Integration von Konfigurationsmanagement Funktionen in Softwareentwicklungsprozesse

Axel Mahler und *Andreas Lampen*
Technische Universität Berlin

Integration von Konfigurationsmanagement Funktionen in Softwareentwicklungsprozesse

Überblick

Die Kontrolle von Änderungsprozessen ist eines der fundamentalen Probleme bei der Organisation industrieller Softwareentwicklung: Anforderungen ändern sich genauso wie Designs, Spezifikationen, Quellcode von Programmen, begleitende Dokumentation etc. Unzureichende Kontrolle über Änderungsprozesse, ihre Auslöser und Realisation hat schwerwiegende Auswirkungen auf den Verlauf von Entwicklungsprojekten. Die Techniken und Werkzeuge des Softwarekonfigurationsmanagements bieten das nötige Instrumentarium um mit der Komplexität stetiger Änderungsnotwendigkeit in Softwareprojekten fertig zu werden. Von großer Wichtigkeit ist dabei die sinnvolle Einbettung der einzelnen Funktionen in den konkreten Arbeitsprozeß der Softwareentwicklung, damit sie nicht als Fremdkörper empfunden (und nachlässig genutzt) werden.

Das Konzept der Entwicklungstransaktion ist das grundlegende Hilfsmittel zur Strukturierung allgemeiner Änderungsprozesse. Es dient als Ansatzpunkt zur harmonischen Einbettung von Kontrollfunktionen in laufende Entwicklungsprozesse. Verschiedene, weit verbreitete Basistechniken des Softwarekonfigurationsmanagements werden dargstellt und in Bezug zum Konzept der Entwicklungstransaktion gesetzt. Problematisch ist bei vielen Werkzeugsystemen die Diskrepanz zwischen funktionaler Bandbreite einerseits und problemloser Anpaßbarkeit der Funktionen an die Bedürfnisse gewachsener Prozessablaufstrukturen in den Entwicklerteams andererseits.

Einen neuartigen Ansatz zur Auflösung dieses Widerspruchs bieten Werkzeugkästen mit sorgfältig aufeinander abgestimmten Einzelfunktionen, die sich mit wenig Aufwand in vorhandene Entwicklungsinfrastrukturen einbetten lassen und die Realisierung maßgeschneiderter Änderungskontrollprozeße gestatten. *ShapeTools* ist ein solcher Werkzeugkasten. Die Funktionen des Systems können sowohl als bereits integrierte Komplettlösung verwendet werden, oder aber exakt in den spezialisierten Prozeßablauf vieler Softwareentwicklungsumgebungen eingepaßt werden.

1. Hintergrund: Die Rolle des Konfigurationsmanagements im Softwareentwicklungsprozeß

Die Produkt- und Organisationsstruktur jedes nennenswerten Softwareentwicklungsprojekts stellt einen erheblichen investiven Wert für die Entwicklungsorganisation – z.B. ein Softwarehaus – dar. Es sollte selbstverständlich sein, daß diese Investition angemessen gegen destruktive Einwirkungen geschützt wird. Während mehr oder weniger alle Entwicklungsprojekte Vorkehrungen treffen, um sich z.B. vor Datenverlusten zu schützen, wird das Risiko von *unkontrollierten Änderungen* an Softwaresystemen weitgehend unterschätzt. Immer wieder gemachte Erfahrungen belegen, daß das sichere Wissen um den jeweils aktuellen Entwicklungs– bzw. Änderungszustand eines Softwareprojekts zu seinen wichtigsten Integritätsmerkmalen gehört [3,12,4]. Im Unterschied zu herkömmlichen Entwicklungsprojekten, z.B. im Maschinenbau, stellt sich hier ein besonderes Problem, da Software nicht stofflich sichtbar, dafür aber um so leichter änderbar ist.

Der Verlust oder das Nichtvorhandensein von zuverläßigen Informationen über den aktuellen Entwicklungszustand eines Softwaresystems und den Status der Entwicklungsaktivitäten behindert einen methodischen Softwareentwicklungsprozeß und führt nahezu sicher zu einem Qualitätsverlust des in Entwicklung befindlichen Produkts. Häufige Folge sind Planungsunsicherheit, andauernde Integrationsprobleme und zum Teil erhebliche Entwicklungsverzögerungen. Tichy illustrierte treffend eine ganze Reihe typischer Probleme, die durch mangelhafte Kontrolle des Änderungsprozesses entstehen können [28]. Die Techniken des Softwarekonfigurationsmanagements (SCM, engl. Software Configuration Management) dienen u.a.

dazu, die entwicklungsprozeßpezifischen Informationen systematisch zu erfassen und aktuell zu halten.

Es gehört zu den Tücken des Softwareentwicklungsprozesses, daß das Bewußtsein für die Wichtigkeit eines effektiven Softwarekonfigurationsmanagements als Kontroll- und Steuerungsinstrument in Entwicklungsprojekten erst mit der Zeit heranreift. In den frühen Phasen der Softwareentwicklung wird striktes Konfigurationsmanagement häufig als bürokratisch und insgesamt hinderlich empfunden und entsprechend vernachlässigt.

Auch wenn Konfigurationsmanagement schon in den ersten Entwicklungsphasen eines Softwareprojekts ein sehr wirkungsvolles Hilfsmittel ist (z.B. durch die Automatisierung einer Reihe von fehleranfälligen, schematischen Tätigkeiten), so zeigt sich seine wirkliche Bedeutung erst in der sog. Wartungsphase. Es ist hier, wo viele Softwareprojekte nach evtl. erfolgreicher Entwicklungsphase in die roten Zahlen geraten. Es wurde immer wieder beobachtet, daß die Wartung eines operationalen Softwaresystems mit weitem Abstand der größte einzelne Kostenfaktor ist. Sommerville berichtet, daß die Wartungskosten typischer Softwaresysteme zwischen zwei und viermal höher liegen als die Entwicklungskosten [25]. Pressman schätzt den Anteil der Wartung auf 70 - 80 % der Gesamtkosten eines Softwareprojekts [22]. Auch wenn es keine detaillierteren veröffentlichten Daten zu dieser Problematik gibt, darf davon ausgegangen werden, daß Konfigurationsmanagementprobleme (direkt oder indirekt; indirekt: ein unstrukturiert geändertes, „zurechtgeflicktes'' System unterliegt einer sehr viel stärkeren Entropie; die Lebenszeit des Systems wird verkürzt – Lehman's Law [17]) einen gewichtigen Anteil dieser Kosten verursachen.

Ein klares Bewußtsein für die Bedeutung des Softwarekonfigurationsmanagements und der frühzeitige Aufbau einer Projektunterstützungsumgebung, die konsequent SCM einsetzt sind wesentliche Voraussetzungen dafür, die Kosten in den späteren Phasen des Entwicklungsprojekts unter Kontrolle zu halten. Es ist ein positives Zeichen, daß sich das Verständnis für die Schlüsselrolle des Konfigurationsmanagements bei der Qualitätssicherung von Software zunehmend verbreitet und sich beispielsweise in Normen wie ISO-9000 niederschlägt.

Ein gut konzipiertes SCM System hilft aber nicht nur das entwickelte Produkt sichtbarer und damit kontrollierbar(er) zu machen, sondern kann den Grundstein einer leistungsfähigen Infrastruktur für kooperatives Arbeiten innerhalb des Projekts darstellen. SCM Systeme haben das Potential die isolierten Aktivitäten der verschiedenen Entwickler sinnvoll zu koordinieren. Darüberhinaus ist es die geradezu natürliche Schnittstelle zwischen Projektmanagement und der eigentlichen Produktentwicklung. Die Entwicklungsdatenbasis – elementarer Teil jedes SCM Systems – ist die Schlüsselkomponente zur Erfassung und Auswertung von Daten zur quantitativen Prozeßkontrolle. Zusammengenommen kann ein SCM System ein großartiger Kristallisationskern für eine integrierte Projektunterstützungsumgebung sein, die einen strukturierten und geplanten Entwicklungsprozeß erst ermöglicht. Es ist kein Zufall, daß einige der erfolgreichsten Softwareentwicklungsumgebungen gewissermaßen um das Konfigurationsmanagementsystem herum aufgebaut wurden.

2. Konzepte: Konfigurationsmanagement in der kommerziellen Softwareentwicklung

Zu den Grundprinzipien des SCM zählen die Identifizierung von Komponenten, Konfigurationen und definierten Referenzlinien eines Softwaresystems (Identification), die Kontrolle und Verfolgung jeglicher Veränderung an identifizierten Einheiten (Change- and Configuration Control), die systematische Festlegung und Sicherung definierter Referenzlinien (Baselining and Configuration Audit), sowie die dynamische Zustandsverfolgung und Berichterstattung über aktuelle Änderungsprozesse (Configuration Status Accounting) [13,28]. Obwohl diese Liste den Eindruck erwecken kann, als legte SCM eine nicht unerhebliche bürokratische Last auf Softwarentwicklungsprojekte, so ist in der Praxis die Einhaltung relativ weniger grundlegender Vorgehensweisen ausreichend, um die genannten Prinzipien zumindest im Ansatz umzusetzen. Es gibt inzwischen zahlreiche gute Softwarewerkzeuge, die einen Großteil der fehleranfälligen Routinearbeit automatisieren.

2.1. Die Basis – das Projektrepositorium

Das elementarste Hilfsmittel zur Unterstützung der hier zu beschreibenden Vorgehensweisen ist die Projektbibliothek (heutzutage *Repositorium* genannt), in der alle Komponenten, deren Versionen sowie ihre Beziehungen untereinander aufbewahrt werden. Funktionales Hauptmerkmal der Projektbibliothek ist ein genau geregeltes Zugriffsverfahren auf ihren Inhalt, das sowohl die Integrität der verwalteten Elemente, als auch die strukturelle Integrität der Bibliothek (also des entwickelten Produkts) als ganzes sicherstellt.

Formalisierte Zugriffsregelungen für die Elemente der Bibliothek unterstützen einen Entwicklungsstil, bei dem jeder Schritt von einem konkreten, explizit formulierten Vorhaben geleitet ist. Jedes einzelne Änderungsvorhaben mündet in einer entsprechenden Version der betroffenen Komponenten und des Gesamtsystems. Der Entwicklungsprozeß stellt sich demnach als eine Folge von *Entwicklungstransaktionen* dar, die jeweils separat verwaltet werden.

Da je nach Ausgestaltung der Transaktionen auf der Projektbibliothek eine Reihe von prozeßspezifischen Daten anfallen und erfaßt werden (z.B. Größe und Dauer bestimmter Änderungen, Häufigkeit von Fehlern, Änderungsanfälligkeit von Komponenten etc.), ist hier ein idealer Ansatzpunkt zur Einführung quantitativer Qualitätssicherungsmethoden (z.B. Kategorisierung von Fehlern, statistische Auswertungen etc.). Darüberhinaus ist das Repositorium bei Vorhandensein einer vernünftigen Reporterzeugung eine zuverlässige Informationsquelle für das Projektmanagement.

2.2. Entwicklungstransaktionen – Ein Modell für kooperative Softwareentwicklung

Die meisten in der Praxis eingesetzten Konfigurationsmanagementsysteme basieren auf dem Konzept der Entwicklungstransaktion. Das gemeinsame Grundmuster eines kontrollierten Entwicklungsschritts sieht wiefolgt aus:

- Änderungsinitiative (Änderungsauftrag, Idee, Fehlermeldung etc.)
- Analyse des Vorhabens (u.a. welche Komponenten sind betroffen, Planung)
- Reservierung der zu ändernden Konfiguration (*checkout*)
- Durchführung der Änderung (*{think−modify−build−test}**)
- Registrierung der neuen Systemversion (in der Regel neue Versionen von mehreren Komponenten; vorher oft Regressionstest und Qualitätsprüfung)

Die konzeptionellen Unterschiede zwischen Konfigurationsmanagementsystemen liegen vor allem in

- dem Grad der Automatisierung bzw. der Unterstützung des beschriebenen Verfahrens,
- den Politiken zur Koordinierung *nebenläufiger* Änderungsprozesse sowie
- den höheren Organisationsfunktionen der Entwicklungsumgebung, die auf dem Grundmuster basieren, z.B. Anbindung an das Projektmanagement (Arbeitsverteilung, Planung, Monitoring etc.).

In den folgenden Betrachtungen soll zwischen dem grundlegenden konzeptuellen Vorgehensmodell einerseits und dessen Realisierung bzw. automatisierter Unterstützung durch konkrete Werkzeuge und Umgebungen andererseits unterschieden werden.

Entwicklungstransaktionen auf Softwarekonfigurationen (auch *Long Transactions* oder *Design Transactions* genannt) sind nicht mit dem Transaktionsbegriff aus der Datenbankenwelt zu verwechseln. Sie beziehen sich nicht auf „kleine'' Datenelemente in einer Datenbank, sondern auf definierte Systemkonfigurationen, die zum Ausgangspunkt eines Entwicklungsschritts gemacht werden. Solche Transaktionen können Stunden und Tage dauern und sich über viele Arbeitssitzungen des ausführenden Entwicklers erstrecken. Entwicklungstransaktionen beziehen sich jeweils auf einen Ausschnitt des Projektrepositoriums, einer konzeptuellen Datenverwaltung für die Projektdaten, die in der Regel nicht als Datenbank im eigentlichen Sinne realisert ist (*non standard data base*).

Der Begriff Transaktion legt bereits nahe, daß bei jedem Entwicklungsschritt von einem gesicherten, definierten Grundzustand ausgegangen wird (Definition der Referenzlinie für die Entwicklungstransaktion), der lokal – d.h. im Arbeitsbereich eines Entwicklers – bearbeitet/geändert wird. Solchermaßen lokal gemachte Änderungen bleiben für das übrige Projekt unsichtbar, bis die Transaktion durch ein formales *commit* abgeschlossen wird, d.h. eine neue Version der Systemkonfiguration definiert ist.

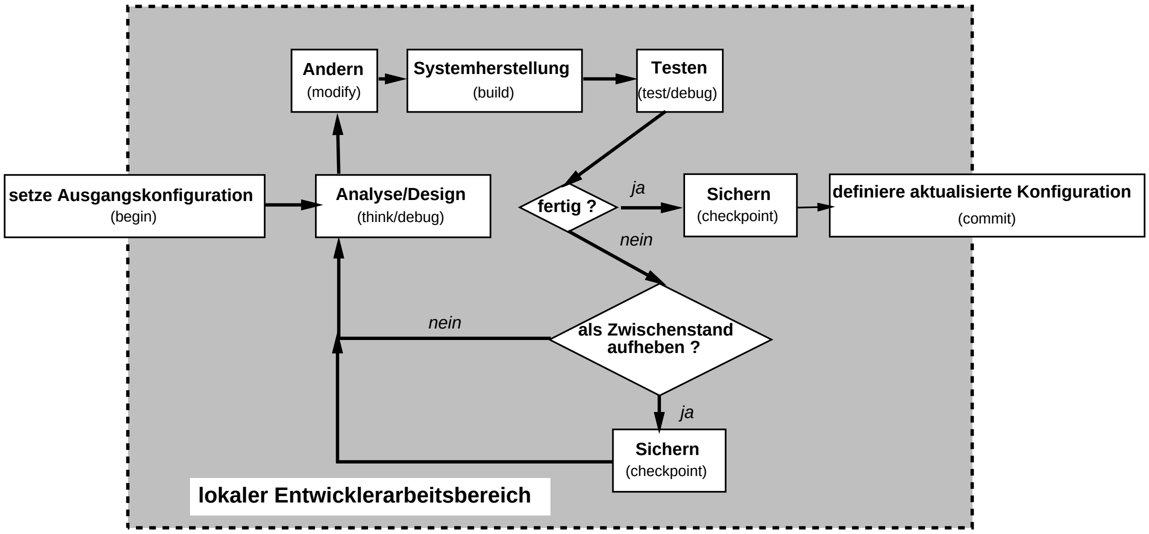

Abb. 2.1. Ablauf einer Entwicklungstransaktion

Die einzelnen Komponenten einer Konfiguration werden für eine bestimmte Änderung jeweils von einem Entwickler bearbeitet. Bei der Arbeit entstehen möglicherweise mehrere lokale Zwischenversionen (*checkpoints*) zum Zwecke eines möglichen späteren Wiederaufsetzens nach fehlgeschlagenen Änderungsversuchen. Meint der Entwickler, einen Änderungsauftrag abgeschlossen zu haben, so publiziert er sein Arbeitsergebnis. Die Beendigung der Entwicklungstransaktion führt zur Definition einer neuen Konfiguration, die sich von der Ausgangskonfiguration dadurch Unterscheidet, daß ein weiterer Änderungsauftrag (z.B. eine Fehlerkorrektur oder eine Weiterentwicklung) ausgeführt wurde. Wenn Buch darüber geführt wird, welche Änderungen genau in welchen Systemversionen realisiert sind und sich über Rückverzeigerungen feststellen läßt, warum bestimmte Passagen in einer Version eines Entwicklungsdokuments so und nicht anders aussehen, ist ein wichtiges Element effektiven Konfigurationsmanagements realisiert.

2.3. Technische Ansätze für Entwicklungstransaktionen

Die Umsetzung des Konzepts der Entwicklungstransaktionen durch Werkzeugsysteme ist sehr unterschiedlich. Die Praxis ist stark vom Einsatz relativ eingeschränkter Werkzeuge geprägt, die die Implementierung von Entwicklungstransaktionen an Softwaresystemen nicht oder nur mit größerem Aufwand unterstützen. Viele Funktionen, die heute gut automatisierbar sind, müssen so manuell ausgeführt werden, was zu Fehlern, geringerer Produktivität und Kontrollverlusten über den Entwicklungsprozeß führt. Allerdings gibt es in der letzten Zeit eine deutlich erkennbare Tendenz, das Softwarekonfigurationsmanagement systematischer als bisher zu unterstützen. Dies schlägt sich auch im Aufkommen einiger sehr leistungsfähiger Konfigurationsmanagementsysteme auf dem Markt nieder.

Auf technischer Ebene gibt es verschiedene Ansätze, wie Entwicklungstransaktionen realisiert werden. Bei der Darstellung der gängigsten Techniken beziehen wir uns auf die Übersichten von Feiler[10] und Dart[8].

2.3.1. Komponenten Ein-/Ausgangskontrolle

Die verbreitetste und einfachste Technik zur Kontrolle der Projektbibliothek ist die Komponentenein– und –ausgangskontrolle (*checkin/checkout*). In der ursprünglichen Form werden hierbei die Komponenten eines Systems isoliert voneinander verwaltet. Das eigentliche Konfigurationsmanagement findet in den Köpfen der Entwickler statt, die sich an die verschiedensten, informell vereinbarten Benutzungskonventionen für die Projektbibliothek halten müssen.

Das Projektrepositorium besteht aus einer Menge von komponentenspezifischen Archiven, in denen die verschiedenen Änderungszustände der Komponenten (Schnappschüsse) festgehalten werden. Bevor ein Entwickler eine Änderung an einer bestimmten Komponente vornehmen kann, muß er diese ßen ausleihen (checkout). Das Repositorium stellt dabei sicher, daß dieselbe Komponente nicht bereits von einem anderen Entwickler zum Zwecke der Änderung ausgeliehen wurde. Bereits gesicherte Versionen bleiben weiterhin für die anderen Mitarbeiter im Projekt lesbar.

Wenn ein Entwickler eine Version zum Anbringen einer Änderung ausleiht, so wird ihm eine änderbare Kopie dieser Version in seinem lokalen Arbeitsbereich zur Verfügung gestellt. Nachdem die geplante Änderung vom Entwickler als abgeschlossen angesehen wird, registriert er diese als neue Version in der Projektbibliothek. Normalerweise wird dabei die ausgeliehene Komponente an die Bibliothek zurückgegeben (d.h. die änderbare Kopie der Komponente wird gelöscht), um sie für weitere Änderungen – evtl. durch andere Entwickler – zugänglich zu machen. Bei der Registrierung einer neuen Version werden Grund und Art der vorgenommenen Änderung in Form eines kurzen Logbucheintrags dokumentiert.

Konflikte

Will ein Entwickler eine Komponente verändern, die bereits von einem anderen Entwickler ausgeliehen wurde, so gibt es zwei Möglichkeiten, wie das Problem aufgelöst werden kann:

1) es wird gewartet, bis die entsprechende Komponente wieder verfügbar ist (strikte Serialisierung von Änderungen, konservatives Kooperationsmodell) oder
2) die Änderungen werden parallel durchgeführt und unabhängig nacheinander gespeichert (s. Abb.2.2), bzw. es wird eine Verzweigung (*branch*) in der Versionenfolge im Archiv erzeugt (s. Abb.2.3; Nebenläufigkeit von Änderungen, optimistisches Kooperationsmodell).

Bei nebenläufigen Änderungsprozessen ist die Verzweigung im allgemeinen der sinnvollere Ansatz, da vermieden wird, daß die zuletzt registrierte Änderung alle anderen Änderungen überdeckt (Abb.2.2). Um die Überdeckung aufzulösen müssen die als aufeinanderfolgende Versionen repräsentierten Änderungen kombiniert werden (*join*).

Verzweigungen sind ein geeignetes Hilfsmittel um zeitlich überlappende Änderungen an derselben Komponente getrennt zu verwalten. Um die verschiedenen Änderungen schließlich in einer gemeinsamen Version zu akkumulieren, müssen die Verzweigungen wieder zusammengeführt und in einer weiteren Version verschmolzen (*merge*) werden. Ohne die Unterstützung durch weitere Mechanismen ist es mit diesen Techniken allerdings recht schwierig den Überblick darüber zu behalten, welche Verzweigungen im Versionenbaum zu welchen logischen Änderungen gehören, welche Verschmelzungen noch anstehen bzw. welche Zweige „abgestorben'' sind.

2.3.2. Systemkomposition

Bei den Versionsmanagementsystemen der „ersten Generation'' ist es ein bezeichnendes Merkmal, daß die Entwicklungsgeschichte der einzelnen Komponenten jeweils isoliert verwaltet wird. Die Kompositionsstruktur des gesamten Softwaresystems wird von einem völlig unabhängigen Werkzeug kontrolliert. Prototypische Vertreter dieser SCM Werkzeuge sind die von UNIX her bekannten Programme SCCS [23], RCS [27] und Make [11]. Die starre Trennung der Verwaltung

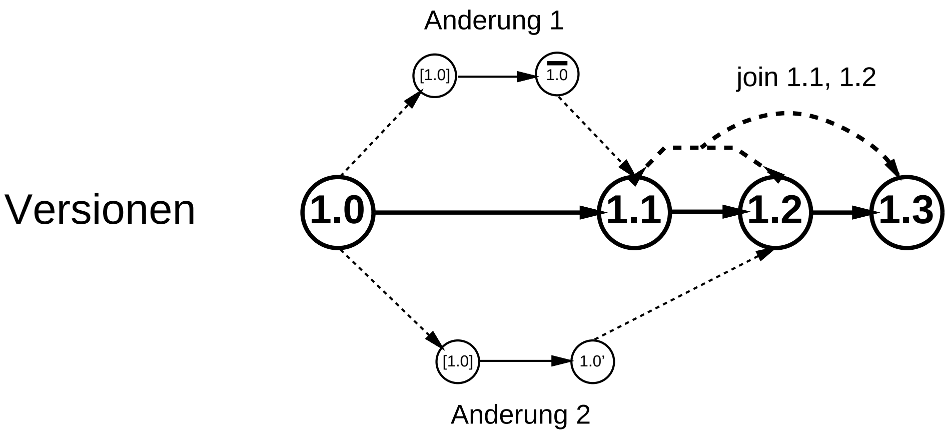

Abb. 2.2. Kombination unabhängiger Änderungen

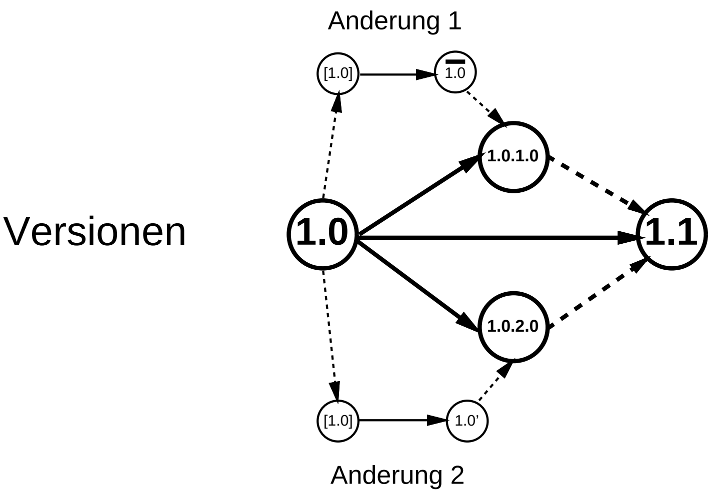

Abb. 2.3. Verschmelzung von Verzweigungen zu einer neuen Version

von Komponentenversionen und der Struktur des Softwaresystems zieht sehr enge Grenzen für ein effektives Konfigurationsmanagement. Zur Systemherstellung (*build*) muß die richtige Version der verschiedenen Komponenten jeweils manuell ausgewählt und installiert werden. Bei größeren Projekten führt dies schnell zu äußerst komplexen Situationen, die häufig in unangenehmen Fehlern (schwer lokalisierbar) und Zeitverlusten münden.

Modernere Systeme verfügen über ein ganzheitliches Konzept sowohl für die Versions- als auch die Systemstrukturverwaltung. Die bearbeiteten Module werden in der Regel zwar immer noch individuell im Projektrepositorium verwaltet, können jedoch mit einer Reihe von vorgangsspezifischen Attributen versehen werden. Die Attributierung von Komponentenversionen erlaubt es, logisch zusammenhängende Versionen (z.B. von gemeinsam geänderten Modulen) entsprechend zu markieren, oder verschiedenste Zustandsinformationen an die

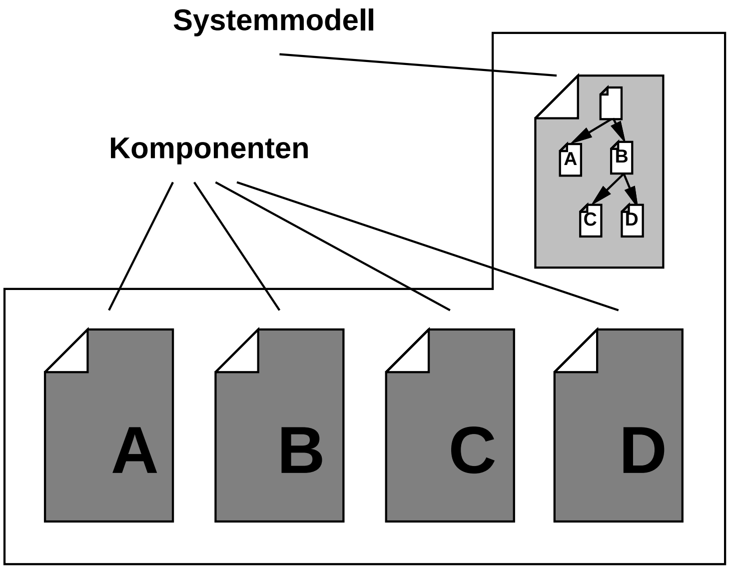

Abb. 2.4. Konfiguration = Komponenten + Systemmodell

Entwicklungsobjekte zu heften (z.B. *getestet*, *abgelehnt* oder *ausgeliefert*). Sofern eine systematische Vergabe von prozeßspezifischen Attributen gegeben ist, können bei der Systemherstellung geeignete Komponentenversionen jeweils dynamisch anhand vorgegebener Kriterien ausgewählt werden. Kriterien zur Auswahl geeigneter Versionen für Zielkonfigurationen mit bestimmten Eigenschaften wären beispielsweise:

> *Von allen Quellkomponenten die ausgeliehene, lokal geänderte Kopie, oder die aktuelle Version der Variante AS/400-german, bzw. falls die Variante für eine Komponente nicht definiert ist, die mit TXSys-2.4pre9 bezeichnete Version; Sonst: melde Konfigurierungsfehler.*

Regelbasierte Systemkonfigurierung hat sich in der Praxis als sehr leistungsfähiges Verfahren bewährt, das es sowohl ermöglicht, die einzelnen Arbeitsbereiche der Entwickler zu koordinieren (Isolation und kontrollierte Integration), als auch Systemzustände mit definierten Eigenschaften festzuhalten (*baselining*). Abb.2.5 zeigt *gebundene Konfigurationen*, die durch Auswertung verschiedener Regeln (bzw. gleicher,dynamischer Regeln zu unterschiedlichen Zeitpunkten) bestimmt wurden. Auf der Basis dieses grundlegenden SCM Modells wurden so erfolgreiche Softwareentwicklungsumgebungen wie Apollos legendäres *Domain Software Engineering Environment* (DSEE) [1] entwickelt. Andere Systeme, die auf diesem Modell basieren sind Adèle [9], Atria's ClearCase [24] oder ShapeTools [15].

2.3.3. Logische Änderungsfolgen

Eine später entwickelte Vorgehensweise zur Realisierung von Entwicklungstransaktionen ist das der logischen Änderungsfolgen (*change sets*). Hierbei handelt es sich um eine Weiterentwicklung des von der Mainframe Betriebssystempflege her bekannten Verfahrens, sog. Patches in alte Versionen der Betriebssoftware einzuspielen. Die Grundidee dieses Modells ist es, auf der Basis eines definierten Ausgangszustands eines Systems (Referenzlinie) jeweils eine logisch in sich geschlossene Änderung durchzuführen. Eine solche Änderung erstreckt sich dabei typischerweise über mehrere Komponenten. Die Gesamtänderung wird als Menge der Einzeländerungen

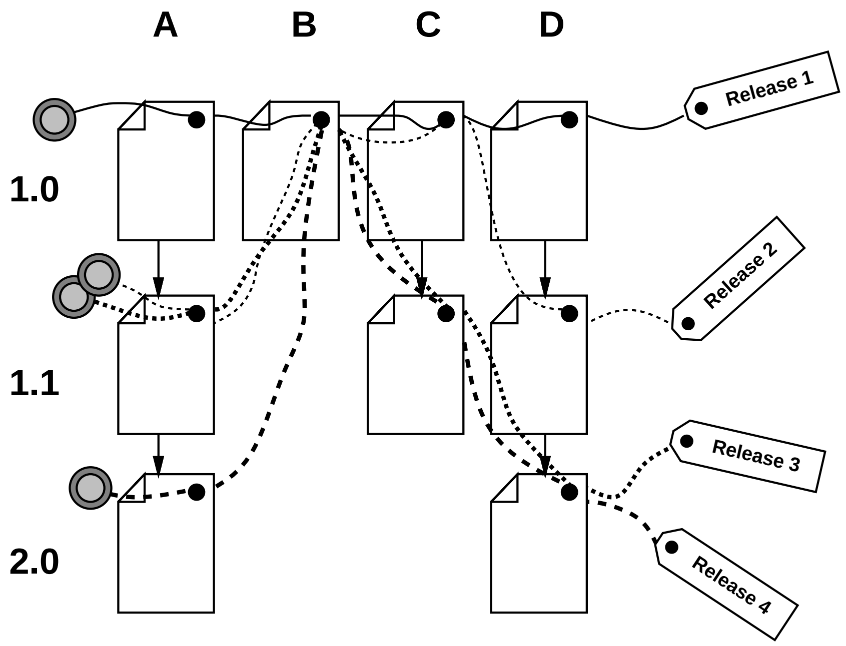

Abb. 2.5. Gebundene Konfigurationen - Konfigurierung und Versionsauswahl

an den betroffenen Komponenten gespeichert. Entsprechend gibt es hier keine Komponentenversionen, sondern lediglich Systemreferenzlinien und benannte Änderungen. Die Fortentwicklung und Pflege eines Softwaresystems spiegelt sich deswegen auch in der Folge der logischen Änderungen wieder.

Nimmt man beispielsweise an, das entwickelte System sei ein Informationssytem, so ließe sich die Entwicklungsgeschichte (zum Teil) darstellen als:

Release1 (Referenzlinie)
Variables Inventarlistenformat (Änderung 1)
Transaktions Deadlockvermeidung (Änderung 2)
...
Optimierung der Stücklistenanfrage (Änderung n)

Weitere Referenzlinien der Entwicklung lassen sich vor diesem Hintergrund gewissermaßen als Formeln beschreiben, etwa:

Release1a = Release1 + Änderung1 + Änderung3 + Änderung9 + ...
Release2 = Release1 + Änderung1 + Änderung2 + Änderung3 + ...

Das Modell der logischen Änderungsfolgen hat besondere Vorteile, ist jedoch auch mit einigen speziellen Problemen behaftet. Wie Feiler in [10] hervorhebt, sind logische Änderungen eine quasi natürliche Umsetzung von Änderungsaufträgen, wie sie vor allem während der Wartungsphase eines Softwareprojekts auftreten. Die direkte Zuordnung von logischen Änderungen zu Änderungsaufträgen führt zu einer problembezogeneren Ordnung des Projektverwaltungssystems.

Ein weiterer prinzipieller Vorteil des Prinzips der logischen Änderungsfolgen ist die Möglichkeit, diese gezielt in mehrere, parallel verwaltete Produktvarianten einzuspielen.

Die Kehrseite der Medaille ist allerdings das Problem, die Verträglichkeit der verschiedenen, oft nebenläufig entwickelten Änderungen sicherzustellen. Insbesondere ist es häufig der Fall, daß bestimmte Änderungen andere, vorher durchgeführte Änderungen voraussetzen. Andere Änderungen können wiederum im Konflikt mit diesen stehen. Um optimal zur Geltung zu kommen, müßten also alle logischen Änderungen voneinander unabhängig sein (was meist nicht der Fall ist). In gewisser Weise wird die gute Problembezogenheit der beschriebenen Vorgehensweise durch einen erhöhten Aufwand bei der Konsistenzsicherung der zu kombinierenden Änderungen komplementiert.

Trotzdem auf einige der oben skizzierten Probleme besonderes Augenmerk verwendet werden muß, treten beim Einsatz von Konfigurationsmanagementsystemen wie Aide-de-Camp [26] oder Softool's CCC [6] in der Praxis nur selten wirkliche Probleme mit dieser Vorgehensweise auf. Die Entwicklungsszenarien sind meist so eingerichtet, daß Konfliktfälle möglichst selten auftreten.

2.3.4. Lange Transaktionen

Einige Konfigurationsmanagementsysteme realisieren unmittelbar das Modell der Entwicklungstransaktion. Zur Initiierung eines Entwicklungsschritts wird explizit eine lange Transaktion gestartet und mit einer transaktionsspezifischen Bearbeitungszone (*Workspace*) assoziiert. Teil der Bearbeitungszone ist eine virtuelle Kopie der Ausgangskonfiguration, deren Komponenten vom Entwickler im Rahmen der Transaktion beliebig verändert werden können, ohne daß irgendwelche Änderungen im zentralen Projektrepositorium sichtbar werden (Isolation von unabhängigen Entwicklungsaktivitäten). Der Begriff *virtuelle Kopie* deutet an, daß wirkliche Kopien nur von denjenigen Komponenten angelegt werden, die tatsächlich geändert werden.

Für den Benutzer (d.h. ein Mitglied eines Projektteams) stellen sich lange Transaktionen in der Regel als hierarchisch strukturierte Objekte dar, deren Struktur die Aufbauorganisation des entwickelten Systems wiederspiegelt. Je nach Werkzeugsystem werden diese strukturierten Objekte als

- Unterverzeichnisse des Dateisystems (z.B. Rational[18]),
- virtuelle Dateisysteme (z.B. NSE [7], ClearCase[24], DSEE [1]), oder
- sog. Sichten (z.B. ClearCase)

in der Entwicklungsumgebung dargeboten. Die Art und Weise, wie das Transaktionskonzept in die Benutzungsumgebung abgebildet wird, hat einen starken Einfluß auf die Einbettbarkeit der SCM Werkzeuge in die verwendete Entwicklungsumgebung, bzw. den zu unterstützenden Entwicklungsprozeß.

Der besondere Vorteil von SCM Systemen, die direkt das Vorgehensmodell der Entwicklungstransaktion realisieren ist ein sehr konsistentes Modell für die Koordinierung nebenläufiger Entwicklungsschritte. Das hohe Abstraktionsniveau des Transaktionskonzepts bietet eine gute Grundlage zur Konzeption unterschiedlicher Vorgehensweisen zur Zusammenführung parallel ausgeführter Entwicklungsschritte und anderer, projektspezifisch spezialisierterer Teilprozesse.

Da die meisten transaktionsbasierten SCM Systeme ihr Datenmodell auf die Möglichkeiten konventioneller kommerzieller Datenbanksysteme abstützen, bedingen sie häufig den Kauf eines solchen Datenbanksystems – was zu nicht unerheblichen Kosten führen kann. Der hohe Grad an konzeptioneller Geschlossenheit birgt weiterhin die Gefahr einer gewissen Unflexibilität in Bezug auf die Einbettbarkeit in laufende Entwicklungsprozesse, da derartige SCM Werkzeuge/Systeme nicht selten als integrierte Komplettlösung angeboten werden. Dies ist insbesondere deswegen ein Problem, da viele Entwicklungsorganisationen erst relativ spät (d.h. nach Beginn der Wartungsphase) konsequentes Konfigurationsmanagement einführen. Dies mag einer der Gründe dafür sein, daß noch immer die einfachsten, Checkin/Checkout basierten, Systeme im konkreten Einsatz dominieren.

2.4. Komplexere Vorgehensweisen

Im vorangegangenen Kapitel haben wir versucht, das Konzept der Entwicklungstransaktion als grundlegende Technik zur Kontrolle von Entwicklungsvorgängen in Softwareprojekten zu skizzieren. Dieses Vorgehensmodell steckt mehr oder minder ausgeprägt in allen halbwegs durchdachten Konfigurationsmanagementsystemen (dabei verstehen wir unter Konfigurationsmanagementsystem nicht unbedingt exakt ein Werkzeugsystem, sondern die gesamte Menge von diesbezüglichen Funktionen in einer Entwicklungsumgebung). In der Praxis haben die Softwareentwicklungsprojekte in der Regel aber noch sehr viel weitergehende, projektspezifische Anforderungen an das Konfigurationsmanmagement. Zu den bestimmenden Faktoren bei der Ausgestaltung des konkreten SCM Systems für ein Projekt gehören (u.a.)

- Teamgröße
- Organisationsstruktur des Projektteams (Rollenmodell)
- Qualifikationsstruktur des Projektteams
- Größe und Komplexität des entwickelten Systems
- voraussichtliche Dauer des Projekts
- Beziehung zum Auftraggeber
- Bedeutung des Projekts
- gewählte Entwicklungsmethode (z.B. Top-Down vs. exploratives Prototyping) usw. usw.

Je nach Ausprägung dieser Randbedingungen wird man unterschiedliche Projektpolitiken wählen, die der Organisationsstruktur des Projekts entsprechen. So werden große oder sicherheitskritische Projekte ein stärker formalisiertes, konservatives Änderungswesen haben, während kleinere Systeme, die von hochqualifizierten Entwicklern betreut werden eher mit optimistischen Kooperationsmustern am besten weiterentwickelt werden dürften.

Nachfolgend sollen einige komplexere SCM Verfahrensweisen exemplarisch diskutiert werden. Jedes Entwicklungsprojekt wird sich mit der Zeit seine eigenen höheren Prozeduren aufbauen. Es ist ein Prüfstein für die Flexibilität von Konfigurationsmanagement Werkzeugen, wie gut sie dafür einsetzbar sind, solche und weitere Vorgehensweisen zu operationalisieren. Stellvertretend für die Vielzahl konkreter Szenarien skizzieren wir Realisierungen eines mäßig konservativen und eines optimistischen Kooperationsmodells.

2.4.1. Moderat konservative Kooperation: Iterative Integration

Das Vorgehensmodell der iterativen Integration basiert auf einer Rollenverteilung inerhalb des Projektteams, die formal zwischen Entwicklern und Systemintegratoren unterscheidet (verschiedene Rollen können dabei von derselben Person wahrgenomen werden). Die zweite Säule dieses Verfahrens ist ein Zustandskonzept für die Versionen der Systemkomponenten.

Zunächst werden die verschiedenen vorzunehmenden Änderungen an die einzelnen Entwickler zur Bearbeitung verteilt. Jeder einzelne Entwickler geht bei der Bearbeitung eines Änderungsvorhabens vor wie in Abb.2.1 dargestellt. Die im Rahmen der Entwicklungstransaktion zu ändernden Komponenten werden vom Entwickler exklusiv reserviert. Dies kann dazu führen, daß andere Entwickler ihren Änderungsauftrag nicht bearbeiten können, da bestimmte Komponenten bereits für eine andere Entwicklungstransaktion reserviert wurden. Der Vorteil dieses Prinzips ist, daß keine automatisierten Versionsverschmelzungen durchgeführt werden müssen.

Beim testen seiner Änderungen bezieht sich jeder Entwickler auf die gesicherten, unveränderten Stände aller Komponenten aus der Ausgangskonfiguration. Wenn die lokalen Änderungen den Schluß nahelegen, daß der Entwicklungsschritt erfolgreich beendet wurde, so schlägt der Entwickler seine Modifikationen zur Übernahme in die neu zu definierende Konfiguration vor.

Es ist die Aufgabe des Systemintegrators, die verschiedenen zur Übernahme in die Zielkonfiguration vorgeschlagenen Änderungen in dieser zu vereinigen. Dazu verfährt er im Prinzip wiefolgt:

1) Festlegung der Reihenfolge, in der die einzelnen Änderungen integriert werden sollen
2) Herstellung des Systems nach folgendem Rezept: man nehme

- alle geänderten Quellkomponenten, die bereits erfolgreich getestet wurden,
- diejenigen Quellkomponentenversionen, die zu der in diesem Durchlauf zu testenden Änderung gehören, sowie
- alle übrigen Quellkomponenten aus der Ausgangskonfiguration.

3) Aktivierung der Regressionstestsuite
4) Auswertung der Testergebnisse: falls der Test erfolgreich durchlaufen wurde, markieren der unter 2b) ausgewählten Komponenten als „erfolgreich getestet“ falls der Test nicht erfolgreich war, benachrichtigen des Entwicklers und zurückweisen der zur Übernahme vorgeschlagenen Quellkomponentenversionen
5) Falls alle Änderungen erfolgreich integriert wurden, Definition und Sicherung der Zielkonfiguration. Ansonsten fahre mit Punkt 2) fort.

Für eine effiziente Durchführung der Integrationstests sind leistungsfähige Systemherstellungswerkzeuge wie *shape* von großer Bedeutung.

2.4.2. Optimistische Kooperation: Kopieren−Ändern−Verschmelzen

Das Kooperationsverfahren „Kopieren−Ändern−Verschmelzen“ (*copy−modify−merge*) arbeitet ohne exklusive Reservierung von Systemkomponenten und benötigt nicht unbedingt eine separate Funktion für den Systemintegrator. Jeder Entwickler hat ungehinderten Zugriff auf alle Komponenten, die für einen Änderungsauftrag modifiziert werden müssen. Dabei entstehen bei parallelen Änderungen Versionsverzweigungen, die im Zuge der Systemintegration verschmolzen werden müssen.

Der deutlichste Unterschied zum vorher beschriebenen Kooperationsszenario ist das Verfahren zur Veröffentlichung von Änderungen. Wie in Abb.2.6 dargestellt, gelingt die Veröffentlichung der Änderung, wenn sich die aktuelle Integrationskonfiguration (Ausgangskonfiguration plus aller bereits veröffentlichter Änderungen) in Bezug auf die zu veröffentlichenden Komponenten nicht gegenüber der Ausgangskonfiguration geändert hat.

Wurden im Rahmen eines nebenläufig bearbeiteten Änderungsauftrags A1 bereits Versionen von Komponenten veröffentlicht, die auch im Zusammenhang mit einer weiteren Änderung A2 modifiziert wurden, so müssen die entsprechenden Änderungen aus A1 vor einer erfolgreichen Veröffentlichung in die Entwicklungstransaktion A2 eingearbeitet werden. Dies geschieht in der Regel durch automatische Versionsverschmelzung.

Das beschriebene Verfahren wird *optimistisch* genannt, da auf ein problemloses automatisches Verschmelzen von Versionen spekuliert wird. Praktische Erfahrungen mit Systemen wie CVS [2] scheinen zu belegen, daß sich dieser Optimismus in der Regel auszahlt. Das Problem der prinzipiellen Unsicherheit textbasierter Verschmelzungsverfahren bleibt jedoch bestehen. Wenn es zu Konflikten kommt (z.B. mehrere Änderungen in derselben Zeile) müssen diese von Hand aufgelöst werden. Noch unangenehmer − weil schwer zu lokalisieren − sind Fehler, die durch semantisch inkompatible Änderungen, die sich konfliktfrei verschmelzen ließen, auftreten.

3. ShapeTools

ShapeTools ist ein Software Werkzeugkasten von fein aufeinander abgestimmten Konfigurationsverwaltungs Funktionen. Es ist vor allem darauf ausgerichtet, SCM Funktionalität in bestehende Software Entwicklungsumgebungen und −organisationen einzubetten. In seiner Gesamtheit ist es aber auch als schlüsselfertige Entwicklungsumgebung zu gebrauchen. ShapeTools implementiert zu weiten Teilen die im bisherigen Verlauf dieses Papiers skizzierten Techniken des SCM. Wichtiges Kriterium bei der Einbettung von ShapeTools in existierende Organisationen ist es, einen *Technologieschock* zu vermeiden und laufende Entwicklungsprozesse durch die Einführung so wenig wie möglich zu beeinträchtigen. Bezüglich der angebotenen Funktionalität ist es erklärtes Ziel, Softwareentwicklung entsprechend ISO 9001 [19] zu unterstützen und damit die Voraussetzungen für einen hohen Qualitätsstandard bei Entwicklung und Plege von Software zu erreichen.

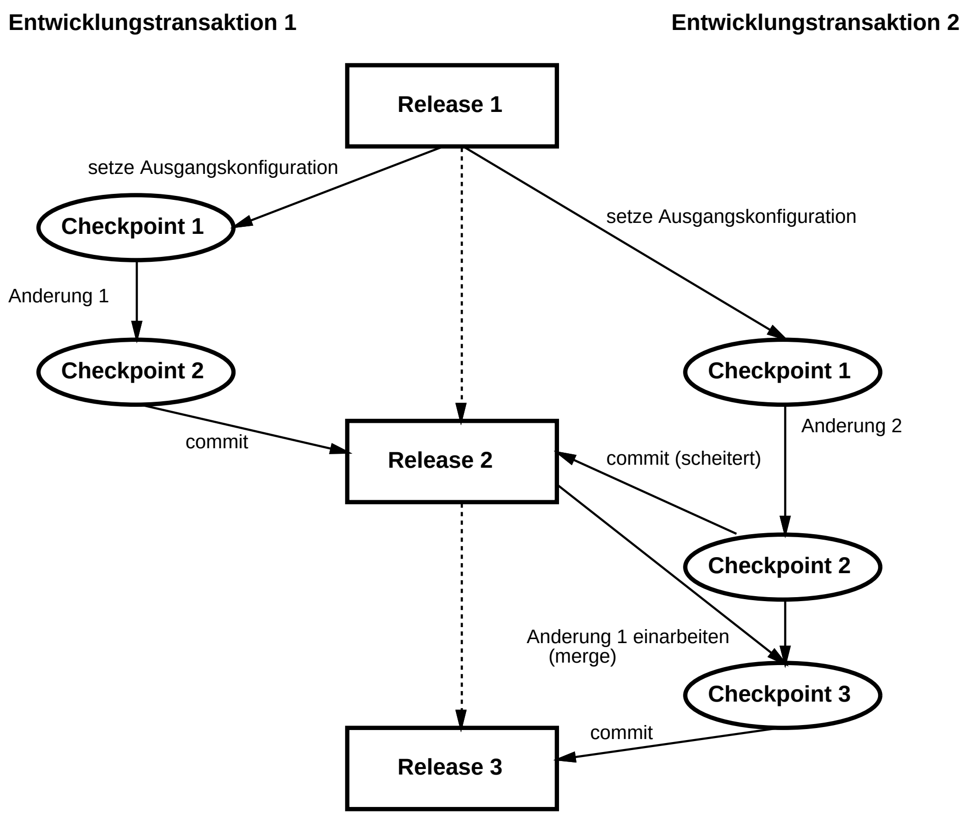

Abb. 2.6. Nebenläufige Entwicklungstransaktionen

Im Folgenden beschäftigen wir uns mit den einzelnen Funktionsbereichen von ShapeTools und ihrer Einbettbarkeit in Entwicklungsorganisationen sowie mit dem Aufbau einer auf ShapeTools basierenden Software Entwicklungsumgebung.

3.1. Repository

Zur Ablage der Daten steht in ShapeTools das *Attribut Dateisystem (Attribute Filesystem – AtFS)* [14] zur Verfügung. AtFS bildet eine konzeptionelle Erweiterung vorhandener Dateisysteme, indem es Versionen von Dateien speichern kann und zu jeder Version eine beliebige Anzahl von Attributen verwaltet. Ein solches, dateiähnliches, attributiertes Gebilde heißt *Attributiertes Software Objekt (ASO)*. Die Zusammenfassung aller Versionen einer Datei wird in AtFS *konzeptionelles Objekt* oder *Historie* genannt.

AtFS ist implementiert als Erweiterung des UNIX Dateisystems und bezieht dabei alle Daten des UNIX Dateisystems in seine Datenwelt mit ein. Reguläre Dateien können auch als ASOe betrachtet werden und beliebig attributiert werden. Gleichzeitig bleibt der direkte Dateisystemzugriff, an AtFS vorbei, offen. Aus Sicht des Attribut Dateisystems besteht eine Historie aus einer *aktiven Version* und einer Reihe von *gesicherten Versionen*. Die aktive Version wird durch eine reguläre Datei repräsentiert und ist sowohl durch AtFS als auch durch Dateisystemzugriff manipulierbar. Gesicherte Versionen existieren nur in der AtFS Sicht, ihr Inhalt ist nicht modifizierbar.

AtFS speichert seine internen Daten, die Attribute und die gesicherten Versionen, in einem eigenen Bereich des Dateisystems, auf den nur über AtFS zugegriffen wird. In jedem Dateisystem kann es mehrere solcher Bereiche geben. Jedes Verzeichnis im Dateisystem, in dem Attribute gesetzt werden sollen oder von wo aus Versionen gesichert werden sollen, muß an einen AtFS Bereich angeschlossen sein. In einem kooperativen Entwicklungsprojekt ist es sinnvoll, zentrale AtFS Bereiche zu haben, an die sich alle beteiligten Entwickler anschließen. Bei einer Zerlegung eines Projekts in Teilsysteme empfiehlt sich ein AtFS Bereich pro Teilsystem (vgl Abb. 3.1).

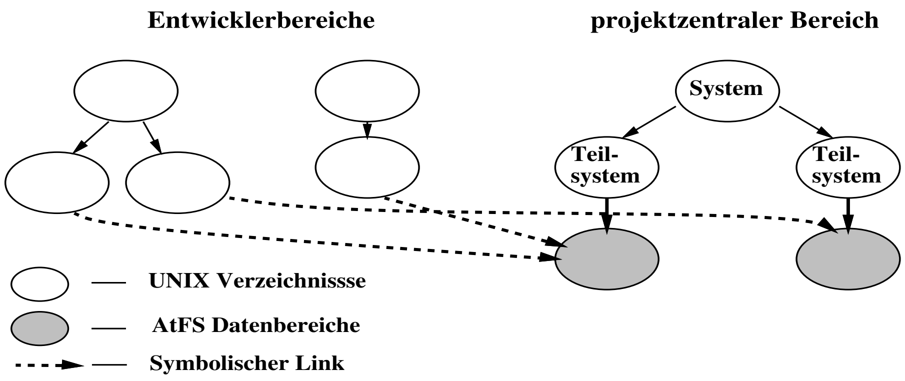

Abb. 3.1. Datenaustausch via AtFS Objektbasis

Über den projektzentralen Entwicklungsbereich findet der Datenfluß zwischen Entwicklern sowie zwischen Entwickler und Projektleitung statt. Aktive Versionen der einzelnen Entwickler sind nur in deren Arbeitsbereichen sichtbar, wogegen gesicherten Zustände potentiell allen anderen angeschlossenen Parteien zur Verfügung stehen. Abhängig ist dies vom *Status* der gesicherten Version, was Thema das nächsten Abschnitts ist.

3.2. Versionen

Die Grundzüge der Versionsbildung mit schreibbaren aktiven und nur lesbaren gesicherten Versionen sind im vorigen Abschnitt schon genannt. Jede gesicherte Version bekommt eine, innerhalb der Historie eindeutige, *Versionsnummer*. Neben der Numerierung können jeder Version einer oder mehrere symbolische Namen, *Versionsaliase*, angeheftet werden. Versionsaliase bezeichnen Versionen innerhalb ihrer Historie eindeutig und können immer anstatt der Versionsnummer verwendet werden.

Neben der Versionsnummer hat jede gesicherte Version einen *Versionsstatus*. Der initiale Status ist *saved*, was bedeutet, daß es sich bei der Version um eine entwicklerprivate Sicherung eines Zwischenstandes handelt. Der Zustand *proposed* zeigt an, daß diese Version vom Entwickler zur Veröffentlichung vorgeschlagen wurde. *Published* sind Versionen, die vormals proposed waren und eine Qualitätsprüfung zur Veröffentlichung bestanden haben. Damit stehen diese Versionen zur Einbindung in komplexe Konfigurationen bereit. Der nächste Zustand, *accessed*, bedeutet daß diese Einbindung stattgefunden hat, und die Version Teil einer im zentralen Bereich definierten Konfiguration des Systems ist. In den Zustand *frozen* werden solche Versionen versetzt, die Bestandteil einer Konfiguration sind, die den Entwicklungsbereich verlassen hat und z.B. bei einem Kunden eingesetzt wird.

Ein wichtiger Punkt bei der Versionsbildung ist die Dokumentation der Änderungen von Version zu Version. ShapeTools in der vorgefertigten Form verfolgt hierbei die Philosophie, daß aktive Versionen nur während einer Entwicklungstransaktion existieren. Bei der Herstellung einer aktiven Version innerhalb einer Entwicklungstransaktion fragt ShapeTools automatisch nach der *Änderungsabsicht*. Beim Abschluß der Transaktion wird dem Entwickler seine ursprünglich geäußerte Änderungsabsicht erneut vorgelegt um daraus die tatsächliche *Änderungsbeschreibung* (wobei die geäußerte Änderungsabsicht nachträglich modifiziert werden kann) zu erzeugen.

Das System von Änderungsabsicht und Änderungsbeschreibung bewirkt zweierlei:

– erstens wird die entstehende Änderungsdokumentation einer Historie zuverlässiger, weil Analyseergebnisse und Entwurfsentscheidungen, die vor der Entscheidung zur Änderung einer Komponente anfallen, unmittelbar dokumentiert werden können, nicht bloß deren technische Auswirkungen
– zweitens entstehen keine Dokumentationslücken wie bei anderen Versionsverwaltungssystemen (z.B. RCS und SCCS). Bereits zum Zeitpunkt der Bearbeitung und nicht erst nach Beendigung einer Transaktion können Projektmanager und Teammitglieder abfragen, welche Arbeiten an den einzelnen Komponenten aktuell stattfinden.

Damit dient dieses Mittel zum Überblick, welche Aktivitäten aktuell die Systementwicklung ausmachen, um diese besser steuern und koordinieren zu können.

3.3. Synchronisation von Entwickleraktivitäten

Im Bereich der Synchronisation von Entwicklungstransaktionen sind in ShapeTools verschiedene Strategien möglich. Sie basieren aber alle auf der Technik der Komponentenein-/ausgangskontrolle. Hier ist zunächst die Technik des ausschließlichen Sperrens einzelner Systemkomponenten während der Bearbeitung. Diese ist in den meisten Fällen die Vernünftigste, sie ist einfach und sicher. ShapeTools verfeinert die Technik der Reservierung von Versionierungsrechten etwas. Sperren gelten jeweils nur für genau eine Generation innerhalb einer Historie. Entwicklung in verschiedenen Generationen, was besonders wichtig ist für Projekte, in denen frührere Konfigurationen bereits im Einsatz sind und gewartet werden müssen, ist ohne weiteres möglich.

Brauchbar ist die Technik des ausschließlichen Sperrens nur, wenn die simultane Änderung einer Komponente in derselben Generation durch zwei oder mehr Entwickler der Ausnahmefall ist. In einigen Entwicklungsprojekten, meist solchen die sich in der Produktphase befinden, wo verschiedene Versionen des Systems bei Kunden im Einsatz sind und gewartet werden müssen, ist die Lage erfahrungsgemäß anders. Hier gibt es Module, die es im Laufe der Zeit zu beachtlicher Größe und Wichtigkeit gebracht haben und die bei fast allen Fehlerkorrekturen oder kundenspezifischen Anpassungen betroffen sind. Hier ist es oft nicht möglich, den ganzen Modul exklusiv zur Bearbeitung zu sperren. ShapeTools unterstützt hierzu eine einfache Variante des *Kopieren, Manipulieren und Verschmelzen* Modells. „Einfach" deshalb, weil Änderungsüberdeckungen in jedem Falle den manuellen Eingriff eines Entwicklers notwendig machen. Ansonsten geht das Verschmelzen von Änderungen automatisch.

3.4. Systemvarianten

Der Begriff von Systemvarianten in der Softwareentwicklung ist recht weitläufig. Klassische Beispiele sind Portierungen eines Systems oder verschiedenpsrachliche Benutzungsschnittstellen. Experimentelle Entwicklungen einzelner Komponenten oder verschiedene Systemausbaustufen sind von ihrem Charakter her ebenfalls Varianten. Dazu kommen Produktvarianten, die aus derselben Quelle unter Verwendung verschiedener Übersetzer oder unterschiedlicher Übersetzungsoptionen generiert wurden. Die Liste ließe sich fortsetzen.

ShapeTools versuchen nicht, dem Variantenbegriff eine spezielle Bedeutung zuzuordnen. Es werden vielmehr eine Reihe technischer Hilfsmittel geboten, alle oben genannten und weitere Variantenszenarien zu modellieren. Dabei bekommt jede einzelne Variante einen *Namen*, und kann zusätzlich einer *Familie* zugeordnet werden. Variantenfamilien haben den Zweck, den gegenseitigen Ausschluß von Varianten festzulegen. Aus jeder Familie kann höchstens eine Variante in eine zu bestimmende Konfiguration übernommen werden.

Die Modellierung der Charakteristika einzelner Varianten hat die Form einer Reihe von Definitionen, die bei der Systemkonfigurierung und –herstellung (s.u.) berücksichtigt werden. Mit diesen Definitionen kann z.B. die Kompositionsliste eines Systems genauso variiert werden, wie die Festlegung, welche Übersetzungswerkzeuge zu verwenden sind und welche Übersetzungsoptionen diese bekommen sollen. Variantendefinitionen sind Teil des Systemmodells, von dem im folgenden Abschnitt die Rede sein wird. Sie sind dahingehend eng mit dem Systemmodell verzahnt, daß sie *Variablen des Systemmodells mit Werten belegen*.

3.5. Systemmodellierung und −konfigurierung

Die Systemmodellierung ist in ShapeTools aufgefächert in die Bereiche Systemidentifizierung, Variantenbildung und Versionsbindung (Versionsauswahl). Zur Systemmodellierung gibt es ein Beschreibungsdokument, das *Shapefile*, in dem alle beteiligten Systemkomponenten, deren gegenseitige Abhängigkeiten, die Variantendefinitionen und Versionsbindungsregeln festgelegt sind. Für komplexe Systeme, die in Form einzelner Teilsysteme modelliert werden, kann das Systemmodells auf mehrere Beschreibungsdokumente, am geeignetsten eins pro Teilsystem, verteilt werden. Die einzelnen Beschreibungsdokumente werden in einem solchen Fall in eine Hierarchie eingeordnet, die es erlaubt, die Systemkonfigurierung für das Gesamtsystem oder komplexe Subsysteme rekursiv über mehrere Teilsysteme vorzunehmen.

Der Systemidentifizierungsteil ist kompatibel zu *make* [11]. Das bedeutet, daß reguläre make Beschreibungsdateien (Makefiles) inkludiert werden können. Bei der Modellierung der Abhängigkeiten zwischen einzelnen Systemkomponenten geht ShapeTools aber über die Möglichkeiten von make und anderen dateibasierten Lösungen hinaus. ShapeTools profitiert hier insbesondere von der Möglichkeit der Attributierung aller beteiligten Objekte. Diese bereitet den Weg zur Modellierung von Abhängigkeiten als Relationen in der Objektbasis. Damit sind diese nicht nur der Konfigurierungs− und Systemherstellungskomponente zugreifbar, sondern dienen als allgemeine Information in der Entwicklungsumgebung.

Die einfachste Form der Versions*bindung* besteht darin, zusätzlich zur Identifikation des konzeptionellen Objekts explizit eine Versionsnummer oder ein Versionsnummernalias anzugeben. Versionsnummern haben den deutlichen Nachteil, nichts über die Qualität der Version, ihr Alter oder den Unterschied zu anderen Versionen auszusagen. Zur expliziten Versionsbinding durch den Benutzer taugen sie daher wenig. Ihre Funktion liegt eher auf der technischen Ebene, wo es darum geht, eine einmal vorgenommene eindeutige Versionsauswahl in kompakter Form aufzubewahren. Versionsaliase, eindeutige symbolische Namen für einzelne Versionen bieten eher die Möglichkeit etwas über die Bedeutung einer Version auszusagen. Sie werden meist als Kennzeichner der Versionen einer Konfiguration verwandt. Hierbei werden in allen konzeptionellen Objekten diejenigen Versionen mit einem gemeinsamen Namen gekennzeichnet, die eine konkrete Konfiguration des Systems ausmachen.

ShapeTools bietet neben symbolischen Namen zwei zusätzliche Formen der Versionsbindung. Die Versionsbindung mit Hilfe eines *Datums* erlaubt es, diejenige Version einer Historie auszuwählen, die zu einem gegebenen Zeitpunkt die aktuelle Gesicherte war. Für einen Entwickler ist es beim Rückgriff auf einem bestimmten Entwicklungszustand leichter ein Datum zu erinnern, als die zu diesem Zeitpunkt aktuellen Versionsnummern. Die zweite zusätzliche Form der Versionsbindung basiert auf attributgesteuerten *Versionsauswahlregeln*. Diese Regeln drücken auf allgemeiner, objektunabhängiger Ebene aus, nach welchen Kriterien die Versionsbindung in konzeptionellen Objekten stattfinden soll. Beispiel für eine einfache Regel ist

> *Wähle die aktive Version.*
> *Wenn es keine aktive Version gibt, wähle die zuletzt gesicherte Version.†*

Dies ist nur ein sehr einfaches, aber im alltäglichen Entwicklerleben oft benutztes, Beispiel für die Ausdrucksfähigkeit von Versionsbindungsregeln. Es zeigt bereits die allgemeine Anwendbarkeit und die Abstützung der Regeln auf Attributen, in diesem Falle dem Versionsstatus. Weitere Ausdrucksmöglichkeiten sind die Formulierung externer Bedingungen und die Einbeziehung von Benutzerinteraktion.

---

†ShapeTools benötigt Versionsbindungsregeln in einer formaleren Schreibweise als der hier Angeführten. Für eine Beschreibung der Syntax und eine detaillierte Erläuterung von Versionsbindungsregeln sei auf [16] verwiesen.

Versionsbindungsregeln können nicht nur bei der Konfigurierung eines Systems angewandt werden. Dem Benutzer von ShapeTools stehen sie *immer* dann zur Verfügung, wenn es gilt aus einer Historie eine konkrete Version auszuwählen.

3.6. Systemherstellung

Die Systemherstellung, also die Durchführung der notwendigen Übersetzungs− und Bindeoperationen zur Herstellung eines lauffähigen Systems, findet in ShapeTools vollautomatisch statt. Voraussetzung ist die Konfigurierung des Sytems mit Systemidentifizierung (unter Berücksichtigung der Varianten) und Versionsbindung. Aus Anwendersicht finden Konfigurierung und Herstellung in einem Schritt statt. Für jeden einzelnen Übersetzungsvorgang stellt ShapeTools den den notwendigen Versionskontext zur Verfügung, wenn nötig durch Etablierung der beteiligten Versionen als Dateien im Dateisystem. Nach Abschluß der Transformation wird der ursprüngliche Dateisystemzustand wieder hergestellt.

Als Ergänzung zur Versionsverwaltung auf Quellobjektebene werden auch verschiedene Versionen *abgeleiteter Objekte* gespeichert (*derived object caching*). Abgeleitete Objekte zeichnen sich dadurch aus, daß sie jederzeit aus ihrem Quellkontext wiederhergestellt werden können. Die zentrale Speicherung mehrerer Versionen abgeleiteter Objekte ermöglicht es zahlreiche Rekompilationen zu vermeiden, da die Kompilationsergebnisse häufig von anderen Teammitgliedern verwendet werden könnnen. Als Methode der Balancierung von Plattenplatzbedarf und Vermeidung von Rekompilationen verfolgt ShapeTools das Konzept der *Pufferung* abgeleiteter Objekte. Zur Pufferung steht eine bestimmte (festlegbare) Menge Plattenplatz für abgeleitete Objekte zur Verfügung. Ist die Plattenplatzgrenze erreicht, wird für jede neu hinzukommende Version eine andere Version gelöscht.

Vor jeder Übersetzung prüft ShapeTools, ob es nicht bereits ein passendes abgeleitetes Objekt im Dateisystem oder im Puffer gibt. Diese Überprüfung findet anhand eines *Ableitungsschlüssels* statt, der als Attribut an jedes abgeleitete Objekt geheftet wird. Dieser Schlüssel wird vor jeder Übersetzung aus den beteiligten Quellversionen und dem Übersetzungskontext gebildet und dient als Suchkriterium für ein passendes abgeleitetes Objekt. Nur wenn dies nicht bereits existiert, wird die Übersetzung tatsächlich angestoßen.

3.7. Releases

Als Beispiel für konkrete Projektunterstützung bieten ShapeTools nicht nur Versionsverwaltung auf Komponentenebene, sondern auch Versionen komplexer Systeme, die zur begrifflichen Unterscheidung *Releases* genannt werden. Dieser Funktionsbereich, im Folgenden ShapeTools *Release Management System (RMS)* genannt, umfaßt die folgende Funktionalität:

- Automatisierte Herstellung von *Prereleases* und *Releases* von Teilsystemen und vom Gesamtsystem.
- System*herstellung* und System*installation* auf der Basis von Pre−/Releases.
- Feststellung der Unterschiede zwischen zwei Pre−/Releases und die Möglichkeit der Herstellung eines *Patches* zur automatischen Überführung des ersten in den zweiten Releasestand (→ Fernwartung).
- Wiederherstellung alter Systemzustände.
- Extrahierung veröffentlichter (Teil-)Systeme aus der Entwicklungsumgebung.
- Systemweite Vereinheitlichung von Versionsauswahl und Variantensteuerung.

ShapeTools RMS realisiert das Konzept der *iterativen Integration*. Dazu werden einige Annahmen über die Projektorganisation gemacht. Die Zerlegung des Softwaresystems ist hierarchisch. Jeder Knoten in der Hierarchie repräsentiert ein Teilsystem, das aus einer Reihe von Komponenten und Subsystemen besteht, die wiederum Teilsysteme sind. Festgeschrieben wird diese Struktur in einer Menge von Systemmodell Dateien (Shapefiles) — genau einem pro Teilsystem. Vom Wurzelknoten der Hierarchie aus erreicht man durch Traversierung des Zerlegungsbaums das komplette System. Entwicklungsarbeit und projektinterne Veröffentlichung werden, wie in

Abschnitt 3.2 beschrieben, von privaten Entwicklerbereichen aus gemacht. Systemintegration und Releasebildung des Gesamtsystems und komplexer Teilsysteme können von zentraler Stelle aus gemacht werden.

Die Bildung von Releases gliedert sich in zwei Schritte, erst die (wiederholte) Bildung von Prereleases, bevor die eigentliche Releasebildung von statten geht. Denkt man sich ein Release als Entwicklungsziel, als *angestrebten* Systemzustand, sind Prereleases ein probates Mittel um den *aktuellen* Systemzustand zu fixieren und mit dem Angestrebten zu vergleichen. Deckt sich das Prerelease mit dem angestrebten Zustand, wird es zum Release. Jedes Prerelease und jedes Release bekommen vom RMS automatisch einen Namen, der aus dem Systemnamen und einer Versionsnummer gebildet wird. Prereleases tragen dabei Namen und Version des angestrebten Release mit einer zusätzlichen Seriennummer. Dieser Name wird jeder beteiligten Version als Versionsalias angeheftet. Über diese Identifikation kann jederzeit bequem auf alle Versionen eines Pre−/Releses zugegriffen werden. Dies spielt eine wichtige Rolle bei der Systemherstellung auf der Basis von Releases und bei der Wiederherstellung konsistenter alter Systemzustände.

Die Releasebildung komplexer Systeme findet durch die Einbeziehung existierender Pre-/Releases der untergeordneten Teilsysteme statt. Erst wenn alle Teilsysteme formal veröffentlicht wurden, macht es Sinn, auf der übergeordneten Stufe ein Pre-/Release zu bilden.

4. Einbettung in existierende Entwicklungsorganisationen

Ziel von ShapeTools ist es, Konfigurationsverwaltung als durchgängiges, einheitliches Prinzip in Software Entwicklungsorganisationen einzuführen, gleichzeitig aber den laufenden Betrieb bereits aktiver Projekte nicht zu behindern. Die SCM Funktionalität soll so behutsam und schrittweise einführbar sein, daß Beeinträchtigungen der regulären Arbeit, Konvertierung von Daten, Umschulung der Mitarbeiter und Einführung neuer Werkzeuge nur in geringem Umfange notwendig werden. ShapeTools sollen einbettbar sein in existierende Umgebungen, mit den dort verwendeten Werkzeugen kooperieren und die dort gewünschten (möglicherweise bereits existierenden) Kommunikationsstrukturen und Verfahrensmodelle unterstützen.

Die Integration der ShapeTools Konfigurationsverwaltungs Funktionalität mit fremden Entwicklungswerkzeugen findet auf verschiedenen Ebenen statt. Es bietet sich die Möglichkeit der sehr *feinen* Abstimmung auf Programmiersprachenebene (C-Ebene). Die *mittlere* Ebene ist die der Werkzeug Erweiterungssprachen (Tcl-Ebene). Relativ *grob* gestaltet sich die Integration auf Kommando(Aufruf-)ebene (Shell-Ebene). Abbildung 4.1 veranschaulicht die verschiedenen Integrationsebenen und ihre Ansatzpunkte in der Architektur von ShapeTools.

Grundvoraussetzung für die Integration eines Werkzeugs ist dessen Kompatibilität mit dem Begriff des Attributierten Software Objekts. Das bedeutet aber *nicht*, daß die Integration mit ShapeTools auf dateisystembasierte Werkzeuge beschränkt bleibt. Obwohl dies ursprünglich die einzige Zielrichtung war, zeigen neuere Konzepte, insbesondere die der Erweiterungssprachen, daß auch Datenbankbasierte Anwendungen in das ShapeTools SCM Konzept eingebunden werden können, sofern ein geeigneter Zugriff auf die bearbeiteten Daten in der Datenbank existiert. Absolut notwendig ist hierzu bloß im Datenmodell der Anwendung ein Konzept von paarweise disjunkten, abgeschlossenen Bearbeitungseinheiten und die Fähigkeit der Attributierung dieser Bearbeitungseinheiten.

Die Grenzen der Integrationsfähigkeit sind meist auf anderer als der Datenebene zu finden. Viele Software Entwicklungsumgebungen (CASE Umgebungen) sind hochintegrierte Systeme mit dem Anspruch, allen Erfordernissen in der Softwareentwicklung gerecht zu werden. Dort sind dann Datenmodell, Vorgehensmodell und Benutzungsschnittstelle so eng verwoben und festgeschrieben, daß dem Anwender kein echter Spielraum zur Modifikation bleibt. Zur Integration mit Systemen solcher Art sind ShapeTools nicht geeignet. In letzter Zeit zeigt sich aber eine Abkehr von solchen ganzheitlichen Lösungsversuchen. Viele Hersteller von CASE Werkzeugen haben erkannt, daß bei Software Entwicklungsumgebungen mehr Flexibilität gefragt ist. Und diese kann nur durch die Möglichkeit der dynamischen Erweiterung und Anpassung der Umgebung erreicht werden.

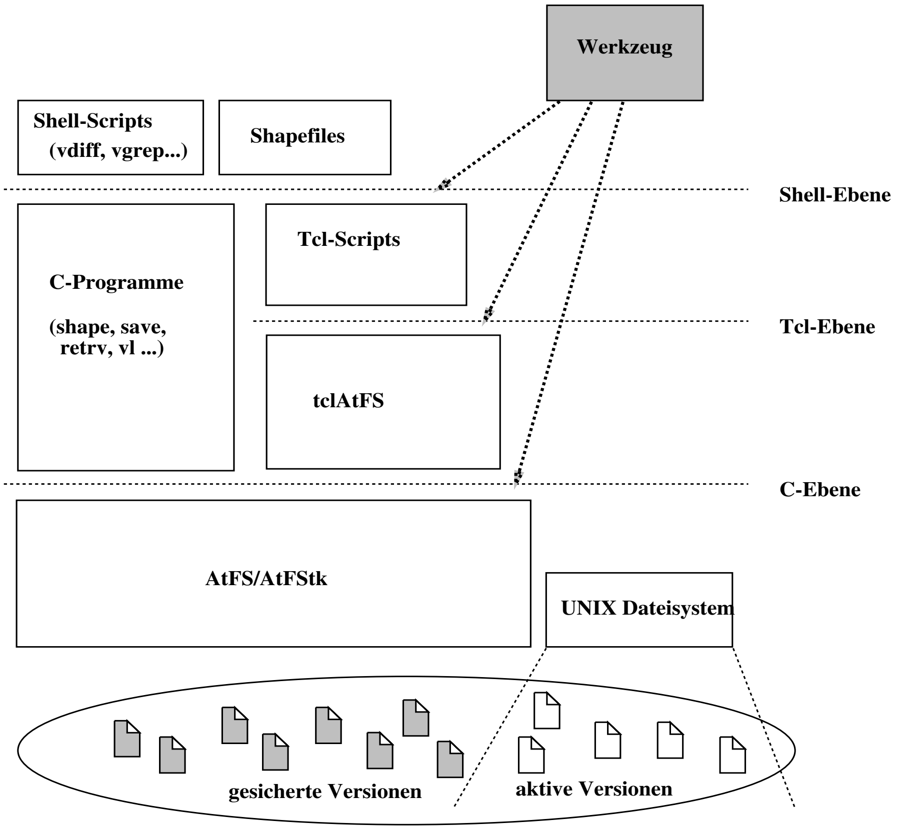

Abb. 4.1. ShapeTools Intergrationsebenen

4.1. Kommandos

Die Einbettung auf Kommandoebene ist die gröbste Form der Integration. Sie ist aber die einzige Möglichkeit wenn Werkzeuge der Entwicklungsumgebung lediglich Zugriff auf Funktionen an der Kommandoschnittstelle zulassen.

Als Technik zur Kommandointegration bieten sich zum einen Shell Skripte, in denen komplexe Abläufe mit Hilfe des Aufrufs von Umgebungskomponenten und ShapeTools Kommandos modelliert werden können. Ein Beispiel hierfür ist das *vdiff* Kommando, dessen Aufgabe es ist, zwei Versionen einer Historie zu vergleichen und die Unterschiede auszugeben. Vdiff stützt sich auf das UNIX diff Kommando, das die Unterschiede zwischen zwei Dateien feststellt. Hierzu werden zunächst die beiden Versionen als temporäre Dateien hergestellt, verglichen und danach wieder gelöscht. Zusätzlich wird die Ausgabe des diff Programms so aufbereitet, daß anstelle der Dateinamen der temporären Dateien die Versionsnamen erscheinen.

Die zweite Form der Kommandointegration sind Transformationsskripte in Shapefiles. Großer Voteil dieser Form ist, daß in Shapefiles Versions– und Variantenkontexte definiert werden können, die bei der Ausführung des Transformationsskripts berücksichtigt werden. Prominenteste Beispiele hierfür sind Übersetzer, die nur auf regulären UNIX Dateien arbeiten. Diesen wird durch den shape Interpreter ein Übersetzungskontext geschaffen, der aus genau den gewünschten Versionen besteht. Hierzu werden zunächst alle betroffenen existierenden regulären Dateien gesichert, danach die verlangten Versionen als reguläre Dateien hergestellt und die Übersetzung angestoßen. Anschließend werden die Übersetzungsergebnisse gesichert (gepuffert) und der ursprüngliche Kontext wieder hergestellt.

4.2. Erweiterungssprachen

Moderne Werkzeuge sind oft mit einer Erweiterungsspache ausgestattet. Erweiterungssprachen dienen dreierlei Zwecken.

- Bereitstellung einer Schnittstelle (unabhängig von der Kommandoschnittstelle) zum Zugriff auf einzelne Funktionen des Werkzeugs.
- anwendungsspezifische Definition neuer komplexer Funktionen auf Basis der Grundfunktionalität.
- Einbindung externer Funktionen an definierten Punkten im Kontrollfluß (Hooks).

Sehr erfolgreich wurde die Integration auf Erweiterungssprachenebene angewandt bei der Integration von ShapeTools und dem Emacs Editor mit einem Lisp Dialekt (Emacs-Lisp) als Erweiterungssprache. Emacs läßt z.B. beim Öffnen einer Datei, wenn unter dem angegebenen Namen kein Dateisystemobjekt gefunden wurde, per Erweiterungs*hook* einen zweiten Versuch zu. ShapeTools nutzt diesen, um auf gesicherte Versionen zuzugreifen und Emacs als „Quasi–Dateien" unterzuschieben. Für den Anwender sehen damit gesicherte Versionen wie reguläre, schreibgeschützte Dateien aus, mit einer Versionsnummer als Dateinamensextension.

Als Beispiel für den zweiten Punkt in der obigen Liste wurde in Emacs, zusätzlich zur regulären Funktionalität der Prozeß der Dokumentation von Historien mit Änderungsabsicht und Änderungsbeschreibung eingebettet. Der Versuch eines Anwenders, den Inhalt einer gesicherten Version modifizierbar zu machen, führt (nach Bestätigung) zur Herstellung einer aktiven Version mit automatischer Abfrage der Änderungsabsicht. Der umgekehrte Weg, die Aufhebung der Modifizierbarkeit einer akiven Version, sichert diese als neue Version, wiederum mit automatischer Abfrage der Änderungsbeschreibung. Die Emacs Integration von ShapeTools ist ein Beispiel für den „schmerzlosen" Einbau grundlegender Konfigurationsmanagementfunktionen in die gewohnten Arbeitsabläufe der Entwickler. Der Anwender muß sich nicht auf die Einhaltung bestimmter Konventionen konzentrieren — die SCM Funktionen sind „einfach da".

Eine andere Erweiterungssprache ist die *Tool Command Language (Tcl)* [20]. Tcl ist dazu entworfen, in bestehende Anwendungen nachträglich als Erweiterungssprache eingebaut zu werden. Da dies auf Programmiersprachenebene passiert, ist dazu entweder Zugriff auf die Quellen der Anwendung nötig oder zumindest auf ungebundene Objektdateien. Eine Tcl Anbindung istin der Regel mit sehr geringem Aufwand zu bewerkstelligen. Vielleicht gerade aus diesem Grund findet Tcl zur Zeit eine sehr rasche Verbreitung. So gibt es bereits Tcl Schnittstellen zum TCP/IP Netzwerkprotokoll, und zum X Window System, um nur zwei Beispiele zu nennen. Letztere, mit dem Namen Tk [21], erlaubt die sehr schnelle Entwicklung graphischer Darstellungen und interaktiver graphischer Benutzungsschnittstellen. Auch die ShapeTools Objektbasis, AtFS, wurde mit einer Tcl Schnittstelle versehen.

Das eigentliche Potential von Tcl liegt in der Zusammenfassung verschiedener Anwendungen unter einer Skriptsprache. Zum Beispiel wurde an der TU Berlin unter Verwendung der erwähnten drei Komponenten Tk, tclTCP und tclAtFS innerhalb weniger Wochen ein funktionsfähiger Prototyp einer integrierten Softwareentwicklungsumgebung auf der Basis von ShapeTools inklusive einer graphischen Benutzungsoberfläche erstellt (*Stage*).

Mit dem Tcl Ansatz soll auch der Anschluß von Datenbanksystemen an ShapeTools gelingen. Ein Ansatz hierzu ist eine Tcl Schnittstelle zum objekt–orientierten Datenbanksystem OBST [5] (*tclOBST*), die, mit tclAtFS kombiniert, wahlweisen Zugriff auf Dateisytemobjekte, Attributierte Softwareobjekte und Datenbankobjekte zuläßt.

4.3. Programmierschnittstelle

Die feinste Art der Integration ist die auf Programmiersprachenebene. ShapeTools legt alle wichtigen internen Programmierschnittstellen offen. Dies sind die Schnittstelle zum Attributierten Dateisystem, eine darauf aufsetzende Toolkit Bibliothek (*AtFStk*) und eine allgemeine ShapeTools Toolkit Bibliothek (*Sttk*). Darauf ist das Shape Programm und das gesamte ShapeTools Versionsverwaltungssystem implementiert. Sie stehen auch offen zur Anpassung oder

Implementierung zusätzlicher Funktionen.

Nicht immer bedeutet Integration auf Programmiersprachenebene einen sehr hohen Aufwand. Die Anpassung des Debuggers *gdb*, der während der vom ihm kontrollierten Ausführung eines Programms den aktuellen Quelltext des Programms anzeigt, ist minimal. Lediglich an einer Stelle, beim Lesen von Quelldateien, mußte statt des einfachen Öffnens einer Datei die Identifizierung und Herstellung der entsprechenden Version eingesetzt werden. Die Änderung betrifft etwa zwanzig Zeilen Quelltext.

5. Schlußbemerkungen

ShapeTools entstand an der Technischen Universität Berlin im Rahmen der Verbundprojekte UniBase und STONE. Beide Projekte wurden vom Bundesminister für Forschung und Technologie unter den Förderkennziffern ITS-8308 (UniBase, 1985-1989) und ITS-8902E8 (STONE, 1990-1993) gefördert. ShapeTools ist freie Software und kann auf allen gängigen UNIX Systemplattformen eingesetzt werden. Die Software, inklusive Dokumentation in elektronischer Form, ist erhältlich per anonymem 'ftp' von *coma.cs.tu-berlin.de*. Auskünfte über weitere Verbreitungskanäle und sonstige Fragen bitte elektronisch an *shape@cs.tu-berlin.de* oder schriftlich an die

Technische Universität Berlin
ShapeTools
Sekretariat FR 5-6
Franklinstraße 28/29
W-1000 Berlin 10

Literatur

1. APOLLO, ''Domain Software Engineering Environment (DSEE) Reference,'' Apollo Computer Inc., Chelmsford MA. (July 1985).
2. Brian Berliner, ''CVS II: Parallelizing Software Development,'' *Proceedings of the USENIX Winter Conference,* USENIX asc., Washington D.C. (January 1990).
3. Edward H. Bersoff, Vilas D. Henderson, and Stanley G. Siegel, ''Software Configuration Management An Investment in Product Integrity,'' Prentice Hall, Englewood Cliffs, N.J. (1980).
4. Frederick Brooks, ''The Mythical Man-Month Essays on Software Engineering,'' Addison Wesley Publ. Company, Reading, Menlo Park, London, Amsterdam (1975).
5. Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, and Walter Zimmer, ''OBST: An Overview,'' STONE Projekt, Report Nr. FZI.039.1, Forschungszentrum Informatik, Karlsruhe, Deutschland (Juni 1992).
6. Softool Corporation, ''Concepts & Facilities for CCC Products on UNIX Systems,'' *Technical Report,* Softool Corporation, Goleta, California (October 1992).
7. William Courington, ''The Network Software Environment,'' *A Sun Technical Report,* Sun Microsystems, Inc., Mountain View, CA. (May 1989).
8. Susan Dart, ''Spectrum of Functionality in Configuration Management Systems,'' *Technical Report CMU/SEI-90-TR-11 ESD-90-TR-212,* Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (December 1990).
9. J. Estublier, ''A Configuration Manager: The Adele Database of Programs,'' *Workshop on Software Engineering Environments for Programming-in-the-Large,* pp. 140-147, GTE Laboratories, Harwichport, Massachusets (June 1985).
10. Peter H. Feiler, ''Configuration Management Models in Commercial Environments,'' *Technical Report CMU/SEI-91-TR-7 ESD-9-TR-7,* Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (March 1991).
11. Stuart I. Feldman, ''MAKE - A Program for Maintaining Computer Programs,'' *Software - Practice and Experience,* 9,3, pp. 255-265 (March 1979).

12. Watts S. Humphrey, ‘‘Managing the Software Process,’’ Addison Wesley Publ. Company, Reading, Menlo Park, London, Amsterdam (1990).

13. IEEE, ‘‘IEEE Standard Glossary for Software Engineering Terminology,’’ IEEE, New York, N.Y. (February 1983).

14. Andreas Lampen, ‘‘Advancing Files to Attributed Software Objects’’ in *Proceedings of the Winter 1991 USENIX Conference,* pp. 219−229, USENIX Association, Berkeley (CA), USA (January 1991).

15. Andreas Lampen and Axel Mahler (Ed.), ‘‘ShapeTools,’’ *Forschungsberichte des Fachbereichs Informatik, Bericht 92-14,* Fachbereich Informatik, Berlin (May 1992).

16. Andreas Lampen, ‘‘Version Binding in ShapeTools,’’ STONE Projekt, Report Nr. TUB.0023.01, Technische Universität Berlin, Berlin, Germany (May 1992).

17. Manny M. Lehman and L. Belady, ‘‘Program Evolution. Process of Software Change.,’’ Academic Press, London, England (1985).

18. Thomas M. Morgan, ‘‘Configuration Management and Version Control in the Rational Programming Environment,’’ *Proceedings of the Ada-Europe International Conference, Munich 1988,* pp. 17-28, Cambridge University Press, Cambridge, England (June 1988).

19. Deutsches Institut für Normung, ‘‘Qualitätssicherungssysteme − Modell zur Darlegung der Qualitätssicheung in Design/Entwicklung, Produktion, Montage und Kundendienst,’’ Beuth Verlag, Berlin, Deutschland (Mai 1990).

20. John K. Ousterhout, ‘‘Tcl: An Embeddable Command Language’’ in *Proceedings of the Winter 1990 USENIX Conference,* USENIX Association, Berkeley (CA), USA (January 1990).

21. John K. Ousterhout, ‘‘An X11 Toolkit Based on the Tcl Language’’ in *Proceedings of the Winter 1991 USENIX Conference,* pp. 105−115, USENIX Association, Berkeley (CA), USA (January 1991).

22. Roger S. Pressman, ‘‘Software Engineering − A Practitioner’s Approach, 2nd Ed.,’’ McGraw-Hill Book Company, New York (1987).

23. Marc J. Rochkind, ‘‘The Source Code Control System,’’ *IEEE Transactions on Software Engineering,* SE-1, pp. 364-370 (1975).

24. Atria Software, Inc., ‘‘ClearCase Concepts Manual,’’ *Document Number 3000-002-A,* Atria Software, Inc., Natick, Massachusetts 01760 (1992).

25. Ian Sommerville, ‘‘Software Engineering, 4th Ed.,’’ Addison Wesley Publ. Company, Reading, Menlo Park, London, Amsterdam (1992).

26. Software Maintenance and Development Systems, Inc., ‘‘Aide-De-Camp Software Management System Product Overview,’’ Software Maintenance and Development Systems, Inc., Concord, Massachusetts (1992).

27. Walter F. Tichy, ‘‘RCS - A System for Version Control,’’ *Software - Practice and Experience,* 15,7, pp. 637-654 (July 1985).

28. Walter F. Tichy, ‘‘Tools for Software Configuration Management,’’ *Proceedings of the International Workshop on Software Version and Configuration Control,* pp. 1-20, German Chapter of the ACM, Grassau, FRG (January 1988).