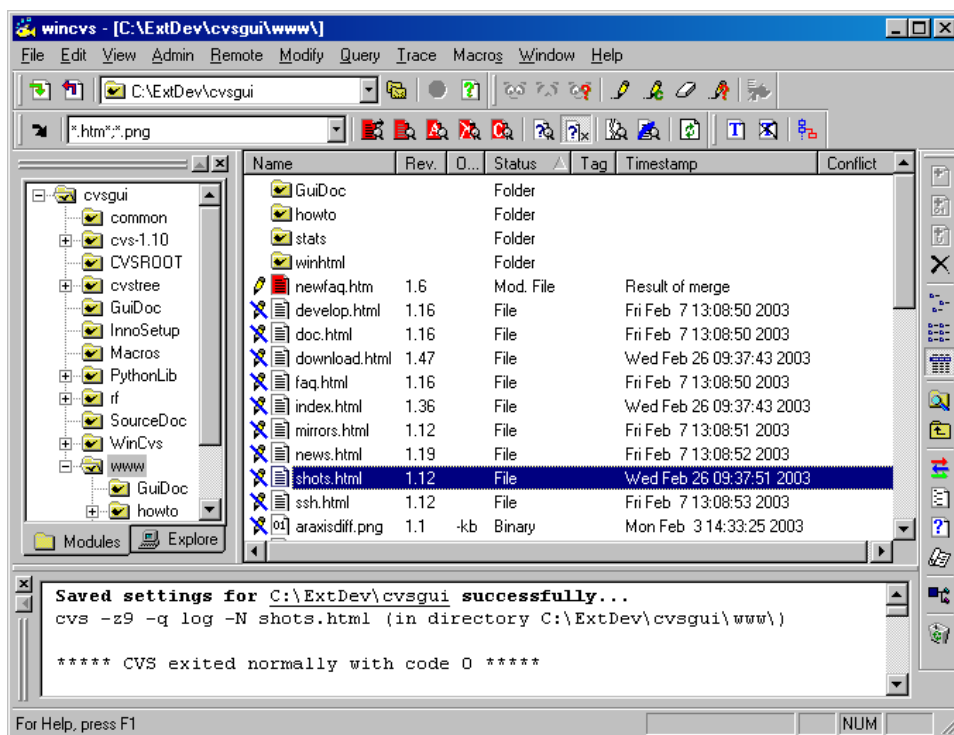


Abbildung 4: WinCVS-Screenshot, s. <http://www.wincvs.org/>

wie Rational ClearCase³ sondern auch maßgeschneiderte Softwareapplikationen, die die Aufgaben des KM, gerade im Bereich der generellen Projektdokumentation, übernehmen. Einige beherrschen zudem automatisierte Testprozesse und -dokumentationserstellung.

³s. <http://www-306.ibm.com/software/awdtools/clearcase/index.html>

Literatur

- [Bal98] BALZERT, HELMUT: *Lehrbuch der Software-Technik*, Band 2. Heidelberg, Berlin: Spektrum, Akademischer Verlag, 1998.
- [Col03] COLLABNET, INC.: *CVS: Concurrent Versions System*. URI: <http://www.cvshome.org/>, 2003.
- [Sch01] SCHIRMACHER, ANDREAS: *Testdokumentation nach ANSI/IEEE 829*. URI: <http://www.softwaretesting.de/article/print/1/-1/7/>, 2001.