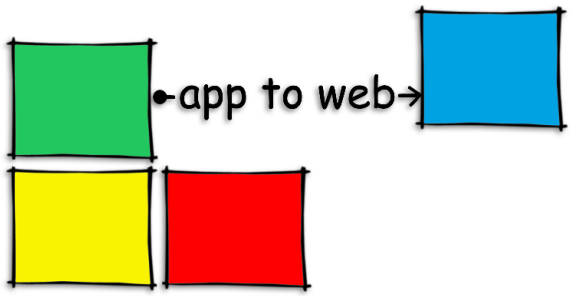
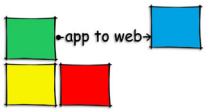


Strategien zur Migration von Altsystemen in komponenten-orientierte Systeme

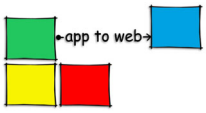
Autoren:
Christoph Andriessens
Markus Bauer
Holger Berg
Jean-François Girard
Michael Schlemmer
Olaf Seng

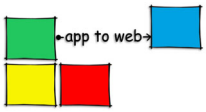
Kennung	MIGRA
Klassifikation	öffentlich
Version	1.0
Datum	20.12.2002

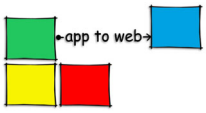
Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministerium für Wirtschaft und Technologie unter den Förderkennzeichen 16IN0047 und 16IN0048 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

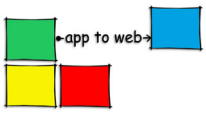


Inhaltsverzeichnis









1 Einleitung

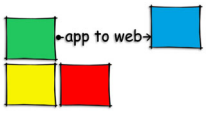
Moderne Webanwendungen können in aller Regel nicht vollständig neu entwickelt werden, sondern es müssen bereits existierende Altsysteme zu Webanwendungen umgebaut oder Teile aus ihnen mit neuen webfähigen Systemen verknüpft werden. Das in den Altsystemen verborgene Know-How, ihre Rolle bei der Abwicklung der Geschäftsprozesse von Unternehmen und ihre im langjährigen Einsatz gereiften Implementierungen tragen dazu bei, dass diese erfolgreich eingesetzten Altsysteme mit Webfunktionalität ausgestattet werden sollen, anstatt neue, webfähige Anwendungen mit ähnlichem Funktionsumfang von Grund auf neu zu entwickeln.

Dazu sind sehr häufig tiefgreifende Änderungen und *Restrukturierungen* an den Altsystemen erforderlich. So entstehen moderne Webanwendungen in vielen Fällen auf Basis moderner Komponenteninfrastrukturen (CORBA, EJB, (D)COM). Bestehender Code muss daher an diese Komponententechnologien angepasst werden und mit den neu entstehenden webfähigen Systemteilen integriert werden. Zur Unterstützung solcher Aktivitäten sind Reengineering-Techniken hilfreich.

Daher geht es im Arbeitspaket *Migration zu Webanwendungen* des Projektes APPLICATION2WEB darum, Reenginering-Techniken zu entwickeln, zu dokumentieren und zu erproben, die die Migration zu komponenten-orientierten Webanwendungen unterstützen. Die Ergebnisse dieses Arbeitspaketes fasst das vorliegende Dokument zusammen. Es gliedert sich in drei grobe Teile:

- *Redokumentation von Altsystemen*
- *Identifikation von Komponentenkandidaten in Altsystemen*
- *Transformation von Komponentenkandidaten in Komponenten*

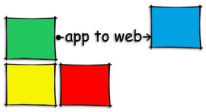
Bei der *Redokumentation von Altsystemen* geht es vornehmlich darum, wie die Grundlagen für Migrations- und Reengineering-Aktivitäten gelegt werden können: Ziel ist es, zu verstehen, wie ein Altsystem aufgebaut ist, und wie es funktioniert. Oft existiert jedoch nur wenig dokumentiertes Wissen über Architektur, Implementierung und Funktionsweise des Altcodes. Daher müssen wir dieses Wissen erst mühsam mit Hilfe von Experten und durch aufwendige Analysen aus dem Quellcode der Systeme zurückgewinnen. Zudem geht es darum, diejenigen Teile zu identifizieren, die später von den Restrukturierungs- und Erweiterungsmaßnamen betroffen sind. Dieses neu gewonnene Wissen muss geeignet modelliert und dokumentiert werden und ist unabdingbare Voraussetzung für sinnvolle Reengineering-Arbeiten. Geeigneten Methoden und Werkzeugen zur Durchführung dieser Aktivitäten widmet sich der erste Teil des vorliegenden Dokumentes.



Zum Kern der Aufgabe, Altsysteme in moderne, komponenten-orientierten Web-Anwendungen zu überführen, kommen wir im zweiten Teil dieses Dokuments. Bei der *Identifikation von Komponentenkandidaten* geht es vornehmlich darum, wie aus dem Quellcode bereits existierender Anwendungen mit Hilfe unterstützender Techniken gemäß bestimmter Kriterien geeignete Komponentenkandidaten ermittelt werden können. In der Regel wird dazu der gesamte Quellcode der zu migrierenden Anwendung mit Hilfe von Clusterverfahren untersucht und in geeignete Fragmente zerlegt. Diese Fragmente bilden dann mögliche Kandidaten für die später zu verfeinernden Komponenten. Solche aus dem Altsystem gewonnenen Komponenten bilden die Voraussetzung dafür, dass dessen Funktionalität zumindest in Teilen in komponenten-orientiert aufgebaute Webanwendungen integriert werden kann.

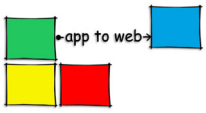
Natürlich können wir die so bestimmten Komponentenkandidaten nicht unmittelbar in unsere neu zu erstellende komponenten-orientierte Webanwendung einbringen. Zuvor müssen wir diese Komponentenkandidaten nämlich in ordentliche, wohldefinierte Komponenten transformieren. Den zu solchen Überarbeitungsaufgaben notwendigen Methoden und Werkzeugen widmet sich der dritte Teil *Transformation von Komponentenkandidaten in Komponenten* dieses Dokumentes.

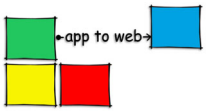
Ein Hinweis für den Leser: Alle drei Teile dieses Dokumentes sind weitgehend in sich abgeschlossen, so daß sie prinzipiell auch getrennt voneinander gelesen und verstanden werden können. Trotzdem empfiehlt es sich für Leser, die mit Reengineering-Fragestellungen weniger vertraut sind, die Teile dieses Dokumentes sequentiell zu lesen – insbesondere der erste Teil definiert Grundbegriffe, die für das weitere Verständnis des Textes hilfreich sind. Für diejenigen Leser, die sich dennoch ausgewählten Aspekten der Thematik dieses Dokumentes widmen wollen, enthalten die Einführungskapitel aller drei Teile als zusätzliche Lesehilfe detaillierte Angaben zu Aufbau und Inhalt der einzelnen Teile. Die Literaturangaben haben wir ebenfalls für jeden der drei Themenblöcke getrennt zusammengestellt.



I Redokumentation von Altsystemen

Christoph Andriessens
Markus Bauer
Jean-François Girard
Olaf Seng





2 Einführung

Bei der *Redokumentation von Altsystemen* geht es vornehmlich darum, wie die Grundlagen für diese Migrations- und Reengineering-Aktivitäten gelegt werden können: Ziel ist es, zu verstehen, wie ein Altsystem aufgebaut ist, und wie es funktioniert. Oft existiert jedoch nur wenig dokumentiertes Wissen über Architektur, Implementierung und Funktionsweise des Altcodes. Daher müssen wir dieses Wissen erst mühsam mit Hilfe von Experten und durch aufwendige Analysen aus dem Quellcode der Systeme zurückgewinnen.Zudem geht es darum, diejenigen Teile zu identifizieren, die später von den Restrukturierungs- und Erweiterungsmaßnamen betroffen sind. Dieses neu gewonnene Wisssen muss geeignet modelliert und dokumentiert werden und ist unabdingbare Voraussetzung für sinnvolle Reengineering-Arbeiten.

Wir beginnen in in Kapitel 2 zunächst mit einer kurzen Einführung in die Disziplin des Software-Reengineerings und definieren ein Phasenmodell zur Durchführung von Reengineering-Aktivitäten. Kapitel 3 definiert Vorgehensweisen und Techniken für zur Redokumentation von Altsystemen und deckt damit die ersten Phasen des Vorgehensmodells ab. Diese Techniken diesen dazu, ein Altsystem mit Hilfe abstrakter Modelle zu beschreiben und anhand geeigneter Analysen mehr über seinen Aufbau und seine Funktionsweise zu erlernen. Kapitel 4 stellt Werkzeuge und Werkzeugprototypen vor, die im Rahmen der Redokumentation von Altsystemen von Interesse sind. Ein umfangreiches Literaturverzeichnis bietet zahlreiche Möglichkeiten zur weiterführenden Vertiefung des Themas.

2.1 Grundlegende Begriffe

Reengineering ist nach IEEE die Untersuchung eines Systems mit dem Ziel, ihm eine neue Form zu geben, und die anschließende Implementierung der neuen Form[1]. Ergänzend dazu führen Chikofsky und Cross viele Standarddefinitionen ein, die in der Reengineering-Community benutzt werden [CI90]. Sie beschreiben die wichtigsten Reengineering-Aktivitäten anhand eines allgemeinen Modells, welches in Abbildung 1 wiedergegeben ist.

1 "the examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

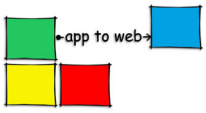
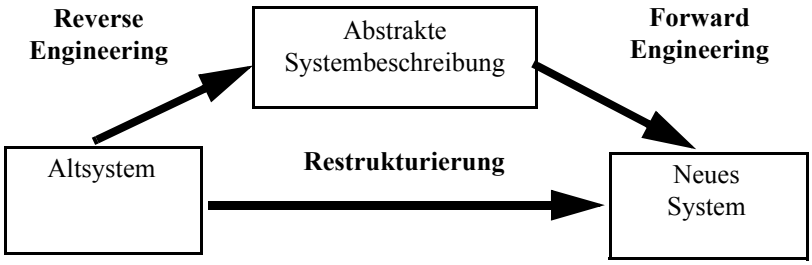


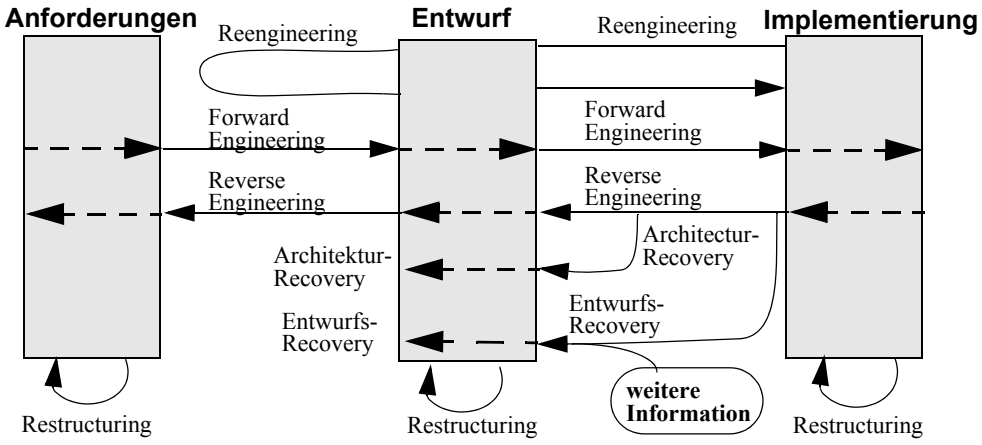
Abbildung 1: Reengineering nach Chykofsky und Cross



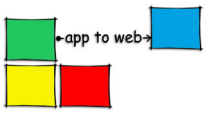
Es zeigt die beiden wichtigsten Vorgehensweisen im Reengineering: *Restrukturierung*, bei der das System auf gleicher Ebene modifiziert wird, und einen zweiphasigen Prozess, bei dem zunächst durch *Reverse Engineering* eine abstrakte Systembeschreibung erstellt wird und anschließend im *Forward Engineering* das neue System gemäß den Modifikationen auf abstrakter Ebene entsteht.

In den meisten Fällen konzentrieren sich Reengineering-Projekte auf bestimmte Aspekte eines Systems, z.B. die Datenspeicherung in Dateien, modifizieren diese Aspekte, ersetzen z.B. flache Dateien durch eine Datenbank, und erhalten die nicht betroffenen Teile der Implementierung. Falls Reverse Engineering angewandt wird, wird es sich darauf konzentrieren, Abstraktionen der relevanten Systemaspekte und ihre Abhängigkeiten mit dem Restsystem zu extrahieren. Somit können die Modifikationen auf der abstrakten Ebene durchgeführt und dokumentiert werden, bevor das Forward Engineering beginnt.

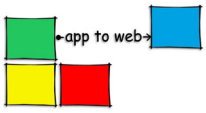
Abbildung 2: Reengineering auf verschiedenen Abstraktionsebenen



Reengineering-Projekte werden mit den unterschiedlichsten Zielen und aus den verschiedensten Gründen durchgeführt. Dennoch lassen sich einige Hauptstoßrichtungen identifizieren, die wir im folgenden kurz zusammenstellen wollen:



- *Integration neuer funktionaler Anforderungen*. Softwaresysteme spielen oft eine besonders wichtige Rolle in Unternehmen, beispielsweise weil sie die grundlegenden Geschäftsprozesse eines Unternhemens abbilden. Häufig sind diese Systeme umfangreich und komplex und zu ihrer Entwicklung waren große Investitionen erforderlich, so dass sie nicht einfach durch Neuimplementierungen abgelöst werden können. Trotzdem müssen durch sie neue inhaltliche Anforderungen abgedeckt werden. Daher sind dann Erweiterungsarbeiten an diesen Systemen erforderlich, die oft umfangreiche Strukturänderungen der Software notwendig voraussetzen bzw. nach sich ziehen.
- *Modularisierung.* Ziel ist hierbei die Zerlegung des Systems in entkoppelte Teile (Subsysteme), welche getrennt (weiter-)entwickelt, gepflegt, getestet und ausgeliefert werden können. Oft besteht eine solche Zerlegung bereits – schließlich haben sich auch die ursprünglichen Entwickler des Systems in aller Regel Gedanken über ein gutes Systemdesign gemacht. Allerdings verwässern ursprüngliche Subsystemzerlegungen leicht, wenn das System über längere Zeit immer wieder erweitert und an neue Anforderungen angepaßt wurde.
- *Performanceverbesserungen.* Häufig sind Performanceprobleme in Softwaresystemen der Ausgangspunkt für Reengineering-Projekte. Oft werden Verbesserungsmaßnahmen zur Performancesteigerung mit weiteren Reorgansiationsvorhaben verknüpft.
- *Anpassung auf neue Technologien und Plattformen.* Bei Systemen, die bereits länger im Einsatz sind, und die nicht einfach durch Neuimplementierungen ersetzt werden können sind oft Anpassungen an neue Technologien erforderlich. Solche Technologieanpassungen können unterschiedliche Ausprägungen annehmen: Migration der Hard- und Softwareablaufumgebung, Wechsel der Programmiersprache bzw. des Programmierparadigmas (Übergang von prozeduralem Code zu objekt-orientiertem Code), Unterstützung neuer Plattformen, Anbindung externer Systeme und dergleichen. Auch die Migration zu internet- bzw. webfähigen Anwendungen stellt einen Technologiewechsel dar.
- *Extraktion von Know-How.* Softwaresysteme enthalten häufig umfangreiches Wissen über die Geschäftsprozesse eines Unternehmens oder andere unternehmenskritische Informationen, die so nirgends explizit dokumentiert sind. In solchen Fällen ist man häufig daran interssiert, dieses Know-How aus dem Quellcode der Systeme wieder zu extrahieren, beispielsweise um damit die Neuentwicklung von Nachfolgesystemen zu vereinfachen oder um die zugrundeliegenden Prozesse an neue Rahmenbedingungen des Unternehmens anpassen zu können.

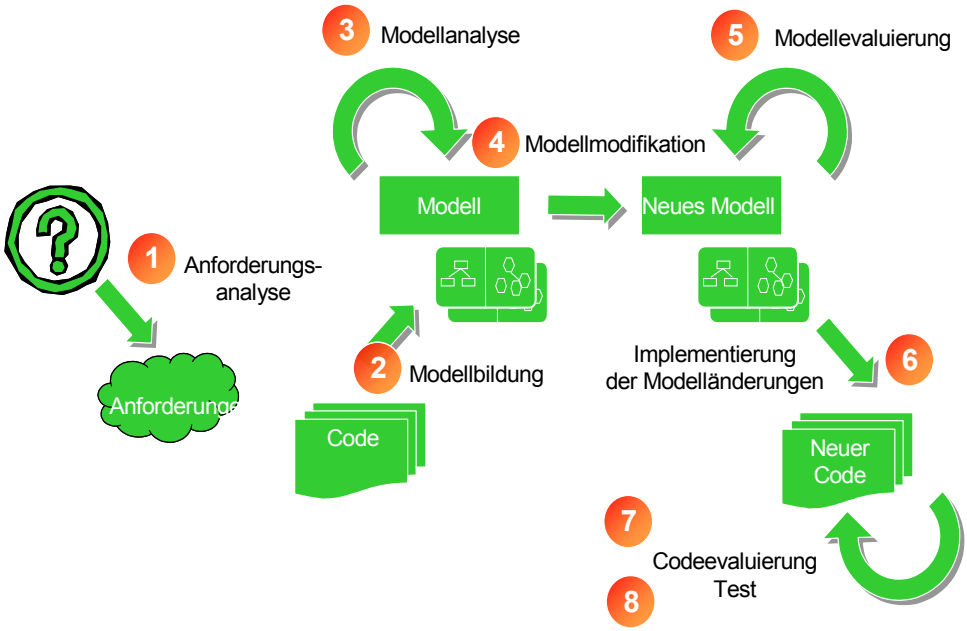


2.2 Phasenmodell

Aufgrund der verschiedenen möglichen Ziele des Reengineering wurden viele spezifische Modelle und Vorgehensweisen entwickelt. So haben beispielsweise Kazman et. al [KWC98] das Hufeisenmodell (horseshoe model) eingeführt, um die Architektur eines Systems zu extrahieren und Hinweise für Modifikationen zu geben. Für die Migration eines prozeduralen Systems zu einem objektorientierten System entwickelten Klösch und Gall ein entsprechendes Modell [GK95].

Trotz der Vielzahl der verschiedenen Vorgehensweisen lassen sich jedoch einige grundlegende Schritte identifizieren, die für die erfolgreiche Abwicklung von Reengineering-Aufgaben erforderlich sind. Im folgenden stellen wir diese Schritte anhand eines Phasenmodells für Reengineering-Projekte vor.

Abbildung 3: Allgemeines Phasenmodell für Reengineering-Projekte



2.2.1 Anforderungsanalyse

Wie wir im vorigen Abschnitt bereits bemerkt haben, gibt es die verschiedenartigsten Zielsetzungen für Reengineering-Projekte. Im Rahmen der *Anforderungsanalyse* müssen die konkreten Zielsetzungen für die Reengineering-Maßnahme genau festgelegt werden, da nur so erreicht werden kann, dass die später durchzuführenden

Reverseengineering- und Restrukturierungsarbeiten effizient und zielgerichtet durchgeführt werden können. So können wir durch eine möglichst exakte Festlegung der Reengineering-Ziele oft erreichen, dass wir in den folgenden Phasen des Reengineering-Prozesses nur wirklich relevante Teile des Systems untersuchen, verstehen und verändern müssen.

Diese Analyse definiert konkrete Ziele, Erfolgskriterien, Randbedingungen und prüft die Machbarkeit.

Im Rahmen der Anforderungsanalyse analysieren wir, *ob* und *warum* eine Reengineeringmaßnahme notwendig und sinnvoll ist. Dabei untersuchen wir, welche funktionalen und nicht-funktionalen Anforderungen vom bestehenden System nicht oder nur unzureichend erfüllt werden. Daraus leiten wir *konkrete Ziele* für unsere Reengineering-Aktivität ab. Zusätzlich definieren wir *Erfolgskriterien* und *Randbedingungen*, mit deren Hilfe wir während der Arbeiten und danach überprüfen können, ob wir unsere Zielsetzung erfüllen. Im Rahmen der Anforderungsanalyse sollte ebenfalls die *Machbarkeit* der Reengineeringziele überprüft werden.

2.2.2 Modellbildung

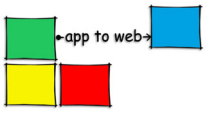
Mit Hilfe der *Modellbildung* verschaffen wir uns das nötige Wissen über den Ist-Zustand des Systems. Oft sind Softwaresysteme unzureichend dokumentiert. Da die ursprünglichen Entwickler häufig nicht mehr verfügbar sind, muss ein grundlegendes Verständnis über das System und seine Zusammenhänge wieder neu erarbeitet und dokumentiert werden *(Redokumentation)*.

Ziel der Modellbildungsphase ist das Erarbeiten geeigneter Modelle des Systems, die seinen Aufbau, die inneren Zusammenhänge und seine Wirkungsweise geeignet wiedergeben und dokumentieren. Die Aktivitäten der folgenden Phasen des Reengineeringzyklus profitieren stark von umfangreichen und aussagekräftigen Modellen.

Entscheidend für eine gute Modellbildung ist, dass möglichst viele aussagekräftige Informationen aus allen verfügbaren Quellen herangezogen werden und zu einem zusammenhängenden Ganzen verdichtet werden. Typische Informationsquellen, die zur Modellbildung herangezogen werden können, sind:

- Entwickler und Domänenexperten
- Bestehende Dokumentation (Entwicklerdokumentation in Form von Anforderungsspezifikationen, Architekturbeschreibungen und Designdokumenten; Benutzerhandbücher)
- Quelltexte des Systems
- Weitere maschinenlesbare Informationen, beispielsweise Datenbank-Schemata, Deployment-Deskriptoren,...

In vielen Fällen wird man aber auf eine gründliche Exploration der Implementierung des Systems in Form von Quelltexten nicht verzichten können. Diese ist jedoch zeitaufwendig und teuer. Mit den Techniken, die in diesem Dokument beschrieben wer-



den (siehe Kapitel 3), kann die Analyse verkürzt und effizienter gestaltet werden. Das Grundprinzip dieser Techniken besteht darin, den Software-Ingenieur durch Werkzeuge so zu unterstützen, dass er die umfangreichen Quelltexte eines Systems schneller sichten und verstehen kann, indem die für ihn relevanten Informationen automatisiert extrahiert (siehe Abschnitt 3.2), verdichtet und aufbereitet (siehe Abschnitt 3.4 bis Abschnitt 3.6) werden können. Dabei müssen Konzepte der Anwendungsdomäne und das verfügbare Entwickler-Know-Hows geeignet mit Konstrukten und Fragmenten des Quellcodes in Beziehung gebracht und verknüpft werden (siehe Abschnitt 3.7).

Nach Abschluß der Modellbildung sollen aussagekräftige Modelle entstehen, die das System auf unterschiedlichen Abstraktionsebenen beschreiben. Dadurch kann der Software-Ingenieur bei weiteren Arbeiten am System vom oftmals verwirrenden Detailreichtum der Quelltexte des Systems abstrahieren.

2.2.3 Modellanalyse

Im Rahmen der *Modellanalyse* identifizieren wir mit Hilfe der zuvor erstellten Modelle diejenigen Teile des Systemes, die modifiziert werden müssen, um die Ziele der Reengineering-Maßnahmen zu erreichen. Solche kritischen Stellen müssen dann im weiteren Verlauf des Reengineering-Projektes genauer analysiert werden. In aller Regel müssen wir aufgrund dieser Analyse neue, geeignetere Zielstrukturen entwerfen, in die wir die bestehenden Strukturen im nächsten Schritt, der *Modellmodifikation*, überführen.

Die Modellanalyse ist stark mit der Modellbildung verzahnt. Oftmals erfordern bestimmte Untersuchungen, die wir in der Modellanalyse durchführen wollen, zusätzliches, tiefergehendes Wissen über das System oder bestimmter Teile davon. Diese Information müssen wir uns dann nachträglich beschaffen, indem wir im Rahmen weiterer Modellbildungsaktivitäten unsere bestehenden Modelle erweitern oder durch neue Modelle ergänzen.

Wie die Modellbildung profitiert die Modellanalyse ebenfalls stark von automatisierbaren Reverse-Engineering-Techniken, wie wir sie in Kapitel 3 beschreiben: Visualisierungen (siehe Abschnitt 3.4), Metriken (siehe Abschnitt 3.6, [LK94]). Eine wichtige Rolle spielen jedoch auch Techniken zur Abhängigkeitsanalyse und Designanfragen [Ciu99], [Ciu02].

2.2.4 Modellmodifikation

Im Rahmen der *Modellmodifikation* gestalten wir das System oder relevante Teile davon um. Durch die Umgestaltung bestehender Systemstrukturen in verbesserte, neue Strukturen erstreben wir so ein System, welches die Ziele unseres Reenginne-

ring-Projektes, die wir im Rahmen der Anforderungsanalyse spezifiziert haben, erfüllt.

Modifikationen an bestehenden Softwaresystemen gestalten sich in aller Regel schwierig. Zum einen ist ein detailliertes Verständnis des betreffenden Systemteils selbst erforderlich, um überhaupt Änderungen an dessen Strukturen oder an seiner Funktionalität durchführen zu können. Zum anderen ziehen oft schon kleinere Änderungen an bestimmten Fragmenten des Systems zahlreiche weitere Änderungen in anderen Systemteilen nach sich.

Um Umstrukturierungen zuverlässiger und effizienter durchführen zu können, empfiehlt es sich, folgende Grundprinzipien zu beachten:

- *Restrukturierung auf Basis geeigneter abstrahierender Modelle.* Umstrukturierungen sollten stets auf einem geeigneten Abstraktionniveau stattfinden, d.h. sie sollten zunächst auf Basis der zuvor erstellten Modelle geplant und durchgeführt werden. Erst in einem weiteren Schritt, im Rahmen der *Implementierung der Modelländerungen*, sollen diese Umstrukturierungen implementiert werden.
- *Verwendung von Werkzeugunterstützung.* Restrukturierungen lassen sich leichter durchführen, wenn entsprechende Werkzeuge zur Verfügung stehen. Restrukturierungswerkzeuge tragen dazu bei, dass Änderungen am System konsistent durchgeführt werden.

Leider sind leistungsfähige Werkzeuge, die Umstrukturierungen gemäß dieser Grundprinzipien zuverlässig ermöglichen, noch Gegenstand der Forschung, so dass die Restrukturierungen vielfach von Hand durchgeführt werden müssen. In den letzten Jahren jedoch wurden einige theoretische Grundlagen zur automatisierten Restrukturierung von Softwaresystemen gelegt, beispielsweise durch die Spezifikation typischer Restrukturierungen in Form von *Refactorings* [Opd92] [Fow99]. Nach und nach werden diese Techniken auch anhand in entsprechenden Werkzeugen unterstützt. Beispiele für solche Werkzeuge sind der Refactoring-Browser für Smalltalk [RBJ97], Inject/J [inj], bzw. JBuilder [jbu] für Java.

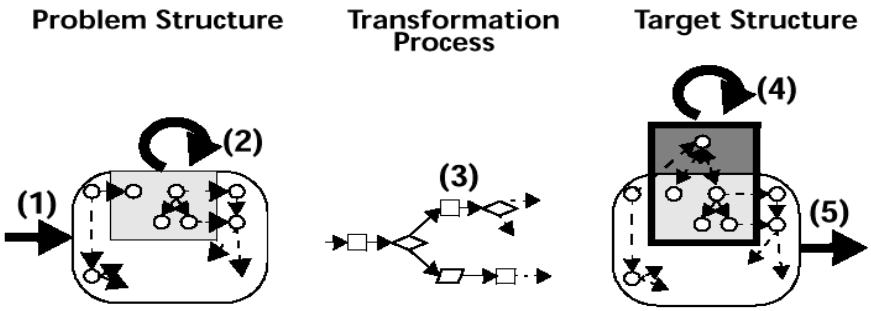
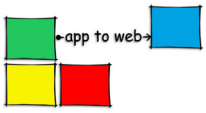


Abbildung 4: Restrukturierungen auf der Ebene abstrahierender Modelle



2.2.5 Modellevaluierung

Bevor die Umstrukturierungen, die wir im Rahmen der Modellmodifikation auf der Ebene abstrakter Modelle vorgenommen haben, in die Praxis umsetzen und implementieren, müssen wir überprüfen, ob unser neues Modell die während der Anforderungsanalyse definierten Reengineering-Ziele und Randbedingungen erfüllt. Wir nennen diese Phase *Modellevaluierung*.

2.2.6 Implementierung der Modelländerungen und Konsolidierung

Nachdem wir die Modifikationen am System, die zur Erreichung unserer Reengineering-Ziele erforderlich sind, auf Modellebene vorgenommen haben, müssen diese auf die Ebene der Implementierung übertragen werden, d.h. die Änderungen auf Modellebene müssen auf Quellcodeebene nachgeführt werden. Dieser Arbeitsgang, die Implementierung der Modelländerungen, ist üblicherweise aufwendig und fehleranfällig. Daher stützen wir uns dabei , wie in Abschnitt 2.2.4 bereits diskutiert, so weit möglich auf Werkzeuge.[1] In vielen Fällen müssen wir jedoch auch Modelländerungen von Hand in passenden Quellcode umsetzen.

Neben Änderungen am Quellcode, die sich unmittelbar aus den Modelländerungen ergeben, müssen in aller Regel weitere Anpassungen vorgenommen werden. Dies betrifft sowohl externe Systemteile, die auf den modifizierten Code zugreifen, als auch andere Bausteine eines Softwaresystems: Datenbank-Schemata, Deployment-Deskriptoren und dergleichen.

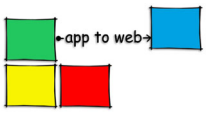
Wir beachten ferner, dass wir neben Änderungen an den Modellen und am Quellcode des Systems auch die entsprechende Dokumentation aktualisieren müssen.

2.2.7 Codeevaluierung und Test

Von entscheidender Bedeutung für den Erfolg von Reengineering-Maßnahmen ist es, dass dabei einerseits die Qualität des Systems nicht beeinträchtigt wird, andrerseits jdie Ziele und Kriterien, die im Rahmen der Anforderungsanalyse etabliert wurden, erfüllt werden.

Durch *Codeevaluierungen und Tests* müssen wir nach der Durchführung von Modifikationen des Systems prüfen, ob diese Ansprüche an das System gewährleistet sind. Wesentlich dafür sind geeignete Teststrategien [Bin99]. Beispielsweise kann mit

1 Die meisten Restrukturierungswerkzeuge vereinen die beiden Schritte Modellmodifikation und Implementierung der Modelländerungen auf Quellcodeebene. Der Software-Entwickler spezifiziert dabei die Restrukturierungen anhand mehr oder minder abstrakter Modelle/Sichten, die Werkzeuge führen dann entsprechende Änderungen (halb-)automatisch auf Quellcodeebene durch.



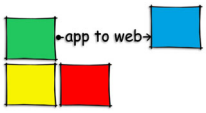
Hilfe automatisierter Regressionstests überprüft werden, ob neue, modifizierte Teile des Systems dasselbe funktionale Verhalten aufweisen, wie die entsprechenden Teile des Ausgangssystems.

2.3 Reengineering in Application2Web

Wir haben zu Beginn dieses Kapitels bereits festgestellt, dass sich die Migration bestehender Anwendungen zu Webanwendungen als Reengineering-Aufgabe betrachten lässt. Im Projekt APPLICATION2WEB erarbeiten wir daher entsprechende Techniken für die einzelnen Phasen unseres Reenginnering-Phasenmodells, die uns dabei unterstützen können, bestehende Anwendungen webfähig zu machen, oder Teile bestehender Anwendungen in neue Webanwendungen einzubringen.

In diesem Dokument beschreiben wir im folgenden Techniken, die insbesondere für die Phasen *Modellbildung* und *Modellanalyse* von nutzen sind – uns also dabei unterstützen, Wissen über ein Softwaresystem, seine Strukturen und seine Funktionsweise zu erschließen und zu konsolidieren. Dieses Wissen ist die Grundvoraussetzung dafür, dass wir bestehende Anwendungen oder Teile davon umgestalten können, um ihre Funktionalität über das Web vefügbar zu machen.

Weitere Inhalte der Reenginnering-Aktivitäten in APPLICATION2WEB decken die späteren Phasen des Reenginnering-Phasenmodells ab. Die Entwicklung moderner Webanwendungen erfolgt nach den Prinzipien der komponenten-orientierten Softwarekonstruktion [Szy98] [ABB+01], die meisten Infrastrukturen und Architekturen für Webanwendungen sehen Komponenten als zentrale Bausteine der Anwendungsentwicklung vor [ejb]. Daher müssen bestehende Anwendungen oder Teile daraus in das komponenten-orientierte Paradigma überführt werden. Nach der Redokumentation und Modellierung des bestehenden Codes muss das System in Komponenten zerlegt werden. Hierzu sind zunächst im Rahmen der *Modellanalyse* Techniken erforderlich, um geeignete Kandidaten für Komponenten im bestehenden Code identifizieren (Arbeitspaket 5.2, *Kriterien und Verfahren zur Indentifikation von Komponenten in Altsystemen*). Im Rahmen der *Modellmodifikation* mit anschließender *Implementierung der Modelländerungen* erfolgt dann die Umarbeitung bestehenden Codes zu Komponenten (Arbeitspaket 5.3, *Verfahren zur Umwandlung der identifzierten Komponentenkandidaten zu Komponenten*).



3 Techniken zur Redokumentation von Altsystemen

In diesem Kapitel beschreiben wir einzelne Techniken und Verfahren, die für die Redokumentation von Softwaresystemen von Nutzen sind.

3.1 Allgemeine Vorgehenweise

Verbunden mit den Techniken und Verfahren ist eine allgemeine Vorgehensweise zur Redokumentation, die wir in diesem Abschnitt kurz beschreiben.

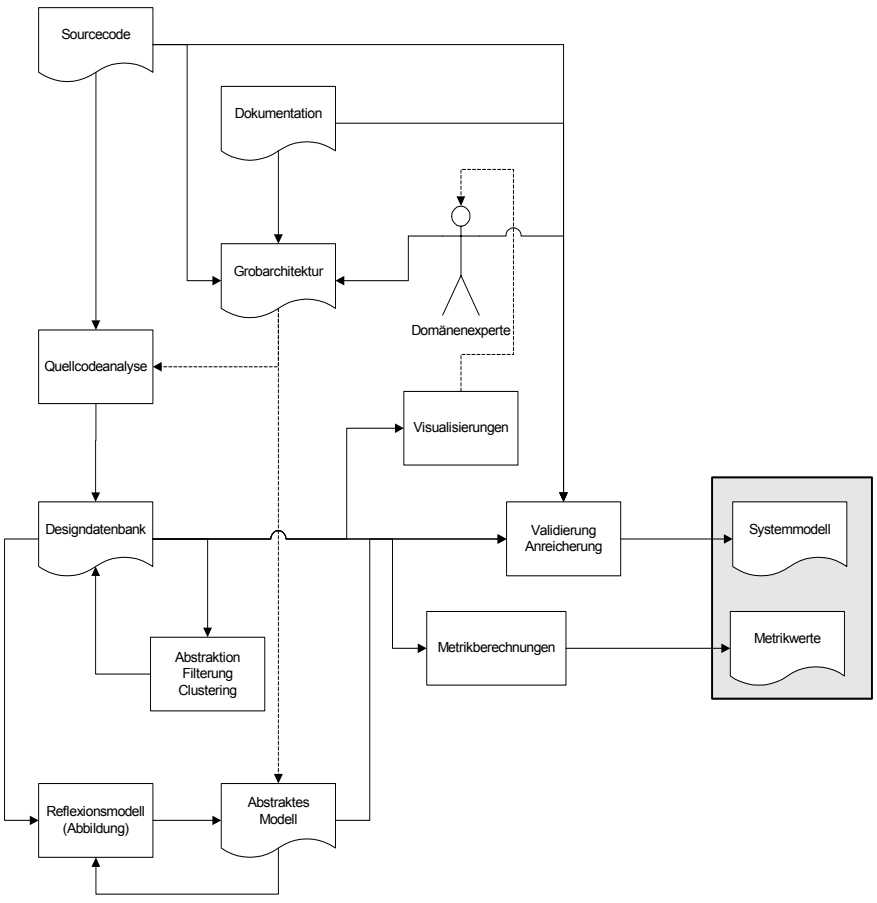
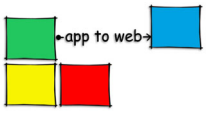


Abbildung 5: Vorgehensweise zur Redokumentation mit beteiligten Artefakten und Vorgängen



Zunächst empfiehlt es sich mit Hilfe der verfügbaren Dokumentation, des Quellcodes und gegebenenfalls eines Domänenexperten eine grobe Architekturbeschreibung zu erstellen, die als Rahmen für die weiteren Schritte dient und das zu redokumentierende System mit seiner Umgebung in Beziehung setzt. Sie sollte zumindest die physikalischen Systemteile identifizieren, die später genauer analysiert werden.

Als Basis für weitere Schritte und als ein erstes Systemmodell wird man in der Regel durch Quellcodeanalyse (siehe Abschnitt 3.2 und Abschnitt 3.3) eine Designdatenbank aufbauen, die die wesentlichen Entitäten und Abhängigkeiten des Systems enthält. Durch Filterung und Abstraktion kann deren Umfang und Detaillierungsgrad reduziert werden. Dies ist insbesondere bei umfangreichen Systemen unerlässlich. Clustering (siehe Abschnitt 3.5) sollte dann eingesetzt werden, wenn grobgranularere Einheiten als die des Quellcodes (Klassen, Packages), z.B. Komponenten oder Subsysteme identifiziert werden sollen. Für die folgenden Arbeiten im Arbeitspaket 5 des Projekts sind besonders diese Komponentenkandidaten interessant.

Eine Alternative oder Ergänzung zur Extraktion einer Designdatenbank aus dem Quellcode stellen Reflexionsmodelle dar (siehe Abschnitt 3.7). Sie eignen sich besonders dann, wenn die abstrakte Architektur des Systems vorher schon zumindest grob bekannt ist und sich die Analyse auf bestimmte Abhängigkeiten beschränkt.

Zur Visualisierung (siehe Abschnitt 3.4) der wiedergewonnenen Systemstrukturen der Designdatenbank oder des abstrakten Modells der Reflexiomsmodelle eignen sich sehr natürlich Graphen. Durch sie können Domänenexperten recht schnell beurteilen, ob die wiedergewonnenen Informationen korrekt sind. Visualisierungen der Graphen lassen bei geeignetem Layout sogar mit bloßem Auge mögliche Komponentenbildungen oder Problemstellen erkennen.

Nach der Validierung des (halb-)automatisch gewonnenen Informationen über das untersuchte System kann das Modell weiter mit Informationen angereichert werden, die automatisch nicht zu gewinnen sind, wie z.B. Aufrufe aus Binärcode heraus an das System oder Instanzierungen von Objekten deren Klasse namentlich in Konfigurationsdateien spezifiziert ist.

Dieses validierte und angereicherte Modell, das Systemmodell, bildet zusammen mit Metrikwerten (siehe Abschnitt 3.6) und gegebenenfalls der anfangs erstellten Grobarchitektur das Ergebnis der Redokumentation. Metrikwerte sind Kennzahlen für das zu untersuchende System, die Rückschlüsse auf (Qualitäts-)Eigenschaften des Systems zulassen und somit ebenfalls zum Systemverständnis und vor allem auch zur Bewertung des System beitragen. Bestimmte Metriken dienen auch als Kriterium zur Bildung von Clustern.

3.2 Quellcodeanalyse, Fact Extraction

Eine Aktivität, die für die Redokumentation von Systemen von wesentlicher Bedeutung ist, besteht darin, (Struktur-)informationen über das System aus seinem Quellcode zu gewinnen. Wir nennen diese Aktivität *Fact Extraction*. Im Prinzip geht es dabei darum, eine Designdatenbank mit Strukturinformationen zu füllen, indem der Quellcode des Systems mit Hilfe von Übersetzerbautechniken analysiert wird.

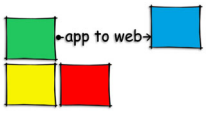
Wir werden im nächsten Abschnitt beschreiben, an welcher Art von Informationen wir üblicherweise interessiert sind und wie wir diese in Form eines Metamodells beschreiben können. In diesem Abschnitt begnügen wir uns damit, festzuhalten, dass es sich bei den Informationen, die wir von einem System für eine Analyse seiner Strukturen benötigen, um die Einheiten (*Entitäten*) aus denen das System aufgebaut ist, und die Beziehungen (*Relationen)* zwischen diesen Entitäten handelt. Entitäten eines Softwaresystems umfassen beispielsweise seine Methoden, Klassen, Pakete. Mögliche Beziehungen sind Aufrufbeziehungen zwischen Methoden, Vererbungsbeziehugnen zwischen Klassen, Import-Beziehungen zwischen Paketen und viele weitere mehr.

Zur Extraktion der Fakten existieren verschiedene Ansätze, die jeweils über Vor- und Nachteile verfügen. Deshalb muss im konkreten Anwendungsfall entschieden werden, welche Technik eingesetzt werden sollte. Die im folgenden beschriebenen Ansätze zur Gewinnung von Informationen über den Quelltext unterscheiden sich im wesentlichen bezüglich zwei Eigenschaften:

- Robustheit: Ist der Ansatz in der Lage mit unvollständigem Quelltext umzugehen, und muss der Quelltext in syntaktisch korrekter Form vorliegen.
- Vollständigkeit: Können mit dem Ansatz alle Informationen gewonnen werden, die zum Übersetzen des Programmes nötig wären.

3.2.1 Syntaktisch basierte Ansätze

Mit vollständigem Parsen kann Quelltext in einen abstrakten Syntaxbaum umgewandelt werden, der die gesamte Information enthält, die zum Übersetzen eines Programmes nötig ist. Hierzu muss die syntaktische Struktur der Programmiersprache in Form einer Grammatik bekannt sein. Je nach Eigenschaften der Grammatik können Techniken wie Rekursiver Abstieg oder Kellerautomaten zur Implementierung der Parser verwendet werden. [GW84]. Ein Beispiel für einen vollständigen Parser für Java ist in ReCoder [LH01] enthalten. Meist ist es einfacher, den Parser aus einer vorhandenen Spezifikation der Grammatik der Sprache generieren zu lassen, als ihn direkt zu implementieren. Beispiele für solche Generatoren sind javacc [Jav01] oder BISON[DS98] .



Ein Parser wird für genau ein Sprache geschrieben. Soll eine andere Sprache oder ein anderer Dialekt bearbeitet werden, so muss ein neuer Parser generiert oder geschrieben werden, d.h. die Wiederverwendbarkeit eines Parsers ist eingeschränkt.

Vollständige Parser sind nicht sehr robust. Sie müssen immer alle Quellen zur Verfügung haben, sonst sind sie nicht in der Lage den kompletten Syntaxbaum aufzubauen und können den Quelltext nicht verarbeiten. Liegt der Quelltext in syntaktisch inkorrekter Form vor, so wird der Parser mit einer Fehlermeldung abbrechen, und die Extraktion der Information kann nicht durchgeführt werden.

Um die Robustheit der Parser zu verbessern, können sogenannte partielle Parser verwendet werden. Diese erzeugen nicht mehr den kompletten Syntaxbaum, da der Quelltext nur teilweise bearbeitet wird. Ein Beispiel hierfür ist der Parser von Sniff für C++. Hier werden alle Deklarationen von Klassen und Methoden verarbeitet, aber keine Methodenrümpfe [Bis92]. Dies macht den Parser robuster gegenüber syntaktischen Fehlern und lässt zu, dass nicht der gesamte Quelltext vorhanden ist.

3.2.2 Lexikalisch basierte Ansätze

Diese sogenannten leichtgewichtigen Ansätze verwenden Mustersuche mit Hilfe von endlichen deterministischen Automaten, um Informationen über den Quelltext zu bekommen.Das Muster gibt an, nach welchen Teilen des Quelltextes gesucht werden soll. Hierzu können Werkzeuge wie grep oder AWK benutzt werden, indem man mit einem regulären Ausdruck z. B. beschreibt, wie die Methoden eines Programmes im Quelltext zu finden sind.In [MN95] wird eine Möglichkeit vorgestellt, mit der man im Gegensatz zu grep und AWK auch Muster beschreiben kann, die sich über mehrere Zeilen erstrecken.

Mit lexikalischen Ansätzen kann Quelltext bearbeitet werden, der nicht syntaktisch korrekt und unvollständig ist, da nur nach den entsprechenden Mustern im zur Verfügung stehenden Quelltext gesucht wird.Allerdings wird kein kompletter Syntaxbaum aufgebaut, die extrahierte Information ist unvollständig und hängt von der Exaktheit der Musterspezifikation ab.

3.2.3 Bemerkungen

Oftmals besitzen Sprachen Eigenschaften, deren Verarbeitung sehr aufwändig ist. Die verschiedenen Ansätze unterscheiden sich oft in der Behandlung dieser Besonderheiten. Ein Beispiel hierfür sind Makros in C++. Um diese exakt zu behandeln, müssen diese expandiert werden. Dies können insbesondere die lexikalischen Ansätze nicht leisten.

Abbildung 6: Übersicht über Verfahren zur Fact Extraction und ihre Eigenschaften

	Vollständiges Parsen	Partielles Parsen	Lexikalische Ansätze	Laufzeit Ansätze
Vollständigkeit der Information	+	**0**	–	**0**
Robustheit	–	**0**	+	–

Zusammenfassend lässt sich sagen, dass je nach Randbedingungen unterschieden werden muss, welche Art von Ansatz benutzt werden sollte. Bei Sprachen, die sehr komplex sind, und für die keine Grammatikbeschreibung vorliegt und bei schnellem Bedarf an Information bieten die lexikalischen Ansätze klare Vorteile.

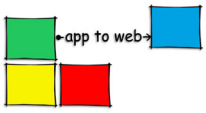
Wird allerdings Wert auf Exaktheit und Vollständigkeit der Information gelegt, so müssen syntaktisch basierte Verfahren eingesetzt werden. Gerade im Bereich des Reengineerings kann aber nicht immer vorausgesetzt werden, dass der Quelltext vollständig vorhanden ist, so dass sich hier partielle Parser und lexikalische Ansätze am besten eignen.

3.2.4 Weitere Ansätze zur Faktenextraktion

Eine alternative Art der Gewinnung der statischen Information besteht darin, Daten über das Programm zur Laufzeit zu sammeln und zu speichern, indem der Programmcode des Systems durch geeignete Anweisungen ergänzt wird. Diese Vorgehensweise wird in der Regel als *Instrumentierung* des Quellcodes bezeichnet. Hierbei kann z. B. Introspektion benutzt werden. In [OL01] wird sie eingesetzt, um zur Laufzeit ein statisches Strukturdiagramm des Quelltextes zur erstellen. Diese Ansätze sind nicht sehr robust, da der Quelltext syntaktisch korrekt, komplett und lauffähig sein muss. Zudem muss sichergestellt werden, dass alle relevanten Kontrollflüsse durch das System während des instrumentierten Programmlaufs abgearbeitet werden. Dies bedeutet, dass eine unter Umständen beträchtliche Menge von Testfällen erzeugt werden muss, um die relevanten Kontrollflüsse abzudecken. Ein weiteres Problem stellt die vergleichsweise große Datenmenge dar, die beim Durchlauf aller Testfälle erzeugt wird.

Trotz dieser in der Praxis recht gravierenden Probleme spielt das Aufzeichnen von laufzeitbezogenen Daten und die Technik des Instrumentierens für die Ermittlung von Fakten über das System durchaus eine wichtige Rolle. Oft wollen wir nämlich die rein statischen Informationen über ein System aus der Quellcodeanalyse durch entsprechende *dynamische Informationen* ergänzen.

Hierzu wird in der Praxis die bereits erwähnte Instrumentierung gerne verwendet, alternativ können auch mit Hilfe von Debuggern, Profilern bzw. den entsprechenden Systemschnittstellen Laufzeitintormationen ermittelt werden. Eine weitere Alternative zur Ermittlung von laufzeitbezogenen Daten ist durch Programmnalysetechniken gegeben, auf die wir im folgenden Abschnitt etwas eingehen wollen.



3.2.5 Statische Approximation von Laufzeitverhalten (Programmanalyse)

Statische Approximation von Laufzeitverhalten (im folgenden auch *Programmanalyse* genannt) ist eine Analyse von Software bzw. Softwaresystemen zur Übersetzungszeit, bei der alleine aus dem Quelltext dieser Systeme Schlüsse auf spezielle Aspekte des Laufzeitverhaltens gezogen werden. Informationen über das Laufzeitverhalten von Softwaresystemen können Details und wichtige Zusammenhänge in der Funktionsweise des Quelltextes offenbaren und so die Redokumentation unterstützen.

Derartige Analysen [NNH99] sind unter anderem in Fällen notwendig, in denen Informationen über das Laufzeitverhalten benötigt, die Beobachtung von Softwaresystemen zur Laufzeit aber nicht möglich ist: Beispielsweise könnten spezieller Hard- oder Softwareabhängigkeiten (beispielsweise fehlende Bibliotheken) vorliegen oder der Quelltext nur teilweise zur Verfügung stehen. Denkbar ist auch, dass es für spezielle Softwaresysteme einfach zu schwierig ist, sie kurzfristig für Untersuchungen zu modifizieren und in Betrieb zu nehmen.

Der Vorteil von statischer Approximation des Verhaltens eines Softwaresystems gegenüber seiner Beobachtung zur Laufzeit ist, daß die durch Approximation erreichten konservativen Abschätzungen des Laufzeitverhaltens zuverlässig für alle möglichen Zustände und Abläufe innerhalb des Systems gelten. Bei hinreichend exakten Abschätzungen sind damit sichere Aussagen über die Eigenschaften der untersuchten Softwaresysteme möglich. Da bei Beobachtungen zur Laufzeit fast nie sicher ist, ob auch wirklich alle möglichen Ablaufpfade untersucht wurden, müssen diese Beobachtungen prinzipbedingt unvollständig bleiben und erlauben damit nur punktuell sichere Aussagen.

Es gibt verschiedene Analyseziele im Rahmen der Programmanalyse. Im folgenden werden einige mit Relevanz zur Redokumentation aufgezählt.

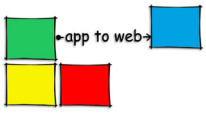
- *Typanalyse / Typinferenz:* Bei der Typanalyse wird versucht, den zur Laufzeit vorliegenden Typ einer Variable zu bestimmen. Diese bei Basistypen wie Ganzzahlen triviale Aufgabe ist bei objektorientierten Sprachen wie C++, Java, Smalltalk (wobei letzteres sogar gar keine statischen Typen kennt) nicht mehr so einfach. Durch die Verwendung von Polymorphie ist es nämlich oft aus dem Quelltext nicht direkt ersichtlich, zu welcher Klasse eine Variable zur Laufzeit tatsächlich gehört.
  Je nach verwendetem Algorithmus lässt sich der Typ eines Objekts in manchen Fällen durch eine Typanalyse nicht eindeutig bestimmen. Stattdessen wird dann untersucht, eine Menge derjenigen Typen anzugeben, die zur Laufzeit möglicherweise vorliegen könnten.
  Bei der Redokumentation ist die Typanalyse interessant, um Polymorphie in Softwaresystemen aufzulösen und zu erkennen, welche Klassen tatsächlich vorliegen können - beispielsweise in einer allgemeinen Containerklasse, die theoretisch

Objekte beliebiger Klassen speichern kann, in der Praxis vielleicht aber nur Objekte von 2-3 verschiedene Klassen aufnimmt.

- *Steuerflußanalyse:* Die Steuerflußanalyse versucht, über eine Bestimmung von Sprungzielen mögliche Pfade durch ein Programm zu ermitteln. Der ermittelte Steuerfluß lässt sich anschliessend als Graph darstellen, dessen Kanten Verzweigungen und dessen Knoten Anweisungen oder (linear ohne Sprünge durchlaufene) Anweisungsfolgen sind.
  Bei objektorientierten Sprachen muss zur Sprungvorhersage immer auch eine Typanalyse zur Auflösung der Polymorphie durchgeführt werden.
  Für die Redokumentation ist es sinnvoll, durch die Steuerflußanalyse einen Überblick über die Abläufe innerhalb eines Softwaresystems zu erhalten und dieses so besser verstehen zu können. Derartig automatisch bestimmte Abläufe können eine Basis für die zu erstellende Dokumentation sein.
- *Identifizierung von „toten" Quelltextteilen:* Oft entstehen im Laufe der Entwicklung von Softwaresysteme Fragmente, die nicht mehr ausgeführt werden. Durch eine Steuerflußanalyse, die möglicherweise mit anderen Analysen (z.B. Lebendigkeitsanalyse von Variablen) kombiniert wird, kann nun versucht werden, solche Quelltextteile zu ermitteln.
  Derartige „tote" Quelltextteile werden zwar nach der Übersetzung häufig beim Binden erkannt und nicht in das ausführbare Programm aufgenommen. Bei der Wartung und der Redokumentation liegt allerdings diese Information noch nicht vor und ist möglicherweise auch nicht feingranular genug. Um überflüssigen Aufwand bei der Redokumentation vermeiden zu können, ist also eine solche Analyse hilfreich.
  Ebenfalls hilfreich ist eine Analyse auf unbenutzte oder konstante Variablen, die i.d.R. aber bereits durch den Übersetzer durchgeführt wird, der seine Ergebnisse als Warnungen ausgibt.

Nachteile der Programmanalyse liegen unter anderem in der Größe der analysierbaren Quelltextstücke: Bei einigen Analysen wächst prinzipbedingt der Speicherbedarf in Abhängigkeit von der Größe der zu analysierenden Quelltextfragmente ungeheuer schnell. Teilweise können nicht mehr als einige hundert Zeilen analysiert werden. Der Grund für diese Komplexität liegt in der teilweise notwendigen Unterscheidung von sogenannten *Ausführungskontexten* [Tra01]: Die einzelnen Blöcke eines Softwaresystems können zur Laufzeit beliebig häufig durchlaufen werden. Bei jeder dieser Ausführungen können sich die Ergebnisse von Berechnungen abhängig vom dem Kontext, in dem sie stattfinden, ändern - beispielsweise können sie von der vorherigen Ausführung oder anderen Teilen des Systems abhängen. Damit definiert jede dieser Ausführungen einen eigenen Ausführungskontext. Manche Analysen versuchen nun, möglichst viele dieser Ausführungskontexte zu unterscheiden, um eine größere Analysegenauigkeit zu erreichen. Ein Beispiel: Bei der Typanalyse nach Golubski [Gol97], die gegenüber derjenigen nach

[Age95]eine erhöhte Kontextsensitivität aufweist, können nur noch Programme bis zu ca. 1500 Zeilen gegenüber 30.000 Zeilen bei Agesen analysiert werden [Tra01].



3.3 Repräsentation von Designinformationen – Metamodelle

Durch die Definition eines Metamodells legen wir fest, welche Informationen über die Struktur eines Softwaresystemes wir erfassen wollen und wie diese Informationen miteinander in Beziehung stehen. Kurzum, wir modellieren, welche Entitäten und Beziehungen wir mittels *Fact Extraction* aus dem Quellcode eines Systems extrahieren wollen: Metamodelle legen somit die *Semantik* der Designdatenbank fest, in der wir Fakten über ein zu analysierendes System sammeln.

Metamodelle können je nach Zielsetzung und verfügbareren Werkzeugen zur Faktenextraktion unterschiedlich detailliert und umfangreich sein. Tatsächlich verwendet die Reengineering-Gemeinde zahllose, verschiedene Metamodelle. In [Tic01] findet sich eine detaillierte Abhandlung über unterschiedliche Ausprägungen von Reengineering-Metamodellen, zudem wird dort ein umfangreiches, programmiersprachenunabhängiges Metamodell [DTS99] beschrieben. Aufgrund seiner Komplexität ist die Konstruktion von Werkzeugen zur Fact Extraction für dieses Metamodell jedoch sehr aufwendig, ebenso gestaltet sich die Verwendung der Fakten in weitergehenden Analysen, wie wir sie im weiteren Verlaufe dieses Kapitels kennenlernen werden, als recht kompliziert.

Im folgenden stellen wir daher zwei Beispiele für einfachere Metamodelle zur Repräsentation von Designinformationen vor, die im Projekt APPLICATION2WEB hauptsächlich zum Einsatz kommen.

3.3.1 Prozedurales Metamodell

In diesem Abschnitt beschreiben wir ein allgemein gehaltenes Metamodell, welches sich besonders dazu eignet, Designinformationen von Systemen darzustellen, die in einer *prozeduralen* Programmiersprache implementiert worden sind. Das Modell ist unter dem Kürzel *RFG (Resource Flow Graph)* bekannt und hat sich bereits in zahlreichen Projekten, die das IESE und seine Partner durchgeführt haben, bewährt [GKS97], [GKS99], [KGW98]. So wurde es bereits zur Repräsentation von Fakten aus Systemen eingesetzt, die in C, FORTRAN und Pascal implementiert wurden. Wir beschreiben hier den wesentlichen Kern des Modell – es kann jedoch auch leicht erweitert werden, um weitere Fakten zu modellieren, die in bestimmten Reengineeringsituationen von Interesse sind.

Wie einige andere Modelle zur Repräsentatuion von Designinformationen beruht RFG auf Konzepten der Graphtheorie, es modelliert Entitäten von Softwaresystemen als Knoten und Beziehungen zwischen diesen Entitäten als Kanten.

3.3.1.1 Entitäten

Konstrukte des Quellcodes und weitere Entitäten werden in RFG als Knoten eines Graphen modelliert. Grundsätzlich unterscheidet RFG *einfache Entitäten* und *zusammengesetzte Entitäten.*

Einfache Entitäten modellieren typische Konstrukte prozeduraler Programmiersprachen:

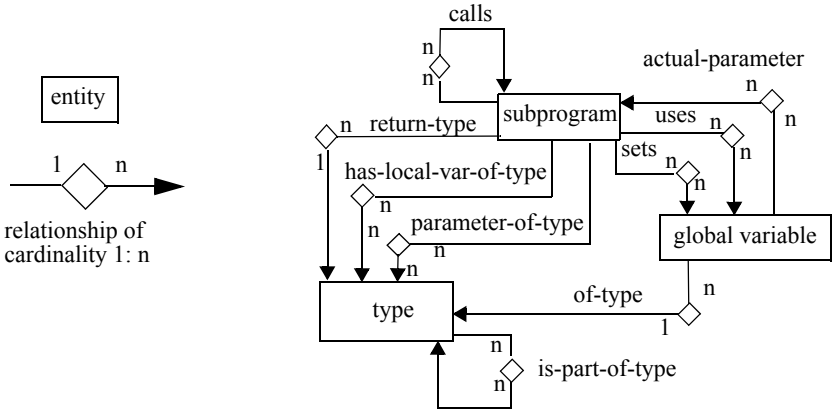
- *Unterprogramme*. In der Regel sind dies die Funktionen und Prozeduren des Systems.
- *Typen*. Hierbei sind in erster Linie benutzerdefinierte Datentypen von Interesse. Für die Programmiersprache C beispielsweise sind dies Typen, die durch die Sprachkonstrukte *typedef, struct, union*, und *enum* deklariert werden.

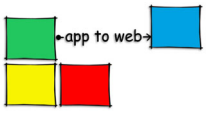
Zusammengesetze Entitäten werden verwendet, um mehrere einfache Knoten zu komplexeren, abstrakteren Strukturen zusammenzufassen. Beispiele für zusammengesetzte Entitäten sind zum einen strukturelle Entitäten wie Dateien (die mehrere Unterprogramme enthalten können), oder Unterverzeichnisse, mit Hilfe derer der Quellcode des Systems gegliedert wurde. Zum anderen können dies logische Entitäten sein, beispielsweise *abstrakte Datentypen* oder Komponenten.

3.3.1.2 Beziehungen

Kanten im graphorientierten RFG-Metamodell beschreiben Beziehungen zwischen den Programmiersprachenkonstrukten. RFG unterscheidet *quellcodebedingte Beziehungen* und *analysebedingte Beziehungen*. Quellcodebedingte Beziehungen bestehen nur zwischen einfachen Entitäten und können direkt aus dem Quellcode des Systems abgeleitet werden. Analysebedingte Beziehungen dagegen können dem Modell erst durch weitere, teilweise umfangreiche Analysen hinzugefügt werden.

Abbildung 7: Metamodell für prozedurale Sprachen.





3.3.1.3 Quellcodebedingte Beziehungen

Das Entity-Relationship-Diagramm in Abbildung 7 zeigt die wichtigsten einfachen Entitäten aus RFG und die quellcodebedingten Beziehungen, die zwischen ihnen bestehen können.Wir wollen diese Beziehungen der Reihe nach genauer betrachten:

- *Aufrufe (calls).* Ein Unterprogramm kann ein oder mehrere andere Unterprogramme aufrufen (auch sich selbst), ebenso kann es von verschiedenen anderen Unterprogrammen aufgerufen werden. Zwischen sich aufrufenden Unterprogrammen besteht dann die Beziehung *calls*.
- *Variablennutzungen und -belegungen (uses/sets).* Ein Unterprogramm kann eine oder mehrere *globale* Variablen nutzen, indem es aus ihnen liest, ohne ihren Wert zu verändern, ebenso kann es auf diese schreibend zugreifen, um sie mit neuen Werten zu belegen. Umgekehrt kann eine Variable von mehreren Unterprogrammen lesend verwendet oder beschrieben/belegt werden. Zwischen Unterprogrammen und Variablen bestehen dann je nach Situation die Beziehungen *uses* und *sets*.
- *Typdeklarationen (of-type).* Eine Variable hat stets genau einen Typ T, der durch eine entsprechende Deklaration im Quellcode des Systems festgelegt wird. Zwischen Variable und Typ besteht dann die Beziheung *of-type*. Es kann verschiedenen Variablen des selben Typs geben.
- *Zusammengesetzte Typen (is-part-of-type).* Ein Typ ST kann innerhalb der Deklaration eines weiteren Typs T verwendet werden. ST ist dann ein Teiltyp von T und zwischen ST und T besteht die Beziehuung *is-part-of-type*. [1]
  - Wenn T (unter anderem) aus einem Teiltyp ST zusammengesetzt wird, dann schreiben wir ST << T.
  - Gilt ST << ST' und ST' << T, dann folgt: ST << T. Die is-part-of-type-Beziehung ist demnach transitiv, entprechende transitive Kanten sind daher stets auch Teil des Modells.

  Ein Typ kann aus mehreren verschiedenen Teiltypen zusammengesetzt werden, umgekehrt kann ein Typ auch Teiltyp mehrerer anderer Typen sein.
- *Parametertypen, Rückgabetypen (parameter-of-type, return-type).* Wenn ein Typ T in der Liste der formalen Parameter eines Unterprogramms S vorkommt, so besteht zwischen S und T die Beziehung *parameter-of-type* . Wenn T der Typ des Rückgabewertes eines Unterprgramms S ist, dann besteht zwischen S und T die Beziehung *return-type*. Ein Typ T kann in verschiedenen Unterprogrammsignatu-

1 Diese Definintion wurde [oWFNW90] entnommen. Dort wird der englische Ausdruck sub-type verwendet. Allerdings ist der Ausdruck sub-type irreführend, daher verwenden wir hier part-type.

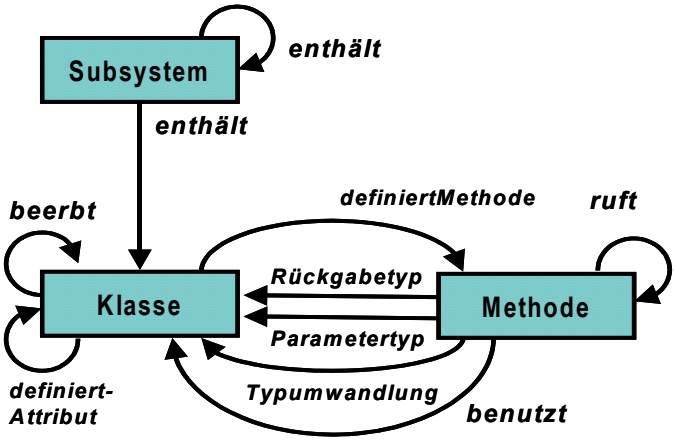
ren vorkommen, umgekehrt kann eine Unterprogrammsignatur mehrere Typen verwenden. Dabei setzt sich die Signatur eines Unterprogramms aus dem Typ des Rückgabewertes und der Typmenge seiner Parameter zusammen.

- *Lokale Variablen (has-local-var-of-type)*. Enthält ein Unterprogramm S eine *lokale* Variable vom Typ T, so besteht zwischen S und T die Beziehung *has-local-var-of-type*.

3.3.2 Metamodell für objektorientierte Systeme

Im folgenden Stellen wir ein relativ leichtgewichtiges Metamodell vor, welches zur Modellierung von Systemen dient, welche mit Hilfe von objektorientierten Programmiersprachen entworfen wurden. Dieses Modell basiert auf den Arbeiten in [BCD+98]. Es wurde nachfolgend immer wieder verfeinert [Ciu02] und wird jetzt als Basis für das Reverse Engineering von Systemen in APPLICATION2WEB herangezogen und verfeinert.

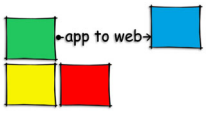
Abbildung 8: Metamodell für objektorientierte Sprachen



3.3.2.1 Entitäten

Unser Metamodell zur Beschreibung objektorientierter Systeme berücksichtigt nur die statischen Programmelemente eines Softwaresystems, also Klassen, Methoden und dergleichen. Dynamische Elemente (z.B. Objekte) betrachten wir zunächst nicht, da Informationen über sie nur mit aufwändigeren Techniken aus dem Quelltext extrahiert werden können (siehe Abschnitt 3.2.5).

- *Klassen:* Das zentrale Konzept in der statischen Struktur eines objektorientierten Systems ist sicherlich die Klasse. Die meisten Beschreibungen objektorientierter Systemstrukturen (z.B. Entwürfe), seien sie nun graphisch (z.B. in UML) oder auf



andere Weise notiert, stützen sich insbesondere auf Klassen und eine Beschreibung der Beziehungen zwischen Klassen

- *Methoden:* Eine Klasse enthält normalerweise eine Menge von Methoden, in denen der eigentliche Programmablauf beschrieben ist. Hybride Programme (z.B. in C++) können außerdem noch freie Prozeduren enthalten. Wir modellieren sie uniform als Methoden, die in keiner Klasse enthalten bzw. definiert sind.
- *Subsysteme*: In großen Systemen muss die Menge der Klassen normalerweise noch weiter strukturiert werden. Zu diesem Zweck gibt es in der Regel eine weitere Unterteilung in Subsysteme. Diese können Elemente der Programmiersprache sein, wie etwa Pakete in Java, oder sie können durch sprachexterne Konventionen definiert sein, wie z.B. durch eine Verzeichnishierarchie oder durch Präfixe in Klassennamen. Falls solche Informationen nicht zur Verfügung stehen, dann können die Techniken zum *Clustering* aus Abschnitt 3.5 wertvolle Hinweise auf Subsystemkandidaten liefern.

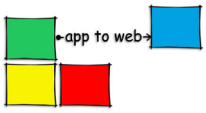
3.3.2.2 Beziehungen

Zwischen den Einheiten eines Softwaresystems gibt es verschiedene Typen von Abhängigkeiten. Wir modellieren sie als Beziehungen zwischen den Einheiten dieses Metamodells. Für jeden Beziehungstyp gibt es im Metamodell eine Relation, die die Beziehungen dieses Typs enthält. Unser Metamodell für objekt-orientierte Systeme enthält folgende Relationen:

- *Vererbung:* Die Beziehung *beerbt* besteht zwischen zwei Klassen A und B, wenn A von B abgeleitet ist. Wir unterscheiden dabei nicht zwischen verschiedenen Arten von Vererbung wie z.B. "*extends*" und "*implements*" in Java, da diese Unterscheidungen immer sprachspezifisch sind.
- *Methodendefinition:* Die Beziehung *definiertMethode* besteht zwischen einer Klasse A und einer Methode m, wenn m in A definiert wird.
- *Attributdefinition*: Die Beziehung *definiertAtttribut* besteht zwischen zwei Klassen A und B, falls A ein Attribut mit dem Typ B definiert. Unser Metamodell modelliert Attribute nicht als eigene Einheiten. Stattdessen werden sie nur über ihren Typ repräsentiert. Diese Vereinfachung hat zwar einige Nachteile. Mehrere Attribute derselben Klasse und vom selben Typ können beispielsweise nicht mehr voneinander unterschieden werden. Es hat sich jedoch gezeigt, daß eine genauere Modellierung mit Attributen als Einheiten (neben Problemen bei der Sprachunabhängigkeit) die Komplexität des Modells unverhältnismäßig erhöht. Insbesondere müssen dann Attribute und (Methoden-) lokale Variable mitsamt aller zugehörigen Beziehungen voneinander unterschieden werden. Aus Gründen der Praktikabilität wurde daher die hier vorgestellte Modellierung gewählt. Da wir an der rein statischen Struktur des Programms interessiert sind, geht diese Abhängigkeit immer zu demjenigen Typ, mit dem das Attribut deklariert ist, nicht z.B. zu dem Typ, mit dem ein in Objekt erzeugt wird. (Eine Objekterzeugung "*new A()*" ist aus

struktureller Sicht ein gewöhnlicher Methodenaufruf an den jeweiligen Konstruktor.)

- *Parametertyp*: Die Beziehung *hatParametertyp* besteht zwischen einer Methode m und einer Klasse A, wenn m einen formalen Parameter des Typs A besitzt. Dies ist neben den Attributen ein weiteres Beispiel, bei dem wir Einfachheit gegen Ausdruckskraft eingetauscht haben. Parameter sind nicht als eigenständige Einheiten modelliert.
- *Methodenaufruf*: Die Beziehung *ruft* besteht zwischen zwei Methoden m und n, falls m eine Anweisung mit einem Aufruf an n enthält. Beispiel: Eine Methode A.m() ruft eine andere Methode B.n(), wenn sie mindestens eine Anweisung oder Ausdruck b.n() enthält, B der statische Typ der Variablen b ist und B.n() diejenige Definition von m() ist, die in der Klasse B sichtbar ist. Die Extraktion des Metamodells erfolgt, wie in Abschnitt 3.2 beschrieben, durch ein sprachabhängiges Analysewerkzeug. Dort erfolgt durch eine Typanalyse auch die Berechnung des Ziels dieser Beziehung. Völlig unberührt davon ist natürlich die Frage, welche Methodendefinition zur Laufzeit gerufen wird. Dies hängt vom dynamischen Typ B' von b ab, wobei in statisch typisierten Sprachen B'eine Unterklasse von B ist. Dieses Metamodell beschreibt jedoch ausschließlich die statische Struktur eines Systems. Der dynamische Typ von b lässt sich darin nur nach oben abschätzen.
- *Rückgabetyp*: Die Beziehung *hatRückgabetyp* besteht zwischen einer Methode m und einer Klasse A dann, wenn m als Funktion einen Rückgabewert vom Typ A liefert.
- *Variablennutzung*: Die Beziehung *benutzt* besteht zwischen einer Methode m und einer Klasse A dann, falls m eine Anweisung enhält, die eine Variable vom Typ A liest, schreibt oder deklariert. Unter Variablen verstehen wir hierbei lokale Variablen, Attribute oder Methodenparameter. Die Unterscheidung von Lesen und Schreiben anhand des Programmtextes ist durchaus nicht trivial, deshalb wird dies auch im vorliegenden Metamodell nicht unterschieden. Wegen der Möglichkeit der impliziten Deklaration und vergleichbarer Komplikationen in einigen Programmiersprachen, wird auch die Deklaration lokaler (und gegebenenfalls freier) Variablen unter diesen Beziehungstyp gefaßt.
- *Typumwandlung*: Die Beziehung *hatTypumwandlung* besteht zwischen einer Klasse A und einer Methode m dann, wenn m eine Typumwandlung (Typecast) zum Typ A verwendet. Auch die Typumwandlung induziert Abhängigkeiten, die für die statische Struktur eines Systems bedeutsam sind. Die Abhängigkeit geht dabei von der Methode, die die Typumwandlung enthält zu dem Typ bzw. der Klasse zu der gewandelt wird.
- *Enthaltensein*: Die Beziehung *enthält* besteht zwischen einem Subsystem A und einem Subsystem oder einer Klasse B, falls A ein Subsystem oder eine Klasse mit Namen B enthält. Diese Relation ist normalerweise, aber nicht zwangsläufig, hierarchisch. Diese Definition von enthält geht davon aus, daß innere Klassen vernachlässigt werden. Eine Klasse kann also keine anderen Klassen enthalten. Soll



das Metamodell auch innere Klassen berücksichtigen, sind jedoch einfach entsprechende Erweiterungen möglich.[1]

3.3.3 Syntaktische Repräsentation der Designinformation

Neben der Festlegung der Semantik der Designinformationen in Form von metamodellen ist in aller Regel auch noch die Festlegung einer geeigneten Syntax zur Repräsentation der Designinformationen erforderlich, insbesondere dann, wenn unterschiedliche Reengineering-Werkzeuge auf die Designinformationen zugreifen sollen. Hierzu existieren zahlreiche Möglichkeiten. Sie reichen von einfachen, strukturierte Textdateien über XML-basierte Formate (ein Beispiel hierfür ist in Abbildung 9 zu sehen, XMI [Gro98]und GXL [HWS00] sind verbreitete XML-basierte Standards zur Repräsenation von Designinformationen) bis hin zu komplizierteren Spezialformaten (FAMIX/CDIF [Tic01]). Im Vergleich zur semantischen Festlegung des Metamodells ist seine syntaktische Repräsentation eher von ungeordneter Bedeutung. Daher verzichten wir im Rahmen dieses Dokuments auf eine tiefergehende Behandlung dieses Themas.

3.4 Visualisierungen

In zahlreichen Reengineering-Projekten haben Entwickler sehr gute Erfahrungen mit Visualisierungen ihrer Softwaresysteme gemacht. Visualisierungen helfen, weil sie komplexe Zusammenhänge (also auch die von Bestandteilen eines Softwaresystems) übersichtlich und für das menschliche Auge geeignet darstellen können.

Visualisierungen können – sofern sie mit geeigneten Abstraktionsmitteln bearbeitet werden (Grouping) – das Systemverständnis beträchtlich erleichtern. Ebenfalls von großer Bedeutung für die Effektivität von Visualisierungen sind neben Abstraktionstechniken auch automatische Layout-Algorithmen.

Wir werden im folgenden kurz darstellen wie sich Visualisierungen nutzen lassen, um schnell bestimmte Eigenschaften über ein System zu erfassen.

1 Innere Klassen sind hauptsächlich aus Java bekannt. Theoretisch erlaubt auch C++ die Definition von Klassen innerhalb von Klassen citeEllisStroustrup90. Die derzeit gängigen Übersetzer kommen damit jedoch nur bei sehr kleinen Programmen zurecht. Praktisch haben innere Klassen in C++ damit keine Bedeutung.

3.4.1 Gewinnen von Basisvisualiserungen

Wir möchten Visualisierungen in erster Linie dazu nutzen, Abhängigkeiten zwischen einzelnen Systembestandteilen zu untersuchen und zu verstehen. Aus diesem Grund wollen wir aus den Designinformationen des Systems einen Abhängigkeitsgraphen gewinnen. Wir gehen dabei davon aus, dass die Designinformationen über das System gemäß Abschnitt 3.2 extrahiert und unter Verwendung eines geeigneten Metamodells zur Beschreibung von Designinformationen (Abschnitt 3.3) zu einem Modell des Systems verdichtet wurden.

In der einfachsten Form können wir ein solches Designmodell eines Systems visualisieren, wenn wir die Entitäten des Modells als Knoten eines Graphen auffassen, und die Beziehungen zwischen den Einheiten als (getypte) Kanten modellieren. Dieser Graph kann dann einfach mit entsprechenden Werkzeugen, sogenannten Grapheditoren, veranschaulicht werden. Beispiele für solche Grapheditoren oder -betrachter, die sich insbesondere im Reengineering-Bereich großer Popularität erfreuen, sind die Systeme *Graphlet* [Gra] und *Rigi* [rig].

Abbildung 9: Designinformation und entsprechendes Fragment aus der entsprechenden Graphvisualisierung

```
<class id="A">
   </class>
   <class id="B">
</class>
...
<inheritsFrom>
   <source_id>B</source_id>
   <dest_id>A</dest_id>
</inheritsFrom>
...
<definesMethod>
   <source_id>A</source_id>
   <dest_id>m</dest_id>
</definesMethod>
```

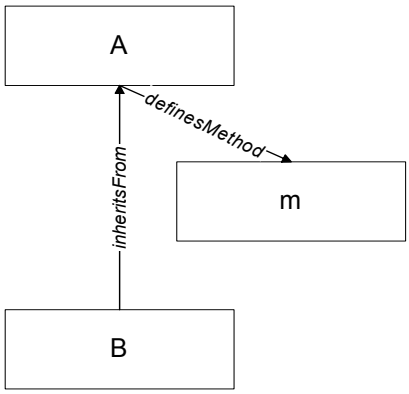
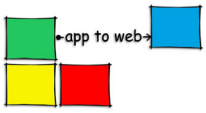


Abbildung 9 zeigt links einen beispielhaften Auszug aus den Designinformationen eines Systems. Dabei verwenden wir das Goose-Metamodell, wie es in Abschnitt 3.3.2dargestellt wurde, und stellen es in einer XML-Syntax dar. Rechts daneben ist eine Visualisierung des entsprechenden Graphfragmentes zu sehen.

In Abbildung 10 visualisieren wir ein größeres System mit Hilfe derselben Vorgehensweise. Dabei modellieren die Knoten des Graphen Klassen des Systems, und die Kanten dazwischen stellen Abhängigkeiten zwischen diesen Klassen dar, wie sie im Goose-Metamodell definiert wurden.



Wir erkennen leicht, dass diese einfache Visualisierungstechnik schnell versagt, sobald es sich um größere Systeme mit zahlreichen Bauteilen und Abhängigkeiten handelt – die zugrundeliegenden Graphen werden einfach zu umfangreich und unübersichtlich.

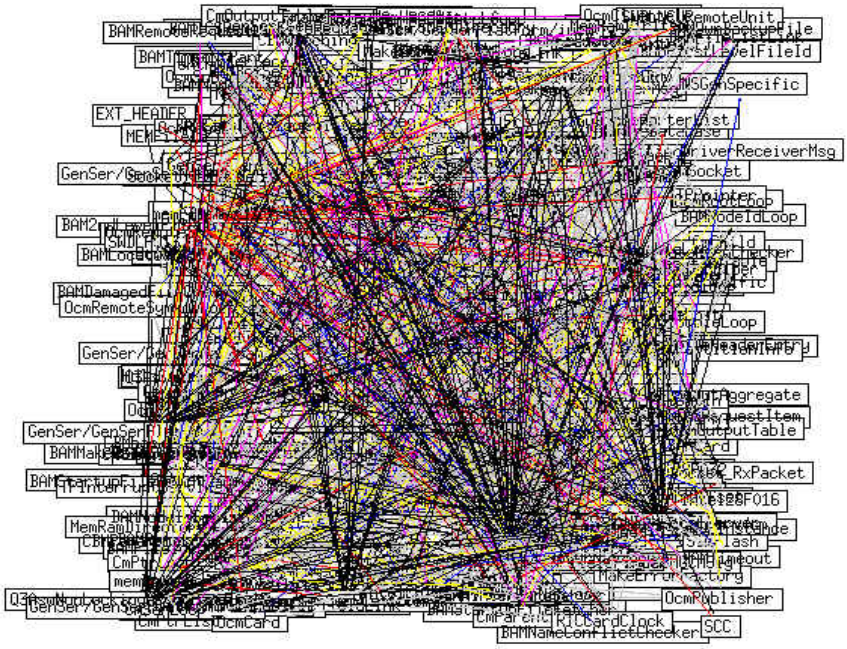


Abbildung 10: Unübersichtliche Basisvisualisierung eines größeren Softwaresystems

3.4.2 Automatisierte Abstraktion: Filterung und Gruppierung

Abbildung 10 verdeutlicht, dass Visualisierungen auf zu niedrigem Abstraktiongrad kaum dazu dienen können, einem Softwareentwickler geeignete Einblicke und Erkenntnisse über ein Softwaresystem zu verschaffen. Es fehlt an geeigneten Abstraktionen – an Konzepten, wie sich die Informationsmenge in den Visualisierungen reduzieren lässt.

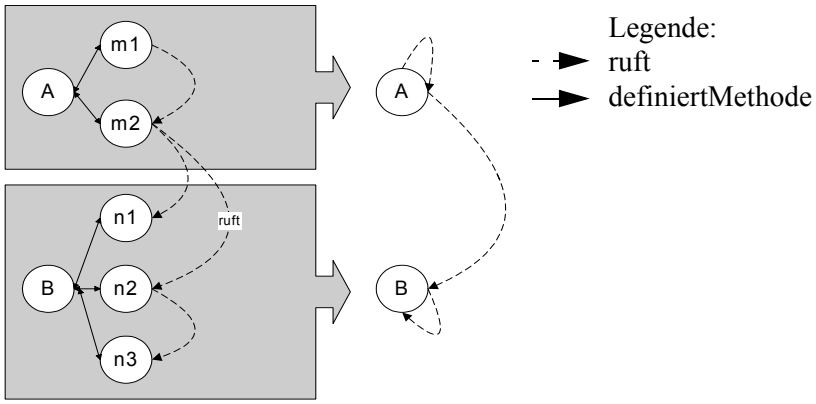
Zur Informationsreduktion in Visualisierungen verwenden wir zwei grundlegende Techniken: *Filterung* und *Gruppierung*. (Beide Techniken können auch direkt auf den Designdatenbanken angewandt werden, um deren Größe zu reduzieren.)

Bei der *Filterung* (oft auch Selektion genannt) werden uninteressante Knoten (meist samt der mit ihnen verknüpften Kanten) einfach aus dem Graphen entfernt. Ebenso ist auch eine Filterung unerwünschter Kantentypen (Beziehungen) möglich.

Die *Gruppierung* ersetzt eine Menge von Entitäten einer niedrigen Abstraktionsebene durch eine Entität einer höheren Abstraktionsebene und überträgt die Abhängigkeiten der niedrigen Abstraktionsebene in entsprechende Abhängigkeiten der höheren Abstraktionsebene.

Ein Beispiel mag dies verdeutlichen: Oftmals sind wir nicht an Abhängigkeiten zwischen den Methoden eines Systems interessiert, sondern an Abhängigkeiten, die zwischen den Klassen des Systems bestehen. Natürlich entsteht ein Großteil der Abhängigkeiten auf Klassenebene durch entsprechende Abhängigkeiten auf Methodenebene – Aufrufbeziehungen beispielsweise sind gemäß des Metamodells aus Abschnitt 3.3.2 nur zwischen Methoden definiert (Beziehung *ruft*), trotzdem ergibt sich daraus eine Abhängigkeit zwischen den beiden beteiligten Klassen. In Abbildung 11 ist eine solche Situation wiedergegeben: Methode m2 der Klasse A ruft die Methoden n2 und n2 der Klasse B. Gruppiert man die Methoden mit ihren entsprechenden Klassen, so ensteht daraus eine Aufrufbeziehung auf Klassenebene zwischen A und B.

Abbildung 11: Gruppierung von Methoden zu Klassen



Gruppierungen, wie wir sie hier am Beispiel der Gruppierung von Methoden zu Klassen illustriert haben, sind auf den unterschiedlichsten Abstraktionsebenen verwendbar. Wir führen hier einige der wichtigsten Beispiele an:

- *Klassen zu Subsystemen/Paketen:* Dies ist vermutlich – zusammen mit der Gruppierung von Subsystemen zu neuen Subsystemen – die wichtigste Anwendung dieser Abstraktionstechnik. Abbildung 12 verdeutlicht die zugrundeliegende Vorgehensweise.

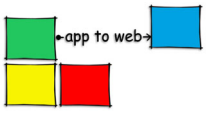
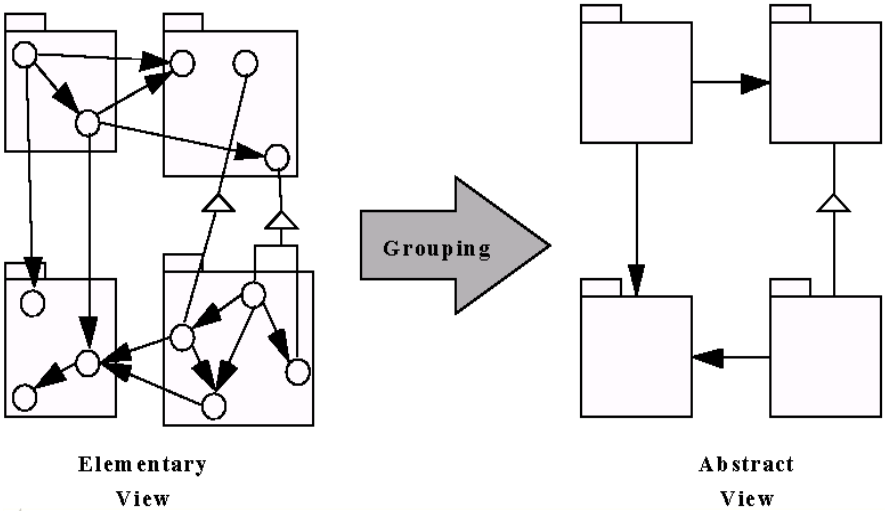


Abbildung 12: Gruppierung von Klassen in Subsysteme



- *Subsysteme zu Subsystemen:* Oftmals sollten in großen Systemen aus Gründen der Übersichtlichkeit kleine Subsysteme oder Module zu gröber granulareren Subsystemen zusammengefaßt werden.
- *Klassen zu Oberklassen:* Durch diese Gruppierung werden Einheiten gemäß bestimmter Eigenschaften oder gemäß ihres Verhaltens gruppiert, weil Klassenhierarchien oft verschiedene Verfeinerungstufen ein- und derselben Aspekte realisieren.
- *Clientseitige und Serverseitige Fragmente zu logischen Klassen:* In verteilten Systemen, die mittels *COM, CORBA, EJB* oder ähnlichen Mechanismen verbunden sind, lassen sich durch geeignete Gruppierungen Abhängigkeiten, die zwischen Clientcode und Server-Code bestehen, die aber mittels gewöhnlicher Quellcodeanalyse gemäß Abschnitt 3.2 nicht ermittelt werden können, weil sie durch die entsprechende Infrastruktur realisiert werden, wieder ins Modell einbringen. In Abbildung 13 ist eine entsprechende Gruppierung für die Client- und Server-Bestandteile in EJB-Systemen dargestellt – genaueres zu solchen Gruppierungen findet sich in [Ciu02].

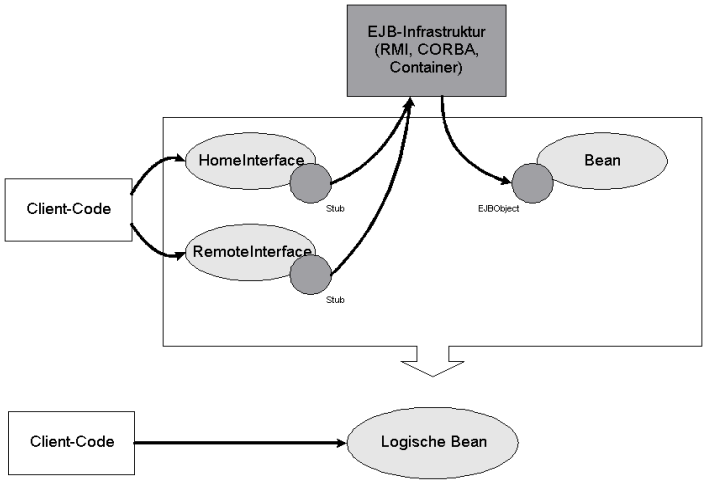


Abbildung 13: Gruppierung von Client- und Server-Bestandteilen in EJB-Systemen

- *Laufzeitkonzepte zu statischen Entitäten:* Durch die Gruppierung von Laufzeitkonzepten zu entsprechenden statischen Entitäten wird eine Verknüpfung von dynamischer und statischer Sicht möglich. Beispielsweise lassen sich so Aufrufe zwischen Objektinstanzen auf Beziehungen zwischen Klassen übertragen. Diese Beziehungen können durchaus von den statischen Beziehungen, wie sie beispielsweise durch die *ruft*-Beziehung gegeben sind, abweichen.

Natürlich können und müssen diese und weitere Gruppierungen auch miteinander kombiniert werden. Um beispielsweise eine aussagekräftige Visualisierung der Abhängigkeiten eines Systems auf Subsystemebene zu bekommen, bedarf es zunächst einer Gruppierung von Methoden zu Klassen, und anschliessend einer Gruppierung der Klassen zu Subsystemen. Auf diese Weise übertragen sich die Abhängigkieten, die zwischen den einzelnen Methoden bestehen, auf Abhängigkeiten zwischen Subsystemen.

Solche Gruppierungen können formal durch Graphenhomomorphismen beschrieben werden [BCD+98] [Ciu02]. Goose enthält entsprechende Werkzeuge, um in Modellen, die der Goose-Metamodelldefinition entsprechen, Gruppierungen unterschiedlicher Ausprägungen durchzuführen.

Abbildung 14 zeigt eine abstrahierte Visualisierung des Systems aus Abbildung 10. Dabei wurden wiederholt Gruppierungen eingesetzt. Zunächst wurden Methoden zu ihren Klassen gruppiert, dann Klassen zu ihren Subsysteme. Die Abbildung veranschaulicht demnach die Abhängigkeiten zwischen den Subsystemen des Systems, berücksichtigt dabei jedoch auch diejenigen Abhängigkeiten, die aus Beziehungen zwischen den einzelnen Klassen bzw. den einzelnen Methoden des Systems entstehen.

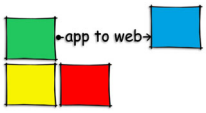
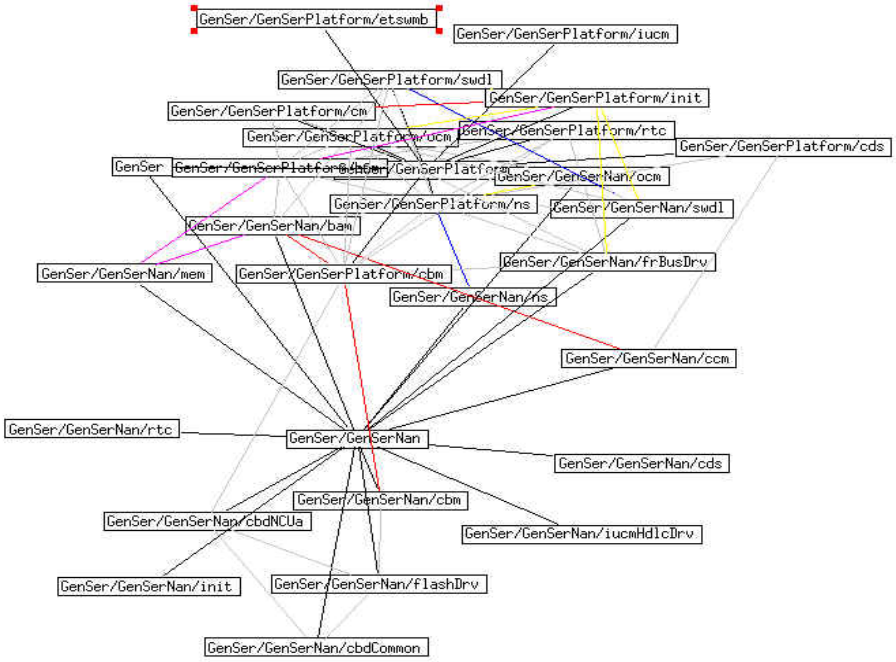


Abbildung 14: Visualsierung nach autmatischer Abstraktion auf Subsystemebene



3.4.3 Layoutalgorithmen

Neben Methoden zur automatischen Abstraktion sind für brauchbare Visualisierungen auch geeignete Layoutalgorithmen erforderlich, die die Knoten der Abhängigkeitsgraphen auf übersichtliche Weise anordnen, so dass sich bestimmte Eigenschaften im Designmodell eines Systems leichter erkennen lassen. So zeigt Abbildung 15 erneut das System aus Abbildung 10 – dieses Mal sind die Knoten im Graphen allerdings mit Hilfe des sogenannten Spring-Embedder-Algorithmus angeordnet worden, der versucht, zusammengehörige Knoten nahe beieinander zu platzieren.

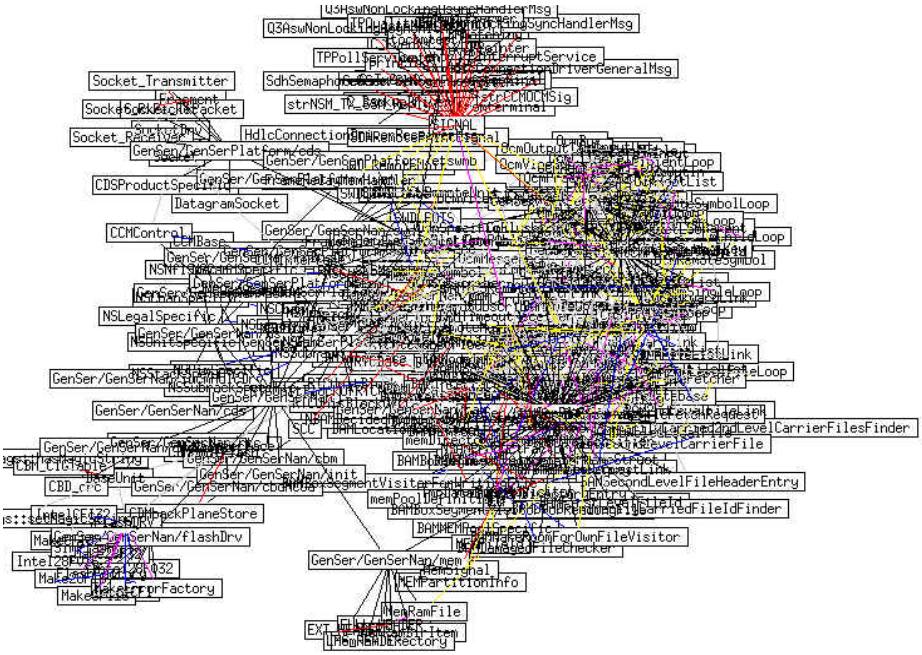
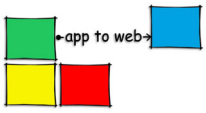


Abbildung 15: Visualsierung mit automatischem Spring-Embedder Layout

Wenn wir uns die Tatsache vor Augen führen, dass Visualisierungen von Softwaresystemen oft auf Graphen beruhen, die aus mehreren tausend Knoten und Kanten bestehen, dann wird schnell klar, dass eine geeignete Anordnung von Hand nicht praktikabel ist.

Aus diesem Grund unterstützen die meisten Werkzeuge zur Visualisierung von Graphen, wie beispielsweise die beiden bereits erwähnten Systeme *Rigi* und *Graphlet* entsprechende Layoutalgorithmen aus folgenden Klassen:

- *Baumlayouts*: Diese Algorithmen versuchen, Klassen so anzuordnen, dass möglichst klare Baum- oder Waldstrukturen entstehen. Sie eignen sich insbesondere zur Darstellung von *Vererbungsbeziehungen* in Systemen. Die meisten dieser Algorithmen setzen voraus, dass der zugrundeliegende Graph sich auch tatsächlich als Wald oder Baum darstellen lässt, d.h. der Graph muss gerichtet sein und darf keine Rückwärtskanten enthalten.
- *Hierarchische Layouts:* Hierarchische Algorithmen verallgemeinern Baumlayouts, indem sie versuchen Knoten in gerichteten Graphen gemäß geeignet in Hierarchieebenen anzuordnen.Im Unterschied zu Baumlayouts sind hier jedoch Rückwärtskanten erlaubt. In Softwaresystemen lassen sich diese Algorithmen gut dazu verwenden, die Abhängigkeitsverhältnisse in *Schichtenarchitekturen* darzustellen.
- *Massekörper-Feder-Modelle, Spring-Embedder-Layouts:* Diese Algorithmen ordnen die Knoten eines Graphen so an, dass zusammengehörige Elemente des Gra-



phens in der Visualisierung auch dicht beieinander liegen. Solche Algorithmen arbeiten iterativ und nutzen zur Anordnung der Knoten das folgende physikalische Modell: Knoten repräsentieren magnetisierte bzw. elektrisch geladene Massekörper, die sich gegenseitig abstoßen. Kanten repräsentieren Spiralfedern, die diese Massekörper miteinander verbinden und der magnetischen bzw. elektrischen Anziehungskraft der Körper entgegenwirken. In Graphen mit gewichteten Kanten entspricht das Gewicht einer Kante der jeweiligen Federzugkraft. Für die Visualisierung von Softwaresystemen sind solche Layouts insbesondere geeignet um stark zusammengehörige Teile des System zu ermitteln und zu veranschaulichen. Modelliert man die Klassen des Systems als Knoten, die Aufrufe zwischen den Methoden der Klassen als Kanten zwischen diesen Knoten (die Anzahl der Aufrufe kann als Kantengewicht mit in die Visualisierung einfließen), so erhält man typischerweise Visualisierungen mit „Klassenhäufungen“, in denen stets Klassen zu finden sind, die stark miteinander interagieren.

3.4.4 Beispiele

In diesem Abschnitt zeigen wir anhand einiger Szenarien, wie Softwareentwickler Visualisierungen einsetzen können, um ihr Verständnis über ein Softwaresystem zu verbessern. Wir illustrieren unsere Vorgehensweise an einer Fallstudie aus dem Telekommunikationsbereich. Alle Beispiele und Abbildungen dieses Abschnitts zeigen Auszüge aus der Steuerungssoftware eines Netzwerknotens.[1]

Visualisierung der Grobstruktur des Systems

Wenn wir ein Reengineering-Projekt beginnen, so wollen wir häufig im ersten Schritt einen Überblick über die grobe Struktur des Systems bekommen. Dazu nutzen wir eine Visualisierung, die stark von den soeben erläuterten Abstraktiontechniken Filterung und Gruppierung als vom automatischen Layout profitiert.

Zunächst gruppieren wir die Klassen des Systems in Subsysteme, wie sie entweder durch Mittel der Programmiersprache, durch die Verzeichnisstruktur der Quellen oder ähnliche Mechanismen definiert sind – gemäß Abschnitt 3.4.2 übertragen wir dabei die Abhängigkeiten auf Methoden und Klassenebene auf die Subsystemebene. Diese Abhängigkeiten nutzen wir dann, um mit Hilfe eines hierarchischen Layoutalgorithmus die Schichtung des Systems darzustellen. Abbildung 16 (Links) zeigt eine solche Visualisierung für unsere Fallstudie. Wir erkennen, dass das System prinzipiell einer Schichtenarchitektur folgt. Natürlich erkennen wir auch, dass das Prinzip der Schichtenarchitektur an einigen Stellen verletzt wird.

1 Da die Abbildungen tatsächliche Visualisierungen eines kommerziellen Softwaresystems zeigen, mussten alle Bezeichner für die Subsysteme und Klassen in den Abbildungen verfremdet werden. Ansonsten haben wir an den Abbildungen keine Änderungen vorgenommen – sie zeigen also in der Tat realistische Situationen aus dem Reverse-Engineering.

Überprüfen von Abhängigkeiten

Sobald wir die Grobarchitektur eines Systemes ermittelt haben (oder wir überprüft haben, ob die in Dokumentation über das System – falls vorhanden – definierte Architektur grob eingehalten wird), können wir uns daran machen diese Architektur genauer zu untersuchen: Wir möchten Abhängigkeiten zwischen Systemteilen ermitteln, die der vorgeschriebenen/gewünschten Architektur widersprechen.

Wir nehmen dazu an, dass unsere Fallstudie folgenden beiden Grundsätzen genügen soll:

1. Das System ist – wie bereits bekannt – in Schichten organisiert. Höhere Schichten implementieren ihre Funktionalität mit Hilfe der unteren Schichten. Umgekehrt gilt alllerdings: untere Schichten sollen *nicht* auf höhere Schichten zugreifen.

2. Das System besteht aus zwei Teilen. Der erste Teil besteht aus einem *Framework*, welches auch in anderen Systemen bzw. Produkten verwendet werden kann. Der andere Teil besteht aus produktspezifischem Code, welcher das Framework instanziert und mit Hilfe des Frameworks ein bestimmtes Produkt implementiert. Daraus ergibt sich, dass Code aus dem Framework nicht von Code aus den produktspezifischen Teilen des Systems abhängen darf. (In Abbildung 16 sind Teile des Frameworks mit dem Suffix *FW*, produktspezifische Teile mit dem Suffix *Product* versehen.)

Mit Hilfe geeigneter Filteroperationen lässt sich die Visualisierung auf der linken Seite von Abbildung 16 leicht so vereinfachen, dass nur Kanten übrig bleiben, die diesen beiden Prinzipien widersprechen. Solche Abhängigkeiten stellen Architekturverletzungen dar; sie sind auf der rechten Seite von Abbildung 16 dargestellt. In unserer Fallstudie greift beispielsweise das Subsystem *Layer/Product* auf *Layer3/Product* zu. Überraschenderweise gibt es auch Subsysteme des Frameworks, die von produktspezifischem Code abhängen: In der untersten Schicht des Systems, in *Layer1*, hängt *Layer1/FW* von *Layer1/Product* ab.

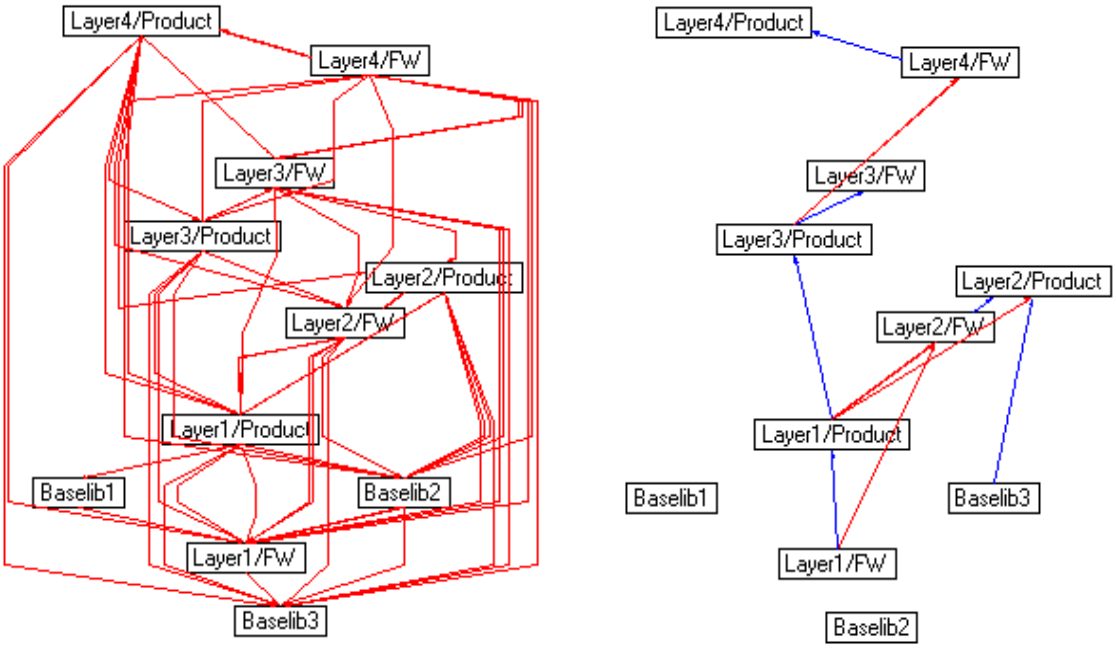
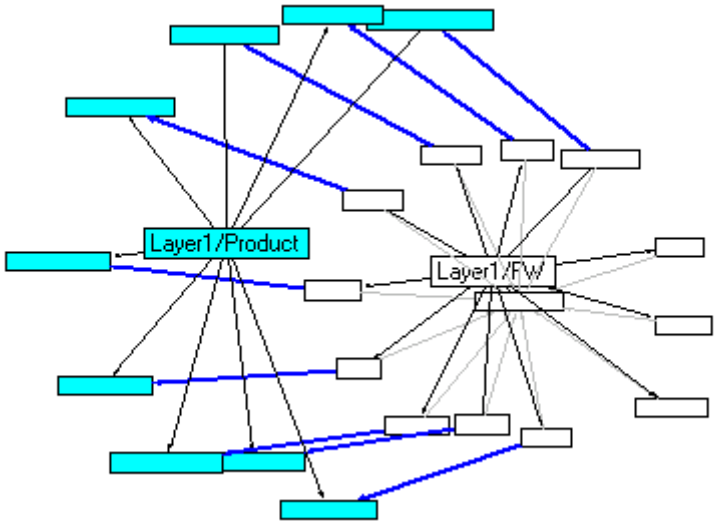


Abbildung 16: Links: Visualsierung der Schichtenarchitektur eines System. Rechts: Überprüfen von Architekturverletzungen.

Natürlich interessieren wir uns genauer für diese Architekturverletzungen. Deswegen betrachten wir die Subsysteme *Layer1/FW* und *Layer1/Product* auf der Klassenebene. Abbildung 17 zeigt die zugehörige Visualisierung. Produktspezifische Klassen aus *Layer1/Procuct* sind dunkel eingefärbt, Klassen des Frameworks aus *Layer1/Product* dagegen weiß. Die Kanten, die radial von den Subsystemknoten *Layer1/FW* bzw. *Layer1/Product* zu den Klassenknoten ausgehen, stellen die *enthält*-Beziehung dar. Die übrigen Kanten sind Vererbungsbeziehungen, alle weiteren Kantentypen wurden herausgefiltert. Dieser Visualisierung entnehmen wir, dass einige Klassen aus dem Framework von produktspezifischen Klassen abgeleitet wurden.

Abbildung 17: Eine Architekturverletzung im Detail



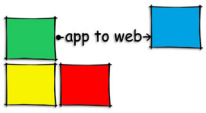
Mit Hilfe von Visualisierungen, Filter- und Abstraktionstechniken können wir solche problematischen Stellen leicht aufdecken. Natürlich können wir erst dann zuverlässige Aussagen über die Konsequenzen dieser Architekturverletzungen machen, wenn wir die entsprechenden Stellen im Quellcode des Systems genauer untersucht haben. Die vorgestellten Techniken haben uns jedoch gezielt auf einige wenige, kritische Stellen hingewiesen, und uns damit mühsames Inspizieren des Quellcodes erspart.

Überprüfen der Subsystembildung

Sehr oft sind wir daran interessiert, zu überprüfen, ob eine existierende Subsystemstruktur ordentlich entworfen und implementiert ist. Dies spielt insbesondere dann eine Rolle, wenn wir später einzelne Module oder Komponenten eines Systems extern wiederverwenden oder durch neue ersetzen. Solche Situationen ergeben sich bei der Migration zu Webanwendung besonders häufig – zum einen können Teile aus Altsystemen in neue, webbasierte Systeme eingebracht werden, zum anderen können Teile des Altsystems durch neue webfähige Module ersetzt werden.

Im folgenden betrachten wir eine Subsystemstruktur als wohlstrukturiert, wenn Klassen *innerhalb* eines Subsystems stark zusammenhängen (also zahlreiche Beziehungen zwischen ihnen bestehen) und wenn es deutlich weniger Abhängigkeiten zu *anderen* Subsystem gibt.[1]

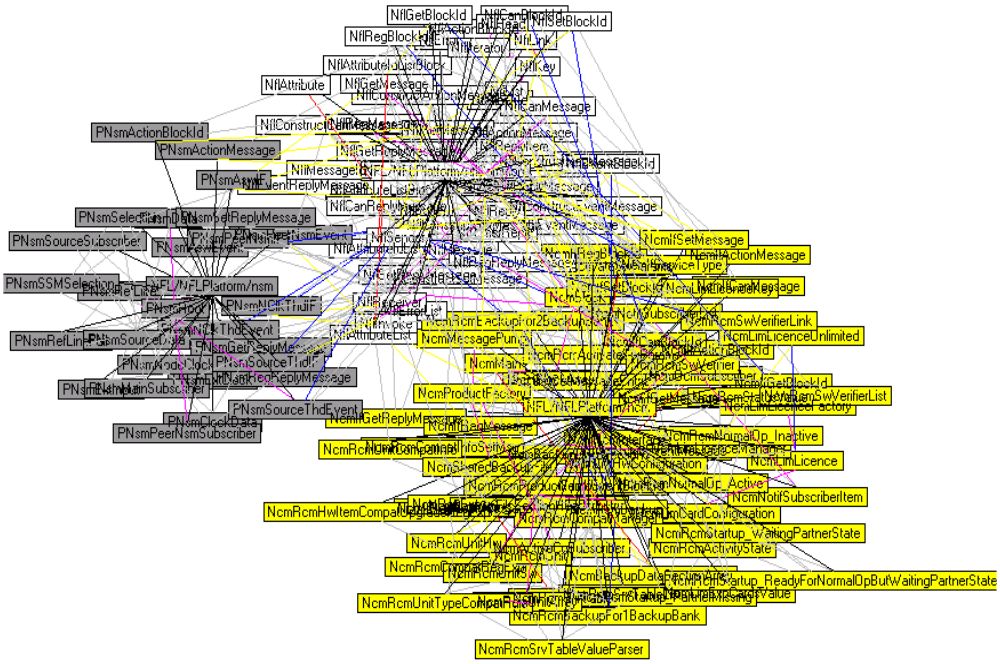
1 Es gibt Fälle, in denen die Klassen eines Subsystemes fast gar nicht miteinander über statische Abhängigkeiten, wie wir sie in unseren Designinformationen üblicherweise finden, in Verbindung stehen. Beispiele hierfür sind Datenbankzugriffsschichten, in denen alle Abhängigkeiten über Datenbankinhalte modelliert werden, oder Subsysteme die einen starken Bibliothekscharakter haben.



Um zu überprüfen, ob Subsysteme diese Anforderungen erfüllen, können wir wieder Visualisierungen einsetzen. Eine weitere Möglichkeit besteht darin, Clustering-Techniken (siehe Abschnitt 3.5) einzusetzen.

Abbildung 18 zeigt eine Visualisierung dreier Subsysteme unserer Fallstudie. Knoten stellen wieder Klassen des Systems dar, Kanten Abhängigkeiten zwischen diesen Klassen. Die Klassenknoten sind gemäß ihrer Subsystemzuordnung eingefärbt. Der Graph wurde mit Hilfe eines Spring-Embedder-Layouts dargestellt. Wir sehen, dass die Klassen der jeweiligen Subsysteme nahe beieinander platziert werden. Dies liegt jedoch an den radial verlaufenden *enthält*-Kanten, die Klassen stark an ihre jeweiligen Subsystemknoten binden.

Abbildung 18: Analysieren der Subsystembildung – erster Schritt



Wir erinnern uns: diese Kanten entstehen ja lediglich dadurch, dass die Entwickler des Systems die Klassen bestimmten Subsystemen (oder Quellcode-Verzeichnissen) zugordnet haben. Wir wollen nun prüfen, ob die Entwickler dabei sinnvoll vorgegangen sind, daher entfernen wir diese „künstlichen“ Kanten und wenden auf den daraus entstehenden Graphen wieder den Spring-Embedder-Algorithmus an. Abbildung 19 zeigt das Ergebnis. Wir erkennen anhand der Farbgebung, dass wieder viele Klassen der jeweiligen Subsysteme nahe beieinander angeordnet werden. Allerding stellen wir auch fest, dass einige dunkel eingefärbten Klassen und einige hell eingefärbte Klassen sind zu Klassen anderer Subsysteme gesellen.

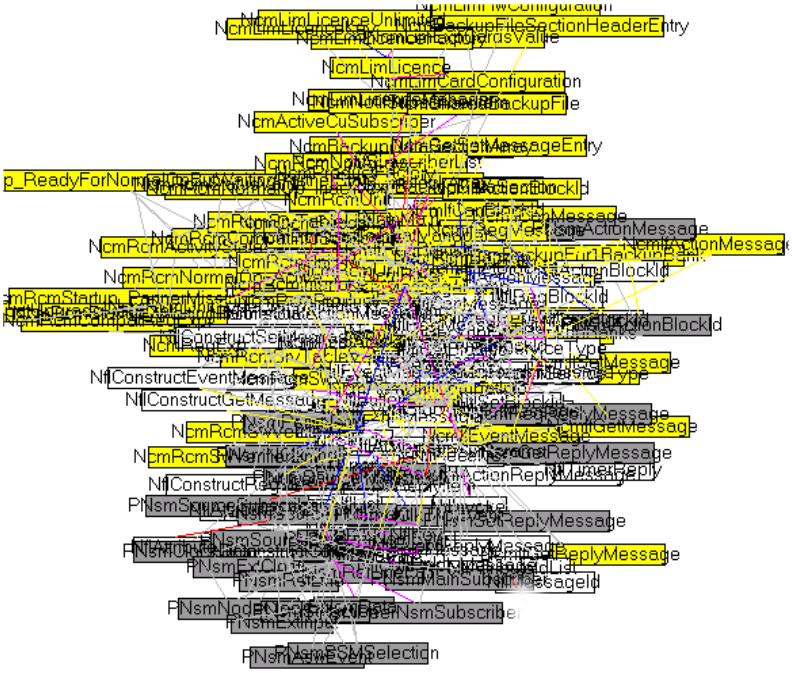


Abbildung 19: Analysieren der Substembildung – zweiter Schritt

Möglicherweise sollten im Rahmen von Restrukturierungsmaßnahmen einige dieser Ausreißer in andere Subsysteme umgruppiert werden. Allerdings ist es auch hier wieder angebracht, die durch Visualisierungen gewonnenen Erkentnisse anhand des Quellcodes nachzuvollziehen und zu bestätigen.

3.5 Clustering

Zur Clusteranalyse gehören eine Reihe von Ansätzen, die dazu dienen, Elemente einer Menge anhand von Ähnlichkeiten hinsichtlich bestimmter Eigenschaften zu klassifizieren. *Clusteringverfahren* spielen bei der *(Re-)Modularisierung* von Softwaresystemen eine große Rolle ([Wig97], [Eve74]) und können wesentlich zum *strukturellen Verständnis* eines Softwaresystems beitragen.

Wir wollen im folgenden kurz skizzieren, wie solche Clusteringverfahren funktionieren. Betrachten wir dazu zunächst einmal die Grundform eines Clustering-Algorithmus, wie sie in Abbildung 20 angegeben ist. Diese Grundform des Algorithmus fasst Entitäten aufgrund eines *Ähnlichkeitskriteriums* zu Clustern zusammen, bis diese dem *Abbruchkriteriu*m, in dem wünschenswerte Eigenschaften der zu bestimmenden Cluster angegeben werden können, genügen. Diese Form des Algorithmus wird als *zusammenfassendes Clustering (hierachical agglomerative clustering, HAC)* bezeichnet.

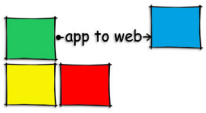


Abbildung 20: Grund form eines Clusteral-gorithmus

1. Jede Entität bildet ein Cluster
2. **Wiederhole**
3. Identifiziere die Cluster Ci ; Cj ,die am *ähnlichsten* zueinander sind
4. Kombiniere Ci ; Cj zu einem neuen Cluster und halte ihr Ähnlichkeitsniveau fest
5. **bis gilt:** alle Cluster Ci verfügen über zufriedenstellende Eigenschaften **oder** es ist nur noch ein Cluster übrig.

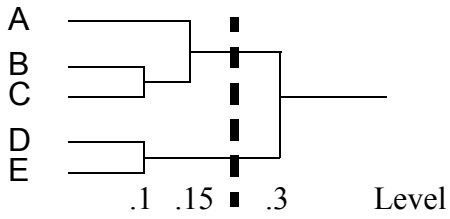
Wenn wir nun diesen Grundalgorithmus auf das Problem, Subsysteme in Softwaresystemen zu identifizieren, übertragen wollen, so können wir folgendermaßen vorgehen:

- Als Entitäten verwenden wir die Klassen des Systems, die entstehenden Cluster sind die gesuchten Subsysteme.[1]
- Das Ähnlichkeitskriterium können wir über *statische Abhängigkeiten* zwischen Klassen definieren: Klassen, die stark gekoppelt sind, gelten als ähnlich.
- Das Abbruchkriterium definieren wir über wünschenswerte Eigenschaften für die entstehenden Subsystemkandidaten, die wir über Software-Metriken bzw. Heuristiken [BCD+98] messen können:
  - geringe externe Kopplung
  - hohe interne Kohäsion
  - vernünftige interne Komplexität
  - Schnittstelleneigenschaften

Das Ergebnis des Algorithmus wird üblicherweise in Form eines *Dendrogrammes* angegeben. Ein Dendrogramm ist ein Baum, dessen Blätter die zu clusternden Ausgangselemente sind und dessen innere Knoten den Clustern entsprechen, die während der Abarbeitung des Verfahrens entstehen. Das Ähnlichkeitsniveau, bei dem ein bestimmtes Cluster gebildet wurde, kann dem Dendrogramm entnommen werden. Es gibt Auskunft darüber, wie gut die Elemente eines Clusters zusammen passen. In Abbildung 21 beispielsweise wurden die Cluster A und (B,C) auf dem Ähnlichkeitsniveau von 0.15 zu einem neuen Cluster zusammengefasst.

1 Natürlich lässt sich das Verfahren auch auf nicht-objektorientiert implementierte Systeme anwenden. In diesem Fall verwendet man als zu clusternde Entitäten Unterprogramme oder Unterprogramme und globale Variablen.

Abbildung 21: Dendrogramm



Clusteringverfahren wie HAC können auf Klassen, Module und Funktionen angewandt werden, um diese anhand ihrer Beziehungen und Abhängigkeiten zu Subsystemen zu gruppieren. Allerdings liefert HAC nicht direkt Subsysteme oder Komponenten des Systems, sondern vielmehr eine Familie von Zerlegungen des Systems in Form des Dendrogrammes. Dieses Dendrogramm muss dann erst noch passend geschnitten werden, um passende Subsysteme oder Komponenten zu erhalten. In Abbildung 21 haben wir einen solchen Schnitt vorgenommen – demnach erhalten wir zwei Komponenten, eine Komponene enthält die Entitäten A,B,C; die andere die Entitäten D,E.

In der Praxis ist es sehr entscheidend, an welcher Stelle man die Schnitte durchführt, sie bestimmen wesentlich das Ergebnis der Modularisierung. Oft zieht man Experten zu Rate, die dann beispielsweise *bottom-up* durch das Dendrogramm gehen und überlegen, ab welchem Niveau Cluster zusammengefügt werden, zwischen denen kein logischer Zusammenhang mehr besteht. In großen Systemen ist dieser Prozess allerdings aufwendig und schwierig, so dass die Tauglichkeit solcher Verfahren in der Praxis oft angezweifelt wird. Dennoch können solche Clusteringverfahren wertvolle Hinweise für die Redokumentation von Softwarearchitekturen geben.

3.6 Softwaremaße, Metriken

Metriken messen bestimmte Eigenschaften eines Software-Projektes, indem sie diese Eigenschaften mit Hilfe von klar definierten, objektiven Messvorschriften auf Zahlen (oder andere Werte) abbilden [LK94]. Beispiele für Softwaremetriken sind u.a. Zahlenwerte, die Auskunft über die Größe und Komplexität einer Software geben oder die Kopplung von Systemelementen berechnen. Wir nennen solche Metriken Software-Produktmetriken, weil sie Eigenschaften des Softwareproduktes messen.

Ein bekanntes Beispiel für eine Software-Produktmetrik ist *LOC*, deren Defintion wir in Tabelle 1 kurz darstellen.

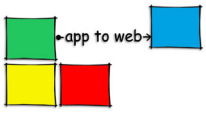


Tabelle 1: Defintion der Metrik LOC

LOC – Lines Of Code	
Bezug	System, Datei, Klasse, Methode
Kategorie	Komplexitätsmetrik
Beschreibung	*Mißt die Größe eines Quellcode-Stücks durch Zählen der Codezeilen.* Da die Größe eines Quellcode-Stücks als Indikator für seine Komplexität dienen kann, wird die *LOC*-Metrik oft als Komplexitätsmaß oder als Maß für den Aufwand, der benötigt wurde, diesen Quellcode zu erstellen, verwendet. Das Zählen der Zeilen wird üblicherweise unter Verwendung eines Codier-Standards vorgenommen, der exakt definiert, was als eine zu zählende Programmzeile in der verwendeten Programmiersprache gilt. Insbesondere muss festgelegt werden, ob Leer- und Kommentarzeilen als Codezeilen zählen, oder ignoriert werden sollen. Dadurch gewährleistet man vergleichbare, klar definierte Meßergebnisse.
Verwandte Metriken	–
Literatur	[Hum95] gibt eine gute und detaillierte Einführung in die LOC-Metrik.

Die Metrikergebnisse können dazu verwendet werden, um sinnvolle Informationen zu gewinnen, mit denen Aktivitäten in Software-Projekten besser geplant und durchgeführt werden können. So kann eine Analyse des Systems mit Hilfe von Metriken von Nutzen sein, um einen ersten Überblick über das System und seine Bestandteile zu bekommen. Dabei ist es oft von Interesse, herauszufinden, welche Bestandteile des Systems (d.h. welche Klassen) die wichtigsten Konzepte des Systems implementieren. Eine Vorgehensweise, dies herauszufinden ist in [Bau99] beschrieben:

In der Regel sind jene wichtigsten Konzepte eines Systems in einigen wenigen *Schlüsselklassen* implementiert worden. Schlüsselklassen können durch folgende Eigenschaften charakterisiert werden:

- Schlüsselklassen verwalten oder verwenden eine größere Anzahl anderer Klassen um die ihnen zugeordnete Funktionalität zu realisieren, also sind sie eng mit anderen Teilen des Systems *gekoppelt*.
- Schlüsselklassen sind relativ *komplex* aufgebaut, da sie einen Großteil der Funktionalität des Systems implementieren müssen.

Wenn wir diese Eigenschaften ausnützen, können wir die Schlüsselklassen eines Systems leicht ermitteln, indem wir sowohl eine Komplexitätsmetrik als auch eine Kopplungsmetrik einsetzen, also für alle Klassen des Systems Komplexitäts- und Kopplungswerte berechnen, und die Meßergebnisse kombinieren.

Als geeignete Metriken haben sich dafür die Komplexitätsmetrik *WMC* und die Kopplungsmetrik *DAC* erwiesen.

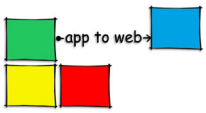


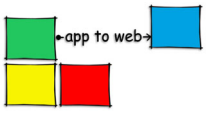
Tabelle 2: Defintion der Metrik WMC

WMC – Weighted Method Count	
Bezug	Klasse
Kategorie	Komplexitätsmetrik
Beschreibung	*Mißt die Komplexität einer Klasse durch Aufaddieren der Komplexität aller Methoden, die in der Klasse definiert sind*. Genauer, $$WMC = \sum_{i=1}^{n} c_i$$ wobei c_i die Komplexität der Methode i bezeichnet. Die Komplexität der Methoden wird üblicherweise durch gewöhnliche Code-Komplexitätsmaße berechnet – beispielsweise durch die *LOC*-Metrik oder durch die *McCabe-Komplexität*. Die McCabe-Komplexität mißt die Komplexität eines Programmstückes anhand der Entscheidungsknoten im Code. Code, der viele *if-then-else*-Konstrukte oder Schleifen enthält wird dadurch als besonders komplex eingestuft.
Verwandte Metriken	*NOM* ist ein Spezialfall dieser Metrik – jede Methode hat die Komplexität 1.
Literatur	[CK94]

Tabelle 3: Defintion der Metrik DAC

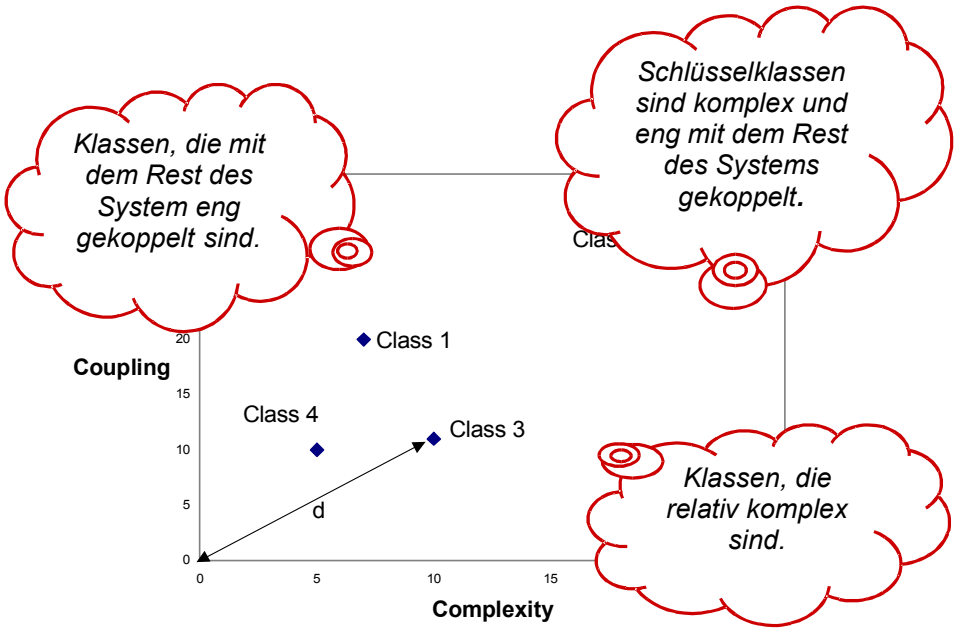
DAC – Data Abstract Coupling	
Bezug	Klasse
Kategorie	Kopplungsmetrik
Beschreibung	Misst die Kopplung zwischen Klassen, die durch Attributdeklarationen entsteht. *DAC* zählt die Anzahl der abstrakten Datentypen, die in einer Klasse definiert sind. Im wesentlichen modelliert eine Klasse einen abstrakten Datentyp, daher entspricht *DAC* der Anzahl komplexer Attribute einer Klasse. Komplexe Attribute sind solche Attribute, die als Typ eine andere Klasse des Systems haben – im Gegensatz zu einfachen Attributen mit elementaren Typen wie `int`, `char`,..
Verwandte Metriken	*RFC – Response Set for a Class [CK94] und CBO – Coupling Between Objects* [CK94] können als Alternativen zu dieser Kopplungsmetrik verwendet werden.
Literatur	[LH93]

Beide Metriken sind relativ einfach zu berechnen und bereits seit einiger Zeit als nützliche Produktmetriken etabliert. Sie sind daher auch in einigen Metrikwerkzeugen enthalten und können daher leicht verwendet werden. So enthält beispielsweise das CASE-Werkzeug *Together* eine geeignete Implementierung beider Metriken. Auf Basis geeigneter Metamodelle (siehe Abschnitt 3.3) lassen sich diese und viele weitere Metriken jedoch auch recht einfach selbst implementieren.



Um die Metriken zu kombinieren, schlagen wir hier einen recht intuitiven Weg vor: Wir verwenden ein Diagramm und führen damit eine grafische Auswertung durch. Die folgende Abbildung zeigt beispielhaft ein Diagramm mit dessen Hilfe wir eine solche Analyse durchführen können. Dazu platzieren wir die Klassen des Systems gemäß ihrer Komplexitäts- und Kopplungsmaße in ein Koordinatensystem. Klassen, die sowohl komplex (hohe WMC-Werte) als auch eng mit dem Rest des Systems gekoppelt sind (hohe DAC-Werte) liegen dann in der rechten oberen Ecke des Diagramms und sind gute Kandidaten für Schlüsselklassen.

Abbildung 22: Grafische Identifikation von Schlüsselklassen mit Hilfe eines Schaubildes



Wir sollten demnach unser Hauptaugenmerk auf diese Klassen legen: Wenn es uns gelingt, diese Klassen zu verstehen, dann haben wir schon beträchtliches Wissen über Aufbau und Arbeitsweise eines Systems gelernt.

Natürlich werden in diese rechte obere Ecke auch einige wenige Klassen platziert, die keine wirklichen Schlüsselklassen darstellen. Bei diesen Klassen kann es sich oft lohnen, zu hinterfragen, warum sie so komplex sind und so viele Abhängigkeiten zu anderen Systemteilen aufweisen. Oftmals stößt man hier auf grundlegende Entwurfsprobleme, die gegebenenfalls beseitigt werden sollten, wenn das System oder Teile davon wiederverwendet oder aktiv weiterentwickelt werden sollen. Insbesondere weisen Systemteile mit starker Kopplung darauf hin, dass sich die Extraktion von

Komponenten mit dem Ziel diese in Web-Anwendungen zu integrieren, als schwierig gestalten kann.

3.7 Reflexionsmodelle

Gail Murphy et al. [MNS95], [Mur96] stellen eine Methode vor, die es einem Ingenieur ermöglicht, sein Verständnis eines existierenden Systems mit Hilfe struktureller Informationen aus dem Quellcode zu verfeinern. Im allgemeinen analysiert man nicht die vollständige Architektur des Systems, sondern konzentriert sich auf die fallspezifisch relevanten Systemaspekte (z.B. wenn spezifische Änderungen geplant sind). Anfänglich konzentrierte sich die Methode auf die Analyse von Funktionsaufrufen. Später wurde sie weiterentwickelt, um unterschiedliche Arten von Beziehungen analysieren zu können.

Input

Die Methode benötigt folgenden Input:

- Informationen, die aus dem Quellcode mittels eines einfachen lexikalischen Extraktionswerkzeugs gewonnen wurden (z.B. Routinen innerhalb eines Moduls und ihre gegenseitigen Aufrufbeziehungen).
- Ein abstraktes Modell des zu analysierenden Teilsystems in Form von erwarteten Komponenten und ihren Verbindungen untereinander.
- Eine Abbildung zwischen den Komponenten des abstrakten Modells und den Elementen des Quellcodes.

Ergebnis

Die Methodik liefert dem Analysten ein verfeinertes Modell, indem er kontinuierlich die Unstimmigkeiten in den Verbindungen zwischen den Komponenten des abstrakten Modells und den Elementen des Quellcodes analysiert und integriert.

Analystenrolle

Der Analyst hat die Aufgabe, ein abstraktes Modell des Systems zu entwickeln und die Abbildung der abstrakten Komponenten auf die Code-Module vorzunehmen. Mit der fortschreitenden Aufdeckung von Unstimmigkeiten zwischen beiden Modellen wird das abstrakte Modell angepasst.

Abbildung 23 veranschaulicht die Methode, die aus folgenden Schritten besteht:

1. Basierend auf System-, Fachgebietwissen und seinen Erwartungen erstellt der Analyst das abstrakte Modell.
2. Anschliessend definiert er eine Abbildung zwischen den Komponenten dieses Modells und den Modulen des Quellcodes. Das setzt natürlich eine grobe Kenntnis der Qellcodestrukturen voraus. Um die Darstellung der Abbildung nicht zu sehr aufzublähen setzt er Patterns ein.

3. Mit einem einfachen lexikalischen Extraktionswerkzeug extrahiert er relevante Informationen über das System (z.B. Funktionsaufrufe) und erstellt daraus ein Modell des Quellcodes [MN95].

   repeat //Schrittweise Verfeinerung: Schritt 4-6 werden wiederholt

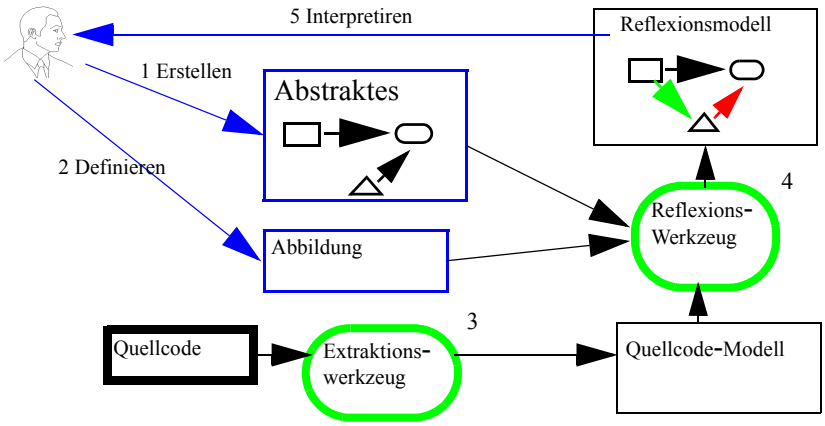
4 Das Reflexions-Werkzeug ermittelt die Unstimmigkeiten zwischen abstraktem und Quellcode-Modell. Das Werkzeug stellt beide Modelle und die Unstimmigkeiten in einem sogenannten Reflexionsmodell dar. Übereinstimmungen der Beziehungen in beiden Modellen, werden als durchgezogene Linien dargestellt. Verbindungen, die im Modell des Quellcodes vorhanden sind, aber im abstrakten Modell fehlen, werden als Abweichungen bezeichnet und als gepunktete Linie dargestellt. Sind umgekehrt im abstrakten Modell Beziehungen enthalten, die im QellcodeModell fehlen, werden diese als Abwesenheiten bezeichnet und als gestrichelte Linie dargestellt.

5 Der Analyst interpretiert das Reflexionsmodell. Abwesenheiten und Abweichungen werden durch eine tiefergehende Analyse der Software weiter untersucht.

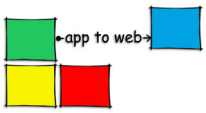
6 Falls das Reflexionsmodell noch nicht seinen Ansprüchen genügt, verfeinert der Analyst das abstrakte Modell und/oder die Abbildung. Z.B. indem er eine Klasse einer anderen Komponente zuordnet. In einigen Fällen kann es auch erforderlich sein, den Extraktionsschritt nochmal durchzuführen, um etwa Informationen über Teile des Quellcodes zu erhalten, die zu Beginn der Analyse noch nicht relevant erschienen.

   until // Übereinstimmung beider Modelle seinen Anforderungen genügt

Abbildung 23:Reflexionsmodell - Verfahren



Erfahrungen Die Methodik wurde bereits bei einigen Systemen auf ausgewählte Teilsysteme angewandt. Namentlich die virtuelle Speicherverwaltung von NetBSD (das komplette

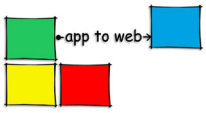


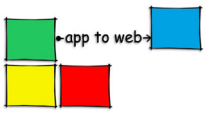
System besteht aus 250.000 Zeilen C-Code, das Teilsystem enthält 15000 Beziehungen zwischen 3000 Elementen), ein Teilsystem des gnu C Compilers gcc(), sowie Excel (1200.000 Zeilen Code).

In einer Fallstudie bei Microsoft wurde sie von einem Entwickler mit 10 Jahren Microsoft Entwicklungserfahrung durchgeführt. Im Rahmen eines experimentellen Reengineering von Excel, bei dem es darum ging, Komponenten aus dem Quellcode heraus zu identifizieren und zu extrahieren, wandte er eine leicht modifizierte Version der Methode an. Innerhalb eines Tages ermittelte er ein erstes Reflexionsmodell von Excel, das er in den folgenden 4 Wochen ständig verfeinerte. Das Modell enthält 15 Komponenten mit 19 Verbindungen zwischen ihnen. Er schätzte, dass er ohne die Methode "ca. 2 Jahre gebraucht hätte, um dasselbe Verständnis für den Quellcode zu entwickeln".

Bewertung

Dieser Ansatz könnte in APPLICATION2WEB verwendet werden, um einen ersten Überblick über den bekannten Teil des Systems zu gewinnen. Dort erscheint es ausserdem sehr vielversprechend, den Systemteil zu untersuchen, der mit Web-Komponenten verbunden werden soll.





4 Werkzeugunterstützung

Viele der Aufgaben aus Abschnitt 3 benötigen Werkzeugunterstützung, um mit vertretbarem Aufwand und verlässlichem Ergebnis durchführbar zu sein. In diesem Kapitel beschreiben wir die Werkzeuge, die wir unter anderem bei Redokumentationsaufgaben erfolgreich eingesetzt haben. Teilweise sind die Werkzeuge selbst entwickelt, teilweise handelt es sich um Fremdwerkzeuge.

4.1 Quellcodeanalyse, Fact Extraction

4.1.1 Goose

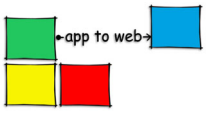
Goose [BBCS] besteht aus einer Menge von Werkzeugen zur Analyse des Entwurfs eines objektorientierten Softwaresystems, die am FZI entwickelt wurden. Grundidee ist die Gewinnung der Basisinformation über den Entwurf aus dem vorhandenen Quelltext des zu analysierenden Systems. Diese Information wird in einer Designdatenbank in textueller Form abgelegt. In diesem ersten Schritte werden die Entitäten des Systems wie Klassen, Methoden, Attribute etc. und ihre Beziehungen untereinander ermittelt. Diese Basisinformation kann weiterverarbeitet werden, in dem z.B. Klassen zu höheren Abstraktionsebenen zusammengefasst, oder potentielle Entwurfsmängel berechnet werden.

Goose kann zum jetzigen Zeitpunkt mit Java und C/C++ umgehen. Für jede dieser Programmiersprachen wurde ein eigener Parser entwickelt. Durch weitere Parser kann das Spektrum der Programmiersprachen erweitert werden.

4.1.1.1 SRFGenerator

Der SRFGenerator ist ein Kommandozeilenwerkzeug zur Fact Extraction aus Java Systemen. Die Information kann sowohl aus dem Quelltext als auch aus dem Bytecode des Systems gewonnen werden, und wird in Form einer Texttabelle auf der Standardausgabe ausgegeben.

Der SRFGenerator basiert auf dem Parser von RECODER [LH01], einem Java Rahmenwerk zur Quelltext-Metaprogrammierung. Da die Quellen der zu untersuchenden Systeme nicht immer komplett und fehlerfrei sind, wurde der vollständige Parser von RECODER robuster gemacht.Wie RECODER ist auch der SRFGenerator komplett in Java geschrieben, so dass er plattformunabhängig eingesetzt werden kann.



Es wurde schon mehrere Systeme mit dem SRFGenerator analysiert. Darunter befanden sich sehr große Systeme mit knapp einer Million Zeilen Quelltext, die in kürzester Zeit und ohne Probleme auf handelsüblichen Rechnern bearbeitet werden konnten.

4.1.1.2 TableGen

Mit TableGen können C/C++ Systeme verarbeitet werden. Es handelt sich wie bei dem SRFGenerator um ein kommandozeilenbasiertes Werkzeug, das den Quelltext des zu analysierenden Systems einliest und die statische Designinformation in Form von Tabellen ausgibt. Da TableGen in C++ geschrieben ist und auf dem FAST-Environment C/C++ Parser basiert, den es nur für Solaris und Windows gibt, ist es nur für diese Plattformen verfügbar.

TableGen skaliert, es konnten problemlos Fallstudien mittleren und grossen Umfangs mit über einer Million Quelltextzeilen bearbeitet werden. TableGen ist wie der SRFGenerator in der Lage, mit unvollständigem, fehlerhaftem Quelltext umzugehen. Es werden verschiedene C++ Stile wie z..B Borland C++ oder Microsoft Visual C unterstützt.

4.1.2 Weitere Werkzeuge zur Extraktion

Am IESE kamen verschiedene weitere Werkzeuge zum Einsatz: Auf FAST basierende Werkzeuge zur Faktenextraktion aus Java. Für C++ werden zwei Werkzeuge benutzt. Zum einen der Export-Mechanismus von Visual Studio für C++, zum anderen ein weiteres auf FAST basierendes Werkzeug. Für Delphi-Systeme wurde am IESE ein eigener Extraktor mit Hilfe von Lex und Yacc entwickelt.

4.2 Visualisierungen

4.2.1 Rigi und Erweiterungen

Für die interaktive Visualisierung hat IESE hauptsächlich Rigi [rig] und selbstsentwikkelte Erweiterungen desselben vervendet.

4.2.1.1 Rigi

Rigi ist ein System, das Reverse Engineering unterstützt, indem es aus extrahierten Quelltextinformationen einen Graphen erstellt, den man analysieren, modifizieren oder durch gezielte Filter an eine Fragestellung anpassen kann. Mit Rigi kann der Benutzer die grafische Darstellung konfigurierien. Er kann die Kanten, die Knoten,

und die Attribute der Knoten definieren. Damit lässt sich Rigi einfach an die jeweilige Programmiersprache und die spezifischen Projektbedürfnisse anpassen. Rigi bietet mehrere interessante Möglichkeiten zur Layoutgestaltung.

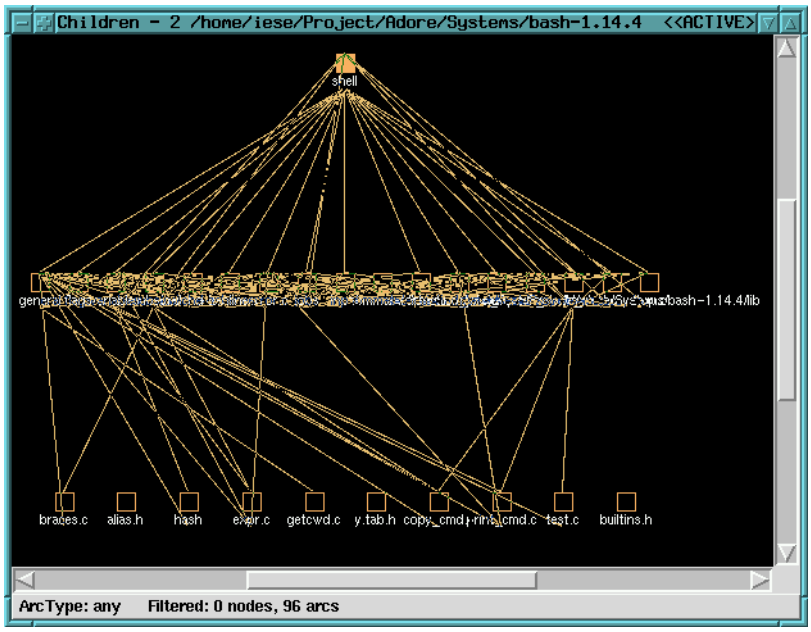


Abbildung 24: Beispiel für einen mehrstufigen Aufrufbaum.

4.2.1.2 Erweiterungen von Rigi

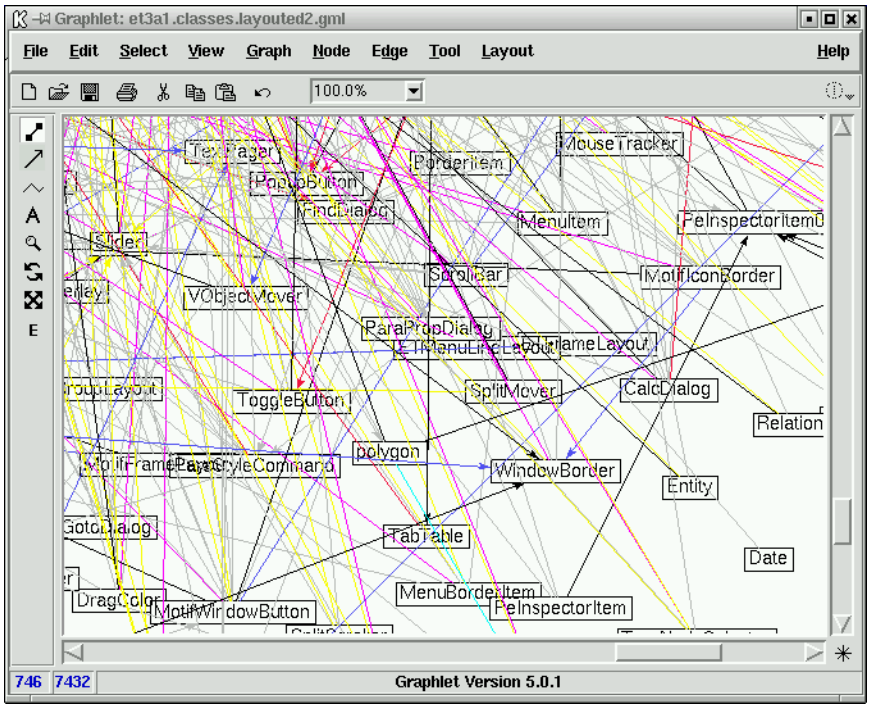
Die Erweiterungen für Rigi teilen sich im wesentlichen in zwei Gruppen auf. Einerseits gibt es Erweiterungen die das Erscheinungsbild der Modelle verändern um Zusatzinformationen durch graphische Attribute darzustellen. Dazu zählen die Knotenmarkierungen (deuten Verbindungen auf anderen Abstraktionsebenen an) und die variablen Knotengrößen (zeigen an, wieviel Unterknoten ein Knoten enhält).

Bei der zweiten Art von Erweiterungen handelt es sich um neu entwickelte Techniken, die einen Benutzer bei der Generierung einer High Level Beschreibung einer Software-Architektur aus einem existierenden Modell eines Software-Systems unterstützen sollen. Dazu zählen Erweiterungen zur vereinfachten Identifikation wichtiger Elemente (z.B. Namenskonventionen und Verbindungsmuster) und zur vereinfachten Komponentenkonstruktion die der Erzeugung einer solchen Beschreibung dienen, sowie die Techniken zur Korrektur einer gewonnenen Software-Architektur.

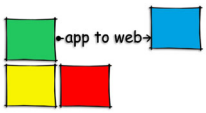
4.2.2 Graphlet

Graphlet [Gra] ist ein Werkzeug zur Bearbeitung von Graphen, das am FZI eingesetzt wird. Es besteht aus Graphscript, einer auf Tcl/Tk basierenden, erweiterbaren Programmiersprache für Graphalgorithmen mit Benutzerschnittstelle, einem skriptfähigen, anpassbaren Editor und einer Menge von Graphalgorithmen.

Abbildung 25: Visualisierung mit Graphlet



Mit Graphlet können im GML-Format vorliegende Graphen angezeigt und bearbeitet werden. Zur übersichtlicheren Visualisierung werden Algorithmen aus den Bereichen Baumlayouts, Hierarchische Layouts und Feder-Masse Layouts angeboten. Leider gibt es keine Möglichkeit, eine Menge von Knoten zu einem einzigen Knoten zu aggregieren, was insbesondere die Bearbeitung von großen Graphen erschwert. Besteht ein Graph aus mehreren tausend Knoten und Kanten, so machen sich starke Geschwindigkeitseinbußen bemerkbar.Graphlet wird innerhalb des Werkzeuges Goose dazu benutzt, die gewonnene Information über das System zu visualisieren.



4.3 Metriken

4.3.1 M-System

Zur Ermittlung von statischen Softwaremassen von objektorientierten Java oder C++ - Systemen hat IESE das M-System entwickelt. Als Input verwendet dieses Werkzeug die mittels Faktenextraktoren gewonnenen Informationen (siehe Abschnitt 3.2). Das M-System berechnet die meisten der in der Literatur vorgeschlagenen OO-Metriken (34 bzgl. Kopplung, 10 bzgl. Kohäsion, 11 bzgl. Vererbung und 5 bzgl. Grösse). Wie in Abschnitt 2.6 schon erwähnt, können diese statischen Messergebnisse des M-Systems von einem Experten für Reverse Engineering Aktivitäten genutzt werden. Sie wurden auch schon bei der Entwicklung von Vorhersagemodellen und zur Verfolgung der Systementwicklung eingesetzt.

4.3.2 Together Control Center

Together [Tog] ist eine integrierte Entwicklungsumgebung, die Werkzeuge für fast alle Phasen und Aktivitäten moderner Softwareentwicklungsprozesse bietet. Es ist in verschiedenen Ausführungen erhältlich. Die abgespeckte Solo Version bietet keine Unterstützung für Metriken, die normale Version hingegen enthält 47 Metriken für Java, 16 Metriken für C++ und weitere für Metriken VB 6.0, .NET und C#.

Zur Messung der Komplexität des Quelltextes werden z. B. einfachere Metriken wie LOC oder komplexere wie McCabe Cyclomatic Complexity unterstützt. Um die kognitive Komplexität zu messen sind viele der Halstead Metriken enthalten. Das verwendete LCOM-Paket (Lack of Cohesion in Methods) [CK94] bietet eine Menge von Kohäsions-Metriken, DAC [LH93] und CBO [CK94] dienen als Kopplungsmetriken. Eine grosse Anzahl der Metriken befasst sich mit den Vererbungsbeziehungen eines Systems. Unter diesen Metriken befinden sich sowohl einfache Metriken wie DIT (Depth of Inheritance Tree) und NOC (Number Of Children), als auch komplexere Metriken wie NOOM (Number of Overriden Methods).

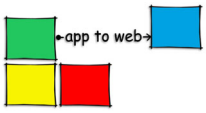
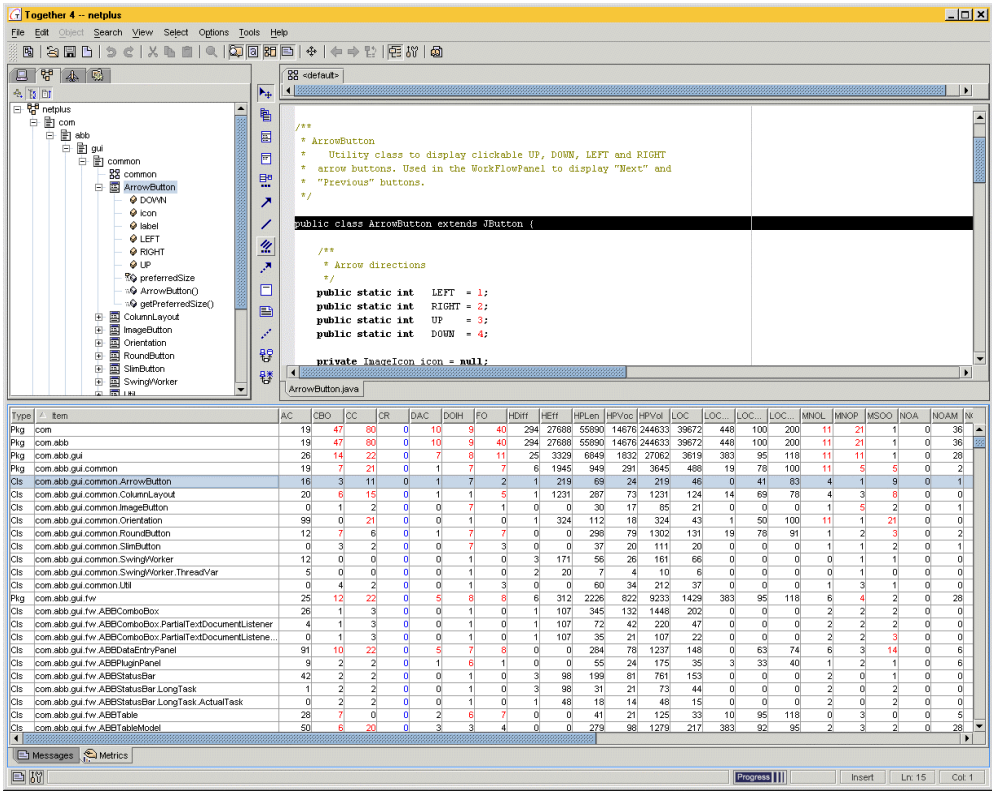


Abbildung 26: Ergebnisse der Together Metriken in Tabellenform



Type	Item	AC	CBO	CC	CR	DAC	DOIH	FO	HDiff	HEff	HPLen	HPVoc	HPVol	LOC	LOC...	LOC...	LOC...	MNOL	MNOP	MSOO	NOA	NOAM	NI
Pkg	com	19	47	80	0	10	9	40	294	27688	55890	14676	244633	39672	448	100	200	11	21	1	0	36	
Pkg	com.abb	19	47	80	0	10	9	40	294	27688	55890	14676	244633	39672	448	100	200	11	21	1	0	36	
Pkg	com.abb.gui	26	14	22	0	7	8	11	25	3329	6849	1832	27062	3619	383	95	118	11	11	1	0	28	
Pkg	com.abb.gui.common	19	7	21	0	1	7	7	6	1945	949	291	3645	488	19	78	100	11	5	5	0	2	
Cls	com.abb.gui.common.ArrowButton	16	3	11	0	1	7	2	1	219	69	24	219	46	0	41	83	4	1	9	0	1	
Cls	com.abb.gui.common.ColumnLayout	20	6	15	0	1	1	5	1	1231	287	73	1231	124	14	69	78	4	3	8	0	0	
Cls	com.abb.gui.common.ImageButton	0	1	2	0	0	7	1	0	0	30	17	85	21	0	0	0	1	5	2	0	1	
Cls	com.abb.gui.common.Orientation	99	0	21	0	0	1	0	1	324	112	18	324	43	1	50	100	11	1	21	0	0	
Cls	com.abb.gui.common.RoundButton	12	7	6	0	1	7	7	0	0	298	79	1302	131	19	78	91	1	2	3	0	2	
Cls	com.abb.gui.common.SlimButton	0	3	2	0	0	7	3	0	0	37	20	111	20	0	0	0	1	1	2	0	1	
Cls	com.abb.gui.common.SwingWorker	12	0	0	0	0	1	0	3	171	56	26	161	66	0	0	0	0	1	1	0	0	
Cls	com.abb.gui.common.SwingWorker.ThreadVar	5	0	0	0	0	1	0	2	20	7	4	10	6	0	0	0	0	1	0	0	0	
Cls	com.abb.gui.common.Util	0	4	2	0	0	1	3	0	0	60	34	212	37	0	0	0	1	3	1	0	0	
Pkg	com.abb.gui.fw	25	12	22	0	5	8	8	6	312	2226	822	9233	1429	383	95	118	6	4	2	0	28	
Cls	com.abb.gui.fw.ABBComboBox	26	1	3	0	0	1	0	1	107	345	132	1448	202	0	0	0	2	2	2	0	0	
Cls	com.abb.gui.fw.ABBComboBox.PartialTextDocumentListener	4	1	3	0	0	1	0	1	107	72	42	220	47	0	0	0	2	2	2	0	0	
Cls	com.abb.gui.fw.ABBComboBox.PartialTextDocumentListene...	0	1	3	0	0	1	0	1	107	35	21	107	22	0	0	0	2	2	3	0	0	
Cls	com.abb.gui.fw.ABBDataEntryPanel	91	10	22	0	5	7	8	0	0	284	78	1237	148	0	63	74	6	3	14	0	6	
Cls	com.abb.gui.fw.ABBPluginPanel	9	2	2	0	1	6	1	0	0	55	24	175	35	3	33	40	1	2	1	0	6	
Cls	com.abb.gui.fw.ABBStatusBar	42	2	2	0	0	1	0	3	98	199	81	761	153	0	0	0	2	0	1	0	0	
Cls	com.abb.gui.fw.ABBStatusBar.LongTask	1	2	2	0	0	1	0	3	98	31	21	73	44	0	0	0	2	0	2	0	0	
Cls	com.abb.gui.fw.ABBStatusBar.LongTask.ActualTask	0	2	2	0	0	1	0	1	48	18	14	48	15	0	0	0	2	0	2	0	0	
Cls	com.abb.gui.fw.ABBTable	28	7	0	0	2	6	7	0	0	41	21	125	33	10	95	118	0	3	0	0	5	
Cls	com.abb.gui.fw.ABBTableModel	50	6	20	0	3	3	4	0	0	279	98	1279	217	383	92	95	2	3	2	0	28	

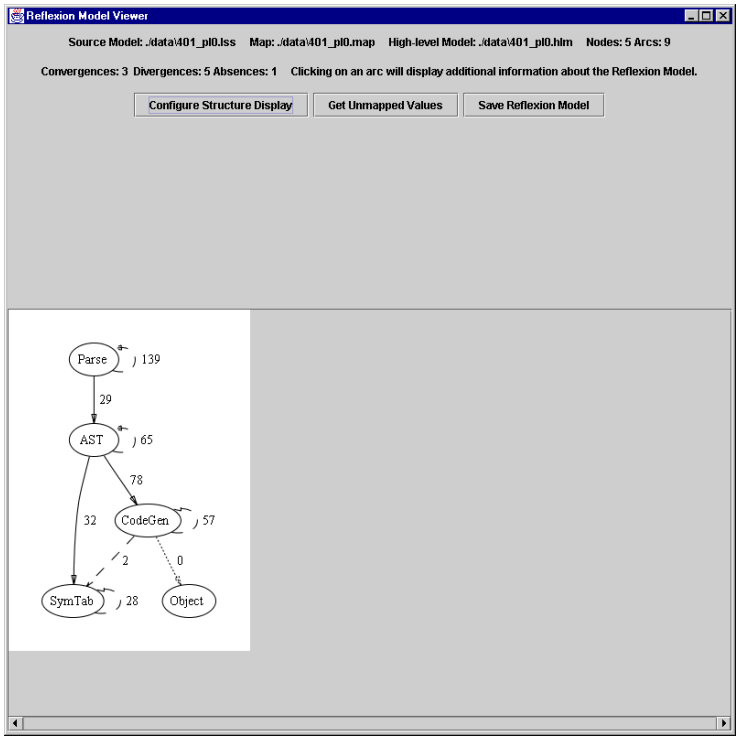
Die Ergebnisse der sich auf Klassen beziehenden Metriken können ausser bei den Kohäsions- und Kopplungsmetriken auf Subsystemebene zusammengefasst werden. Leider gibt es in Together kein Prozessmodell und keine Vorgehensweise, die dem Benutzer die Auswahl der Metriken erleichtert. Auswahl und Interpretation bleiben dem Benutzer überlassen, so dass dieser über fundierte Kentnisse im Bereich Metriken verfügen muss, um gute Ergebnisse zu erhalten.

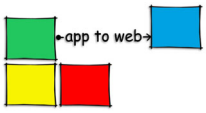
4.4 Reflexionsmodelle

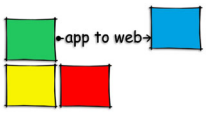
Für die Berechnung von Reflexionsmodellen (siehe Abschnitt 3.7) kann das Werkzeug jRMTool verwendet werden, das in Abbildung 27 dargestellt ist. Das IESE

arbeitet an einer erweiterten Reimplementierung, die die interaktive Modifikation von Abbildungen und Modellen vereinfacht.

Abbildung 27
JRMTool GUI







5 Literatur

[AAB+00] Michalis Anastasopoulos, Christoph Andriessens, Holger Bär, Christian Bunse, Jean-Francois Girard, Isabel John, Dirk Muthig, and Tim Romberg. Überblick Stand der Technik. Technical report, Application2Web, December 2000.

[ABB+01] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Juergen Wuest, and Joerg Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2001.

[Age95] Ole Agesen. The cartesian product algorithm: Simple and precise typing of parametric polymorphism. Number 952, pages 2–26. Springer-Verlag, New York, N.Y., 1995.

[Bau99] Markus Bauer. Analysing software systems using combinations of metrics. In Oliver Ciupke and Stéphane Ducasse, editors, *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, FZI Report, May 1999.

[BBCS] Holger Baer, Markus Bauer, Oliver Ciupke, and Olaf Seng. Goose. http://www.fzi.de/prost/tools/goose/.

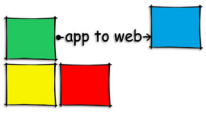
[BCD+98] Holger Bär, Oliver Ciupke, Serge Demeyer, Stéphane Ducasse, Radu Marinescu, Robb Nebbe, Tamar Richner, Matthias Rieger, Benedikt Schulz, Sander Tichelaar, and Joachim Weisbrod. The famoos handbook of reengineering, September 1998.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[Bis92] Walter R. Bischofberger. Sniff: A pragmatic approach to a c++ programming environment. In *C++ Conference*, pages 67–82, 1992.

[CI90] E. Chikofsky and J. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, January 1990.

[Ciu99] Oliver Ciupke. Automatic Detection of Design Problems in Object-Ori-



ented Reengineering. In Donald Firesmith, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 18–32, Santa Barbara, CA, August 1999. IEEE Computer Society.

[Ciu02] Oliver Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen*. PhD thesis, Universität Karlsruhe, 2002.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[DS98] Charles Donnelly and Richard Stallman. Bison the yacc-compatible parser generator. http://www.gnu.org/manual/bison/html_mono/bison.html, 1998.

[DTS99] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Bern, August 1999.

[ejb] Ejb homepage. http://java.sun.com/products/ejb/.

[Eve74] B. Everitt. *Cluster Analysis*. Heinemann Education Books Lted, 1974.

[Fow99] Martin Fowler. *Refactorings (Working Title)*. Addison-Wesley, 1999.

[GK95] H. Gall and R. Klˆsch. *Objektorientiertes Reverse Engineering*. Springer-Verlag, Berlin Heidelberg, 1995.

[GKS97] J.-F. Girard, R. Koschke, and G. Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In *Working Conference on Reverse Engineering*, pages 66–75, Amsterdam, The Netherlands, Oct. 1997.

[GKS99] J.-F. Girard, R. Koschke, and G. Schied. A Metric-based Approach to Detect Abstract Data Types and State Encapsulations. *journal of automated software engineering*, 6:357–386, Oct. 1999.

[Gol97] Wolfgang Golubski. Datenflu?analyse in objektorientierten programmiersprachen. Habilitationsschrift an der Universität Siegen, 1997.

[Gra] Graphlet. http://www.infosun.fmi.uni-passau.de/Graphlet/.

[Gro98] Object Management Group. XML Metadata Interchange (XMI). Technical Report ad/98-10-05, Object Management Group, February 1998.

[GW84] G. Goos and W. Waite. Compiler construction, 1984.

[Hum95] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Massachusetts, 1995.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Sch:urr. GXL: Towards a standard exchange format. In *Proceedings WCRE'00*, November 2000.

[inj] Injectj homepage. http://injectj.fzi.de.

[Jav01] Java compiler compiler - the java parser generator. http://www.web-gain.com/products/java_cc/, 2001.

[jbu] Jbuilder homepage. http://www.borland.com/jbuilder/.

[KGW98] R. Koschke, J.-F. Girard, and M. Wuerthner. An intermediate representation for reverse engineering analyses. In *Working Conference on Reverse Engineering*, pages 241–250, Hawai, USA, Oct 1998. IEEE, IEEE Computer Society Press.

[KWC98] R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of WCRE'98*, pages 154–163. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[LH93] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. *Proceedings of the First International Software Metrics Symposium.*, pages 52–60, May 1993.

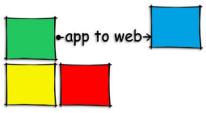
[LH01] Andreas Ludwig and Dirk Heuzeroth. Recoder. http://recoder.sourceforge.net/, 2001.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1994.

[MN95] G. Murphy and D. Notkin. Lightweight source model extraction. In *ACM SIGSOFT '95: Proceedings of the Third Symposium on the Foundations of Software Engineering (FSE3), m (Washington, D.C.; Oct 10-13, 1995)*, October 1995.

[MNS95] G.C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT*, Washibgton D.C., 1995. ACM.

[Mur96] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, July 1996.



[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[OL01] Doug Orleans and Karl Lieberherr. Dj: Dynamic adaptive programming in java. http://www.ccs.neu.edu/research/demeter/biblio/DJ-reflection.html, 2001.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[oWFNW90] University of West Florida Norman Wilde. Understanding program dependencies. Technical Report SEI-CM-26 ADA235700, Software Engineering Institute (Carnegie Mellon University), August 1990.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. http://st-www.cs.uiuc.edu/users/brant/Refactory/Refac toring-Browser.html, April 1997.

[rig] Rigi homepage. http://www.rigi.csc.uvic.ca.

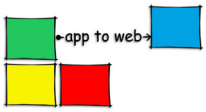
[Szy98] Clemens A. Szyperski. *Component Software*. Addison-Wesley, 1998.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[Tog] Togethersoft home page. http://www.togethersoft.com/.

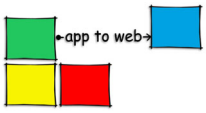
[Tra01] Martin Trapp. *Optimierung objektorientierter Programme: ‹berset-zungstechniken, Analysen und Transformationen*. Xpert.press. Springer Verlag, 2001.

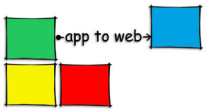
[Wig97] T. A. Wiggert. Using Clustering Algorithms in Legacy Systems Remodularization. In *Working Conference on Reverse Engineering*, pages pp. 33–43, Amsterdam, October 1997. IEEE Computer Society Press.



II Identifikation von Komponentenkandidaten in Altsystemen

Markus Bauer
Jean-François Girard





6 Einführung

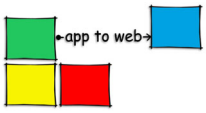
Wir erinnern uns: die Zielsetzung des Arbeitspaketes *Migration zu Webanwendungen* des Projektes APPLICATION2WEB war es, Reengineering-Techniken zu entwickeln, zu dokumentieren und zu erproben, die die Migration zu komponenten-orientierten Webanwendungen unterstützen. Ein wesentlicher Bestandteil dieser Migrationsaufgabe ist es, bereits existierenden Quellcode so umzuformen, daß dessen Funktionalität zumindest in Teilen in komponentenorientiert aufgebaute Webanwendungen integriert werden kann.

Zu diesem Zweck müssen in einem ersten Schritt aus dem existierenden Quellcode Komponentenkandidaten gewonnen werden, die zusammenhängende, wiederverwendbare Teile der Funktionalität des Altcodes repräsentieren. Diese Kandidaten müssen dann in weiteren Schritten evaluiert und überarbeitet werden, so daß sich nach Abschluß der Migrationsarbeiten wiederverwendbare Komponenten ergeben.

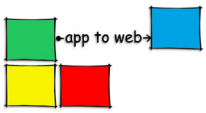
In diesem Dokument *Identifikation von Komponentenkandidaten* geht es vornehmlich darum, wie aus dem Quellcode bereits existierender Anwendungen mit Hilfe unterstützender Techniken gemäß bestimmter Kriterien geeignete Komponentenkandidaten ermittelt werden können. In der Regel wird dazu der gesamte Quellcode der zu migrierenden Anwendung mit Hilfe von *Clusterverfahren* untersucht und in geeignete Fragmente zerlegt. Diese Fragmente bilden dann mögliche Kandidaten für die später zu verfeinernden Komponenten.

Bevor mit der Identifikation von Komponentenkandidaten begonnen werden kann, ist ein Grundverständnis über Funktionsweise und inneren Aufbau des Systems erforderlich. Entsprechende Grundlagen haben wir im ersten Teil dieses Dokumentes schon erarbeitet. Die dort vorgestellten Techniken zur Extraktion von Fakten aus dem Quellcode und zur Aufbereitung und Repräsentation dieser Fakten in strukturierter Form mittels geeigneter Metamodelle sind auch unmittelbare Voraussetzung für die Anwendung der in dem vorliegenden Dokument dargestellten Techniken zur Komponentenidentifikation.

Nach der Identifikation von Komponentenkandidaten müssen diese evaluiert werden, und es müssen diejenigen Komponentenkandidaten bestimmt werden, die in weiteren Schritten zu wiederverwendbaren Komponenten überarbeitet werden sollen. Hierbei leisten die in [AB+01] vorgestellten Techniken ebenfalls wertvolle Hilfen. Die Überarbeitung der ausgewählten Komponenten ist Gegenstand eines weiteren Dokumentes aus APPLICATION2WEB, COMPTRANS, welches die Ergebnisse der Aktivität *Verfahren zur Umwandlung von Komponenten* zusammenfassend darstellt.



Das vorliegende Dokument *Identifikation von Komponentenkandidaten* gibt in Kapitel 7 zunächst einen Überblick über die in APPLICATION2WEB erarbeitete Vorgehensweise zur Identifikation von Komponentenkandidaten und illustriert die dabei verwendete Basistechnik der *Clusteranalyse*. In den folgenden Kapiteln verfeinern wir diese Vorgehensweise und gehen auf ihre einzelnen Schritte genauer ein: In Kapitel 8 fassen wir die aus dem ersten teil des Dokumentes bekannten Techniken zur Faktenextraktion und zur Repräsentation dieser Fakten mittels Strukturmodellen nochmals kurz zusammen, da diese für das weitere Verständnis des Textes wesentlich sind. In Kapitel 9 stellen wir Kriterien auf, mit deren Hilfe Clusteranalysen auf Basis der Strukturmodelle geeignete Komponentenkandidaten identifizieren können. In Kapitel 10 vertiefen wir die eingangs bereits eingeführte Technik der hierarchischen Clusteranalyse. In Kapitel 11 und Kapitel 12 beschreiben wir erste Resultate, die wir mit Hilfe unserer Vorgehensweise zur Komponentenidentifikation erzielen konnten und geben Hinweise darauf, wie diese Ergebnisse verbessert werden können. In Kapitel 13 stellen wir andere, größtenteils ähnliche Ansätze zur Komponentenidentifikation zusammen, bevor wir in Kapitel 14 die wesentlichen Erkenntnisse dieses Dokuments nocheinmal zusammenfassen. Ein umfangreiches Literaturverzeichnis bietet zahlreiche Möglichkeiten zur weiterführenden Vertiefung des Themas.



7 Grundlagen

In diesem Kapitel geben wir einen Überblick über die grundlegende Vorgehensweise zur Identifikation von Komponentenkandidaten. Die einzelnen Bestandteile und Schritte dieser Vorgehensweise werden wir dann in den folgenden Kapiteln weiter vertiefen.

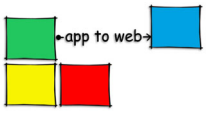
7.1 Grundlegende Vorgehensweise

Unsere Grundidee zur Identifikation von Komponentenkandidaten aus dem Quellcode existierender Softwaresysteme besteht darin, daß wir versuchen, die im Quellcode definierten strukturgebenden Elemente (beispielsweise Klassen und Methoden in objektorientierten Systemen, Funktionen bzw. Unterprogramme oder Übersetzungseinheiten bzw. Dateien in prozeduralen Systemen) des Systems so zu Gruppen zusammenzufassen, daß semantisch zusammengehörende Elemente möglichst in ein- und dieselbe Gruppe plaziert werden. Dadurch entsteht eine Zerlegung (*Dekomposition*) des Systems in Einheiten grober Granularität. Diese Einheiten stellen bei geeigneter Wahl der Zerlegung des Systems geeignete Komponentenkandidaten dar, die in weiteren Schritten evaluiert und überarbeitet werden können und in vollwertige Komponenten überführt werden können.

Da sich aus dem Quellcode eines Systems semantisch zusammengehörende Elemente nicht direkt ermitteln lassen, versuchen wir, mit Hilfe von Quelltextanalysen Abhängigkeiten zwischen den strukturgebenden Elementen zu ermitteln und ziehen diese dann heran, um mit Hilfe von bestimmter Techniken – den sogenannten *Clusterverfahren* – die Elemente des Systems zu gruppieren, um geeignete Zerlegungen des Systems in Komponentenkandidaten zu erhalten.

Die dabei verwendeten Clusterverfahren haben bereits eine lange Tradition und finden heute vielfache Verwendung bei der Erfassung und Auswertung von Meßwerten und bei Data-Mining-Problemstellungen [KR90]. Ursprünglich geht es dabei darum, Daten mit ähnlichen Eigenschaften zu Gruppen zusammenzufassen um so auf Basis einer reduzierten Datenmenge leichter Schlüsse ziehen zu können. Hierzu gibt es eine Reihe gut untersuchter Standardverfahren (siehe beispielsweise [Wig97]). All diesen Verfahren ist gemeinsam, daß die zu untersuchenden Daten anhand eines sogenannten *Ähnlichkeitsmaßes* gruppiert werden.

Wir wollen daher im folgenden Ähnlichkeitsmaße definieren, die die statischen Abhängigkeiten zwischen den strukturgebenden Elementen eines Softwaresystems



ausdrücken. Mit Hilfe bewährter Clustertechniken können wir dann die strukturgebenden Elemente eines Softwaresystems sinnvoll zu Komponentenkandidaten gruppieren.

Um Clusterverfahren zur Identifikation von Komponentenkandidaten einsetzen zu können, können wir gemäß der vorangehenden Überlegungen und mit Hilfe der Vorarbeiten aus dem ersten Teil dieses Textes folgende Vorgehensweise ableiten:

6. *Gewinnen eines Strukturmodells* des Systems durch Techniken zur Faktenextraktion. Ein solches Strukturmodell läßt sich als Graph darstellen, dabei bilden die strukturgebenden Elemente wie Klassen, Module oder Dateien des Systems die Knoten, und die Abhängigkeiten zwischen diesen Elementen die Kanten des Graphen.

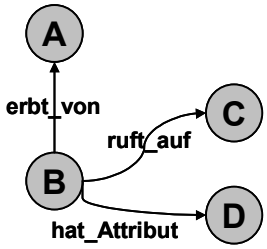


Abbildung 28: Beispiel für einen stark vereinfachten Strukturgraphen im Falle eines OO-Systems

7. *Definition eines geeigneten Ähnlichkeitsmaßes* (= Kriterium, nachdem geclustert wird) zwischen den Knoten des Strukturmodells. Das Ähnlichkeitsmaß beschreibt, auf welche Weise die Abhängigkeiten zwischen Systemteilen in das Clusterverfahren eingehen.

8. *Anwenden eines Clusteralgorithmus* zur Gruppierung der Knoten des Strukturbaums gemäß des Ähnlichkeitsmaßes. Ziel ist es, bzgl. des Ähnlichkeitsmaßes ähnliche Knoten in eine Gruppierung zu plazieren.

9. *Interpretieren der Ergebnisse*: Aus den Clustern des vorangehenden Schrittes ergeben sich letztlich Komponentenkandidaten. Oftmals müssen dazu jedoch weitere Analyseaktivitäten durchgeführt werden, damit aus diesen Gruppierungen geeignete Komponentenkandidaten abgeleitet werden können.

7.2 Clustern – Basistechnik zur Komponentenidentifikation

Wir skizzieren im folgenden kurz, wie Clusterverfahren funktionieren. Betrachten wir dazu zunächst einmal eine einfache Variante eines Cluster-Algorithmus, wie sie in Abbildung 29 angegeben ist. Diese Variante des Algorithmus faßt Entitäten auf-

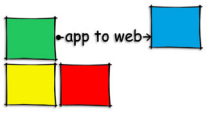
grund eines *Ähnlichkeitskriteriums* schrittweise zu Clustern zusammen, bis diese dem *Abbruchkriterium*, in dem wünschenswerte Eigenschaften der zu bestimmenden Cluster angegeben werden können, genügen. Diese Form des Algorithmus wird als *zusammenfassendes Clustering (Hierarchical Agglomerative Clustering, HAC)* bezeichnet.

Abbildung 29: Grund form eines Clusteralgorithmus *(HAC)*

1. Jede Entität bildet ein Cluster
2. **Wiederhole**
3. Identifiziere die Cluster C_i ; C_j, die am *ähnlichsten* zueinander sind
4. Kombiniere C_i ; C_j zu einem neuen Cluster und halte ihr Ähnlichkeitsniveau fest
5. **bis gilt:** alle Cluster C_i verfügen über zufriedenstellende Eigenschaften **oder** es ist nur noch ein Cluster übrig.

Wenn wir nun diesen Grundalgorithmus auf das Problem, Subsysteme in Softwaresystemen zu identifizieren, übertragen wollen, so können wir folgendermaßen vorgehen:

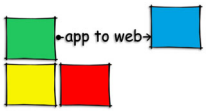
- Für objektorientierte Systeme verwenden wir als Entitäten die Klassen des Systems, die entstehenden Cluster sind die gesuchten Subsysteme. Ebenso läßt sich das Verfahren auch auf nicht-objektorientiert implementierte Systeme anwenden. In diesem Fall verwendet man als zu clusternde Entitäten Funktionen, Unterprogramme oder Übersetzungseinheiten (Dateien, Module) und globale Variablen.
- Das Ähnlichkeitskriterium können wir über *statische Abhängigkeiten* zwischen den Entitäten definieren: Klassen, die stark gekoppelt sind, gelten als ähnlich; im prozeduralen Fall können beispielsweise Funktionen als ähnlich gelten, die sich besonders häufig aufrufen oder Funktionen und Unterprogramme, die eine gewisse Anzahl von globalen Variablen gemeinsam nutzen. Alternativ lassen sich die jeweils zu verschmelzenden Cluster in Schritt 3 des Algorithmus auch so auswählen, daß das durch die Verschmelzung resultierende Cluster bezüglich folgender Kriterien zur Komponentenbildung das beste (unter den jeweils möglichen) ist:
  - Geringe externe Kopplung: Eine Komponente sollte von ihrer Umgebung weitgehend entkoppelt sein und nur durch wenige, wohldefinierte Aufrufbeziehungen mit dem Rest des Systems verwoben sein.
  - Hohe interne Kohäsion: Im Vergleich zur externen Kopplung dürfen die in der Komponente enthaltenen Elemente stark miteinander interagieren.
  - Vernünftige interne Komplexität: Da eine Komponente häufig als eigenständige Einheit entwickelt wird, sollte sie von überschaubarer interner Komplexi-



tät sein. Erst zusammen mit anderen Komponenten ähnlicher Komplexität entsteht die durchaus komplexe Gesamtfunktionalität des Systems.

  – Schnittstelleneigenschaften: Eine Komponente sollte über übersichtlich gestaltete Schnittstellen verfügen. Üblicherweise bieten Komponenten eine genau definierte Menge an Schnittstellenfunktionen, die einen Großteil der internen Komplexität der Komponente verbergen.

- Das Abbruchkriterium definieren wir über wünschenswerte Eigenschaften für die entstehenden Subsystemkandidaten, die wir über Software-Metriken bzw. Heuristiken [BBC+99] messen können.



8 Strukturmodelle

In Kapitel 7 haben wir bereits kurz beschrieben, welche Arten von strukturgebenden Elementen wir mit Hilfe der Clusterverfahren zu Komponentenkandidaten gruppieren wollen. Solche strukturgebenden Elemente sind die Basisbauteile eines Softwaresystems und können mittels Techniken zur Faktenextraktion, wie wir sie im ersten Teil dieses Textes bereits beschrieben haben, aus dem Quellcode zusammengestellt und mittels eines Strukturmodells dargestellt werden.

Wir stellen im folgenden nochmals die wesentlichen Elemente der Strukturmodelle für objektorientierte und prozedural programmierte Systeme zusammen, soweit sie für die Identifikation von Komponentenkandidaten durch Clustertechniken relevant sind.

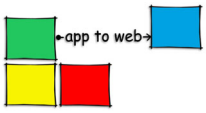
8.1 Strukturmodell für objektorientierte Systeme

Als wesentliche Bestandteile objektorientierter Systeme können wir folgende Elemente identifizieren:

- *Klassen:* Das zentrale Konzept in der statischen Struktur eines objektorientierten Systems ist sicherlich die Klasse.
- *Methoden:* Eine Klasse enthält normalerweise eine Menge von Methoden, in denen der eigentliche Programmablauf beschrieben ist.

Für unsere Zwecke abstrahieren wir hier von der Methodensicht und ordnen die Methoden den jeweiligen Klassen, in denen sie definiert, sind zu. Wir beschränken uns somit darauf, die Klassen eines objektorientierten Systems mit Hilfe von Clusterverfahren zu Komponentenkandidaten anzuordnen. Unser Strukturmodell kennt als Entitäten folglich ausschließlich Klassen, die durch folgende Beziehungen miteinander verknüpft sein können:

- *Vererbungsbeziehungen.*
- *Aufrufe:* Zwischen zwei Klassen *A* und *B* besteht eine Aufrufbeziehung, wenn eine Methode aus der Klasse *A* eine Methode der Klasse *B* aufruft.
- *Typdeklarationen:* Zwei Klassen *A* und *B* stehen über eine Typdeklaration miteinander in Beziehung, wenn innerhalb der Klasse *A* Deklarationen unter Verwendung des Typs *B* erfolgen. Dies umfaßt die Deklaration von Attributen, Parameter- und Rückgabetypen.



- *Variablenzugriffe, Objekterzeugung:* Solche Beziehungen bestehen zwischen zwei Klassen *A* und *B* dann, wenn Methodenimplementierungen aus *A* auf Variablen (Attribute) einer Klasse *B* zugreifen oder Objekte vom Typ *B* erzeugen.

8.2 Strukturmodell für prozedurale Systeme

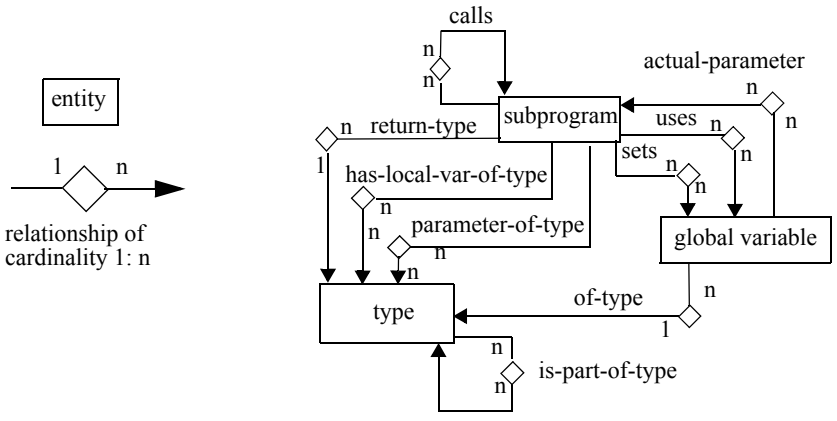
Zur Analyse prozeduraler Systeme mittels Clusterverfahren können wir den in im ersten Teil dieses Dokumentes eingeführten *Resource Flow Graph (RFG)* verwenden. Der RFG modelliert wesentliche Elemente prozeduraler Systeme, wie *Unterprogramme* bzw. *Funktionen*, *globale Variablen* und benutzerdefinierte *Datentypen*, wie sie beispielsweise in C mittels der Sprachkonstrukte *typedef, struct* und *union* definiert werden können.

Zwischen diesen Elementen, die im RFG als Knoten repräsentiert werden, können verschiedene Beziehungen bestehen. Die wesentlichen hierbei sind:

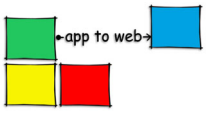
- *Aufrufbeziehungen* zwischen Unterprogrammen bzw. Funktionen
- *Variablennutzungen* zwischen Unterprogrammen und globalen Variablen
- *Typdeklarationen* zwischen Unterprogrammen und Typen oder globalen Variablen und Typen.

In Abbildung 30 sind die wesentlichen Elemente des RFG und deren Beziehungen untereinander in graphischer Form dargestellt.

Abbildung 30: Schema des RFG



Eine weitere Möglichkeit besteht zusätzlich darin, *Übersetzungseinheiten* bzw. Module gemäß ihrer *Includeabhängigkeiten* zu logischen Komponenten zu clustern.



9 Kriterien zur Komponentenidentifikation und Ähnlichkeitsmaße

Komponentenstrukturen werden aus unterschiedlichen Motiven heraus gebildet. Analysiert man diese Motive genauer, so findet man heraus, daß sich Komponentenstrukturen durch die folgenden Eigenschaften auszeichnen:

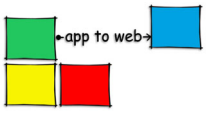
- geringe externe Kopplung
- hohe interne Kohäsion
- handhabbare interne Komplexität
- Gruppierung von gemeinsam verwendeten Elementen
- Gruppierung nach gemeinsamen Merkmalen, beispielsweise nach gemeinsam verwendeten Elementen

Diese Eigenschaften können wir nutzen, um nach Komponentenkandidaten zu suchen. Voraussetzung dafür ist allerdings, daß beim Entwurf der Altanwendung diese Prinzipien nicht völlig vernachlässigt wurden.

9.1 Kriterien und Ähnlichkeitsmaße für objektorientierte Systeme

Wir leiten nun aus den beiden Kriterien geringe Kopplung, hohe interne Kohäsion ab, wie man geeignete Ähnlichkeitsmaße definieren kann, wenn es sich um objekt-orientierte Systeme handelt.

Wir definieren dazu zunächst ein Ähnlichkeitsmaß zwischen jeweils zwei Klassen des Systems. Dieses Ähnlichkeitsmaß definieren wir als gewichtete Summe aus mehreren Einzelmaßen, welche die unterschiedlichen Typen von Abhängigkeiten, die zwischen den Klassen eines objektorientierten Systems bestehen können, berücksichtigen. Unser Ähnlichkeitsmaß zwischen Klassen müssen wir dann noch entsprechend auf Cluster übertragen um auch die Ähnlichkeit zwischen Clustern, also Gruppen von Klassen, berechnen zu können, so wie dies beispielsweise in Zeile 3 des in Abbildung 29 skizzierten HAC-Algorithmus erforderlich ist.



9.1.1 Ähnlichkeitsmaße zwischen Klassen

Durch die hier dargestellten Berechnungen werden die unterschiedlichen Abhängigkeiten zwischen den Klassen des Systems, wie sie im Strukturmodell existieren, in ein Ähnlichkeitsmaß abgebildet, auf Basis dessen das eigentliche Clustern durchgeführt werden kann. Welche Arten von Abhängigkeiten in das Ähnlichkeitsmaß einfließen, und wie stark diese gewichtet werden sollen, beschreiben wir im folgenden näher.

Abbildung 31: Abhängigkeiten zwischen Klassen in OO-Systemen und deren Gewichtung in Clusteralgorithmen

Abhängigkeit *i*	**Erläuterung**	**Gewicht**
Vererbung		niedrig
Typdeklarationen	Attribute, Variablen, Parameter, Rückgabetypen	niedrig
Methodenaufrufe		mittel
Attributzugriffe		hoch
Objekterzeugung		hoch

Die Tabelle in Abbildung 31 gibt an, welche Abhängigkeiten in für objekt-orientierte Systeme prinzipiell berücksichtigt werden können. Für jede Art von Abhängigkeit *i* aus dieser Tabelle berechnen wir dann mit Hilfe einer Metrik einen Meßwert $\omega_i(A, B)$, der die Stärke der Abhängigkeit *i* zwischen zwei Klassen *A, B* angibt. Diese Meßwerte werden mit Hilfe einer gewichteten Summe zum Ähnlichkeitsmaß *sim(A,B)* zwischen zwei Klassen *A , B* zusammengefaßt:

$$sim(A, B) = \sum_i \omega_i(A, B) \times p_i$$

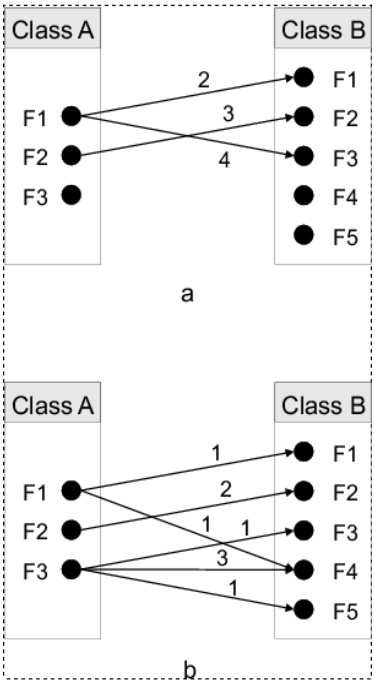
Mit Hilfe des Wertes p_i kann dabei ein Gewicht angegeben werden, welches einer Abhängigkeit vom Typ *i* zugeordnet werden soll. Die Tabelle in Abbildung 31 gibt qualitative Hinweise auf eine sinnvolle Gewichtung der Abhängigkeiten.

9.1.2 Ähnlichkeitsmaße für einzelne Abhängigkeitsbeziehungen

Wir wollen noch kurz am Beispiel von Methodenaufrufen[1] darauf eingehen, wie die Berechnung der $\omega_i(A, B)$ erfolgen kann. Wir beobachten zunächst, daß der Grad der Kopplung zwischen zwei Klassen durch Methodenaufrufe stark von der Lokalität der Aufrufe abhängt. In Abbildung 32 sind zwei unterschiedliche Fälle dargestellt, in denen jeweils zwei Klassen durch eine identische Anzahl von Methodenaufrufen miteinander verknüpft sind. Im Fall b) erscheint die Kopplung zwischen den beiden

Klassen jedoch stärker, weil die Aufrufabhängigkeiten breiter unter den Methoden der Klassen *A* und *B* gestreut sind.

Abbildung 32: Lokalität von Aufruf Abhängigkeiten zwischen zwei Klassen



Um dieser Beobachtung Rechnung zu tragen, definieren wir die durch Methodenaufrufe entstehende Abhängigkeit wie folgt:

$$\omega_{Calls}(A, B) = \frac{Calls_A(B)}{|A|} \times \frac{Callers_A(B) + Called_B(A)}{|A| + |B|}$$

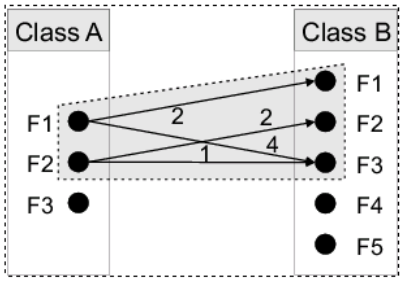
Dabei bezeichnet $Calls_A(B)$ die Anzahl von Aufrufen zu Methoden von B aus Methoden von A, $Callers_A(B)$ die Anzahl von Methoden in A, welche Aufrufe zu Methoden aus B enthalten, $Called_B(A)$ die Anzahl von Methoden aus B welche von A

1 Wir betrachten hier *statische* Aufrufbeziehungen, wie wir sie anhand der deklarierten Typen der an einem Aufrufkonstrukt beteiligten Variablen direkt aus dem Quelltext des Systems ableiten können: zwischen zwei Methoden *A.m()* und *B.n()* besteht eine Aufrufbeziehung, wenn der Methodenrumpf von *A.m()* einen Ausdruck der Form *b.n()* enthält, und *B* der statische (deklarierte) Typ von *b* ist

aus gerufen werden, und $|A|$ bzw. $|B|$ bezeichnen die Anzahl der Methoden in A bzw. B.

Abbildung 33 gibt eine intuitive Rechtfertigung für diese Formel: Die Kopplung zwischen A und B durch Methodenaufrufe ist proportional zum Flächeninhalt des hervorgehobenen Trapezes, welches bei entsprechender Anordnung der beteiligten Methoden ein gutes Maß für die Lokalität der Aufrufe zwischen A und B darstellt.

Abbildung 33: Intuitive Rechtfertigung der Metrik für Aufrufabhängigkeiten



Ähnliche Überlegungen lassen sich auch für andere Abhängigkeiten anführen, beispielsweise zur Berechnung der Kopplung durch Attributzugriffe.

9.1.3 Ähnlichkeitsmaß zwischen Clustern

Wir müssen unser Konzept des Ähnlichkeitsmaßes zwischen Klassen nun noch auf die Ähnlichkeit zwischen Clustern, also Mengen von Klassen übertragen, damit wir dieses Maß im in Abbildung 29 dargestellten HAC-Algorithmus verwenden können.

Die einfachste Möglichkeit besteht darin, die Ähnlichkeit zwischen zwei Clustern X und Y über die durchschnittliche Ähnlichkeit innerhalb aller Paare, die sich aus den Elementen der Cluster bilden lassen, zu definieren. Dies ergibt folgende Formel

$$Sim(X, Y) = \frac{\sum_{(A \in X, B \in Y)} Sim(A, B)}{|X| \times |Y|}$$

Alternativ können wir bei der Verschmelzung der zwei ähnlichsten Cluster C_i, C_j zu einem neuen Cluster C in Schritt 4 des Algorithmus aus Abbildung 29 die Ähnlichkeit des neu entstehenden Clusters C zu jedem anderen Cluster X durch eine sogenannte *Update-Regel* berechnen, die dazu die Ähnlichkeiten zwischen C_i und X

sowie C_j und X verwendet. Folgende beiden Update-Regeln haben sich dabei bewährt:

$$sim(C, X) = \frac{sim(C_i, X) + sim(C_j, X)}{2}$$

$$sim(C, X) = max(sim(C_i, X), sim(C_j, X))$$

Abschließend bemerken wir noch, daß unsere Definition des Ähnlichkeitsmaßes *sim*, sobald sie in einem Cluster-Algorithmus (siehe Abbildung 29) verwendet wird, Klassen zu Subsystemkandidaten gruppiert, die sich durch eine starke interne Kopplung *(Kohäsion)* auszeichnen.

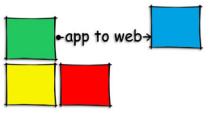
9.2 Kriterien und Ähnlichkeitsmaße für prozedurale Systeme

Ähnlichkeitsmaße für prozedurale Systeme lassen sich in vergleichbarer Weise definieren. Grundlage ist hierfür der *Resource Flow Graph (RFG)* zur Repräsentation prozeduraler Systeme. Die Definition des Ähnlichkeitsmaßes ergibt sich aus drei Bestandteilen:

- dem Ähnlichkeitsmaß zwischen *zwei Entitäten* des Strukturmodells (für den RFG sind solche Entitäten beispielsweise Unterprogramme bzw. Funktionen und benutzerdefinierte Typen)
- jeweils ein Maß für verschiedene Teilaspekte, die ins Ähnlichkeitsmaß zwischen zwei Entitäten einfließen
- das Ähnlichkeitsmaß zwischen zwei Gruppen bzw. Clustern von Entitäten des Strukturmodells, welches jeweils aus den Ähnlichkeiten der in den Clustern enthaltenen Elemente berechnet werden kann.

9.2.1 Ähnlichkeitsmaß zwischen zwei Entitäten

Das Ähnlichkeitsmaß zwischen zwei Entitäten des RFG ergibt sich, ähnlich wie im objektorientierten Fall, aus unterschiedlichen Aspekten, die unterschiedliche Arten von Abhängigkeiten ins Ähnlichkeitsmaß einbringen. Wir unterscheiden folgende Aspekte:



- direkte Abhängigkeiten zwischen zwei Entitäten, so wie sie sich aus dem RFG direkt ergeben,
- indirekte Abhängigkeiten, die sich über gemeinsame Abhängigkeiten zu dritten Entitäten ergeben,
- informelle Abhängigkeiten, die sich nicht direkt aus dem Quelltext bzw. RFG ergeben, sondern die aus externen Informationsquellen abgeleitet werden können, beispielsweise aus Namenskonventionen für Bezeichner und dergleichen.

Diese Aspekte fließen über eine gewichtete Summe ins Ähnlichkeitsmaß zwischen zwei Entitäten A, B ein:

$$Sim(A,B) = \frac{x_1 \cdot Sim_{indirect}(A,B) + x_2 \cdot Sim_{direct}(A,B) + x_3 \cdot Sim_{informal}(A,B)}{x_1 + x_2 + x_3} \quad (\boldsymbol{F2})$$

Mittels der Gewichte x_i kann der Einfluß der einzelnen Abhängigkeitsaspekte reguliert werden. Da das Maß für jeden Abhängigkeitsaspekt zwischen 0 und 1 liegt, also normalisiert ist, ist auch das Abhängigkeitsmaß zwischen zwei Entitäten normalisiert.

9.2.2 Maße für die unterschiedlichen Abhängigkeitsaspekte

Direkte Abhängigkeiten

Direkte Abhängigkeiten zwischen zwei Entitäten *A, B* entstehen durch direkte Verbindungen im RFG. Ihren Beitrag zum Ähnlichkeitsmaß bezeichnen wir mit $Sim_{direct}(A,B)$. Er ergibt sich aus der gewichteten Summe aller tatsächlich vorliegenden Kantentypen zwischen den Knoten der Entitäten *A* und *B* im RFG, dividiert durch die gewichtete Summe aller möglichen Kantentypen zwischen den Knotentypen von *A* und *B*.

$$Sim_{direct}(A,B) = \frac{W(link(A,B))}{W(\text{all-links}(nodetype(A), nodetype(B)))} \quad (\boldsymbol{F3})$$

wobei *link (A, B)* die Menge aller tatsächlich vorliegenden Kantentypen zwischen *A* und *B* bezeichnet, und die Menge *all-links()* alle Kantentypen enthält, die zwischen den Knotentypen von *A* und *B* theoretisch aufgrund der Definition des RFG bestehen können. Auf Basis dieser Menge werden gewichtete Summen

$$W(X) = \sum_{x \in X} weight(x) \quad (\boldsymbol{F4})$$

berechnet, wobei mittels $weight(x) \geq 0$ eine Gewichtung vorgenommen werden kann, die bestimmten Abhängigkeiten mehr Einfluß zuschreibt als anderen.

Für die Definition dieser Gewichtung können unterschiedliche Vorgehensweisen gewählt werden: Eine Möglichkeit besteht darin, eine Formel aus Shannons Informationstheorie [Sha72] für *weight(x)* zu verwenden:

$$weight(x) = -\log(probability(x))$$

wobei *probability(x)* die Häufigkeit des Vorkommens des Kantentyps *x* im RFG angibt. Dahinter steckt die Überlegung, daß selten verwendete Entitäten bzw. Kantentypen wichtiger sind, als solche die häufig verwendet werden. Ein Beispiel mag dies verdeutlichen: Eine Fehlervariable, die fast überall im System verwendet wird, spielt sicher eine geringere Rolle für Abhängigkeitsbetrachtungen, als eine globale Variable, die von einer kleinen Menge von Unterprogrammen des Systems gemeinsam genutzt wird.

Alternativ dazu können wir an bestimmte Kantentypen feste Gewichte binden. Beispielsweise werden wir, wenn wir ADTs und deren zugehörige Operationen in prozeduralen Systemen identifizieren wollen, Kanten, die zu benutzerdefinierten Typen führen, höher bewerten als gewöhnliche Aufrufkanten.

Wir können beide Vorgehensweisen auch kombinieren, indem wir die Werte für den Shannon-Ansatz mit denen für feste Gewichte nach Kantentypen multiplizieren. Diese Vorgehensweise vereint die Vorteile der beiden Verfahren: wir können beim später folgenden Clusterprozeß bestimmten Mustern Rechnung tragen und trotzdem selten vorkommenden Kantentypen eine höhere Bedeutung beimessen.

Indirekte Abhängigkeiten

Indirekte Abhängigkeiten berücksichtigen den Anteil an gemeinsamen Merkmalen, die zwei Entitäten *A, B* gemeinsam haben. Im RFG bedeutet dies, daß solche gemeinsamen Merkmale jeweils Nachbarn von *A* und *B* sind.

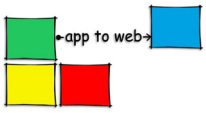
Ausgedrückt wird dieser Abhängigkeitsaspekt durch folgende Formel:

$$Sim_{indirect}(A, B) = \frac{Common(A, B)}{Common(A, B) + d \cdot Distinct(A, B)} \qquad (\boldsymbol{F5})$$

wobei *Common (A,B)* die gewichtete Summe gemeinsamer Merkmale von A und B bezeichnet und *Distinct (A,B)* die gewichtete Summe abweichender Merkmale. $d \geq 0$ ist ein Parameter, mit dem der Einfluß abweichender Merkmale reguliert werden kann. Im RFG sind *Common* und *Distinct* wie folgt definiert:

$$Common(A, B) = W(edges\text{-}with(A) \cap edges\text{-}with(B))$$
$$Distinct(A, B) = W(edges\text{-}with(A) \oplus edges\text{-}with(B))$$

Der Operator $\oplus$ bezeichnet die symmetrische Differenz für Mengen. Der Ausdruck *edges-with(B)* steht für die Menge an Paaren *(E, N)*, in denen *N* ein Knoten ist der



zum Knoten *B* durch eine Kante vom Typ *E* verbunden ist. Diese Paare werden durch eine gewichtete Summe aufsummiert, deren Gewichte gemäß der oben dargestellten kombinierten Gewichtung gemäß Shannons Formel und festen Gewichten für jeden Kantentyp festgelegt werden.

Informelle Abhängigkeiten über Namenskonventionen

Programmierer lassen oft semantische Informationen über die Bedeutung von Programmkonstrukten in die Namensgebung von Verzeichnissen und Dateien der Quelltexte und in die Bezeichnung von Funktionen, Variablen und Typen einfließen. Dies ermöglicht es ihnen und anderen Programmierern, sich besser in den Quellen zurecht zu finden. Üblicherweise werden diese Informationen in Reengineeringtechniken oft vernachlässigt. Wir beschreiben im folgenden exemplarisch, wie man aus Bezeichnern zusätzliche Abhängigkeitsinformationen gewinnen kann.

Wir verwenden folgende Formel, um Abhängigkeiten, die sich durch Ähnlichkeiten in Bezeichnern ergeben, zu quantifizieren und ins Ähnlichkeitsmaß einfließen zu lassen:

$$Sim_{informal}(A,B) = \frac{\sum_{t \in T} x_t \cdot Sim_t(A,B)}{\sum_{t \in T} x_t} \qquad T = \left\{ \begin{array}{l} words \\ suffix \end{array} \right\} \qquad (\boldsymbol{F6})$$

Oft sind Bezeichner zusammengesetzte Begriffe, deren einzelne Worte durch Unterstriche (_) oder passende Groß-/Kleinschreibung abgetrennt werden. Beispiele für solche Bezeichner sind `insert_word()` bzw. `InsertWord()`. Das folgende Maß profitiert von solchen Konventionen und berechnet die Ähnlichkeit zwischen zwei Entitäten anhand gemeinsamer Teilworte:

$$Sim_{words}(X,Y) = \frac{|\text{words}(X) \cap \text{words}(Y)|}{|\text{words}(X) \cup \text{words}(Y)|} \qquad (\boldsymbol{F7})$$

Hintergrund hierfür ist, daß solche Bezeichner oft Hinweise auf zusammengehörende Codeteile liefern, beispielsweise werden `stack_push()` und `stack_pop()` sicherlich eng verwandte Operationen bezeichnen.

In anderen Fällen drückt sich die Gemeinsamkeit zwischen Bezeichnern oft durch gemeinsam verwendete Präfixe oder Postfixe aus. Dieser Beobachtung trägt das folgende Maß Rechnung:

$$Sim_{suffix}(X, Y) = \begin{cases} 1 & X = Y \\ \dfrac{\text{prefix}(X, Y) + \text{postfix}(X, Y)}{1 + \text{prefix}(X, Y) + \text{postfix}(X, Y)} & X \neq Y \end{cases} \qquad (\boldsymbol{F8})$$

wobei *prefix()* und *postfix()* die Länge der gemeinsamen Präfixe und Postfixe der Bezeichner der Entitäten *X* und *Y* bezeichnet.

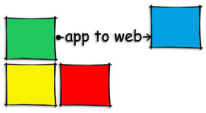
9.2.3 Ähnlichkeitsmaß zwischen Clustern

Wir müssen das soeben definierte Ähnlichkeitsmaß zwischen einzelnen Entitäten auf ein Ähnlichkeitsmaß zwischen Clustern (also Gruppen bzw. Mengen von Elementen) zu übertragen. Die Ähnlichkeit zwischen Clustern wir im HAC-Algorithmus für alle späteren Iterationen benötigt, um die zwei ähnlichsten und damit zu verschmelzenden Cluster zu identifizieren. Dazu verwenden wir folgende Formel:

$$GSim(S_1, S_2) = \frac{\sum_{(s_i \in S_1, s_j \in S_2)} Sim(s_i, s_j)}{size(S_1) \times size(S_2)} \qquad (\boldsymbol{F1})$$

Diese Formel berechnet die Ähnlichkeit zwischen zwei Clustern S_1, S_2 über die durchschnittliche Ähnlichkeit innerhalb aller Paare, die sich aus den Elementen der Cluster bilden lassen.

Andere Ähnlichkeitsmaße sind denkbar, in der Praxis haben wir mit dieser Definition jedoch recht gute Erfahrungen gemacht.



10 Clustering

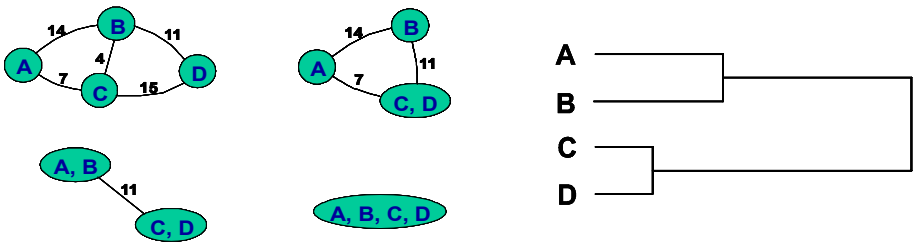
Unter dem Begriff *Clusterverfahren* faßt man eine Reihe von Ansätzen, die dazu dienen, Elemente einer Menge anhand von Ähnlichkeiten hinsichtlich bestimmter Eigenschaften zu klassifizieren, zusammen. Clusterverfahren spielen bei der *(Re-) Modularisierung* von Softwaresystemen eine große Rolle ([Wig97], [Eve74]) und können wesentlich zum *strukturellen Verständnis* eines Softwaresystems beitragen.

Solche hierarchischen Clusteralgorithmen berechnen prinzipiell eine Hierarchie von Clusterings, in der die jeweils nächsthöhere Hierarchiestufe aus der vorausgehenden hervorgeht, indem die zwei ähnlichsten Cluster zu einem größeren Cluster verschmolzen werden. So entsteht ein binärer Baum aus Clustern, dessen Ebenen jeweils unterschiedlich feine Clusterings darstellen.

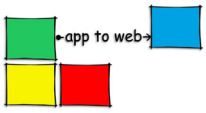
Abbildung 34 zeigt auf der linken Seite den Ablauf eines hierarchischen Clusteralgorithmus anhand eines kleinen Beispiels. Der Algorithmus beginnt mit einer initialen Zerlegung aus 4 Clustern: *{{A}, {B}, {C}, {D}}*. Im nächsten Schritt werden die Cluster *{C}* und *{D}* zusammengefaßt, so daß folgende Zerlegung entsteht: *{{A}, {B}, {C, D}}*. Anschließend entsteht durch die Verschmelzung von *{A}* und *{B}* folgende Zerlegung aus 2 Clustern: *{{A, B}, {C, D}}*. Im letzten Schritt entsteht durch die Verschmelzung dieser beiden verbleibenden Cluster folgende Zerlegung *{{A, B, C, D}}*.

Auf der rechten Seite der Abbildung sehen wir einen Binärbaum, mit Hilfe dessen sich das Ergebnis des Algorithmus veranschaulichen läßt.

Abbildung 34: Beispielablauf eines hierarchischen Cluster-Algorithmus

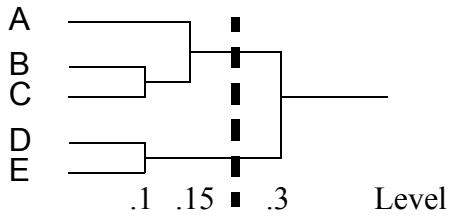


Dieser Binärbaum, der als Ergebnis des Algorithmus entsteht, wird üblicherweise in Form eines *Dendrogrammes* angegeben. Ein Dendrogramm ist ein Baum, dessen Blätter die zu clusternden Ausgangselemente sind und dessen innere Knoten den Clu-



stern entsprechen, die während der Abarbeitung des Verfahrens entstehen. Das Ähnlichkeitsniveau, bei dem ein bestimmtes Cluster gebildet wurde, kann dem Dendrogramm entnommen werden. Es gibt Auskunft darüber, wie gut die Elemente eines Clusters zusammen passen. In Abbildung 35 beispielsweise wurden die Cluster A und (B,C) auf dem Ähnlichkeitsniveau von 0.15 zu einem neuen Cluster zusammengefaßt.

Abbildung 35: Dendrogramm



In Verbindung mit den Ähnlichkeitsmaßen aus Kapitel 9 können Clusteringverfahren wie HAC auf strukturgebende Elemente wie Klassen, Module und Funktionen angewandt werden, um diese anhand ihrer Beziehungen und Abhängigkeiten zu Subsystemen zu gruppieren. Allerdings liefert HAC nicht direkt Subsysteme oder Komponenten des Systems, sondern vielmehr eine Familie von Zerlegungen des Systems in Form des Dendrogrammes. Dieses Dendrogramm muß dann erst noch passend geschnitten werden, um passende Subsystem- oder Komponentenkandidaten zu erhalten. In Abbildung 35 haben wir einen solchen Schnitt vorgenommen – demnach erhalten wir zwei Komponentenkandidaten, eine Komponente enthält die Entitäten A,B,C; die andere die Entitäten D,E.

In der Praxis ist es sehr entscheidend, an welcher Stelle man die Schnitte durchführt, sie bestimmen wesentlich das Ergebnis der Modularisierung. Oft zieht man Experten zu Rate, die dann beispielsweise *bottom-up* durch das Dendrogramm gehen und überlegen, ab welchem Niveau Cluster zusammengefügt werden, zwischen denen kein logischer Zusammenhang mehr besteht. In großen Systemen ist dieser Prozeß allerdings aufwendig und schwierig, so daß die Tauglichkeit solcher Verfahren in der Praxis oft angezweifelt wird. Dennoch können solche Clusteringverfahren wertvolle Hinweise auf Komponentenkandidaten geben.

11 Fallbespiele zur Evaluierung der Konzepte

Um die zuvor skizzierten Techniken zu erproben, haben wir mit Hilfe eines Tool-Prototypen *JAMES* ein Java-System, *Inject/J* [GK], analysiert. Ziel war es dabei, das System mittels Cluster-Verfahren in Subsystemkandidaten zu zerlegen. Durch einen Vergleich der so berechneten Zerlegung des Systems mit der originalen, vom Entwickler des Systems definierten Paketstruktur erhoffen wir uns erste Rückschlüsse über die Tauglichkeit unseres Verfahrens und der zugrunde liegenden Techniken.

Inject/J ist ein Java-System zur automatischen Adaption von Java-Programmen. Wir haben dieses System aus folgenden Gründen als Fallstudie ausgewählt:

- Inject/J ist ein unabhängig entwickeltes, praktisch einsetzbares Softwaresystem (ca. 50000 LOC, 170 Klassen). Know-How über seinen Aufbau steht uns in Form eines Entwicklers zur Verfügung.
- Das System ist seit einigen Jahren in Entwicklung. Wir gehen davon aus, daß das System weder besonders mangelhaft strukturiert ist, noch daß es völlig frei von zerwarteten Strukturen ist.

Inject/J wurde mit Hilfe unterschiedlicher Clusteralgorithmen (siehe auch Kapitel 13) gemäß der in Abschnitt Abschnitt 7.1 vorgestellten Vorgehensweise untersucht. Die vielversprechendsten Ergebnisse erhielten wir während der Anwendung von hierarchischen Cluster-Verfahren. Allerdings bereitete uns die Interpretation der Ausgaben der hierarchischen Algorithmen größere Probleme. Die während des Clustering-Vorgangs entstehenden Dendrogramme sind umfangreich. Da wir momentan keine hinreichend guten Heuristiken zur Auswahl der Schnitte durch die Dendrogramme in JAMES zur Verfügung haben, haben wir das Dendrogramm von Hand analysiert und entsprechende Schnitte gesetzt.

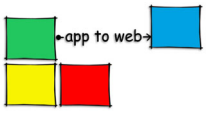
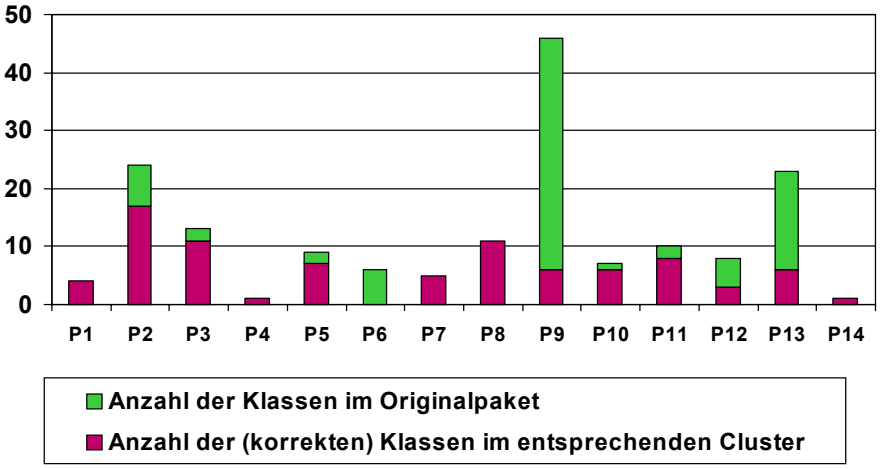


Abbildung 36: Hierarchisches Clustering – Ergebnisse für die Fallstudie Inject/J



In Abbildung 36 ist eine Auswertung der auf diese Weise entstandenen Subsystemzerlegung zu sehen. Das Diagramm zeigt auf der Rubrikenachse die durch den Entwickler definierten Pakete. Für jedes Paket zeigt das Diagramm zunächst die Anzahl der Klassen, die in dem jeweiligen Paket definiert sind. Dieser Wert wird kontrastiert mit der Anzahl derjenigen Klassen, die anhand des hierarchischen Cluster-Verfahrens korrekt in passende Pakete plaziert worden sind. Beispiel: Paket P2 enthält per Entwicklerdefinition 24 Klassen. Mit Hilfe von JAMES konnte ein Cluster identifiziert werden, welches 18 dieser Klassen enthält. Die fehlenden 6 Klassen wurden von JAMES anderen Clustern zugeordnet, desweiteren gab es einige Klassen, die irrtümlich P2 zugeordnet wurden.[1]

Betrachten wir das Diagramm in Abbildung 36, so stellen wir fest, daß 11 der 14 Pakete von JAMES recht gut identifiziert werden konnten. Bei den Paketen P6, P9, P13 lieferte JAMES dagegen kaum brauchbare Ergebnisse.

Zusammengefaßt: Automatisches Clustern liefert Partitionen des Systems, die nur teilweise mit der vom Entwickler vorgesehenen Strukturierung des Systems übereinstimmen.

1 Diese Information ist nicht im Diagramm dargestellt.

12 Ansätze zur Verbesserung des Clusterverfahren

Hier werden wir ein paar Überlegungen skizzieren, wie Clusterverfahren hinsichtlich folgender Punkte verbessert werden können. Teilweise entstanden dieser Überlegungen während der Bearbeitung von Fallstudien (siehe Kapitel 11) oder sie entstammen anderen Arbeiten, die sich mit der Identifikation von Subsystemen oder Komponenten auseinandersetzen (siehe beispielsweise [MMCG99] bzw. Kapitel 13).

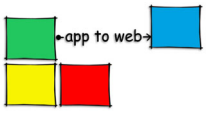
Zum einen konnten wir während der Durchführung unserer Fallstudien beobachten, daß die verwendeten Clusterverfahren mit bestimmten Teilen der Systeme nur schwer umgehen konnten. Einige Subsysteme bzw. Pakete, die von Entwicklern definiert wurden, konnten durch die Algorithmen überhaupt nicht identifiziert werden. Wir stellten fest, daß solche problematischen Subsysteme meistens bibliotheksartigen Charakter hatten.

Zum anderen haben wir bisher nur wenig Ansatzpunkte vorgesehen, wie wir die Ergebnisse unserer Verfahren verbessern können, wenn wir bereits vorhandenes Entwicklerwissen einbringen. Solches Wissen steht oft (zumindest lückenhaft) zur Verfügung und sollte daher nutzbringend in die Clusterverfahren integriert werden können.

Umgang mit Bibliotheken und Rahmenwerken

Während der Bearbeitung einiger Fallstudien konnten wir feststellen, daß die Gruppierung von Bibliotheken zu Clustern mittels Clusteralgorithmen in Verbindung mit den kopplungsbasierten Ähnlichkeitsmaßen aus Kapitel 9 versagt. Dies liegt daran, daß viele Elemente der Bibliothek aufgrund zahlreich vorhandener Beziehungen mit Teilen des Clientcodes der Bibliothek zusammen gruppiert werden.

Eine Möglichkeit, mit diesem Problem umzugehen, besteht darin, bibliotheksartige Teile des Systems entweder durch den Entwickler-Interaktion im voraus zu markieren oder mit Hilfe von einfachen Heuristiken zu identifizieren, und diese dann gesondert zu behandeln. Diese markierten Teile können dann vorab in ein ausgezeichnetes Cluster *Bibliothek* (oft auch *omnipresent module* genannt) plaziert werden und vom allgemeinen Clusterprozeß ausgeschlossen werden.



Alternativ können solche bibliotheksartigen Teile auch durch Clusterverfahren gruppiert werden , indem die Ähnlichkeitskriterien so umgestaltet werden, daß der Clusterprozeß die Elemente der betreffenden Systemteile nach dem Kriterium „alle Elemente, die häufig gemeinsam verwendet werden, kommen in dasselbe Cluster“ anordnet.

Um zu erkennen, welche Teile des Systems Bibliothekscharakter haben, können wir beispielsweise folgende Heuristik verwenden: Wir betrachten das Strukturmodell des Systems als Graph und zeichnen diejenigen Knoten darin, die einen Eingangsgrad über einem gewissen Schwellwert *O* haben, als zu Bibliotheken zugehörig aus. Dies sind dann genau die Systemteile, die besonders häufig verwendet werden, und somit mit großer Wahrscheinlichkeit Bibliothekscharakter haben.

Bei Rahmenwerken *(Frameworks)* in objektorientierten Systemen treten ähnliche Effekte auf. Zum einen sind hier starke Kopplungen durch Aufrufe zwischen dem Rahmenwerk und dem Code der eigentlichen Anwendung, welcher das Rahmenwerk instanziert, zu erwarten. Zum anderen dürfte die Kopplung durch zahlreiche Vererbungsbeziehungen zwischen Rahmenwerk und Anwendungscode verstärkt werden. Vermutlich lassen sich Rahmenwerke während des Clusterns ähnlich behandeln wie Bibliotheken; zusätzlich kann der starken Vererbungskopplung zwischen Rahmenwerk und Anwendungscode begegnet werden, indem die Vererbung innerhalb des Ähnlichkeitsmaßes weniger stark gewichtet wird.

Ähnliche Überlegungen sind auch für andere *„Musterstrukturen“* innerhalb der Systeme denkbar, beispielsweise für treiberartige Module, welche den Kontrollfluß durch das System durch eine zentrale Ereignisbearbeitung steuern und somit enge Beziehungen zu fast allen anderen Modulen aufweisen.

Integration von Entwickler Know-How

In vielen Fällen verfügen Entwickler (die oft auch Nutzer der Clusterverfahren sind) über allerhand nützliches Wissen über den Aufbau des Systems oder können sich solches Wissen aus externen Quellen (Dokumentation, Erfahrung, Domänen-Know-How) beschaffen. Es ist daher wünschenswert, dieses Wissen zur Verbesserung der automatischen Cluster-Verfahren heranzuziehen.

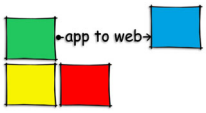
Dazu sehen wir folgende Möglichkeiten:

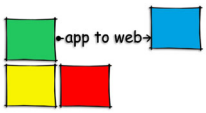
- Der Entwickler weiß in etwa, was für Subsysteme es gibt, aber nicht mehr, welche Elemente dazu gehören. Dieses Wissen kann der Nutzer in den Clusterprozeß einbringen, indem er sogenannte "*Kondensationskeime*" festlegt, um

die sich Cluster bilden sollen. In objektorientierten Systemen kann der Benutzer dazu beispielsweise einzelne Klassen auszeichnen, die er kennt und von denen er vermutet, daß sie zu unterschiedlichen logischen Teilen des Systems gehören. Er ordnet dazu beispielsweise einige GUI-Klassen, einige Klassen aus bestimmten Bereichen der eigentlichen Anwendungslogik und einige Klassen aus der Datenbankschicht einer Geschäftsanwendung jeweils unterschiedlichen Ausgangsclustern zu, die dann später durch ein sogenanntes *optimierendes Clusterverfahren* (siehe Kapitel 13) überarbeitet werden können.

- Der Entwickler weiß von manchen Systemfragmenten noch, daß Sie konzeptionell zusammengehören. Solches Wissen läßt sich oft anhand externer Informationsquellen, wie beispielsweise aus der Verzeichnisorganisation der Quelltexte, aus Namenskonventionen oder existierender Dokumentation herauslesen. Dieses Wissen kann der Entwickler in den Clusterprozeß einbringen, indem er spezifiziert, welche Elemente des Systems während des Clusterns nicht getrennt werden dürfen.

In beiden Varianten ergeben sich Nebenbedingungen für die Clusterverfahren, die oft zu besseren Resultaten führen können. Optimierende Clusterverfahren, wie das in Abschnitt 13.3 beschriebene Clusterverfahren des Werkzeugs *Bunch* eignen sich hierfür besonders gut.





13 Weitere Clusterverfahren

In diesem Abschnitt werden wir weitere Verfahren zur Identifikation von Komponentenkandidaten beschreiben. Alle diese Verfahren arbeiten mit Hilfe verschiedenster Clusteralgorithmen und verwenden unterschiedliche Kriterien bzw. Ähnlichkeitsmaße während des Clusterns. Zur Darstellung und Einordnung der unterschiedlichen Verfahren verwenden wir die in [Wig97] vorgeschlagene Klassifikation der Clusteralgorithmen in *Hierarchische Algorithmen*, *Graphtheoretische Algorithmen* und *Optimierende Algorithmen*.

Auf *hierarchische Algorithmen* sind wir in Kapitel 10 schon genauer eingegangen.

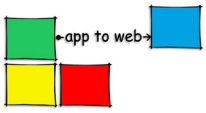
Graphtheoretische Algorithmen arbeiten auf Graphen und berechnen Cluster als Teilgraphen anhand graphtheoretischer Konzepte (Zusammenhangskomponenten, minimale Spannbäume). Geeignete Graphen gehen aus dem in durch das Strukturmodell eines Systems definierten Graphen hervor, indem dessen Knoten durch Kanten verknüpft werden, falls das Ähnlichkeitsmaß zwischen jeweils zwei Knoten (bzw. Klassen) einen bestimmten Schwellwert überschreitet.

Optimierende Algorithmen verwenden ein (gegebenes oder zufällig berechnetes) initiales Clustering und versuchen, jenes durch sukzessive Anpassungen zu verbessern, beispielsweise durch Verschieben von Einheiten zwischen Clustern, so daß eine Zielfunktion, die aus dem Ähnlichkeitsmaß gebildet werden kann, optimiert wird.

Um die unterschiedlichen Verfahren zu vergleichen und gegenüberzustellen, führen wir die folgendes Beschreibungsformat ein:

Fokus Dieser Eintrag enthält eine Kurzbeschreibung des jeweiligen Verfahrens. Diese beschreibt Zielsetzung und Umfang des Ansatzes, wie er von seinen ursprünglichen Autoren konzipiert wurde (z.B. war der Ansatz dazu konzipiert, um die Gesamtarchitektur von Systemen wiederherzustellen oder war er ursprünglich für eine Teilaufgabe geplant, wie der Dokumentation von Interaktionen zwischen ausgewählten Subsystemen?) Ferner geben wir an, welche Aspekte (z.B. Subsysteme, Module, Komponenten, Konnektoren, Architekturstile) direkt mit Hilfe den Ansatz identifiziert werden können. Obwohl die einzelnen Verfahren durchaus für unterschiedliche Ziele (z.B. zur Qualitätssicherung und dergleichen) entwickelt wurden, können die nachfolgend beschriebenen Ansätze durchaus mit Erfolg zur Identifikation von Komponentenkandidaten herangezogen werden.

Eingabe Dieser Eintrag spezifiziert die Information, die durch das Verfahren verwertet wird. Mögliche Informationsquellen umfassen extrahierte Code-Fakten über ein System



und Informationen, die aus externen Quellen (also nicht aus dem Quellcode des Systems), wie beispielsweise aus Namenskonventionen oder aus der Architektur aus der Sicht des Wartungsteams gewonnen werden.

Ausgabe Der Ausgabe-Eintrag beschreibt die Ergebnisse des Ansatzes, ihre Charakteristika und ihr Format. Beispiele wichtiger Komponenten-Charakteristika umfassen, ob ihre Anordnung flach oder hierarchisch ist, ob verschiedene Komponenten Elemente gemeinsam benutzen (überlappende Komponenten) oder ob diese Komponenten einem festen Muster folgen.

Schritte des Ansatzes Dieser Eintrag beschreibt die Schritte, die in einem Ansatz verwendet werden oder bietet Pseudocode für den verwendeten Algorithmus.

Analystenrolle Die verschiedenen Verfahren variieren bzgl. des Maßes an Intervention, die von den Analysten bzw. Softwareingenieuren verlangt wird. Am einen Ende des Spektrums bieten einige Ansätze Werkzeuge zur Visualisierung von Systemeinheiten und zur Manipulation dieser Visualisierung, aber lasten dem Analysten die Verantwortung zur eigentlichen Komponentenidentifikation auf. Am anderem Ende bieten einige Ansätze Werkzeuge, die automatisch eine Sicht der Architektur wiederherstellen, welche durch einen Experten validiert werden kann. Innerhalb dieses Spektrums kann ein Analyst verschiedene Rollen spielen. Die Hauptrollen bestehen darin, von Details des Systems zu abstrahieren, ein Modell anzugeben, welches als Eingabe für die Verfahren genutzt werden kann, die Ergebnisse der Verfahren zu validieren und zu korrigieren.

Erfahrungen Dieser Eintrag nennt die Größe und Art der Systeme (z.B. monolithisch, verteilt, transaktionsbasiert, Client-Server), auf die der Ansatz beispielsweise in Form von Fallstudien angewandt worden ist. Er berichtet über möglicherweise erfolgte experimentelle Validierung und bekannte Einsätze des Verfahrens im industriellen Rahmen. Er kommentiert auch die Relevanz der Erfahrungen oder Fallstudien.

Evaluierung Unter diesem Eintrag werden die Stärken und Schwächen des Ansatzes bzgl. der Architektur-Rückgewinnung diskutiert. Skalierbarkeit, Wiederholbarkeit und Stabilität bzgl. kleiner Änderungen sind wünschenswerte Schlüsselcharakteristika eines Verfahrens zur Komponentenidentifikation.

13.1 Hierarchische Verfahren

Wie bereits erwähnt, lassen sich die hierarchischen Ansätze leicht verstehen, implementieren und liefern recht ordentliche Resultate. Daher findet man in der Reengineering-Literatur entsprechend viele Arbeiten, die auf hierarchische Clusteralgorithmen aufbauen. Im allgemeinen leiden hierarchische agglomerative clusterbasierte

Ansätze an der Tatsache, daß ihre Ausgabe aus einem Dendrogramm besteht, welches in Komponenten umgewandelt werden muß. Keiner der Ansätze beschreibt jedoch, wie diese Konversion vorgenommen werden kann. Eine Lösung besteht darin, einen Experten zur Auswahl der Komponenten im Dendrogramm heranzuziehen, aber dies ist aufwendig. Eine Alternative wäre, einen automatischen oder semiautomatischen Ansatz zu entwickeln, um die Konvertierung vorzunehmen, aber bis dieser Ansatz entwickelt ist, kann man nicht direkt von automatisch gewonnenen Komponentenkandidaten sprechen.

Arch

Fokus

Schwanke [Sch91] schlägt einen Ansatz vor, die Systemstruktur prozeduraler Systeme zu analysieren, um Verbesserungen oder neue Modularisierungen vorzuschlagen. Dieser Ansatz wird durch ein Werkzeug namens *Arch* unterstützt.

Eingabe

Die Eingabe zu diesem Ansatz besteht aus einem ungerichteten Graphen, bei dem die Knoten Unterprogrammen, Typen und globalen Variablen entsprechen, und die Kanten den Aufrufen oder Referenzen auf Typen oder Variablen in Unterprogrammen.

Ausgabe

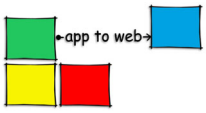
Die Ausgabe dieses Ansatzes ist ein Dendrogramm von Unterprogrammen. Die Komponenten, welche aus diesem Dendrogramm abgeleitet werden können, stellen Modulkandidaten des Systems dar.

Analystenrolle

Die Rolle des Analysten hängt davon ab, welche der drei Variationen (interaktives Reclustering, interaktiv-radikales Reclustering und Batch-Clustering) des Ansatzes verwendet wird. In allen Fällen umfaßt die Analystenrolle das Validieren der Endergebnisse und die Interpretation des Dendrogrammes. In der Batch-Clustering-Version ist dies seine einzige Rolle. Beim interaktiv-radikalen Clustering muß er außerdem entscheiden, ob jedes Clusterpaar, das vom Werkzeug präsentiert wird, kombiniert werden soll oder nicht. Beim interaktiven Clustering muß der Analyst nur solche Clusterkombinationen nachprüfen, die sich im jeweils vorherigen Clustering in verschiedenen Clustern befanden.

Ansatz

Der Ansatz führt eine hierarchische Cluster-Analyse von Unterprogrammen durch, basierend auf ihrer direkten Beziehung (d.h. Aufrufe) und den Beziehungen, die sie sich mit dritten Entitäten teilen, was Unterprogramme, Typen und globale Variablen sein können. Der Ansatz unterscheidet sich vom einfachen HAC dadurch, daß er terminiert, sobald “die existierenden Cluster zufriedenstellend sind”.



Die verwendete Ähnlichkeitsfunktion für zwei Unterprogramme *A* und *B* ist wie folgt definiert:

$$Sim(A,B) = \frac{W(a \cap b) + k \times Linked(A,B)}{n + W(a \cap b) + d \times (W(a-b) + W(b-a))}$$

wobei

$a \cap b$ die Menge an Features (Typen, Variablen) ist, die *A* und *B* gemeinsam ist.

$a-b, b-a$ die Mengen an Features sind, welche jeweils unterschiedlich zu *A* und *B* sind.

$w_x > 0$ die Gewichtung des Features x ist.

$W(X) = \sum_{x \in X} w_x$ die Gewichtung von X ist.

$Linked(S,N) = \begin{cases} 1, \text{ if A calls B or B calls A} \\ 0, \text{ otherwise} \end{cases}$ die direkten Beziehungen erfaßt.

k, d, n Parameter zur Feinabstimmung sind.

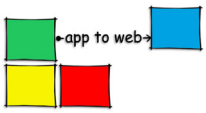
Die Ähnlichkeit zwischen Gruppen/Clustern von Unterprogrammen berechnet sich aus der maximalen Ähnlichkeit zwischen jedem Paar der zwei Cluster (Update-Regel).

Erfahrungen

Dieser Ansatz wurde auf den Quellcode des *Arch*-Werkzeugs selbst angewandt (Dieses in C geschriebene Werkzeug umfaßt 64 Unterprogramme, verteilt auf 7 Module). Nach manueller Anpassung der individuellen Gewichtung von 10% der Features (80 insgesamt) erreichte der Ansatz völlige Übereinstimmung mit den Autoren des Werkzeugs. Die manuelle Anpassung der Feature-Gewichtungen kann die Skalierbarkeit des Ansatzes stark einschränken. In einem späteren Artikel führt der Autor ein neuronales Netzwerk ein, um die Gewichtung des Features durch überwachtes Lernen zu ermitteln.

Evaluierung

Die Anpassung der Gewichtung stellt einen aufwendigen Aspekt dieses Ansatzes dar. Das Problem der Komponentenselektion aus dem Dendrogramm und der damit verbundene Aufwand schränken den Anreiz des Ansatzes ein.



Clustering mit Datenbindung

Fokus

Hutchens und Basili [HB85] schlagen zwei verwandte Ansätze zum Clustering von Unterprogrammen in prozeduralen Systemen vor, die auf Datenbindung basieren (d.h. der Kommunikation mittels globaler Variablen). Das Ziel dieser Ansätze besteht darin, eine hierarchische Beschreibung eines Systems (Clustering) zu erzeugen, welche dazu verwendet werden kann, die Eigenschaften des Systems (z.B. Stärke der Kopplung) zu studieren, Maße zu definieren und den Bereich des Systems zu untersuchen, an dem das Clustering nicht mit der von den Programmierern geschaffenen Dekomposition übereinstimmt.

Eingabe

Die Eingabe zu den Verfahren von Hutchen und Basili besteht aus einem Daten-Abhängigkeitsgraphen, welcher die Datenbindung zwischen Unterprogrammen und dem Quellcode des Programms erfaßt. Die gewählte Datenbindung wird “tatsächliche Datenbindung” genannt. Es gibt eine tatsächliche Bindung zwischen den Unterprogrammen p und q durch eine Variable x, wenn p nach x schreibt und q aus x liest.

Ausgabe

Die Ausgabe dieser Ansätze ist ein Dendrogramm von Unterprogrammen. Die Komponenten, welche aus diesem Dendrogramm abgeleitet werden können, stellen Modulkandidaten des Systems dar.

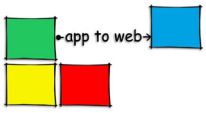
Analystenrolle

Der Analyst filtert zuvor jene Hilfsroutinen (z.B. Bibliotheken und ADTs) aus, die nicht mit diesen Ansätzen geclustert werden sollen. Nach Anwendung der automatisierten Verfahren vergleicht der Analyst das Dendrogramm mit den Komponentenstrukturen, wie sie vom Experten (z.B. dem Wartungsteam oder dem Systemarchitekten) verstanden werden.

Ansätze

Die beiden Ansätze, wiederberechnete Bindungen und erwartete Bindungen genannt, verwenden eine gemeinsame Clustering-Strategie und unterscheiden sich nur in der verwendeten Unähnlichkeitsmetrik. Diese Strategie arbeitet auf zwei Matrizen: eine enthält die Datenbindung zwischen Unterprogrammen und eine zweite die Unähnlichkeit zwischen ihnen. Der Algorithmus kann wie folgt skizziert werden:

```
1  Filtere jene Unterprogramme heraus, die zu Hilfsroutinen gehören
2  Setze jedes übriggebliebene Unterprogramm in ein eigenes Cluster, diese
   Cluster bilden die Blätter des Dendrogramms
3  Initialisiere die Binde-Matrix mit der tatsächlichen Datenbindung
   zwischen jedem Clusterpaar
4  Solange es mehr als ein Cluster gibt
5     Errechne die Unähnlichkeitsmatrix aus der Binde-Matrix
6     Fasse die beiden Cluster mit der kleinsten Unähnlichkeit zusammen
7     Füge einen Knoten entsprechend dieser Mischung in das Dendrogramm auf
      der Stufe = Unähnlichkeit ein
8     Aktualisiere die Bindematrix bzgl. dieser zusammengefaßten Cluster
9  Ende
10 Für jeden Knoten im Dendrogramm
```




```
11    Wenn Stufe des Sohns > Stufe des Vaters dann
12       Lege den Vater mit dem Sohn auf der Stufe des Sohns zusammen
13 Ende
```

Die Ähnlichkeit zwischen Clustern wird direkt errechnet, weil nach jeder Iteration die Bindematrix neu berechnet wird. Auf die Angabe einer expliziten Update-Regel kann somit verzichtet werden.

Für die wiederberechnete Datenbindung werden die Elemente der Unähnlichkeitsmatrix wie folgt berechnet:

$$d(i,j) = \frac{sum_i + sum_j - 2b(i,j)}{sum_i + sum_j - b(i,j)}$$

wobei

sum_x die Anzahl an Datenbindungen des Clusters x ist

$b(i,j)$ die Anzahl an Datenbindungen *(i, v, j)* oder *(j, v, i)* zwischen zwei Clustern über alle möglichen Variablen v ist.

Für die erwartete Datenbindung werden die Elemente der Unähnlichkeitsmatrix wie folgt berechnet:

$$d(i,j) = \frac{sum_i + sum_j - b(i,j)}{(dim(b(i,j)) - 1) \times b(i,j)}$$

wobei $dim(b(i,j))$ die Dimension der Bindematrix ist.

Diese Ansätze sind prinzipiell Vertreter aus der Klasse der hierarchischen Clusterverfahren[1].

Erfahrungen

Dieser Ansatz wurde auf zwei FORTRAN-Systeme (100 kLOC bzw. 64 kLOC) der NASA/SEL und auf 19 Implementationen eines Compilers aus Studentenprojekten angewandt. Aus den Fallstudien schließen die Autoren, daß es “so erscheint, daß Clu-

1 Diese Ansätze unterscheiden sich von vergleichbaren Clusterverfahren aus der Klasse der hierarchischen Algorithmen dadurch, daß Cluster aus späteren Interationen höhere Ähnlichkeitsmaße besitzen als Cluster aus vornagehenden Iterationen..In den Schritten 10 bis 12 wird diese Anomalie korrigiert.

stering mit Datenbindung die logischen Module eines Systems identifizieren können.” Dieser Ansatz funktioniert mit Systemen, bei denen der Datenaustausch hauptsächlich über globale Variablen erfolgt.

Evaluierung Das Problem, die Komponenten aus dem Dendrogramm zu selektieren, sowie der damit verbundene Aufwand schränken den Anreiz des Ansatzes ein.

13.2 Graphtheoretische Algorithmen

k-Cut-Modularisierung

Fokus In [Jer99] schlägt Jermaine diese Technik vor, um ein System ein System in seine Hauptkomponenten derart zu zerlegen, daß die Anzahl der Aufrufs-Kanten zwischen den erzeugten Komponenten reduziert wird. Als solche ist diese Technik am besten geeignet für monolithische prozedurale Systeme oder individuelle Programme eines verteilten Systems. Es läßt sich jedoch auf objekt-orientierte Systeme leicht übertragen.

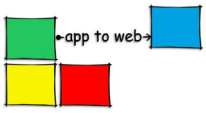
Eingabe Die Eingabe besteht aus einem Aufufgraphen des Systems, dessen Knoten die unterschiedlichen Unterprogramme/Funktionen des Systems darstellen.

Ausgabe Die Ausgabe besteht aus den Gruppen an Unterprogrammen, welche die Hauptkomponenten des Systems darstellen.

Analystenrolle Der Analyst gibt eine angemessene Zahl *k* an Komponenten vor. Diese Zahl läßt sich beispielsweise mit Hilfe von Diagrammen, die Meßwerte geeigneter Qualitätsmetriken gegen unterschiedliche *k* darstellen, wählen. Er interpretiert die Ergebnisse. Jedoch gibt es keinen Vorschlag, wie der Analyst die Ergebnisse modifizieren kann und diese als Basis für einen eines Verbesserungslauf mit der Technik vorschlagen kann.

Ansatz Diese Technik verwendet eine Approximation des *minimalen k-Schnittes*, um das System in *k* Komponenten zu modularisieren. Diese Approximation erzeugt eine Zerlegung, bei der die Anzahl der Aufruf-Kanten, welche die Komponentengrenze kreuzt, sich innerhalb eines Faktors 2 vom Minimum bewegt. Die Technik benötigt folgende 4 Schritte:

```
1  Errechne eine Menge von Gomory-Hu-Schnitten für den Graphen
2  Sortiere die Gomory-Hu-Schnitte nach zunehmender Gewichtung
```




```
3  // Wähle die angemessene Zahl an Schnitten k
4  Errechne eine Soziabilitäts-Metrik der Komponenten für verschiedene
   Schnitte und wähle die Schnitte aus, welche den besten Soziabilitäts-
   Wert bieten
5  Für das selektierte k schneide den Graphen bzgl. der am wenigsten
   aufwendigen Gomory-Hu-Schnitte
```

Dies stützt sich auf folgendes (rekursive) Unterprogramm *Produziere einen Gomory-Hu-Schnitt für einen Aufruf-Graph G*:

```
1  Prozedur Gomory-Hu-Schnitt
2  Wähle zwei willkürliche Kanten v1 and v2 aus dem Graph G
   // Errechne den minimalen 2-Schnitt des Graphen mit der Einschränkung,
      daß v1 und v2 darauf liegen
3  // auf beiden Seiten des Schnitts:
4  Wiederhole
5     Wähle irgendeinen Pfad von v1 nach v2
6     Laufe diesen Pfad entlang
7     Für jede hierbei durchlaufene Kante
8        Wenn die Kante ungerichtet ist
9           Mache die Kante gerichtet in die Gegenrichtung des Laufs
10       Sonst
11          Mache die Kante ungerichtet
12 Bis es keine Pfade mehr von v1 nach v2 gibt
13 Untergraph1 = erreichbare Knoten von v1
14 Untergraph2 = unerreichbare Knoten von v1
15 Entferne die Schnitt-Kanten vom Graph G
16 Gomory-Hu-Schnitt (Untergraph1)
17 Gomory-Hu-Schnitt (Untergraph2)
```

Ein Vorteil dieser Technik ist, daß sie eine Familie an Schnitten ermittelt, so daß die Entscheidung bzgl. des angemessenen *k* nach der Berechnung erfolgen kann, und daß die Entscheidung für ein neues *k* keine großen Neuberechnungen impliziert.

Erfahrungen

Diese Technik wurde für kleine Systeme (mit 13, 32 und 55 Funktionen) ausprobiert, um die Ergebnisse per Hand analysieren zu können, und für den Browser *Mosaic* (1644 Funktionen, 112 kLOC C-Code), um die Skalierbarkeit zu testen. Der Autor schließt aus dem Skalierbarkeitstest, daß der Algorithmus eine Laufzeit etwa quadratisch bzgl. der Größe des Programm-Aufruf-Graphen hat. Die Technik kann zu einer sehr unausgeglichenen Dekomposition führen: sehr kleine und sehr große Komponenten.

Evaluierung

Diese Technik ist ansprechend: Sie bietet eine Dekomposition, welche die Kontroll-Kopplung minimiert, und sie besitzt eine akzeptable Performanz. Jedoch kann die Technik zu einer unausgeglichenen Dekomposition führen, welche stark von der konzeptionellen Architektur divergiert. Die Tatsache, daß der Ansatz keine Möglichkeit

bietet, Anregungen aus dem Know-How der Entwickler zu integrieren oder korrigierte Dekompositionen aufzunehmen, ist auch ein schwerer Nachteil.

13.3 Iterative Partitionierung, optimierende Verfahren

Eigenvektor-Partitionierung

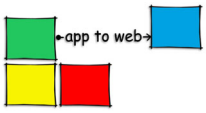
Dieser Ansatz basiert nicht auf hierarchischen Clustering-Analyse, wie die in Abschnitt 13.1 dargestellten Verfahren. Es handelt sich um einen optimierendes Verfahren, welches oft auch iterative Partitionierung genannt wird [Wig97]. Algorithmen dieser Klasse (siehe auch [Eve74] für Details) erzeugen eine anfängliche Partition und versuchen dann, jeden Knoten zu verschieben, um die Partitionierung hinsichtlich eines gegebenen Optimierungskriteriums (auch Zielfunktion genannt) zu verbessern. Der Ansatz terminiert, wenn keine weitere Verbesserung möglich ist.

Fokus Dieser Ansatz, vorgeschlagen von Belady und Evangelisti [BE81], konzentriert sich darauf, eine gegebene Anzahl an Komponenten zu erzeugen, welche eine gegebene maximale Anzahl an Unterprogrammen und Datenstrukturen enthalten. Dieser Ansatz basiert auf einem Algorithmus, der für das Clustering von Digitalschaltungen zum Entwurf von Chip-Designs entwickelt wurde.

Eingabe Die Eingabe besteht aus zwei Parametern (die gewünschte Anzahl an Komponenten *N*, und der oberen Grenze der Anzahl von Elementen in jeder Komponente *UB*) sowie eines bipartiten Graphen, bei dem die Knoten den Unterprogrammen und der Datenstruktur entsprechen und die Kanten dem Datenzugriff eines Unterprogramms auf eine Datenstruktur. Der bipartite Graph wird als eine Verbindungsmatrix repräsentiert.

Ausgabe Die endgültige Ausgabe setzt sich zusammen aus *N* Komponenten, welche jeweils bis zu *UB* Elemente (d.h. Unterprogramme und Datenstrukturen) enthalten. Die mittels dieses Ansatzes erzeugten Komponenten stellen Modulkandidaten des Systems dar.

Analystenrolle Die Rolle des Analysten besteht darin, die Anzahl an zu erzeugenden Clustern und die Maximalgröße jedes Clusters anzugeben, sowie die Resultate zu validieren, ggf. mit neuen Parametern das Verfahren neu anzustoßen und eines der Resultate auszuwählen.



Ansatz

Der Ansatz ist eine Instanz einer Clustering-Methode, bei der die verwendete Distanz in Form eines Eigenvektors definiert wird. Der Ansatz ist auf eine Komplexitätsmetrik angewiesen.

Dieser Ansatz erzeugt Subsysteme, welche aus Elementen (Unterprogrammen oder Datenstrukturen) bestehen, die in einem *N*-dimensionalen Raum dicht beieinander liegen. Die Plazierung jedes einzelnen Elements in einem *N*-Raum wird durch die Berechnung von N Eigenvektoren (die Autoren schlagen typischerweise 5 vor) erreicht. Der *i*-te Wert dieser Vektoren entspricht dem *i*-ten Element in der Verbindungsmatrix. Die Autoren variieren die Anzahl an Clustern und passen die Maximalgröße des Clusters an, um den Cluster ausgeglichen zu halten. Sie schlagen eine Komplexitätsmetrik als Hilfe zur Auswahl der besten Partition vor.

Erfahrungen

Dieser Ansatz wurde auf die etwa 500 Unterprogramme und 270 Kontrollblöcke eines IBM-Betriebssystems namens VTAM angewandt. Die Autoren berichten jedoch nicht davon, daß die von dem Ansatz erzeugte Partition von irgendjemandem validiert wurde, der die Struktur des Systems kannte oder, daß die Struktur durch mehr als der von ihnen vorgestellten Komplexitätsmetrik bewertet wurde.

Evaluierung

Die durch den Artikel gebotene Evaluierung bietet beschränkte Einsichten in die Resultate, da sie nur gegen eine Metrik geprüft wurden, welche nicht mit irgendeiner Entwicklungs- oder Wartungseigenschaft korreliert war. Der Ansatz wird nicht klar genug in den Artikeln [DH72] [DH73] [BE81] beschrieben, um eine direkte Implementierung zu liefern. Zur Implementierung müssen viele Schritte mittels der in den Artikeln präsentierten Beschreibung erraten werden. Demzufolge würde ein Implementierungserfolg ebenso viel von der Kreativität des Entwicklers abhängen wie von der Idee des ursprünglichen Autors.

Bunch

Fokus

Dieser Ansatz, entworfen von Mancodiris et al [MM+98] [MMCG99] konzentriert sich auf die Identifizierung der Hauptkomponenten und ihrer Abhängigkeiten in einem objektorientierten oder prozeduralen System. Die Autoren entwickelten *Bunch*, ein Werkzeug, das diesen Ansatz unterstützt.

Eingabe

Die Eingabe besteht aus einem Graphen, bei dem die Kanten die Existenz einer Abhängigkeit zwischen zwei Entitäten (Quelltext-Dateien, Unterprogrammen, Klassen) anzeigen, aber nicht die Art dieser Abhängigkeit (d.h. das Verfahren unterscheidet nicht, ob es sich um einen einzelnen Aufruf oder viele Aufrufe und Datenzugriffe handelt).

Ausgabe

Die Ausgabe besteht aus einer Dekomposition des Systems in Komponenten einer einzigen Hierarchieebene, welche die Module des Systems bilden. Die mit diesem Ansatz erzeugten Komponenten gehören zur Modulsicht eines Systems.

Analystenrolle

Die Rolle des Analysten besteht darin, einige Parameter für das Werkzeug zu wählen, ein Teilmodell zu liefern, indem allgegenwärtige Module entweder als Clients (Treiber) oder Hilfsmodule/Bibliotheken (Supplier) klassifiziert werden, und ggf. einige Module manuell zu gruppieren, basierend auf seinem Wissen des Systems vor Start der automatischen Analyse. Nach Beendigung der automatischen Analyse validiert er die Ergebnisse.

Ansatz

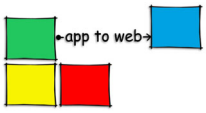
Dieser Ansatz verwendet einen Clusteralgorithmus, der das System so in Cluster zerlegt, daß die dadurch entstehende Dekomposition des Systems die interne Kohäsion der Komponenten (Intrakonnektivität) maximiert und die externe Kopplung zwischen den Komponenten (Interkonnektivität) minimiert. Aus diesen wünschenswerten Konnektivitätseigenschaften entsteht eine Modularisierungs-Qualitätsmetrik. Mit Hilfe eines naiven Algorithmus könnte im Prinzip eine optimale Zerlegung des Systems hinsichtlich dieser Metrik einfach berechnet werden, indem alle möglichen Zerlegungen des Systems mit dieser Metrik bewertet werden und die beste ausgewählt wird. In der Praxis scheitert dieser naive Ansatz jedoch am dafür erforderlichen Rechenaufwand. Bunch verwendet daher einen genetischen Algorithmus, der eine hinsichtlich der Metrik zufriedenstellende – aber im allgemeinen sub-optimale – Zerlegung des Systems berechnet. Dazu wird eine Population von (zufällig gewählten) Zerlegungen des Systems schrittweise verbessert, indem zwischen den einzelnen Clustern Entitäten verschoben werden. Die aussichtreichsten Zerlegungen werden mit Hilfe der Modularisierungsqualitätsmetrik ausgewählt und bilden so als nächste Generation die Basis für die nächste Iteration des Verfahrens. Das Verfahren endet nach einer vorgegebenen Anzahl von Generationen oder wenn keine Verbesserungen mehr erzielt werden können.

Der Analyst liefert dem Werkzeug folgende Eingabeparameter:

- O – die Grenze[1] für die Anzahl der Abhängigkeiten, ab der Module als allgegenwärtig klassifiziert werden.
- N – die Anzahl an Generationen, welche durch den genetischen Algorithmus erkundet werden sollen
- P – die Größe der Population (Anzahl unterschiedlicher Ausgangszerlegungen, die in jeder Generation verwendet wird)

Mit diesen Eingaben führt der Ansatz folgende Schritte aus:

1 Diese Grenze ist ein Vielfaches der Durchschnittszahl an Abhängigkeiten, in die die Module des Systems einbezogen sind.



1. *Bunch* identifiziert allgegenwärtige Module mittels einfacher Heuristiken über die Anzahl der Abhängigkeiten und mittels Vergleich mit der Grenze *O* und präsentiert sie dem Analysten.

2. Der Analyst klassifiziert die allgegenwärtigen Module als Clients (Treiber) oder Suppliers (Hilfsmodule).

3. Der Analyst gruppiert bestimmte Module vorab gemäß seines Domänenwissens. Standardmäßig wird *Bunch* diese Cluster nie verwerfen, aber der Analyst kann es *Bunch* erlauben, Module zu einigen dieser Cluster hinzuzufügen, wenn es die Modularisierungsqualität verbessert.

4. *Bunch* führt den genetischen Algorithmus für *N* Generationen durch, wobei die Ergebnisse jeder Generation mit der Modularisierungs-Qualitatsmetrik beurteilt werden.

Die Modularisierungs-Qualitätsmetrik (MQ) ist wie folgt definiert:

$$MQ = \begin{cases} \dfrac{\sum_{i=1}^{k} A_i}{k} - \dfrac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & k > 1 \\ A_1 & k = 1 \end{cases}$$

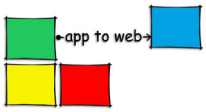
Intrakonnectivität (A) $\quad A_i = \mu_i / N_i^2$

Interkonnectivität (E) $\quad E_{i,j} = \begin{cases} 0 & i = j \\ \varepsilon_{i,j} / 2N_i N_j & i \neq j \end{cases}$

In diesen Formeln entspricht N_i der Anzahl der Knoten im Cluster *i*, μ_i repräsentiert die Anzahl der Kanten im Cluster *i* und $\varepsilon_{i,j}$ die Zahl von Kanten zwischen Cluster *i* und Cluster j.

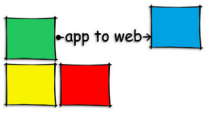
Erfahrungen

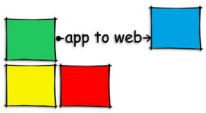
Die Autoren berichteten davon, *Bunch* auf verschiedene C- und C++-Systeme angewandt zu haben, die aus 13 bis 153 Modulen bestanden. Diese umfassen die Rechtschreibprüfung *ispell* (22 Module), das Versionsverwaltungssystem *rcs* (27 Module), *bison* (37 Module), ein Graph-Zeichenwerkzeug namens *dot* (42 Module) und ein proprietäres System aus 153 Modulen.



Evaluierung

Die Qualität der mit diesem Ansatz produzierten Ergebnisse sind sehr stark auf die verwendete Metrik angewiesen, weil diese Metrik unter den unterschiedlichen Zerlegungen des Systems, welche innerhalb einer Generation entstehen, die jeweils beste zum Überleben auswählt und in nachfolgenden Generationen verbessert. Wenn dieser Ansatz in der Praxis nützlich sein soll, sollte diese Metrik die meisten der wichtigen Charakteristika einer guten Architekturbeschreibung für ein gegebenes System umfassen. Positiv zu vermerken ist, daß Bunch brauchbare Möglichkeiten bietet, um einerseits bestimmte Modulkandidaten (Bibliotheken, Treiber) gesondert zu behandeln, und daß Entwickler-Know-How durch die Vorabselektion von Komponenten geeignet integriert werden kann.





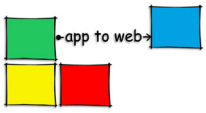
14 Zusammenfassung und Ausblick

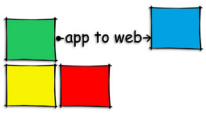
In diesem Dokument haben wir eine Vorgehensweise beschrieben, mit Hilfe derer Komponentenkandidaten im Quellcode von Altsystemen identifiziert werden können. Als zentrale Technik verwenden wir *Clusteralgorithmen*, mit deren Hilfe wir zusammenhängende strukturgebende Elemente des Systems (wie Klassen oder Funktionen) zu Komponentenkandidaten gruppieren. Wir haben illustriert, wie sich dieses Verfahren sowohl auf objektorientierte, als auch auf prozedurale Systeme anwenden läßt. Zusätzlich haben wir vergleichbare Ansätze zusammengestellt und bewertet, so daß insgesamt ein *breites Spektrum an Verfahren* zur Verfügung steht, mit deren Hilfe Komponentenkandidaten in Softwaresystemen identifiziert werden können.

Die Qualität der mittels dieser Verfahrens erzielbaren Ergebnisse hängt stark davon ab, welche *Kriterien* dazu verwendet werden, um zu entscheiden, welche strukturgebenden Elemente zusammen in ein Cluster bzw. in einen Komponentenkandidaten plaziert werden sollen.

Das von uns beschriebene Verfahren bietet – wie alle Verfahren in diesem Bereich – weiteres Verbesserungspotenzial hinsichtlich der folgenden Punkte:

- Optimierung und Anpassung der Kriterien, mit deren Hilfe die Clusteralgorithmen die Elemente des Systems zu Komponentenkandidaten anordnen. Dies ist insbesondere deshalb lohnenswert, damit wir unterschiedlichen Strukturmustern und Programmierstilen, wie sie in Systemen verwendet werden, Rechnung tragen können. Dies erfordert jedoch eine systematische Evaluierung verschiedener Kriterien für unterschiedliche Fallstudien.
- Schaffung geeigneter Interaktionsmöglichkeiten für Softwareentwickler zur Integration bestehenden Entwurfs-Know-Hows in die Verfahren zur Komponentenidentifikation. Einige Ansätze hierfür haben wir im Rahmen dieses Dokumentes bereits diskutiert. Solche Ansätze profitieren stark von geeigneten Visualisierungen der Softwarestrukturen eines Systems. Daher sehen wir in der Verknüpfung von Werkzeugen zur Softwarevisualisierung mit den hier dargestellten Clusterverfahren ein interessantes Gebiet für zukünftige Forschungsarbeiten.





15 Literatur

[AAB+00] Michalis Anastasopoulos, Christoph Andriessens, Holger Bär, Christian Bunse, Jean-Francois Girard, Isabel John, Dirk Muthig, and Tim Romberg. Überblick Stand der Technik. Technical report, Application2Web, December 2000.

[AB+01] Christoph Andriessens, Markus Bauer, , Jean-Francois Girard, and Olaf Seng. Redokumentation von altsystemen. Technical report, Application2Web, June 2001.

[BBC+99] Holger Bär, Markus Bauer, Oliver Ciupke, Serge Demeyer, Stéphane Ducasse, Radu Marinescu, Robb Nebbe, Tamar Richner, Matthias Rieger, Benedikt Schulz, Sander Tichelaar, and Joachim Weisbrod. The famoos handbook of reengineering, September 1999.

[BE81] L. Belady and C. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 1981.

[DH72] W.E. Donath and A.J. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connections matrices. Technical report, IBM Watson Research Centre, 1972.

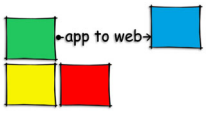
[DH73] W.E. Donath and A.J. Hoffman. Lower bounds for the partitioning of graphs. Technical report, IBM Watson Research Centre, 1973.

[Eve74] B. Everitt. *Cluster Analysis*. Heinemann Education Books Ltd, 1974.

[GK] Thomas Genssler and Volker Kuttruff. Injectj homepage. http://injectj.fzi.de.

[HB85] D. H. Hutchens and B. R. Basili. System Structure Analysis: Clustering with Data Binding. In I*EEE Transactions on Software Engineering,* pages 749–757, Aug. 85.

[Jer99] C. Jermaine. Computing program moduliarizations using the k-cut method. In *Proceeding of the 6th Working Conference on Reverse Engineering*, Oct. 1999.



[KR90] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, 1990.

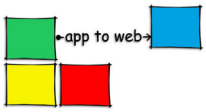
[MMCG99] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of the IEEE International Conference on Software Maintenance 1999*, pages 50–59, Oxford, Aug. 1999.

[MM+98] S. Mancoridis, B.S. Mitchell, , C. Rorres, Y. Chen, and E.R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Procedings of the 6th International Workshop on Program Comprehension*, 1998.

[Sch91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the International Conference on Software Engineering*, pages 83–92, May 1991.

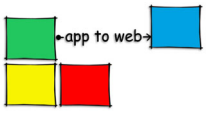
[Sha72] C.E. Shannon. *The mathematical theory of communication*. Urbana University of Illinois Press, 1972.

[Wig97] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *Working Conference on Reverse Engineering*, pages pp. 33–43, Amsterdam, October 1997. IEEE Computer Society Press.



III Transformation von Komponentenkandidaten in Komponenten

Holger Berg
Michael Schlemmer
Olaf Seng



16 Einführung

In den ersten beiden Teilen dieses Dokumentes haben wir grundlegende Techniken erarbeitet und vorgestellt, mit denen wir Codefragemente aus Altsystemen extrahieren können, um sie in moderne, komponenten-orientiert aufgebaute Webanwendungen integrieren zu können.

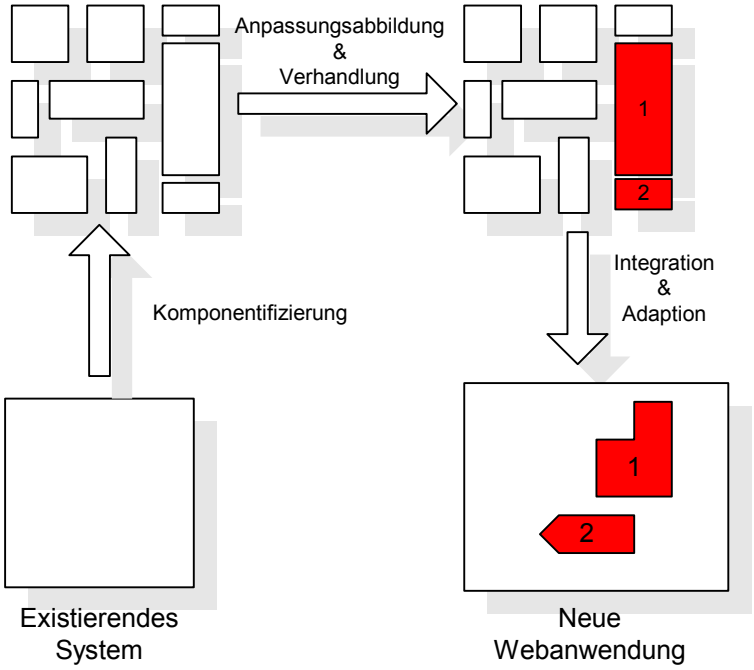
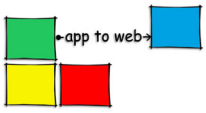


Abbildung 37:Schematische Darstellung der Wiederverwendung von Altsystemteilen

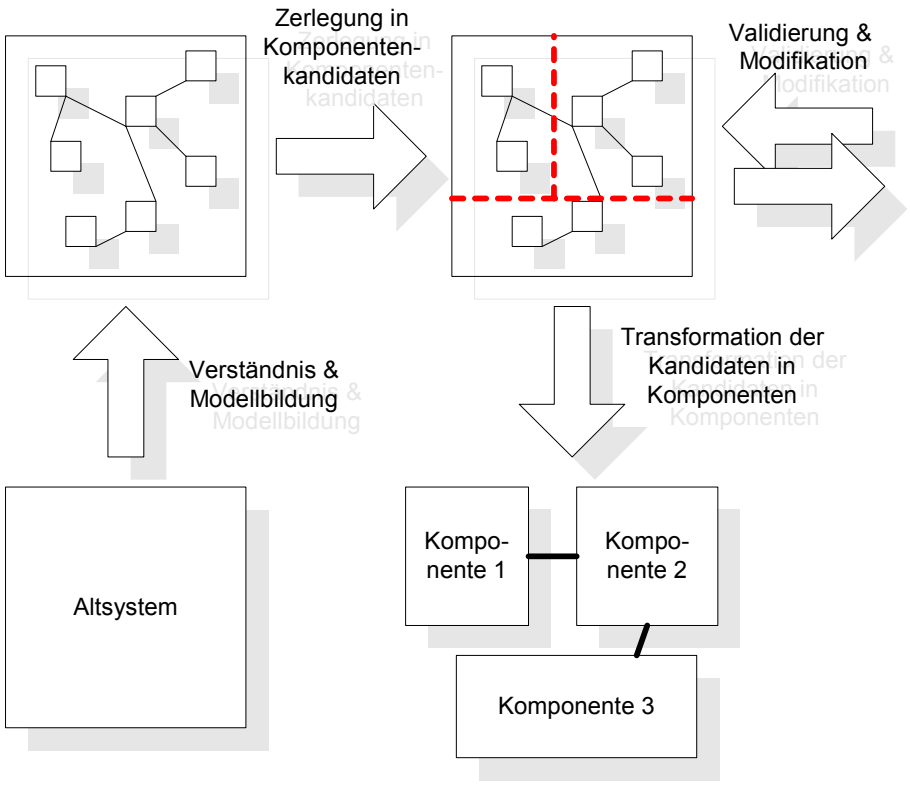
Abbildung 37 zeigt schematisch den Prozess zur Wiederverwendung von Quelltextfragmenten des Altsystems. Auf der linken Seite ist das existierende System - die monolithische Altanwendung - zu sehen, aus dem die wiederzuverwendenden Quelltextteile stammen. Auf der rechten Seite ist die neue Webanwendung zu sehen, in der der Quelltext wiederverwendet werden soll.

In einem ersten Schritt muss das alte System in Komponenten zerlegt werden. Wurde die Komponentifizierung erfolgreich durchgeführt, dann muss in einem zweiten Schritt geklärt werden, welche der extrahierten Komponenten in der neuen Weban-



wendung eingesetzt werden können. Diese sogenannte Anpassungsabbildung und Verhandlung wird in [Bay01] ausführlich beschrieben. Kann die Komponente nicht die Spezifikation erfüllen, die von ihr erwartet wird, weil z.B. die Namen der Schnittstellen nicht passen, dann kann ein Adaptionsschritt nötig sein, um die Komponente entsprechend anzupassen [Sen02].

Abbildung 38: Prozess zur Umwandlung von existierenden Systemen in Komponenten



In diesem Dokument *Transformation von Komponentenkandidaten zu Komponenten* wird auf die Komponentifizierung näher eingegangen. Hinter diesem Begriff stecken mehrere einzelne Aktivitäten, die nicht trivial sind. Diese sind in Abbildung 38 zu sehen und in eine zeitliche Reihenfolge zueinander gesetzt.

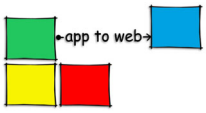
Die erste und wichtigste Aktivität ist es, sich ein Verständnis des Quelltextes anzueignen. Nur wenn man die groben Zusammenhänge und die Funktionalität des Quelltextes versteht, kann man ihn in einem zweiten Schritt entweder von Hand oder automatisch in sinnvolle Komponenten zerlegen. Werkzeuggestützte Vorgehensweisen zur Gewinnung von Komponentenkandidaten haben wir bereits ausführlich im zweiten

Teil dieses Dokumentes beschrieben. Ergebnis der dort skizzierten Techniken zur *Identifikation von Komponentenkandidaten in Altsystemen* sind stets zusammengehörige Mengen von Klassen und Methoden.

Hat man den Quelltext in eine Menge von Komponentenkandidaten partitioniert, dann sollte man diese Partitionierung noch einmal überprüfen. Insbesondere bei werkzeuggestützten automatischen Partitionierungen muss noch einmal untersucht werden, ob ein berechneter Komponentenkandidat den Designüberlegungen des Entwicklers, der den Quelltext in Komponenten zerlegt, entspricht und nur zusammengehörige Klassen enthält. Es könnten auch zusammengehörige Klassen auf mehrere Komponenten verteilt worden sein, die besser zu einer Komponente zusammengelegt werden sollten. Diese Entscheidungen können ohne Benutzereingaben nicht getroffen werden und werden von Fall zu Fall anhand von Entwurfsentscheidungen des Entwicklers gefällt.

In einem letzten Schritt muss man entscheiden, welche Restrukturierungen am Quelltext vorgenommen werden müssen. Diese sind immer dann nötig, wenn die Struktur des Quelltextes, der durch die Komponentenkandidaten gegebenen Struktur widerspricht, z.B. wenn Klassen eines Komponentenkandidaten in zwei verschiedenen Namensräumen liegen, sollte man sie in einem Namensraum zusammenlegen. Diese Restrukturierungen können sowohl manuell als auch werkzeuggestützt erfolgen, allerdings ist eine werkzeuggestützte Vorgehensweise vorzuziehen, weil eine manuelles Vorgehen fehleranfälliger und aufwändiger ist.

In diesem Dokument liegt der Schwerpunkt auf der Umwandlung der Komponentenkandidaten in Komponenten. In Kapitel zwei werden wir zunächst beschreiben, was wir unter einer Komponente verstehen. Diese Beschreibung ist nötig, weil es zum jetzigen Zeitpunkt noch keine eindeutige Definition des Komponentenbegriffs gibt. Kapitel drei beschäftigt sich mit der Evaluation der Komponentenkandidaten und mit der Frage, warum und wie die Struktur des zugrundeliegenden Systems abgeändert werden muss. Wie die Restrukturierungen im Detail aussehen, ist Bestandteil von Kapitel vier. Ein Beispiel zur Umwandlung eines existierenden Systems in Komponenten wird in Kapitel fünf beschrieben. In Kapitel sechs wird noch einmal kurz auf die Adaption von Komponenten eingegangen, die nötig sein kann, wenn Komponenten in einem fremden Kontext eingesetzt werden sollen. In Kapitel sieben werden die existierenden Werkzeuge vorgestellt, die während der Umwandlung in Komponenten nützlich sein können. Den Abschluss des Dokuments bildet eine kurze Zusammenfassung und ein Ausblick auf weitere Arbeiten.



17 Komponentenmodell

In diesem Abschnitt wird das von uns verwendete Komponentenmodell vorgestellt. Dabei muss festgelegt werden, wie eine Komponente definiert ist, und welche Beziehungen zwischen den einzelnen Komponenten existieren können.

17.1 Charakterisierung der Komponenten

Für den Begriff Komponente existiert in der Informatik leider noch keine einheitliche Definition. Es gibt eine Vielzahl von Definitionen, die alle gewisse Gemeinsamkeiten aufweisen, wie z.B. das Komponenten funktionale Kompositionseinheiten sind. Allerdings gibt es auch Unterschiede, da die jeweiligen Definitionen unterschiedliche Aspekte beleuchten, und zum Teil auch Dinge wie Ablaufumgebungen (Corba, DCOM, ...) betrachten.

Wir definieren nun den innerhalb dieses Dokumentes verwendeten Komponentenbegriff. Unsere Definition orientiert sich zum einen an folgender Definition, die aus [Szy98] entnommen ist:

„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“

Zum anderen berücksichtigen wir bei unserer Definition den Sachverhalt, dass wir aus existierenden Systemen Komponenten gewinnen wollen. Unter Komponenten stellen wir uns logisch zusammenhängende, funktionale Einheiten vor, die Berechnungen ausführen. Sie sind deshalb eher grobgranular und enthalten eine Menge von Klassen des Altsystems.

Eine Komponente ist nach unserem Verständnis durch folgende Eigenschaften charakterisierbar:

1. Die in der Komponente enthaltenen Einheiten (Klassen, Methoden, ...) besitzen untereinander eine hohe Kopplung, zu Einheiten anderer Komponenten ist die Kopplung möglichst gering.

2. Die nach außen angebotene Funktionalität ist in einer eigenen Schnittstelle zusammengefasst, der sogenannten *provides-Schnittstelle*.

3. Die von der Komponente benötigte Funktionalität ist in einer eigenen Schnittstelle zusammengefasst, der sogenannten *requires-Schnittstelle*.

Teile eines Systems, die die obigen Bedingungen erfüllen, werden im Folgenden als Komponenten bezeichnet. Mit diesen Eigenschaften wird gewährleistet, dass die Komponenten gut miteinander kombinierbar sind. Eine geringe Kopplung zu anderen Systemteilen erleichtert es, die Komponente in einem anderen Kontext einzusetzen. Im einfachsten Fall enthält die Komponente die gesamte Funktionalität, die sie benötigt, selbst. Dann kann sie überall eingesetzt werden, da keinerlei Abhängigkeiten zu anderen Komponenten bestehen.

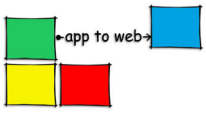
Das Zusammenfassen der angebotenen Funktionalität an einer Schnittstelle erleichtert die Entscheidung, ob die Komponente die Leistungen erbringen kann, die man von ihr erwartet. Sie bestimmt auch, wie man die Komponente zu benutzen hat. Durch die Spezifikation der benötigten Funktionalität wird die Suche nach existierenden Komponenten, die die Schnittstellenanforderungen erfüllen, bzw. eine Neuentwicklung stark vereinfacht, und somit wird es einfacher eine Komponente in einem anderen Kontext zu benutzen.

17.1.1 Bemerkungen

In der Literatur findet man z.B. in [Aea01] die Unterscheidung zwischen den Ausprägungen einer Komponente zur Laufzeit (Instanz) und zur Entwicklungszeit (Typ). Da wir die Komponenten aus existierendem Quelltext gewinnen, sind die Instanzen der Komponente schon bestimmt. Ziel ist es, aus dem Quelltext eine nachträgliche, möglichst exakte und aussagekräftige Beschreibung des Typs zu bekommen. Diese Beschreibung ist erforderlich, wenn man entscheiden will, ob man eine Komponente einsetzen kann.

Komponenten haben zum einen die Eigenschaften von Klassen, sie bieten Schnittstellen nach außen an und verbergen ihre Repräsentation. Sie haben aber auch Modulcharakter, weil sie Namensräume für mehrere Klassen aufspannen. Im Gegensatz zu Subsystemen, wie sie in der UML definiert sind [FS99], können Komponenten auch eigenes Verhalten besitzen, und dienen nicht nur als Behälter für Klassen. Ein Beispiel hierfür sind z.B. Funktionen zur Versionierung von Komponenten.

Da die Begriffe *requires-Schnittstelle* und *provides-Schnittstelle* ein wesentlicher Bestandteil des Modells sind, werden diese im Folgenden noch einmal ausführlich betrachtet. Beide bestimmen die Beziehungen, die zu anderen Komponenten auftreten können.



17.1.2 Requires-Schnittstelle

In der *requires-Schnittstelle* ist die Funktionalität spezifiziert, die eine Komponente nicht selbst bereitstellen kann, sondern von andere Systemteilen (Komponenten, Bibliotheken, ...) benötigt. Somit besteht die *requires-Schnittstelle* in erster Linie aus einer Menge von Systemteilen. Es macht aber durchaus Sinn, diese Menge zu verfeinern, da nicht alle Dienste einer Komponente oder Bibliothek benutzt werden, sondern nur ein Teil davon. Je exakter bekannt ist, was die Komponente von außerhalb benötigt, desto einfacher fällt eine Entscheidung, ob eine der benötigten Komponenten durch eine andere Komponente ersetzt werden kann.

Deshalb soll eine *requires-Schnittstelle* (zumindest für objektorientierte Systeme) genau aus folgenden Dingen bestehen:

- Allen Typen, die innerhalb der Komponente verwendet werden, und nicht in der Komponente selbst definiert werden. Hierzu zählen insbesondere auch Ausnahmen und Ereignisse.
- Allen Methoden, die innerhalb der Komponente aufgerufen werden, und nicht innerhalb der Komponente selbst definiert werden.
- Allen Attributen, die innerhalb der Komponente verwendet werden, aber nicht innerhalb der Komponente definiert werden.

Dieses Wissen steckt implizit über viele Stellen verteilt im Quelltext der Komponente. Es ist jedoch wünschenswert, diese Information zusammenhängend an einer Stelle zu sammeln.

17.1.3 Provides-Schnittstelle

Im Gegensatz zur *requires-Schnittstelle* bestimmt die *provides-Schnittstelle* die Funktionalität, die eine Komponente anderen Komponenten oder Programmteilen zur Verfügung stellt. Auch diese Schnittstelle sollte so exakt wie möglich spezifiziert werden, um eine Benutzung der Komponente möglichst einfach zu gestalten. Im Prinzip besteht diese Schnittstelle aus allen Entitäten der Komponente, die öffentlich sichtbar sind. Hierzu gehören:

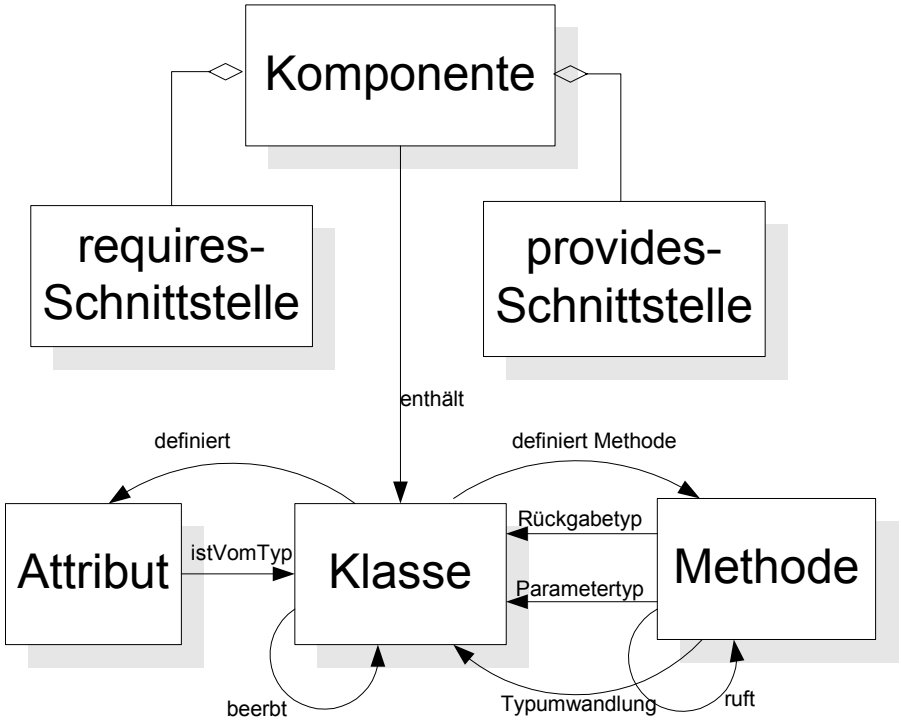
- Alle öffentlichen Klassen, auch Ausnahmen und Ereignisse
- Alle öffentlichen Methoden
- Alle öffentlichen Attribute

Diese Schnittstelle sollte mit Bedacht gewählt sein, da sie bestimmt, welche Zugriffsmöglichkeiten ein Klient auf eine Komponente hat. Methoden, Klassen und Attribute, die nicht aufgerufen werden sollen, müssen explizit aus dieser Schnittstelle entfernt werden. Diese Schnittstelle sollte auch nicht mehr geändert werden, da schon existierende Klienten die Komponente dann nicht mehr benutzen können.

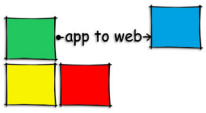
17.2 Strukturmodell

In Kombination mit dem Strukturmodell aus [ABGS01] ergibt sich das in Abbildung 39 dargestellte Modell zur Modellierung von Systemen. Die Entität Subsystem ist durch den Komponentenbegriff ersetzt worden, die Entitäten *requires-* und *provides-Schnittstelle* sind hinzugekommen. Mit Hilfe dieses Modells können dem Benutzer zunächst berechnete Komponentenkandidaten präsentiert werden. Weiterhin enthält das Modell ausreichend Information, um die Zerlegung in Kandidaten zu überprüfen und eventuell nötige Restrukturierungen auszuwählen. Insbesondere die explizite Modellierung der *requires-* und *provides-Schnittstelle* bietet eine aussagekräftige Beschreibung der Komponente.

Abbildung 39:Strukturmodell



Dieses Modell ist insofern sprachunabhängig, als mit seiner Hilfe beliebige objektorientierte Systeme dargestellt werden können. Diese Eigenschaft hat allerdings auch zur Folge, dass das Modell nicht alle Entitäten und Beziehungen einer bestimmten Sprache erfaßt, z.B. gibt es keine Möglichkeit für Java-Systeme Pakete darzustellen.



Insbesondere für die in Kapitel 4 beschriebenen Restrukturierungen ist der Detailgrad des Modells nicht ausreichend. Hier werden z.B. bei der Beschreibung von Vorbedingungen, die erfüllt sein müssen, damit eine Restrukturierung ausgeführt werden kann, weitere Informationen benötigt. Auch die Sprachunabhängigkeit muss gelegentlich aufgegeben werden, da manche Restrukturierungen, wie z.B. eine Klasse in eine Enterprise Java Bean umzuwandeln sprachspezifisch sind. Aus diesen Gründen orientiert sich die Beschreibung der Restrukturierungen an einem formaleren, vollständigeren Modell, das in [Kut02] beschrieben ist.

.

17.3 Beziehung zu anderen Komponentenmodellen

17.3.1 Beziehung zu Kobra

Innerhalb des Projektes App2Web wurde noch ein anderes Komponentenmodell namens Kobra [Aea01] entwickelt. Beiden Modellen ist gemein, dass Komponenten große, logische Einheiten darstellen, die einen eigenen Zustand und eigenes Verhalten besitzen können. Der Hauptunterschied liegt darin, dass in Kobra Komponenten vornehmlich zum Entwurfszeitpunkt betrachtet werden. Diese können dann zum Zeitpunkt einer Implementierung durch mehrere sogenannte physikalische Komponenten implementiert werden. In unserem Modell hingegen entstehen die Komponenten erst nachträglich. Jede gefundene und spezifizierte Komponente steht für eine Menge existierender Klassen im Quelltext. Die Komponenten unseres Modells stellen somit Bausteine dar, die eine Kobra Komponente entweder ganz oder teilweise realisieren könnten.

Ein Unterschied besteht in den möglichen Beziehungen zwischen den Komponenten. Bei Kobra existiert eine sogenannte Enthaltensein-Hierarchie. In ihr wird beschrieben, welche Komponente welche anderen Komponenten enthalten. Diese Hierarchie formt in Kobra einen Baum. Eine Komponente darf nur die Dienste einer anderen Komponente aufrufen, wenn diese auf gleicher Ebene oder tiefer als diese liegt. Die Ebene liegt um so tiefer, desto mehr Knoten zwischen ihr und der Wurzel der Hierarchie liegen. Diese Einschränkung existiert in dem hier beschriebenen Modell nicht. Komponenten können zwar ebenfalls in einer Enthaltensein-Hierachie angeordnet sein, Dienste können allerdings von Komponenten einer beliebigen Hierarchieebene angefordert werden.

17.3.2 Beziehung zu existierenden physikalischen Infrastruktursystemen

Infrastruktursysteme wie EJB, Corba, COM/DCOM legen einen Verdrahtungsstandard für die Kommunikation zwischen Komponenten fest. Zusätzlich bieten sie eine Menge von nützlichen Diensten zur Entwicklung moderner, verteilter Anwendungen. So bieten sie z.B. Möglichkeiten zur verteilten Kommunikation über Rechnergrenzen hinweg oder automatische Persistenzmechanismen.

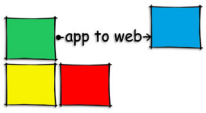
Damit die aus dem Quelltext gewonnenen Komponenten in der Praxis eingesetzt werden können, müssen diese an die Infrastruktursysteme angepasst werden. Obwohl unser Schwerpunkt nur auf der Transformation des Quelltexts liegt, um zu Komponenten gemäß unserer vorherigen Definition zu kommen, müssen diese in einem zweiten Transformationsschritt an eine Infrastruktur der Wahl angepasst werden. Auch diese Anpassungen können werkzeuggestützt erfolgen.

18 Evaluation der Komponentenkandidaten und Auswahl der Restrukturierungen

Die Verfahren zur Komponentifizierung liefern eine Partitionierung des Systems. Die Struktureinheit, anhand derer die zusammenhängenden Gruppen gebildet werden, ist die Klasse. Jede Gruppe enthält also eine gewisse Anzahl von Klassen, und da die Klassen ihrerseits wieder andere Strukturelemente wie z.B. Methoden enthalten können, implizit weitere Elemente.

Diese vorgeschlagenen Gruppen müssen zuerst einmal evaluiert und modifiziert werden, bevor der Quelltext so transformiert wird, dass der Zusammenhang der Klassen auch in der Struktur des Quelltextes zu finden ist. Diesen Schritt nennen wir *Evaluation der Klassengruppen*.

Als nächstes müssen die Schnittstellen der Komponentenkandidaten evaluiert werden. Die Elemente der *requires*- und *provides-Schnittstellen* müssen anhand des Quelltextes der Klassen ermittelt werden. Eventuell müssen diese Schnittstellen



modifiziert werden, damit beispielsweise die Benutzung der Komponente vereinfacht wird, oder aber überflüssige Abhängigkeiten zu anderen Komponenten, die Bestandteil der *requires-Schnittstelle* sind, entfernt werden können. Diese Aktivitäten werden unter dem Begriff *Evaluation der Schnittstellen* erfasst.

Wie in Kapitel 1 beschrieben, können die durch die Schritte 1 und 2 entstehenden Quelltextteile nun sehr gut wiederverwendet werden. Damit sie aber in existierenden Komponenteninfrastrukturen verwendet werden können, müssen sie an diese Infrastruktur angepasst werden. Die Schnittstellen müssen z.B. zu den innerhalb der Komponenteninfrastruktur geltenden Verdrahtungsstandards konform gemacht werden.

Zusammenfassend ergeben sich also folgende Aktivitäten:

1. Evaluation der Klassengruppen

2. Evaluation der Schnittstellen

3. Abbildung auf Komponenteninfrastruktur

Die Aktivitäten sollten idealerweise linear in dieser Reihenfolge ausgeführt werden, allerdings kann es in Schritt zwei erforderlich sein, dass man den ersten Schritt noch einmal ausführen muss, weil man erst jetzt feststellt, dass die gewählte Einteilung der Klassen in Komponentenkandidaten nur suboptimal war.

Ist der zweite Schritt erfolgt, dann kann ausgehend von diesem Ergebnis eine Abbildung auf eine Komponenteninfrastruktur erfolgen. Diese Abbildung kann für mehrere verschiedene Infrastrukturen erfolgen, zum Beispiel einmal auf Corba [OMG], und ein anderes Mal auf EJB [MH00].

18.1 Semi-automatische Programmtransformationen

Die oben genannten Arbeitschritte erfordern in der Regel eine Änderung oder eine Erweiterung des betreffenden Softwaresystems, was, betrachtet man Größe und Komplexität der meisten Systeme, eine große Herausforderung darstellt, da die Auswirkungen der Änderungen schwer überschaubar sind und manuelle unsystematische Änderungen meist schon wegen des erforderlichen Arbeitsaufwands ausscheiden. Desweiteren möchte man die Funktionalität des existierenden Softwaresystems aufrecht erhalten. Um die gleichbleibende Funktionalität sicherstellen zu können und den Aufwand vertretbar zu machen, müssen Transformationen von Softwaresystemen stets systematisch, also algorithmisch, durchgeführt werden. Solche Transformationsalgorithmen, die die Semantik eines Softwaresystems erhalten, nennt man Refactorings. Ein Refactoring zeichnet sich dadurch aus, dass es Vorbedingungen für seine Durchführung gibt, die garantieren, dass bei ihrer Erfüllung, das Softwaresystem nach den algorithmischen Änderungen, definierten Nachbedingungen genügt,

es also das gleiche Verhalten wie vorher zeigt. Refactorings können prinzipiell überall dort eingesetzt werden, wo das Softwaresystem, an dem gearbeitet wird, keinen harten Zeit- oder Speicherrestriktionen unterliegt, da die Refactorings nur das Verhalten erhalten, aber nicht garantieren können, dass sich nicht der Ressourcenverbrauch ändert.

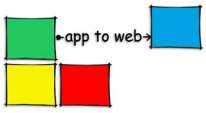
Die Größe und Komplexität der zu verändernden Softwaresysteme legt den Wunsch nach automatisierten Refactorings nahe, die als zusätzliche Funktionalität z.B. in Programmierumgebungen, integriert werden können. Eine manuelle Durchführung der Transformationsalgorithmen (Programmtransformation) ist aufgrund der Fehleranfälligkeit bei der Durchführung und dem Arbeitsaufwand in der Regel nicht anzuraten. Bei der Anwendung automatisierter Refactorings durch einen Benutzer spricht man von semi-automatischer Programmtransformation. Im Gegensatz hierzu versteht man unter einer voll automatischen Programmtransformation eine Programmtransformation bei der ohne menschliches Zutun entschieden wird, welche Refactorings auf welche Programmteile anzuwenden sind. Diese vollkommen automatisierte Vorgehensweise ist jedoch nicht zu empfehlen, da es sinnvoll ist, den Überblick über die Änderungen im Programm zu behalten, was bei vollautomatischen Transformationen nicht der Fall ist. Es existieren im Moment keine zuverlässigen Algorithmen, mit deren Hilfe man zweifelsfrei entscheiden kann, wann eine Programmtransformation sinnvoll erscheint und wann nicht. Trotz der fehlenden automatischen Erkennung von möglichen Anwendungsstellen, gibt es eine Reihe von Indizien, die auf die Notwendigkeit der Anwendung von Refactorings hindeuten. Zwei zur Durchführung von Restrukturierungen einsetzbare Werkzeuge werden in Kapitel 7 beschrieben.

Im weiteren Verlauf dieses Kapitels wird beschrieben, welche Gründe in Bezug auf die Transformation in Komponenten es für eine Restrukturierung des Quelltextes geben kann. Die jeweils passenden Restrukturierungen werden nur kurz beschrieben, eine detailliertere Betrachtung erfolgt in Kapitel 4.

18.2 Evaluation der Klassengruppen

Die Entscheidung, zwei Klassen A und B dem gleichen oder verschiedenen Komponentenkandidaten zuzuordnen, treffen die Algorithmen nur auf Basis der statisch aus dem Quelltext ermittelbaren Abhängigkeiten. Deshalb sollte ein menschlicher Benutzer den Sinngehalt der Partitionierung noch einmal überprüfen. Diese Überprüfung erfolgt zunächst auf Klassenebene, eventuell in einem zweiten Schritt sogar auf Methodenebene. Sie erfordert genaue Kenntnisse über das System. Falls diese nicht vorhanden sind, können sie mit den in [ABGS01] beschriebenen Methoden und Werkzeugen gewonnen werden.

Zum besseren Verständnis der Zusammenhänge innerhalb eines Komponentenkandidaten und seiner Beziehungen zu anderen Komponenten, wurde am FZI im Rahmen des Arbeitspaketes *Transformation der Komponentenkandidaten* das Werkzeug *Echi-*



dna entwickelt. *Echidna* ist ein plattformunabhängiges Visualisierungswerkzeug für Softwaresysteme, das Einheiten und Abhängigkeiten des Systems in Form eines Graphs veranschaulicht. Es ist insbesondere für großes Systeme geeignet, da der Abstraktionsgrad der Sicht auf das System frei konfigurierbar ist, und man ihn beliebig für Teilbereiche erhöhen oder erniedrigen kann. So kann man z.B. initial nur die Komponenten und ihre Beziehungen anzeigen lassen, um anschließend in eine Komponente „hereinzuzoomen“, und sich die darin enthaltenen Klassen anzeigen zu lassen.

Nachdem man ein gewisses Verständnis des Systems gewonnen hat, kann man die einzelnen Kandidaten überprüfen. Eine Modifikation der Kandidaten kann folgende Gründen haben:

- Eine Klasse A ist einem Komponentenkandidaten K1 zugeordnet, weil dieser z.B. die Funktionalität von A häufig benutzt. Konzeptionell gehört A aber zu einer anderen Komponente.
- Man stellt fest, dass der Komponentenkandidat in sinnvolle Teile zerlegt werden kann.
- Ein resultierender Komponentenkandidat ist zu feingranular und formt keine abgeschlossene, logische Einheit, und muss deshalb mit einem anderen Komponentenkandidaten verschmolzen werden.

Diese Modifikationen bleiben ohne Auswirkungen auf den Quelltext des Systems. Hat man die Modifikation allerdings abgeschlossen, dann sollte der Quelltext so angepasst werden, dass seine Strukturierung der Einteilung in Komponentenkandidaten nicht widerspricht. Besitzt die Sprache Strukurierungsmittel für Klassen, wie in Java Pakete, dann sollten diese so umgeformt werden, dass jede Komponente in einem eigenen Paket liegt. Dies ermöglicht eine Ausnutzung der Sichtbarkeitsregeln von Java, die es z.B. erlauben, dass Entitäten nur innerhalb eines Paketes sichtbar sind. Auf Komponenten übertragen bedeutet dies, dass es somit Klassen geben kann, die nur innerhalb der Komponente sichtbar sind. Gibt es keine speziellen Sprachmittel zur Strukturierung von Klassen, dann sollte die Strukturierung anhand der Dateien und Verzeichnisse vorgenommen werden. Jede Komponente sollte hierzu in einem eigenen Verzeichnis liegen. Die zugehörigen Restrukturierungen sind *Erzeugen / Löschen von Entitäten* und *Klasse verschieben*. Da die Strukturierungseinheiten oft auch Namensräume aufspannen, müssen bei Veränderungen auch die Stellen im Quelltext angepasst werden, an denen Klassen oder Strukturierungseinheiten benutzt und eingebunden werden. Analog zu den Veränderungen auf Klassenebene kann man auch Veränderungen an der Methodenzugehörigkeit vornehmen. Die hierzu nötige Restrukturierung ist *Methode verschieben*.

18.3 Evaluation der Schnittstellen

Wie in unserem Modell beschrieben, besitzt jede Komponente zwei verschiedene Schnittstellen. Die *provides-Schnittstelle* besteht aus allen in der Komponente enthaltenen Entitäten, die von außerhalb benutzt werden können. Die *requires-Schnittstelle* hingegen fasst alle Entitäten zusammen, die nicht Teil der Komponente sind, die sie aber benötigt um ihre Funktion zu erfüllen.

Die Gestalt dieser Schnittstellen bestimmt, wie einfach die Komponente in beliebigen Kontexten eingesetzt werden kann. Je kleiner die *requires-Schnittstelle* ist, d.h. je weniger Elemente sie besitzt, desto einfacher ist es, die Komponente wiederzuverwenden. Allerdings ist es keine Lösung, alle benutzten Teile in die Komponente aufzunehmen, weil diese eventuell auch von anderen Komponenten benutzt werden sollen und die Komponente sonst zu groß wird. Das Ziel besteht also in der sinnvollen Entfernung oder Abschwächung von unerwünschten Abhängigkeiten.

Die Form der *provides-Schnittstelle* bestimmt, wie einfach die Dienste und Elemente einer Komponente benutzt werden können. In diesem Fall kann aber eine kleinere Schnittstelle nicht mit besserer Benutzbarkeit gleichgesetzt werden. Die Schnittstelle muss sich immer am Sinn und Zweck der Komponente orientieren.

Somit ergeben sich für die Schnittstellen folgende Kriterien, gemäß denen diese optimiert werden sollen:

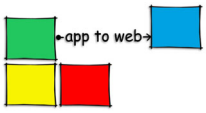
- *requires-Schnittstelle*: so klein wie möglich
- *provides-Schnittstelle*: so einfach wie möglich

Der initiale Zustand dieser Schnittstellen kann anhand des vorhandenen Quelltexts automatisch bestimmt werden. Ein hierzu verwendbarer Algorithmus wird in den nächsten beiden Abschnitten beschrieben. Der Rest des Kapitels widmet sich möglichen Restrukturierungen des Quelltexts, mit denen die Schnittstellen kleiner, bzw. einfacher gemacht werden können.

18.3.1 Berechnung der *requires-Schnittstelle*

Gibt es eine Beziehung einer Entität der Komponente zu einer Entität einer anderen Komponente, dann müssen diese „fremden“ Elemente in die *requires-Schnittstelle* aufgenommen werden. Der Algorithmus wird zum besseren Verständnis in natürlicher Sprache beschrieben. Berechnung der *requires-Schnittstelle* R für eine Komponente K:

- Für alle Klassen B, die in einer Komponente K enthalten sind, und für alle Typabhängigkeiten von B zu einer Klasse C, die nicht in K liegt, füge die Klasse C zu R



hinzu. Typabhängigkeiten entstehen durch Deklarationen, Casts, Vererbungsbeziehungen, Objekterzeugungen, usw.

- Für alle Klassen B, die in der Komponente K enthalten sind, und alle Methodenzugriffe von Methoden der Klasse B auf eine Methode M einer Klasse C, die nicht in K liegt, füge die Methode M zu R hinzu.
- Für alle Klassen B, die in der Komponente K enthalten sind, und alle Variablenzugriffe auf eine Variable V aus einer Klasse C, die nicht in K liegt, füge V und den Typ von V zu R hinzu.

18.3.2 Berechnung der *provides-Schnittstelle*

Grundsätzlich gehören alle Entitäten einer Komponente zur *provides-Schnittstelle,* die öffentlich sichtbar sind. Ob Elemente öffentlich sind, kann in den meisten objektorientierten Programmiersprachen über bestimmte Schlüsselwörter festgelegt werden. Somit kann eine Berechnung der *provides-Schnittstelle* sehr einfach erfolgen, indem:

- alle öffentlichen Methoden
- alle öffentlichen Attribute
- alle öffentlichen Klassentypen

aufgesammelt und zur *provides-Schnittstelle* hinzugefügt werden. Durch diese Berechnung wird herausgefunden, was ein Klient (z.B. eine andere Komponente) potenziell benutzen darf.

Eine weitere Möglichkeit, die *provides-Schnittstelle* zu berechnen, besteht darin, nur die Entitäten aufzunehmen, die auch tatsächlich von außerhalb der Komponente angesprochen werden. Für Sprachen, die keine Modifizierer zu Regelung der Sichtbarkeit besitzen, ist dies die einzige Möglichkeit der Berechnung. Man muss also den gesamten Quelltext des Altsystems nach Stellen durchsuchen, an denen Methoden, Attribute oder Klassentypen der Komponente referenziert werden.

18.3.3 Umstrukturierung der *provides-Schnittstelle*

Nachdem anhand des Quelltextes festgestellt wurde, welche Form die *provides-Schnittstelle* der Komponente hat, kann diese Schnittstelle überprüft und modifiziert werden. Die folgende Aufstellung von Verbesserungsmöglichkeiten erhebt keinen Anspruch auf Vollständigkeit, sie stellt nur eine Menge von möglichen, denkbaren Dingen vor, die man ändern kann, um die Schnittstelle gemäß der obigen Kriterien zu verbessern.

18.3.3.1 Sichtbarkeit der Elemente

Indem man die Sichtbarkeit von Strukturelementen abändert, kann man wesentlichen Einfluss auf die Gestalt der *provides-Schnittstelle* nehmen. In erster Linie sollte man überprüfen, ob manche der als öffentlich deklarierten Methoden, nicht besser verborgen werden sollten, da sie z.B. nur interne Hilfsroutinen implementieren, die nur von Methoden innerhalb der Klasse aufgerufen werden. Andererseits kann man auch noch einmal überprüfen, ob als privat deklarierte Methoden nicht doch so allgemein geschrieben worden sind, dass man sie anderen Klassen zur Verwendung anbieten kann.

In manchen Sprachen hat man zusätzlich zu Klassen höhere Strukturierungsmechanismen, z.B. gibt es in Java Pakete, die einen Namensraum für Klassen aufspannen. Somit kann man auch auf Klassenebene die Entscheidung treffen, ob eine Klasse außerhalb oder nur innerhalb eines Paketes sichtbar sein soll. Oft kann es besser sein, eine Verwendung von Klassen nicht zuzulassen, da diese nur von anderen Klassen innerhalb des Paketes benutzt werden sollen.

Bei nachträglichen Einschränkungen der Sichtbarkeitsregeln ist darauf zu achten, dass das System nach der Änderung noch übersetzbar ist. Wird eine Entität, deren Sichtbarkeit von öffentlich auf verborgen abgeändert werden soll, von anderen Entitäten außerhalb der Klasse oder des Paketes schon benutzt, dann darf die Sichtbarkeit nicht eingeschränkt werden.

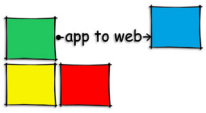
Die hierzu verwendbaren Restrukturierungen sind *Strukturelement verbergen* bzw. *Strukturelement veröffentlichen*.

18.3.3.2 Vorhandene Schnittstelle unzureichend

Eine radikale Änderung der *provides-Schnittstelle* kann erzielt werden, indem man alle bisherigen öffentlichen Elemente ganz oder teilweise verbirgt, und eine neue Zugriffsschicht für die Komponente erstellt. Dies kann z.B. in Form einer Fassadenklasse erfolgen [GHJV94]. Eine Fassadenklasse enthält alle zur Benutzung der Komponente nötigen Methoden und bildet eine einfachere Benutzungsschnittstelle, als wenn die einzelnen Klassen der Komponente direkt angesprochen werden müssten. Die Fassadenklasse selbst enthält keine wesentlichen Berechnungen. Ihre Aufgabe besteht in der Weiterleitung ankommender Aufrufe zu den vorhandenen Klassen der Komponente.

Ein solches Einziehen einer neuen Schnittstelle wird oft zur Einbindung von Codefragmenten benützt, die in einer anderen Sprache geschrieben sind. Beispielsweise können auf diese Weise Cobol-Programme von Visual Basic aus benutzt werden. [ACCL01]

Denkbar ist auch eine Art mehrstufige Fassade, die aus mehreren Klassen besteht. Diese Klassen können sich zum einen ergänzen, z.B. kann die Hauptfassade bei Bedarf eine weitere Fassadenklasse zurückliefern, die einen eigenen Teil der Funktio-



nalität kapselt. Dies kann verhindern, dass die Anzahl der Methoden einer Klasse zu groß wird [Rie96]. Anderseits können die verschiedenen Fassadenklassen jede für sich eine abgeschlossene Sicht auf die Komponente anbieten, die jeweils nur im Detailgrad variiert. Beispielsweise könnte eine Komponente zur Berechnung von Steuern eine Schnittstelle für Privathaushalte und eine weitere für Firmen enthalten.

Ein weiterer Nutzen, den man aus diesen neuen Schnittstellen ziehen kann, ist die Möglichkeit Methoden mit aussagekräftigeren Namen zu versehen, und zusätzlich die alte Schnittstelle zu erhalten. Das Erhalten der alten Schnittstelle ist wichtig, wenn die Komponente schon von anderen Systemteilen benutzt wird. Restrukturierung: *neue provides-Schnittstelle einziehen.*

18.3.3.3*Provides-Schnittstelle* enthält viele Typen

Bei der Gestaltung der Schnittstelle kann ein Ziel sein, dass der Nutzer möglichst wenige in der Komponente definierte Klassen kennen soll [MH00]. Werden diese Typen als Parametertyp oder Rückgabetyp verwendet, so kann es von Vorteil sein, wenn man versucht sie in ihre Basistypen zu zerlegen. Man gibt dann zwar den Vorteil der Kapselung und Abstraktion auf, allerdings gibt es z.B. bei Webanwendungen die Entwurfsrichtlinie, dass Anwendungen, die auf Client-Rechnern laufen, möglichst einfach sein sollen, und deshalb nur mit Basistypen arbeiten sollen. In diesem Fall ist eine solche Restrukturierung durchaus sinnvoll, obwohl sie klassischen OO-Entwurfsrichtlinien widerspricht. Die hierzu verwendbare Restrukturierung ist *Parametertypen vereinfachen.*

18.3.3.4Zustand aus Komponente verlagern

Insbesondere im Kontext von Webanwendungen kann es gewünscht sein, dass eine Server-Komponente den Zustand ihrer Klienten nicht mitverwalten muss, z.B. aus Performancegründen. Wenn es sich dabei um einen Zustand handelt, dessen Berechnung nicht sehr aufwändig ist, kann versucht werden, die Zustandsverwaltung auf den Klienten zu verlagern. Beispielsweise kann eine Komponente zur Darstellung von Kartenausschnitten die Speicherung eines „Zoom“-Faktors dem Klienten überlassen.

Deshalb kann es nützlich sein, Attribute und somit den Zustand der Komponenten, in die entsprechenden Methodenparameter der Methoden zu verlagern, die ein Klient aufrufen kann, obwohl dies nicht den klassischen OO-Entwurfsrichtlinien entspricht. Insbesondere wenn eine Komponente einen Web-Service [W3C02b] realisieren soll, ist es wünschenswert, wenn die Zustandsverwaltung auf der Klientenseite erfolgt. Die zugehörige Restrukturierung ist *Zustand in Methodenaufruf verschieben*.

18.3.3.5Standardisierte Schnittstellen zwischen Komponenten

Soll ein System in bestimmte Komponenten und Module unterteilt werden, ist das Vorhandensein einer eindeutigen, standardisiert ansprechbaren Schnittstelle wünschenswert. Durch Bereitstellung einer solchen Schnittstelle wird auch eine bessere Kapselung des Systems nach außen ermöglicht. Aufrufende Methoden (eventuell von remote-Systemen) brauchen nur diese einheitliche Schnittstelle zu kennen. Die internen Klassen und Methodenaufrufe bleiben dem Benutzer verborgen. Diese Schnittstelle kann durch eine *dispatcher* genannte Methode realisiert werden. Diese Methode soll in jede Klasse eingefügt werden und einen standardisierten Aufruf aller Methoden der Klasse ermöglichen. Die zugehörige Restrukturierung ist *Dispatch-Methoden einfügen.*

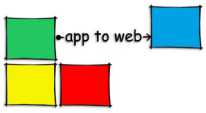
Möchte man tatsächlich nur eine einzige Schnittstelle für eine Menge von Klassen, so kann man zusätzlich zu den Klassen-eigenen *dispatcher-Methoden*, eine globale d*is-patcher-Methode* einführen, welche die jeweiligen Aufrufe auf die klassen-eigenen dispatcher-Methoden verteilt. Diese globale Methode ist dann zuständig dafür, die gesuchte Klasse aus den übergebenen Parametern zu ermitteln und dann die lokale dispatcher Methode mit den gleichen Parametern aufzurufen. Es wäre auch denkbar, eine globale dispatcher Methode einzufügen, welche alle anstehenden Aufgaben übernimmt. Dies könnte sich jedoch (vor allem bei verteilter Software) negativ auf das Laufzeitverhalten der Anwendung auswirken. Außerdem müssten bei geringen lokalen Änderungen einer beliebigen Klasse Änderungen in dieser hochkomplexen globalen Methode vorgenommen werden. Mit der vorgeschlagenen Variante, dass zusätzlich zu den lokalen dispatcher Methoden eine Verteilermethode implementiert wird, betreffen jegliche Änderungen nur die Klasse selbst. Man erhält durch die hier gewählte Variante also besseres Laufzeitverhalten und besser wartbaren Code. Restrukturierung: *Dispatch-Methoden einfügen*.

18.3.4 Restrukturierungen der requires-Schnittstelle

18.3.4.1Kopplung zwischen Komponenten

Oft ist es nicht möglich die Kopplung zwischen zwei Komponenten vollständig aufzulösen. Der einzige Ausweg besteht dann darin, die Kopplung, und somit die Abhängigkeiten abzuschwächen. Um Abhängigkeiten abzuschwächen gibt es verschiedene Möglichkeiten, drei von Ihnen werden nun vorgestellt.

Damit eine Komponente A, die Dienste einer Komponente B in Anspruch nimmt, nicht direkt von dieser Komponente abhängt, kann die *provides-Schnittstelle* von B in eine Menge von Schnittstellenklassen extrahiert werden. Somit hängt A von einer Schnittstellenkomponente ab, die von der Komponente B implementiert wird. Die Kopplung ist abgeschwächt, weil die Komponente B jetzt einfacher ausgetauscht werden kann. Die hierzu nötig Restrukturierung ist *Schnittstelle extrahieren.*



Eine Möglichkeit, einen schwach gekoppelten Benachrichtigungsmechanismus einzurichten, besteht im Einbau eines Observer/Observable Paares [GHJV94]. Eine Schnittstelle legt fest, wie sich Klienten bei einem Observer registrieren können, und wie die Methode zur Benachrichtigung bei Änderung des Zustandes heisst. Die Kopplung wird einseitig schwächer, weil die beobachtete Komponente ihre Beobachter nicht mehr direkt kennt, sondern nur prinzipiell weiß, dass es Beobachter geben kann. Die verwendete Restrukturierung heisst: *Observer einbauen*.

Vererbungsbeziehungen zwischen zwei Komponenten sind ein umstrittenes Thema. In J2EE [MH00] ist es z.B. nicht vorgesehen, Vererbung zwischen zwei Komponenten zu benutzen. Da in unserem Fall jedoch Komponenten aus existierendem objektorientiertem Quelltext gewonnen werden, ist es sehr wahrscheinlich, dass zwischen den einzelnen Komponenten Vererbungsbeziehungen erforderlich sind.

Soll die Anzahl dieser Beziehungen, die über Komponentengrenzen hinweg verlaufen, verringert werden, bestehen drei Möglichkeiten. Entweder modifiziert man die Zerlegung in Komponentenkandidaten, und versucht die Komponentengrenzen so zu ziehen, dass es möglichst wenige Vererbungsbeziehungen zwischen den Kandidaten gibt. Oder man verändert die Vererbungsbeziehungen entsprechend, indem man den Quelltext umwandelt. Die hierfür einzusetzende Restrukturierung ist *Oberklasse verschieben*, mit der unter bestimmten Voraussetzungen die Oberklasse verändert werden darf. Eine dritte Möglichkeit besteht darin, die Vererbung durch Delegation zu ersetzen. So kann über die Komponentengrenzen hinweg über Methodenaufrufe kommuniziert werden, und es ist einfacher, die Komponenten auf verschiedene Rechner zu verteilen. Restrukturierung: *Vererbung durch Delegation ersetzen*.

18.3.4.2Unerwünschte Abhängigkeiten zu Systemteilen

Je mehr Abhängigkeiten zu anderen Komponenten bestehen, desto schwieriger wird eine Wiederverwendung einer Komponente in einem anderen Kontext. Deswegen sollte immer überprüft werden, ob manche Abhängigkeiten nicht unnötig sind, und deshalb entfernt werden können. Beispielsweise sollte eine Komponente der Geschäftslogik keine direkten Aufrufe an GUI-Komponenten ausführen. Diese Form von Aufrufen wiederspricht einer Schichtenarchitektur, und ist immer wieder in existierenden Systemen anzutreffen, z.B. wenn eine Geschäftslogik-Komponente selbst einen Dialog zur Interaktion mit einem Benutzer öffnet. Zugehörige Restrukturierung: *GUI-Abhängigkeiten entfernen*

18.3.4.3Benötigte Komponenten sind schwer austauschbar

Wenn eine Komponente A eine oder mehrere Komponenten benutzt, dann werden diese Nutzungsstellen typischerweise weit im Quelltext verstreut sein. Diese Tatsache erschwert ein Ersetzen einer oder mehrerer Komponenten durch andere Komponenten, die nicht exakt die Schnittstelle der aktuell benutzten Komponenten besitzen. Ein Austausch kann vereinfacht werden, wenn für die ganze oder einen Teil der *requires-Schnittstelle* ein Requires-Proxy eingesetzt wird. Dieser verbirgt, welche Komponenten die einzelnen Funktionen der *requires-Schnittstelle* erbringen. Somit

kann man sich bei einem Austausch einer Komponente auf den Proxy konzentrieren, und hier die Verdrahtung entsprechend abändern.

Der Vorteil besteht darin, dass nur lokale Änderungen an einer Stelle im Quelltext vonnöten sind, und nicht der gesamte Quelltext von Änderungen betroffen ist. Da so eine zusätzliche Abstraktionsschicht eingezogen wird, kann es aufgrund der nötigen Delegation zu Geschwindigkeitseinbußen kommen. Ein Einsatz lohnt sich nur, wenn abzusehen ist, dass die requires-Komponenten sich ändern werden. Restrukturierung: *requires-Schnittstelle explizit machen*

18.4 Abbildung auf Komponentenstandards

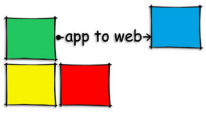
Sollen aus bestehenden Systemen gewonnene Komponenten in vorhandenen Komponentenstandards eingesetzt werden, dann müssen ihre Schnittstellen zu diesen Standards konform gemacht werden. Diese Abbildung auf Standards ist wünschenswert, weil diese zusätzliche Dienste wie Nebenläufigkeit, Persistenz etc. zur Verfügung stellen.

Für Java gibt es unter anderem die Möglichkeit Komponenten in einer Java Bean [Szy98] zusammenzufassen. Die wesentlichen Vorteile eines Einsatzes von Java Beans sind:

- Ereignisse: Java-Beans verwenden eine Art des Observer-Musters, um Ereignisse in Form von Objekten zu verschicken
- Eigenschaften: Beans veröffentlichen ihre Eigenschaften mit Hilfe von Paaren von `get()` und `set()` Methoden. Diese können zur Laufzeit verändert werden, Ereignisse auslösen und das Verhalten der Bean verändern
- Introspektion: Es besteht die Möglichkeit zur Laufzeit herauszufinden, welche Eigenschaften, Ereignisse und Methoden eine bestimmte Bean unterstützt.
- Persistenz: Beans und ihr Zustand können jederzeit gespeichert, und bei Bedarf zu späterer Zeit wieder geladen werden.

Die Restrukturierung *Klasse in Bean umwandeln* erleichtert das Umwandeln einer Klasse in eine Bean.

Für größere Anwendungen, insbesondere auch Webanwendungen ist der Enterprise Java Beans Standard jedoch besser geeignet. Er erleichtert unter anderem die Kommunikations über Rechnergrenzen hinweg und ermöglicht Nebenläufigkeit. Insbesondere die *provides-Schnittstelle* einer Komponente ist ein Kandidat für eine Session Bean, damit auch Klienten auf anderen Rechnern die Funktionalität der Komponente benutzen können. Eine Vorstufe zur Umwandlung kann die Restrukturierung *neue provides-Schnittstelle einziehen* sein, um die Komponente mit einer geeigneten Schnittstelle zu versehen. Anschliessend kann *Klasse in Session Bean umwandeln*



ausgeführt werden, um die Klasse in eine Session Bean gemäß dem EJB-Standard zu überführen.

19 Restrukturierungen

Dieses Kapitel enthält eine Reihe von Restrukturierungen, die bei der Umwandlung von Komponentenkandidaten in Komponenten nützlich sind. Die Restrukturierungen selbst sind nicht alle neu, sondern teilweise schon ein [Fow99] beschrieben. Trotzdem werden auch die schon bekannten hier noch einmal aufgeführt, und zwar immer unter dem Gesichtspunkt, dass sie zur Transformation von Komponentenkandidaten in Komponenten eingesetzt werden sollen.

Teilweise werden die nachfolgenden Restrukturierungen von Werkzeugen unterstützt und können voll- oder halbautomatisch ausgeführt werden. Manche Restrukturierungen erfordern jedoch Handarbeit. Im Prinzip können die Restrukturierungen unabhängig von einer objektorientierten Sprache beschrieben werden. Manche der beschriebenen Restrukturierungen machen jedoch nicht in allen Sprachen Sinn, z.B. *Klasse Verschieben*, da es in C++ kein Konzept der Namensräume für Klassen wie in Java (Pakete) gibt.

19.1 Erzeugen / Löschen von Entitäten

Wichtige Grundrestrukturierungen sind das Erzeugen und Löschen von Entitäten, wie z.B Attribute neu erzeugen, nicht mehr benötigte Pakete löschen etc. Beim Erzeugen muss darauf geachtet werden, dass es zu keinen Namenskonflikten kommt. Wird eine neue Entität mit dem Namen A neu erzeugt, muss sichergestellt werden, dass im betroffenen Namensraum nicht schon eine Entität mit identischem Namen vorhanden ist. Beim Löschen von Entitäten ist darauf zu achten, dass die Entität nicht mehr benutzt wird. Für Methoden bedeutet dies z.B., dass es keine Aufrufe an diese Methode im gesamten System gibt. Dies setzt voraus, dass man den gesamten Quelltext zur Verfügung hat, der die Methode jemals potenziell benutzen kann.

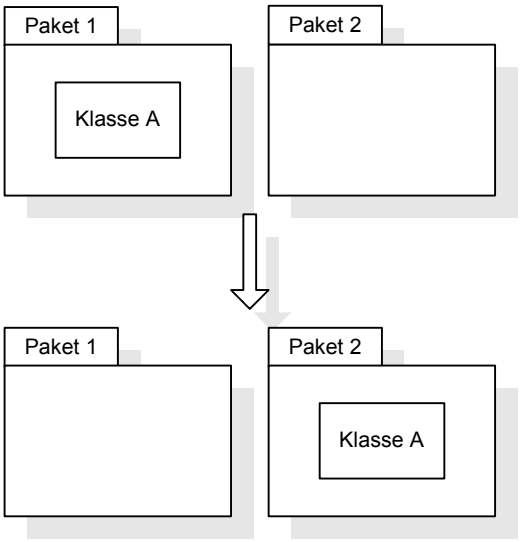
19.2 Klasse verschieben

Klassen zu verschieben macht nur in Sprachen Sinn, die auch Strukturierungsmechanismen für Klassen zur Verfügung stellen. In Java existieren hierfür sogenannte Pakete, die einen Namensraum für Klassen aufspannen.

Motivation

Klassen zu verschieben ist immer dann nötig, wenn eine Klasse nicht in das Paket passt, in das sie vom Programmierer eingeordnet wurde. Insbesondere bei der Transformation in Komponenten gilt, dass sich zwei Komponenten nicht einen Namensraum teilen sollen, sondern jede ihren eigenen besitzen soll.

Abbildung 40:Restrukturierung *Klasse verschieben*

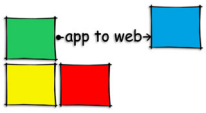


Vorbedingung

Eine Klasse kann nur verschoben werden, wenn es im Zielpaket noch keine Klasse mit identischem Namen gibt. Zusätzlich muss gewährleistet werden, dass noch immer alle existierenden Zugriffe auf die Klasse anschließend möglich sind, da z.B. in Java Methoden oder Attribute nur innerhalb eines Paketes sichtbar sein können.

Ausführung

Zuerst können die Klassen auf Dateiebene verschoben werden, da gemäß Konvention jedes Paket in einem eigenen Verzeichnis liegt. Anschließend muss die Paketdeklaration, die angibt zu welchem Paket die Klasse gehört, abgeändert werden und es muss in allen anderen Klassen, die diese Klasse benutzen, dafür gesorgt werden, dass das neue Paket der Klasse in den zur Verfügung stehenden Namensraum eingebunden wird. An allen Stellen im Quelltext, an denen die Klasse voll mit **`<Paket-Name>.<Klassenname>`** referenziert wird, muss der Paketname entsprechend angepasst werden.



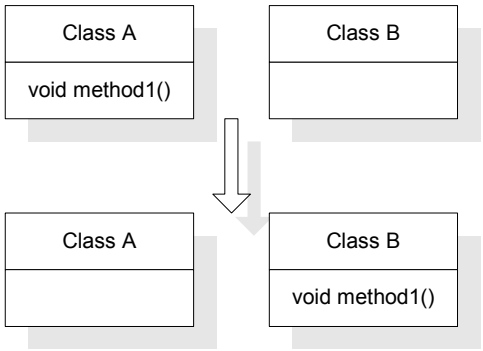
19.3 Methode verschieben

Motivation

Eine Methode kann aus mehreren Gründen von einer Klasse in eine andere Klasse verschoben werden. Man kann versuchen eine Klasse zu vereinfachen, oder die Kopplung zwischen zwei Klassen zu verringern [Fow99] [Kut02]. Auf Komponentenebene kann ein Verschieben von Methoden sowohl innerhalb einer Komponente oder zwischen zwei Komponenten sinnvoll sein.

Diese Restrukturierung ist in zwei Versionen sinnvoll, wobei die Methode in beiden Fällen in die angegebene Zielklasse verschoben wird. Allerdings wird in der ersten Version in der ursprünglichen Klasse ein Skelett der Methode beibehalten, welche die bisherigen Zugriffe an die neue Zielklasse (bzw. eine Ausprägung der Zielklasse) delegiert. Ein Vorteil dieser Methode ist, dass durch Übergabe einer Selbstreferenz das ursprüngliche Objekt bei Bedarf noch benutzt werden kann. In der zweiten Version ist dies nicht möglich, da hierbei auch alle Benutzungsstellen angepasst werden, so dass der Zugriff nur noch auf die Zielklasse erfolgt.

Abbildung 41:Restrukturierung *Methode verschieben*



Vorbedingungen

In der Zielklasse darf noch keine Methode mit dem Namen der zu verschiebenden Methode existieren. Ist die Zielklasse eine Ober- oder Unterklasse, müssen speziell auf diesen Fall zugeschnittene Restrukturierungen angewandt werden, die z.B. in [Fow99] oder [Kut02] beschrieben sind. Entscheidet man sich für die delegierende Variante, muss gewährleistet sein, dass die ursprüngliche Klasse auf die Zielklasse Zugriff hat.

Wird die Variante gewählt, in der die Methode ganz aus der ursprünglichen Klasse entfernt wird, dann muss sicher sein, dass alle bisherigen Nutzer dieser Methode auch weiterhin auf die Methode zugreifen können. Zudem darf die Methode nicht polymorph verwendet worden sein, ansonsten würde die Polymorphie aufgebrochen werden. Die Methode darf auch keine Methode einer abstrakten Schnittstelle oder einer abstrakten Oberklasse implementieren.

Ausführung In der delegierenden Variante muss zuerst in der Zielklasse die neue Methode mit einem zusätzlichen Parameter **`ursprung`** eingebaut werden. Dieser zusätzliche Parameter enthält eine Referenz auf eine Ausprägung der ursprünglichen Klasse. Die Zugriffe auf Elemente der eigenen Klasse müssen in Zugriffe auf **`ursprung`** umgewandelt werden. In der urpsrünglichen Klasse wird der Methodenrumpf entfernt, und durch einen delegierenden Aufruf mit Übergabe einer Selbstreferenz ersetzt.

Bei der zweiten Variante wird die Methode komplett in die Zielklasse kopiert. Die Selbstzugriffe müssen angepasst werden, und alle bisherigen Nutzungsstellen müssen auf die Zielklasse umgebogen werden.

19.4 Strukturelement verbergen

Motivation Um den Zugriff auf Strukturelemente einzuschränken, bieten die gängigen objektorientierten Programmiersprachen Möglichkeiten, die Sichtbarkeit der Entitäten anzugeben. Damit kann verhindert werden, dass unerwünschte Zugriffe auf Elemente erfolgen, z.B. auf Methoden einer Komponente, die nicht von außen zugänglich sein sollen.

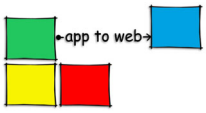
Vorbedingungen Es muss gewährleistet sein, dass alle existierenden Zugriffe auf die zu verbergenden Strukturelemente wie Methodenaufrufe und Attributzugriffe nach der Ausführung noch gültig sind.

Ausführung Der Zugriff wird durch sogenannte Modifizierer geregelt. Damit Elemente nur noch in einer Klasse sichtbar sind, muss der Modifizierer **`private`** verwendet werden. In Java existieren noch zusätzliche Stufen der Sichtbarkeit: **`package`** und **`protected`**. Damit kann erreicht werden, dass Elemente nur in einem Paket oder innerhalb einer Vererbungshierarchie sichtbar sind.

19.5 Strukturelement veröffentlichen

Motivation Stellt man fest, dass private Methoden oder Attribute doch von öffentlichem Nutzen sind, ist es sinnvoll diese nachträglich an der *provides-Schnittstelle* verfügbar zu machen.

Vorbedingungen In Java ist es möglich, dass Klassen nur innerhalb eines Paketes sichtbar sind. Ändert man diese Sichtbarkeit auf **`public`**, dann kann es an anderen Programmstellen zu Uneindeutigkeiten kommen, weil schon eine Klasse mit dem selben Namen existiert, die nicht vollqualifiziert angesprochen wird.



Ausführung

Die Ausführung besteht einfach darin, dass der entsprechende Modifizierer ausgetauscht wird, beispielsweise von **`private method1()`** nach **`public method1()`**.

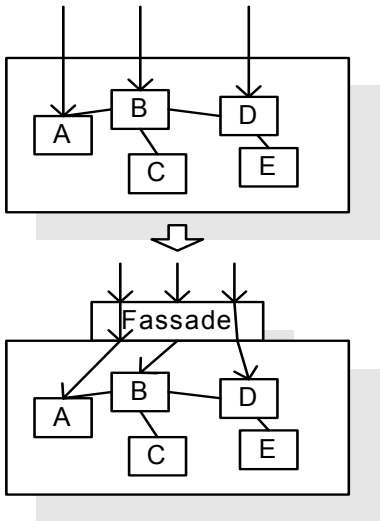
19.6 Neue provides-Schnittstelle einziehen

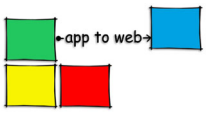
Motivation

Eine neue *provides-Schnittstelle* muss immer dann erstellt werden, wenn die vorhandene Schnittstelle den Anforderungen nicht genügt, weil sie z.B. zu komplex ist, oder die Namenskonventionen nicht zu denen des Klienten passen. Typischerweise werden eine oder mehrere Fassadenklassen hinzugefügt [GHJV94], die alle nach außen angebotenen Elemente zusammenfassen. Zuerst muss geklärt werden, wie viele Fassadenklassen benötigt werden. Mehrere Fassadenklassen sind aus folgenden Gründen sinnvoll:

- Die Komponente enthält zwei separate Abstraktionen, die nicht zu einer Schnittstelle zusammengefasst werden sollen. Dies kann auch die Übersichtlichkeit erhöhen, weil die Anzahl der Methoden in jeder Fassadenklasse überschauber bleibt.
- Es sollen verschiedene *provides-Schnittstellen* bereitgestellt werden. Jede bietet eine andere Sicht auf die Komponente, denkbar ist z.B dass sie sich im Funktionsumfang unterscheiden, weil nicht jeder Benutzer der Komponente den vollen Funktionsumfang benötigt, und er deshalb eine einfachere Schnittstelle benutzen kann.

Abbildung 42:Restrukturierung *neue provides-Schnittstelle einziehen*





Vorbedingungen So lange die bestehende *provides-Schnittstelle* bestehen bleibt, ist nur darauf zu achten dass das Hinzufügen der neuen Schnittstelle zu keinen Namenskonflikten führt.

Ausführung Zuerst müssen die neuen Schnittstellenklassen mit einem geeignetem Namen erzeugt werden. Dann muss festgelegt werden, welche Entitäten an der neuen Schnittstelle

zur Verfügung gestellt werden sollen. Für jede augewählte Methode muss eine neue Methode in einer oder mehreren Fassadenklassen hinzugefügt werden, die den Aufruf an die ursprüngliche Methode delegiert. Falls Attribute angeboten werden sollen, geschieht dies ebenfalls, indem entsprechende **`get()`** und **`set()`** Methoden in den Fassadenklassen ergänzt werden. Die Namen der Methoden müssen dabei nicht identisch sein, sondern können in der Fassadenklasse auch unterschiedlich sein.

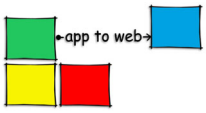
Problematisch wird die Sache, wenn die bisherigen Nutzungsschnittstellen dieser Komponente ebenfalls angepasst werden sollen. Dann muss zuerst wieder abgeklärt werden, dass noch alle benötigten Elemente an der Fassade verfügbar sind. Anschließend müssen die Methodenaufrufe und Attributzugriffe auf die neue Schnittstelle umgebogen werden.

19.7 Parametertypen vereinfachen

Motivation Die Typen, die zwischen Klienten und Komponente ausgetauscht werden, sollen z.B. im Rahmen von Webanwendungen möglichst einfach sein [MH00]. Deshalb kann versucht werden, komplexe Parameter, die nur zur Datenübergabe benutzt werden, in die in ihr enthaltenen einfachen Basistypen (String, integer, double, ...) zu zerlegen. Wird z.B. folgende Klasse als Parameter verwendet:

```
class A {
     ...
     String s;
     int b;
     ...
}
```

Dann kann eine Methode **`method1(A a)`** in eine Methode **`method1(String s, int b)`** umgewandelt werden. Diese Restrukturierung ist die Umkehroperation zur Restrukturierung *Parameterobjekt einführen* [Fow99]**.** Man könnte sogar noch weiter gehen, und fordern dass alle Parameter als Textstrom übergeben werden müssen, dies ist z.B. beim Methodenaufruf via SOAP [W3C02a] der Fall, bei der der komplette Aufruf als XML-Dokument übergeben wird.



Vorbedingungen Es muss sicher sein, dass die Klasse nur zum Datentransport verwendet wird, und der Klient keine Funktionsaufrufe benutzt. Die Methode sollte nicht polymorph verwendet werden, ansonsten muss man dafür sorgen, dass die entsprechenden Methoden der Ober- und Unterklassen ebenfalls verändert werden. Bei Methoden, die abstrakte Oberklassen oder Schnittstellen implementieren, ist Vorsicht geboten, weil diese Schnittstellen eventuell an vielen, nicht änderbaren Stellen benutzt werden. Zusätzlich darf die Komponente auf die Parameterobjekte nur lesend zugreifen.

Ausführung Zuerst müssen die in der Klasse enthaltenen Attribute bestimmt werden, deren Typ ein Basistyp ist. Sind auch Attribute mit zusammengesetztem Typ vorhanden, dann kann versucht werden diese rekursiv weiter zu zerlegen. Anschließend müssen alle Methoden der *provides-Schnittstelle* der Komponente transformiert werden, die die zu zerlegende Klasse als Typ in ihrer Signatur besitzen. Diese Signatur muss verändert werden, indem der alte Parameter entfernt wird, und durch die Menge der neuen Basisparameter ersetzt wird.

Damit die bisherigen Nutzungstellen der Komponente gleich bleiben können, und auch Methoden innerhalb der Komponente nicht verändert werden müssen, können die ursprünglichen Methoden unverändert in der Klasse belassen werden. Es wird eine zusätzliche Methode mit der veränderten Signatur eingesetzt, die das Transportobjekt am Anfang der Methode initialisiert und dann per Delegation den Aufruf an die ursprüngliche Methode übergibt.

19.8 Zustand in Methodenaufruf verschieben

Motivation Die von einer Komponente erbrachte Funktionalität hängt oft von ihrem inneren Zustand ab. Dieser Zustand kann von außen oder innen verändert werden. Gerade im Bereich der Webanwendungen kann es gewünscht sein, dass eine Komponente die Verwaltung des von außen veränderbaren Zustandes an den Klienten übergibt. Benutzt z.B. eine Methode `foo()` einer Komponente K die Attribute **a** vom Typ `A`, und **b** vom Typ `B`, die vom Klienten vor der Benutzung gesetzt werden müssen, dann kann die Signatur der Methode folgendermaßen abgeändert werden: `foo(A a, B b)`.

Vorbedingung Bei der hier vorgestellten Restrukturierung wird vorausgesetzt, dass die Komponente keine schreibenden Zugriffe auf die Zustandsattribute durchführt, die dem Aufrufer mitgeteilt werden müssen.

Ausführung Zuerst muss manuell identifiziert werden, welche Attribute von Klienten modifiziert werden können, und in welchen Methoden diese referenziert werden. Anschließend müssen alle diese Attribute mit identischem Namen in die Parameterliste der Methode aufgenommen werden. Verwendet man für die Parameter die gleichen Namen, dann müssen im Methodenrumpf keine weiteren Änderungen erfolgen.

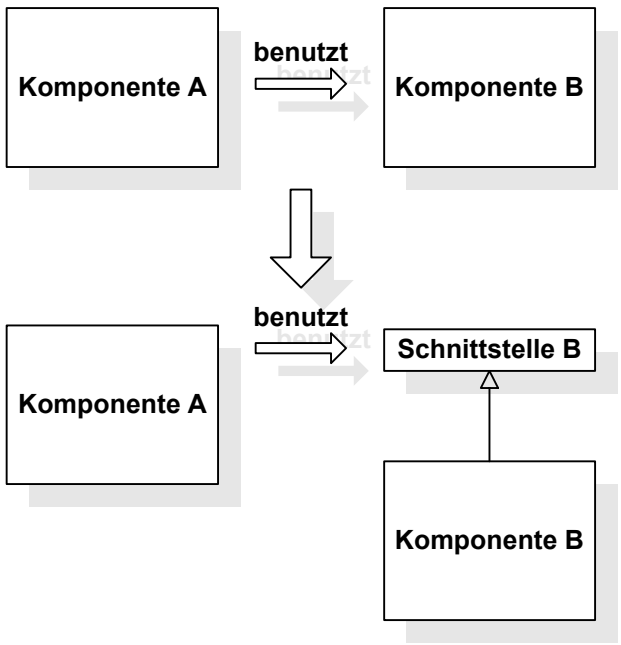
Anschließend können die Attribute aus der Klasse entfernt werden. Gibt es schon Nutzer dieser Funktionen, dann müssen diese Nutzungsstellen angepasst werden.

19.9 Schnittstelle extrahieren

Motivation

Um die Abhängigkeiten abzuschwächen, die von einer Komponente K1 zu einer Komponente K2 bestehen, kann man eine abstrakte Schnittstellenklasse einführen, die die Funktionen von K2 beschreibt, jedoch keine Bildung einer Ausprägung vorsieht. K1 arbeitet dann nur mit der abstrakten Schnittstelle, und muss bei Erzeugung mit einer Referenz auf eine Komponente ausgestattet werden, die die Schnittstelle implementiert [SGCH01].

Abbildung 43:Nachträgliches Einziehen einer Schnittstelle

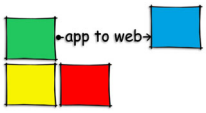


Vorbedingung:

Es wird vorausgesetzt, dass die Schnittstelle der Komponente in genau einer Fassadenklasse vorhanden ist.

Ausführung

Die Schnittstelle einer Komponente kann extrahiert werden, indem zuerst eine abstrakte Schnittstellenklasse oder wie in Java möglich ein `interface` erzeugt wird. In diese Schnittstelle müssen dann alle öffentlichen Methodenköpfe der Fassadenklasse, bis auf die Konstruktoren übernommen werden. Anschließend müssen Schnittstelle und Fassadenklasse miteinander in Beziehung gesetzt werden. Im Falle



einer abstrakten Schnittstelle muss eine Vererbungsbeziehung eingesetzt werden, falls ein `interface` verwendet werden kann, muss eine implementiert-Beziehung verwendet werden.

19.10 Dispatch-Methoden einfügen

Motivation

Eine Klasse soll mit einer einheitlichen Schnittstelle versehen werden, die folgenden Anforderungen genügt:

- Sie soll den Aufruf jeder Methode einer Klasse ermöglichen; mit Ausnahme der Konstruktoren und der Methode main.
- Sie soll nur einmal pro Klasse implementiert werden (keine Überladung). Das bedeutet, dass Möglichkeiten gefunden werden müssen, um Rückgabewert und Parameter dieser Methode zur standardisieren.
- Alle Aufrufe aller Methoden sollen möglichst einfach in die neuen Aufrufe der Methode dispatcher ersetzt werden können.

Vorbedingungen

Bei der hier beschriebenen Transformation wird vorausgesetzt, dass die Klasse nicht polymorph verwendet wird.

Ausführung

Der Kopf der Methode sei folgendermaßen aufgebaut:

```
public static ReturnValue

dispatcher(<class> o, String methodName, Vector param)
```

Dabei ist **`<class>`** durch den Namen der entsprechenden Klasse zu ersetzen, in welcher sich die jeweilige dispatcher-Methode befindet.

Der erste Parameter dient zur Übergabe des aufgerufenen Objektes. Ist die aufzurufende Methode statisch kann hier der Wert **`null`** übergeben werden. Der Name der Methode wird durch den zweiten Parameter als String übergeben. Um eine einheitliche, standardisierte Darstellung der Methode dispatcher zu gewährleisten, übergeben wir die Parameter zusammengefasst im Container **`java.util.Vector`**.

Intern prüfen wir dann mit verschiedenen **`if`**-Verzweigungen welche der Methoden in der jeweiligen Klasse angesprochen werden soll und führen den Aufruf aus. Dazu müssen wir zum einen die Parameter aus dem Vector extrahieren und zum anderen den Rückgabewert der Methode in den standardisierten Rückgabetyp **`ReturnVa-`**

lue umwandeln. Die Klasse ReturnValue (welche als Hilfsdatei in einem speziellen Package mitgeliefert wird) enthält Konstruktoren und Methoden, welche jeden in Java möglichen Typ in ein Objekt dieser Klasse einpacken kann. Außerdem stellt diese Klasse auch Methoden zur Extraktion zur Verfügung.

Beispiel

```
ReturnValue retV = new ReturnValue(42);

int i = retV.intVal();

ReturnValue retV2= new ReturnValue(myObject);// wobei MyObject von

MyClass m = (MyClass) retV2.objectVal();// Klasse MyClass ist
```

Bei den Parametern greifen wir auf die Klasse Par zurück, welche uns beliebig lange Parameterlisten in einen Vector einpacken und wieder auspacken kann. Einen Vector kann man nur mit Objekten füllen, so dass wir für die nativen Datentypen Javas jeweils einen eigenen Objektcontainer benötigen. Dies wird dadurch gelöst, dass wir für jeden einzelnen nativen Datentyp einen Objektcontainer zur Verfügung stellen, welcher neben einfachen Konstruktoren auch jeweils eine Methode enthält, die den Wert als nativen Typ zurückliefern. Benutzen wir die statische Methode m in Par, welche zum Einpacken dient, so werden die Datentypen automatisch konvertiert. Die Methode m akzeptiert ein bis zwei Parameter und liefert einen Vector zurück. Das gleiche gilt bei Benutzung der statischen Methode getIt, welche eine Rückkonvertierung vornimmt. Mit der Methode m werden an den zu übergebenden Vector nach und nach alle Parameter rekursiv angehängt. Dadurch wird gewährleistet, dass wir theoretisch unbeschränkte Möglichkeiten der Parameterzusammenstellung und Parameteranzahl haben.

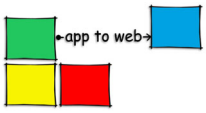
Beispiel:

```
Vector v = Par.m(23,Par.m(myObject,Par.m('s')));
```

Hier sieht man deutlich, wie Par.m arbeitet. Als erstes haben wir einen Integerwert. Dieser wird in einen ReturnValue umgewandelt und in den Vector eingepackt. Als nächstes und letztes Element wird in den an v zugewiesenen Vector ein anderer Vector gepackt. Dieser Vector enthält wieder zwei Elemente. Hier sind es ein Objekt (myObject) und einen Char, welcher zum Einpacken ebenfalls automatisch in ReturnValue umgewandelt wird. Die Reihe der Vectoren im Vector ließe sich beliebig oft fortsetzen. Beim Extrahieren der Daten durch die statische Methode Par.getIt(v) passiert nun folgendes: der übergebene Vector v wird derart zerlegt, dass das erste Element im standardisierten Container ReturnValue zurückgeliefert wird. Des weiteren schneidet man dieses Element quasi vom Vector v ab; der Vector v ist nach Anwendung der Methode um das entnommene Element kleiner geworden.

Abbildung 44:Ausführliches Beispiel

```
class Person {
…
public void setData(String firstName, String lastName,int age) {…}
public int getAge() {return this.age;}
```




```
public static void main() {
       Person p = new Person();
//            p.setData("Harry","Potter",11);-> alte Version des Aufrufs
              dispatcher(p,"setData",Par.m("Harry",Par.m("Poter",11))); // neu!
//            int i = p.getAge();
              int i = dispatcher(p, "getData", null).intVal();// neu !
              …
       }
       public static ReturnValue dispatcher(Person o, String methodName, Vector v) {
              ReturnValue retVal = null;
       if (methodName.equals("setData")) {
              o.setData((String) Par.getIt(v).objectVal(),
              (String) Par.getIt(v).objectVal(),Par.getIt(v).intVal());
       } else if (methodName.equals("getData")) {
              retVal = new ReturnValue(o.getData());
       }
       …
       return retVal;
       }
}
```

Dieses Beispiel verdeutlicht zugleich die allgemeine Arbeitsweise der Methode dispatcher. Wie man sieht hat die Methode dispatcher immer den Rückgabetyp ReturnValue, der mit den zur Verfügung stehenden Methoden leicht umgewandelt werden kann. Gibt es keinen Rückgabewert, wird null zurückgeliefert. Außerdem besitzt die Methode dispatcher genau drei Parameter. Das aufrufende Objekt, den Namen der Methode als String und schließlich unsere als Vector gepackte Parameterliste. Intern erfolgt die Abwicklung durch if-Verzweigungen wie in obigem Beispiel zu sehen ist. Der Aufruf wird mit den zur Verfügung stehenden Mitteln wieder nachgebaut.

Die Änderung der internen Aufrufe der Methoden mag nicht sonderlich sinnvoll erscheinen. Sie dient hier lediglich Demonstrationszwecken. Es zeigt, wie Anwendungen von außen über unsere Methode dispatcher auf bestimmte Methoden zugreifen können, im Vergleich zu der alten Variante auf diese bestimmten Methoden zuzugreifen.

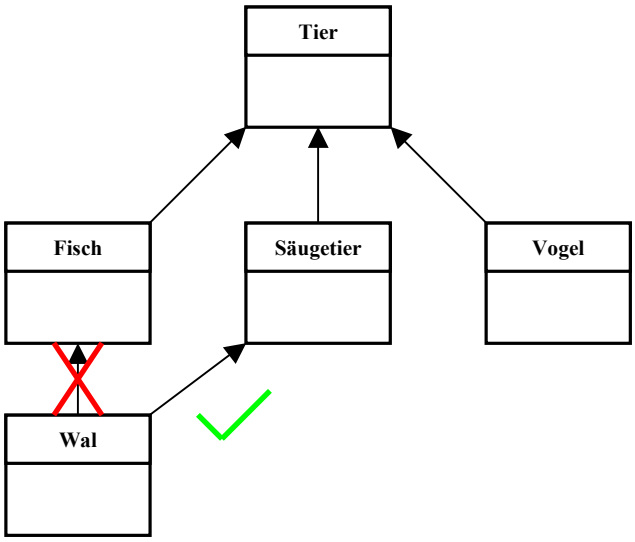
19.11 Oberklasse verschieben

Motivation

Mit Hilfe der Restrukturierung *Oberklasse verschieben* kann die Oberklasse einer Klasse verändert werden, ohne dass sich die Funktionalität des Systems verändert. Dies kann erforderlich werden, wenn sich z.B. die initiale Modellierung der Vererbungsbeziehungen als unpassend herausstellt, oder wenn die Anzahl von Vererbungsbeziehungen zwischen Komponenten verringert werden soll.

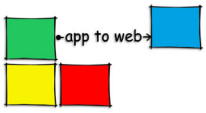
Zur Veranschaulichung der Restrukturierung dient das einfache Beispiel aus Abbildung 45. Die Klasse Wal wurde fälschlicherweise als Unterklasse der Klasse Fisch modelliert, der Fehler wird bemerkt und die Klasse Säugetier soll zur neuen Oberklasse von Wal gemacht werden.

Abbildung 45:Ursprüngliche und verbesserte Vererbungshierarchie



Vorbedingungen Damit die Transformation durchführbar ist, müssen folgende Vorbedingungen erfüllt sein:

- die Datentypen der Attribute der verschobenen Klasse müssen mit gleichnamigen Datentypen der/den neuen Superklasse(n) übereinstimmen
- Argumentlisten und Rückgabewerte von Methoden der verschobenen Klasse müssen identisch sein mit den Argumentlisten und Rückgabewerten von gleichnamigen Methoden in der/den neuen Superklasse(n)
- alle Attribute, die von der neuen Superklasse geerbt werden, sind in der verschobenen Klasse und ihren Subklassen identisch deklariert (siehe erste Bedingung)
- Zuweisungen von Instanzierungen der Klasse die verschoben wurde, müssen typsicher bleiben. Typsicherheit in diesem Zusammenhang, lässt sich gut an einem Beispiel verdeutlichen. Wir nehmen an, dass das Refactoring bereits ausgeführt wurde. Folgende Zuweisung ist dann nicht mehr typsicher, da der Typ einer Subklasse S immer nur an Stelle eines Typs einer Superklasse von S verwendet werden kann:




```
void fangen(Fisch f)
    { ... }
Wal w = new Wal();
fangen(w);
```

Folgendes Beispiel ist nach der Ausführung des Refactorings immer noch typsicher, da die Klasse Wal eine Subklasse der Klasse Tier bleibt:

```
void fangen(Tier t)
{ ... }
Wal w = new Wal();
fangen(w);
```

Ausführung

Nach dem Überprüfen der Vorbedingungen beginnt der eigentliche Transformationsalgorithmus, dessen Korrektheit von der Einhaltung der Vorbedingungen abhängig ist. Im Folgenden wird der Algorithmus auf abstraktem Niveau, d.h. ohne auf konkrete Suchoperationen auf dem Syntaxbaum einzugehen, erläutert. Dabei werden folgende Bezeichnungen verwendet:

Sei C die zu verschiebende Klasse, S1 die alte Superklasse und S2 die neue Superklasse.

- ermittele die Menge M der Methoden und Attribute, die von der **alten** Superklasse S1 (und ihren Superklassen) geerbt werden, aber **nicht** mehr von der **neuen** Superklasse S2 (und ihren Superklassen) geerbt werden.
- für jede Methode m in M, kopiere m in C
- ändere die Superklasse von C von S1 nach S2
- lösche in C alle Methoden und Attribute, die von S2 (und ihren Superklassen) identisch geerbt werden

19.12 Observer einbauen

Motivation

Der Einbau eines Observermusters [GHJV94] kann die Kopplung zwischen zwei Komponenten nicht völlig aufheben, sie aber einseitig abschwächen. Eine Komponente übernimmt die Rolle des Observables, das von anderen Komponenten, den Observern, beobachtet wird. Die Observer sind immer stark an ihr Observable gekoppelt, weil sie z.B. wissen müssen, wie auf seine Daten zugegriffen werden kann. Das Observable hingegen hat im Zusammenspiel mit den Observern jedoch nur die Aufgabe, sie bei Änderungen zu informieren. Ohne eine spezielle Schnittstelle muss die beobachtete Komponente alle ihre Beobachter kennen. Der Einsatz des Observermu-

sters hat zur Folge, dass die beobachtete Komponente nur weiß, dass es prinzipiell Beobachter gibt, aber nicht welche konkreten Komponenten dies sind.

Vorbedingungen

Die hier vorgestellte Restrukturierung setzt voraus, dass eine Schnittstelle für das Observermuster bereits vorhanden ist:

```
interface Observable {
     public void attach(Observer){};
     public void detach(Observer){};
     public void notify();
}

interface Observer {
     public void update {};
}
```

Mit den Methoden **`attach()`** und **`detach()`** können sich Observer bei einem Observable registrieren. Die **`notify`**-Methode benutzt das Observable, um die Observer zu benachrichtigen. Sie wird standardmäßig so aussehen, dass in allen Observern **`update()`** aufgerufen wird.

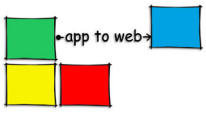
Zusätzlich wird vorausgesetzt, dass die *provides-Schnittstelle* in einer Fassadenklasse zusammengefasst ist.

Ausführung

Es muss dafür gesorgt werden, dass die obige Schnittstelle richtig implementiert wird. Dazu muss die Komponente, die die Rolle des Observables übernimmt, folgendermaßen verändert werden.

- Ein Container, der die Observer aufnehmen kann (z.B ein Vector) muss hinzugefügt werden. Dieser muss im Konstruktor initialisiert werden.
- Die Methoden **`detach()`** und **`attach()`** und der zugehörige Rumpf zum registrieren und deregistrieren der Observer müssen hinzugefügt werden.
- Die Methode **`notify()`** muss ergänzt werden. In ihrem Rumpf wird die **`update()`** Methode aller Observer aufgerufen.
- An den Stellen, an denen eine für die Observer wichtige Zustandsänderung erfolgt, muss ein Aufruf an **`notify()`** hinzugefügt werden. Die Frage, wann eine Zustandsänderung wichtig ist, kann nur von einem Benutzer beantwortet werden. Deshalb wird auch das Hinzufügen dieses Aufrufes manuell geschehen müssen.
- Die Komponente, bzw. ihre Schnittstellenklasse muss die obige Observable-Schnittstelle implementieren.

Auf Seiten des Observers muss zunächst im Quelltext vermerkt werden, dass die zugehörige Schnittstelle implementiert wird. Anschließend muss die **`update()`**



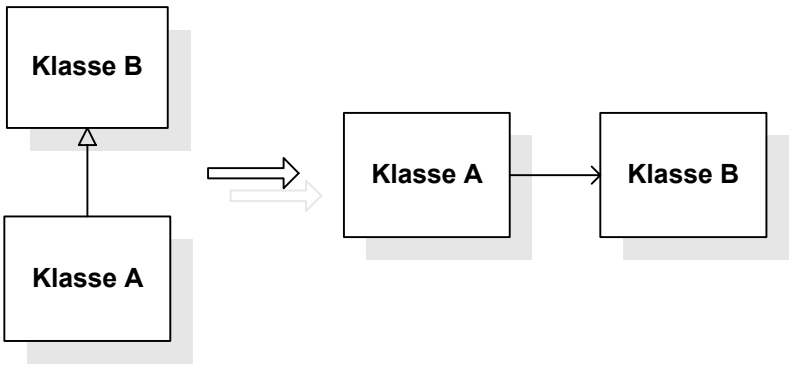
Methode hinzugefügt werden. Ihr Rumpf ist für jeden Observer typischerweise anders, hier wird der Zustand des Observables abgefragt und abhängig von ihm werden Aktionen im Observer ausgelöst.

19.13 Vererbung durch Delegation ersetzen

Motivation

Eine Vererbungsbeziehung durch Delegation zu ersetzen, hat das Abschwächen der Abhängigkeiten zwischen zwei Komponenten zum Ziel. Eine Komponente K1 soll von keiner Komponente K2 aus einer anderen Komponente erben, da nur Delegationsbeziehungen durch verteilte Kommunikation über Rechnergrenzen hinweg mittels Methodenaufrufen erfolgen können. Will man eine Komponente an diesen Standard anpassen, muss man dafür sorgen, dass die Bedingung nicht verletzt wird, und alle Vererbungsbeziehungen eliminiert werden. Ein zusätzlicher Faktor, der für den Einsatz von Delegation spricht, ist die Möglichkeit die Komponente, von der ursprünglich geerbt wurde, einfacher austauschen zu können.

Abbildung 46:Vererbungsbeziehung durch Delegation ersetzen



Vorbedingungen

Zur Delegation muss in der erbenden Klasse ein neues Attribut erzeugt werden, hier muss dafür gesorgt werden, dass es zu keinerlei Namenskonflikten kommt. Es muss sichergestellt werden, dass die ehemalige Unterklasse per Delegation auf alle Methoden der Oberklasse zugreifen kann und es darf kein Aufruf eines Konstruktors der Oberklasse (In Java: `super()`) vorhanden sein.

Ausführung

Zuerst muss ein Attribut `d` vom Typ der Oberklasse erzeugt werden, an das die Delegation erfolgen soll. Gegebenenfalls muss es in den vorhandenen Konstruktoren erzeugt werden. Danach werden alle Zugriffe auf ererbte Methoden und Attribute, die in der Unterklasse erfolgen, durch einen entsprechenden Zugriff auf `d` ersetzt.

Anschließend werden für alle Methoden der Oberklasse, auf die in der direkten Unterklassen oder einer noch tiefer liegenden Unterklasse zugegriffen wird, Methodenrümpfe in der direkten Unterklasse erzeugt, deren Rumpf aus einer Delegation über das Objekt d an die Oberklasse besteht. Alle Aufrufe an Methoden der Oberklasse, die mit `super()` eingeleitet werden, müssen durch einen Aufruf an `d` ersetzt werden. Anschließend kann die Vererbungsbeziehung eliminiert werden.

19.14 GUI-Abhängigkeiten entfernen

Motivation

Abhängigkeiten zu anderen Komponenten erschweren generell den Einsatz einer Komponente in fremdem Kontext. Insbesondere Abhängigkeiten zu GUI-Komponenten sollen in Komponenten der Geschäftslogik nicht vorhanden sein, da diese unabhängig von der verwendeten Oberfläche verwendet werden sollen. Direkte Aufrufe von Dialogen oder Mitteilungsfenstern, die nichts mit der Funktionalität zu tun haben, sondern nur Fehler melden oder weitere Parameter fordern, sollten entfernt werden.

Vorbedingung

Die identifizierten GUI-Teile tragen nicht zur Funktionalität bei, sondern bieten nur Statusmeldungen oder dienen zur Datenein- und Ausgabe. Es muss vorher bekannt sein, welche Komponenten (Klassen) reine Benutzerschnittstellenklassen sind.

Ausführung

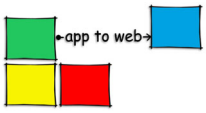
Zuerst müssen alle Stellen im Quelltext der Komponente aufgesucht werden, an denen GUI-Komponenten referenziert werden. Dann muss manuell identifiziert werden, ob es sich nur um eine Mitteilung oder eine Aufforderung zur Dateneingabe handelt. Ist es eine Fehlermeldung, dann sollte anstelle der Mitteilung eine Ausnahme ausgelöst werden. Es muss dafür gesorgt werden, dass diese Ausnahme über die Komponentengrenze hinweg an den Aufrufer geleitet wird.

Werden Dialoge in der Anwendungslogik aufgerufen, die nicht zu einem normalen Ablauf gehören, und nur dann erscheinen, wenn aufgrund eines Fehlers nicht alle Daten vorhanden sind, dann sollte dieser Aufruf durch Auslösen einer Ausnahme ersetzt werden.

19.15 Requires-Schnittstelle explizit machen

Motivation

Die *provides-Schnittstelle* einer Komponente liegt meistens explizit in Form einer oder mehreren Fassadenklassen vor. Für die *requires-Schnittstelle* ist dies in der Regel nicht der Fall. Die Zugriffe auf Elemente der *requires-Schnittstelle* sind über die ganze Komponente verteilt, was den Austausch einer oder mehrerer Komponenten nicht gerade einfach macht. Die Idee ist es, die ganze oder zumindest einen Teil der *requires-Schnittstelle* zu einer requires-Fassade zusammenzufassen, auf die Elemente der Komponente zugreifen. Diese Fassade regelt dann per Delegation, wie die

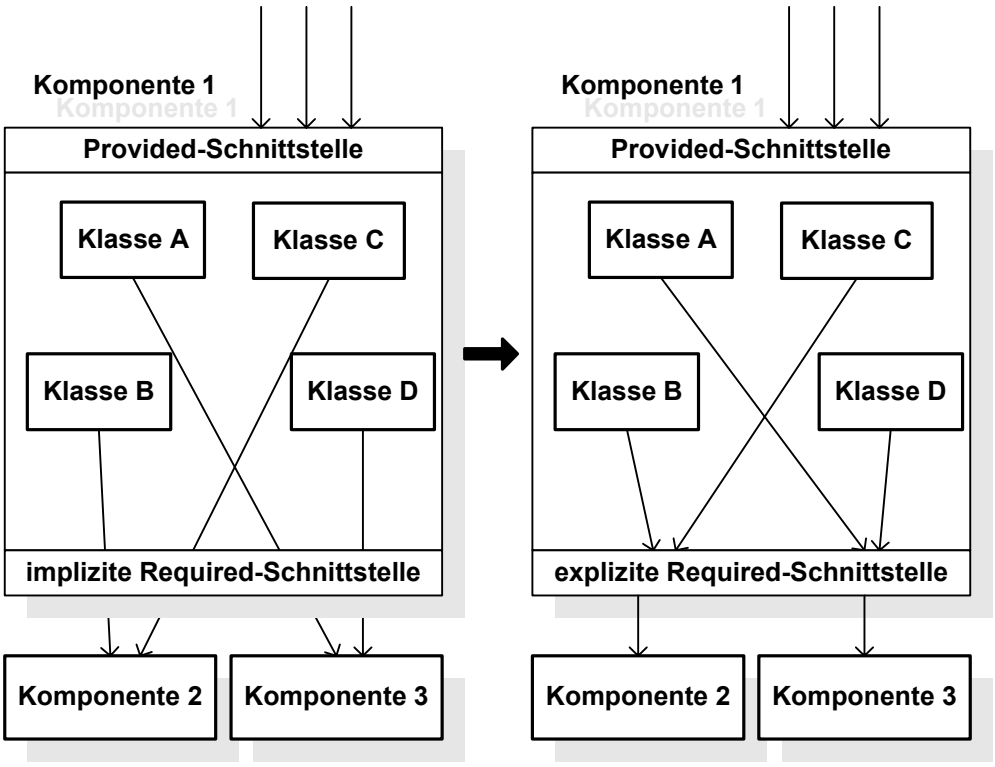


benötigte Funktionalität von einer oder mehreren Komponenten erbracht wird. Der Vorteil liegt nun darin, dass ein Austausch einer benötigten Komponente lokal an der requires-Fassade durchgeführt werden kann.

Vorbedingung

Die hier beschriebene Restrukturierung setzt voraus, dass die *requires-Schnittstelle* nur aus Methodenaufrufen besteht. Attributzugriffe und Typdeklaration müssen gesondert behandelt werden.

Abbildung 47:Restrukturierung requires-Schnittstelle explizit machen



Ausführung

Zuerst muss eine Fassadenklasse für die *requires-Schnittstelle* erzeugt werden. Diese muss beim Erzeugen einer Ausprägung der Komponente auch instanziiert werden. Bei der Erzeugung der requires-Fassade muss sie die entsprechenden Referenzen auf die benötigten Komponenten erhalten, insbesondere müssen in der Fassade Attribute erzeugt werden, die die Referenzen aufnehmen können.

Für alle Methoden, die Teil der *requires-Schnittstelle* sind, müssen identische Methodenköpfe in der Fassade erzeugt werden. Der Rumpf besteht aus einer Delegation an die Komponenten, in denen die Methodenrümpfe implementiert sind. Anschließend müssen die Zugriffe auf die Methoden der *requires-Schnittstelle* auf Zugriffe auf die Ausprägung der *requires-Fassade* umgestellt werden.

19.16 Klasse in Bean umwandeln

Motivation Eine existierende Java-Klasse soll in eine Java Bean umgewandelt werden. Die Restrukturierung hat zum Ziel, alle öffentlichen Attribute nicht mehr direkt zugreifbar zu machen, sondern entsprechende get() und set() Methoden zur Methode hinzuzufügen, und alle Nutzungsstellen anzupassen

Vorbedingung Es wird vorausgesetzt, dass die umzuwandelnde Klasse nicht polymorph verwendet wird.

Ausführung Um eine Klasse A in eine Bean umzuwandeln, muss folgendes getan werden:

- Für alle Attribute, die in A enthalten sind, wird eine Methode

  `<Attributtyp> get<Attributname>()`

  hinzugefügt, die eine Referenz auf das Attribut liefert.

- Für alle Attribute, die in A enthalten sind, wird eine Methode

  `set<Attributname>(<Attributtyp> <Attributname>)`

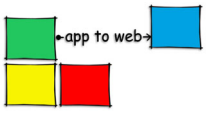
  hinzugefügt, die den Parameter an das Attribut zuweist.

- An allen Stellen außerhalb von A, die eines der Attribute von A referenzieren, wird die entsprechende **`get`** oder **`set`** Methode eingebaut.

19.17 Klasse in Session Bean umwandeln

Motivation Eine aus vorhandenem Quelltext gewonnene Komponente, bzw. gewonnener Komponentenkandidat, kann nicht ohne spezielle Maßnahmen in verteilten Anwendungen eingesetzt werden. Eine Möglichkeit, die Funktionalität über Rechnergrenzen hinweg verfügbar zu machen besteht in der Umwandlung der Komponente, bzw. ihrer Fassadenklasse in eine Session Bean [Lot00].

Vorbedingung Es wird vorausgesetzt, dass die *provides-Schnittstelle* aus genau einer Fassadenklasse besteht, und dass die Komponente keinen Zustand besitzt, der dauerhaft persistent gemacht werden muss. Die Restrukturierung erzeugt eine Session Bean, die zum EJB-Standard 1.1 konform ist.



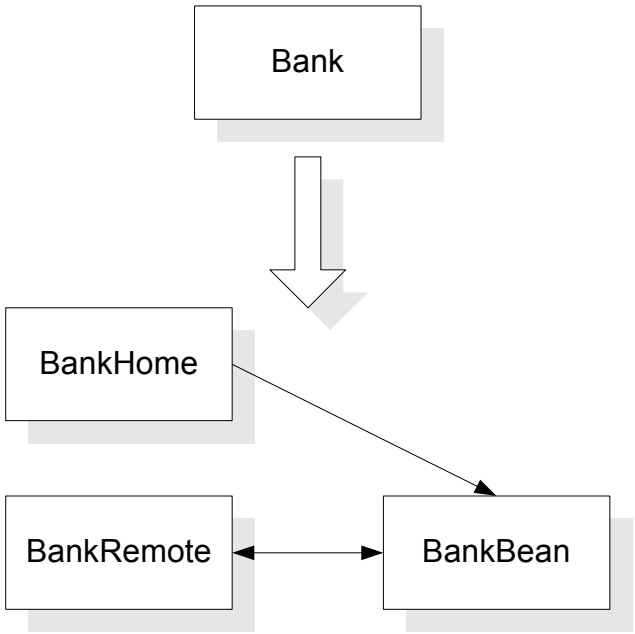
Ausführung

Wie in dem Bank-Beispiel in Abbildung 48 zu sehen, besteht eine Session Bean aus drei Teilen: einem Home-Interface zur Erzeugung der Bean, einem Remote-Interface zur Veröffentlichung der Methoden, und einer Beanklasse, die die Implementierung enthält. Die Schnittstellen können komplett neu generiert werden, die Fassadenklasse wird in eine Beanklasse umgewandelt.

Die Ausführung wird anhand eines einfachen Beispiels aus [Lot00] erklärt:

```
public class Bank
{
  public Bank(String bankName)
  { // -- construct the bank --
  }

  public String createAccount(int balance)
  { // -- create an account and return accno --
    return "00000"; }

}
```

Abbildung 48:Restrukturierung Klasse in SessionBean umwandeln



Zuerst wird das Remote-Interface generiert. Dies ist ein Java-Interface, das von **`javax.ejb.EJBObject`** erbt, und in das alle öffentlichen Methodenköpfe der Fassadenklasse exklusive der Konstruktoren übernommen werden müssen.

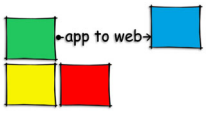
```
import javax.ejb.*;

public interface BankRemote extends javax.ejb.EJBObject
{
    public java.lang.String createAccount(int p0)
        throws java.rmi.RemoteException;

}
```

Als nächstes kann das Home-Interface erzeugt werden, das die Methoden zum erzeugen der Bean bereitstellt. Für jeden öffentlichen Konstruktor wird ein entsprechender Methodenkopf im Home-Interface erzeugt. Die Namen der Köpfe müssen in **`create`** umgewandelt werden, und die benötigten Ausnahmen müssen in den throws-Teil aufgenommen werden.

```
import javax.ejb.*;

public interface BankHome extends javax.ejb.EJBHome
{
    public BankRemote create(String p0)
        throws javax.ejb.CreateException,
                java.rmi.RemoteException;
}
```

Im letzten Schritt muss die bereits existierende Klasse in eine Bean umgewandelt werden. Die Klasse muss die Schnittstelle **`javax.ejb.SessionBean`** implementieren und es muss ein Attribut zur Speicherung des SessionContexts hinzugefügt werden. Die Konstruktoren müssen in **`ejbCreate(<Parameter>)`** umbenannt werden, und es müssen eine Reihe von zusätzlichen ebj-spezifischen Methoden generiert werden (**`activate, passivate, setSessionContext`**, ...)

```
public class BankBean implements javax.ejb.SessionBean
{
  private SessionContext sessionContext;

  // -- business methods --
  public java.lang.String createAccount(int p0)
  {// -- create an account and return accno --
    return "00000"; }

  // -- ejb create --
  public void ejbCreate(java.lang.String p0)
    throws javax.ejb.CreateException
  {
```




```
    // -- construct the bank --
  }

  // -- ejb methods --
  public void setSessionContext(SessionContext ctx) {
    this.sessionContext=ctx; }

  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}
}
```

Diese sind nur die ersten Schritte. Damit die Bean voll einsatzfähig wird, müssen z.B. noch Deploymentdeskriptoren erstellt werden.

20 Fallstudie

In diesem Abschnitt wird der gesamte Prozess vom Finden von Komponentenkandidaten bis zur Umwandlung in Komponenten anhand einer Fallstudie näher beschrieben. Die Auswahl der Fallstudie wurde anhand folgender Kriterien getroffen:

- Die Aufbau der Anwendung sollte als typische dreischichtige Architektur (Datenhaltung - Anwendungslogik - Benutzerschnittstelle) realisiert sein
- Der Quelltext sollte frei verfügbar und in Java oder C++ geschrieben sein, damit die Werkzeuge zum Finden der Komponentenkandidaten zum Einsatz kommen konnten

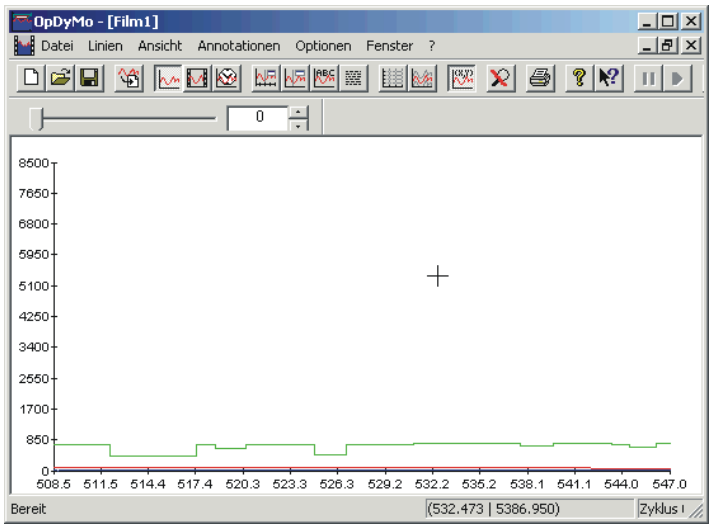
Ursprünglich fiel die Wahl auf das in Java geschriebene ERP-System Compiere [com]. Jedoch wurde schnell festgestellt, dass die erwartete Anwendungslogikschicht nicht in Java implementiert wurde, sondern dass die Logik über das gesamte System inklusive der Clientanwendung in Form von SQL-Anfragen verteilt ist. Somit war klar, dass ein Finden von Komponentenkandidaten hier nicht erfolgsversprechend ist, weil die in [BG02] beschriebenen Werkzeuge voraussetzen, dass die Anwendungslogik objektorientiert implementiert ist.

Deshalb wurde ein anderes System gewählt. Dieses System mit dem Namen Rheingold ist eine Eigenentwicklung des Forschungszentrum Informatik zur Visualisierung von Flussdaten. Der Quelltext ist verfügbar, jedoch fehlt bei diesem System die Datenhaltungsschicht. Daten werden hier in Form von Textdateien verwaltet. Das Fehlen der Datenhaltungsschicht hat allerdings keinen Einfluss auf das Finden von Komponentenkandidaten, da der Schwerpunkt auf der Suche nach Komponenten liegt, die Anwendungslogik bereitstellen.

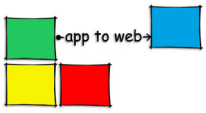
20.1 Beschreibung der Fallstudie

Rheingold entstand als Auftragsarbeit für die Bundesanstalt für Wasserbau. Ursprüngliches Ziel war die Visualisierung von Flussdaten, mittlerweile können jedoch auch diverse Berechnung für fahrende Schiffe, wie z.B Fahrtdauer, berechnet werden.

Abbildung 49:Screen shot von Rheingold



Rheingold ist eine typische in C++ geschriebene MFC-Anwendung. Die hinter einer Zerlegung in Komponenten stehende Motivation war der Wunsch, Teile dieser Anwendung wiederzuverwenden, um die Funktionalität Rheingolds zumindest auszugsweise in einer Webanwendung zur Verfügung zu stellen. Insbesondere sollte es möglich sein, Geschwindigkeitsberechnungen für parametrisierbare Schiffstypen auf verschiedenen Flüssen auszuführen, und die Flussdaten im Webbrowser anzeigen zu lassen.



20.2 Suche nach Komponentenkandidaten

20.2.1 Erster Versuch

Um die Suche nach Komponentenkandidaten vorzubereiten, musste zuerst aus dem Quelltext mittels Faktenextraktion [ABGS01] die Designdatenbank aufgebaut werden. Die aus der Faktenextraktion gewonnenen Daten wurden ohne weitere Vorverarbeitungsschritte als Eingabedaten für die in [BG02] beschriebenen nichthierarchischen Clusteringalgorithmen verwendet. Um die Algorithmen automatisch anzuwenden wurde das vorhandene Werkzeug James [Tri01] benutzt. Es wurden nur nichthierarchische Algorithmen verwendet, da sie im Gegensatz zu den hierarchischen Algorithmen eine Menge von Komponentenkandidaten liefern. Bei den in James verwendeten hierarchischen Verfahren muss in einem anschließenden, manuellen Interpretationsschritt mittels noch nicht näher definierten Heuristiken bestimmt werden, welche Form die einzelnen Komponentenkandidaten haben.

Eine Auswertung der berechneten potentiellen Komponentenkandidaten ergab, dass die Algorithmen in diesem Fall nicht in der Lage waren, eine Trennung zwischen Benutzerschnittstelle und Anwendungslogik zu bestimmen. Bei stichprobenartigen Überprüfungen der Komponentenkandidaten wurde festgestellt, dass Benutzerschnittstellenklassen und Klassen der Anwendungslogik zu einem Kandidaten zusammengefasst wurden. Auch bei unterschiedlicher Gewichtung der in den Algorithmus einfließenden Beziehungen zwischen den Klassen konnte eine Trennung nicht erreicht werden, Benutzersschnittstellenklassen und Anwendungslogikklassen sind zu stark miteinander verwoben.

Da jedoch klar ist, dass die Benutzerschnittstellenklassen nicht in die Komponentenkandidaten aufgenommen werden sollen, wurde beschlossen den Benutzerschnittstellenteil von Hand abzutrennen, und die übrigen Klassen erneut mit den Clusteringalgorithmen zu bearbeiten.

20.2.2 Abtrennen der Benutzerschnittstellenklassen

Glücklicherweise lagen für Rheingold schon von Hand bestimmte Vorschläge einer Aufteilung in zwei Schichten vor. Diese wurden einige Monate vorher von einem Studenten bestimmt, dessen Aufgabe eine manuelle Zerlegung des Systems in Subsysteme war.

Um zu klären, ob eine Aufteilung des Systems entlang der Grenze zwischen Benutzerschnittstellen- und Logikklassen überhaupt sinnvoll ist, musste zunächst unter-

sucht werden, welche Arten von Abhängigkeiten von Klassen der Anwendungslogikschicht zu Klassen der Benutzerschnittstellenschicht vorhanden waren. Gibt es viele dieser Abhängigkeiten, spricht dies dafür dass die Aufteilung in Schichten nicht vorhanden ist, und die Klassen wirklich zusammenarbeiten müssen. Abhängigkeiten in der anderen Richtung hingegen sind natürlich, weil die Benutzerschnittstellenklassen die Klassen der Anwendungslogik kennen müssen, an die sie Aufgaben delegieren.

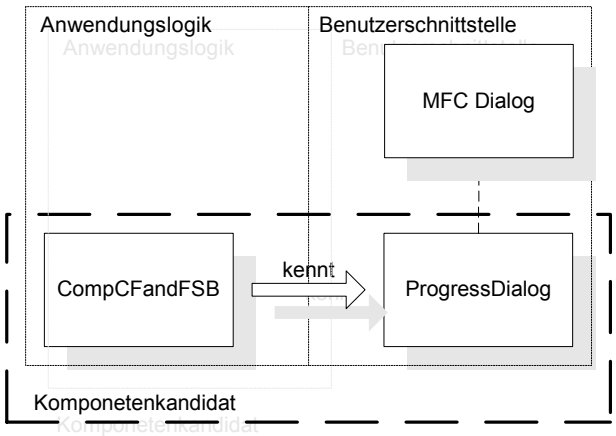


Abbildung 50:Unerwünschte Abhängigkeit zwischen Anwendungslogik und Benutzerschnittstelle

Eine genauere Betrachtung der Beziehungen ergab, dass die Anwendungslogikklassen die Benutzerschnittstellenklassen oft direkt kannten, und z.B. Dialogboxen zur Dateneingabe oder Mitteilungsfenster für Fehlermeldungen selbständig per Funktionsaufruf öffnen, anstatt dies z.B. per Ereignissmechanismen zu veranlassen. In Abbildung 50 ist ein Teil eines Komponentenkandidaten zu sehen, bei dem die Klasse `CompCFandFSB` Berechnungen durchführt und selbst steuert, wie der Berechnungsfortschritt angezeigt wird.

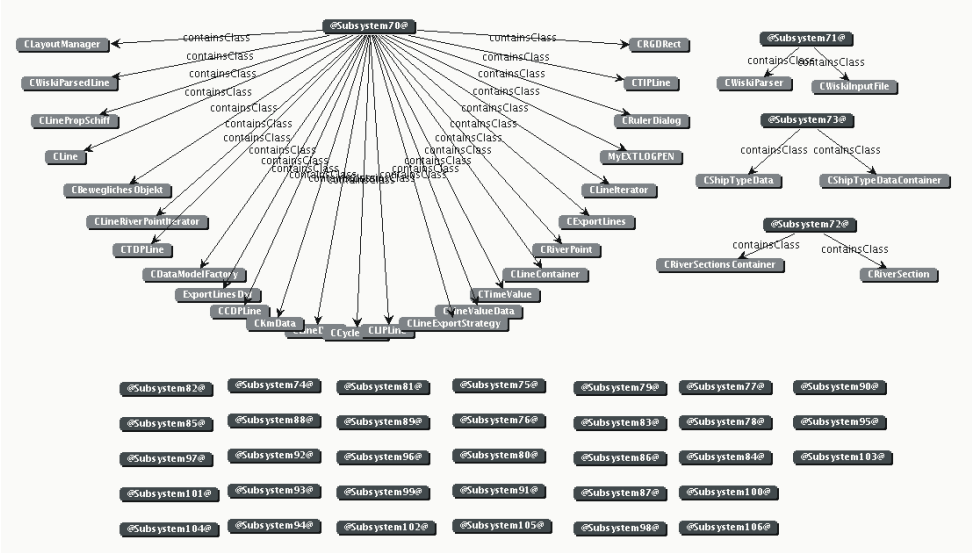
Es konnte allerdings nicht festgestellt werden, dass Benutzerschnittstellenklassen wichtige Funktionalität implementieren. Deswegen wurde die vorgeschlagene Einteilung in zwei Schichten noch einmal überprüft und leicht modifiziert. Anschließend wurden die Benutzerschnittstellenklassen aus der Designdatenbank entfernt, um eine Suche nach Komponentenkandidaten in der Anwendungslogikschicht durchzuführen.

20.2.3 Zweiter Versuch

Bei der erneuten Anwendung der Clusteringalgorithmen auf die in der Designdatenbank enthaltenen Klassen, wurde bei mehrfachen Durchläufen mit unterschiedlichen Gewichten für die Beziehungen zwischen den Klassen folgendes Ergebnis erzielt: Die Menge der Klassen wurde stets in einen sehr großen Komponentenkandidaten

mit ca. 20 Klassen, drei bis vier Komponentenkandidaten mit 2 Klassen und 30 Komponentenkandidaten mit einer Klasse, partitioniert.

Abbildung 51:Ergebnis des zweiten Clusterings



Nach einer Betrachtung des Quelltextes der Klassen des grossen Komponentenkandidaten und der aus einer Klasse bestehenden Komponentenkandidaten wurde festgestellt, dass sehr wohl Abhängigkeiten in beide Richtungen bestehen. Die aus zwei Klassen bestehenden Kandidaten wurden richtig zusammengefasst, da es sich jeweils um einen Datentyp und seinen Behälter handelte. Insgesamt musste aber festgestellt werden, dass eine Zerlegung wenig sinnvoll ist, da keine wirklich abgeschlossenen Einheiten gefunden werden konnten.

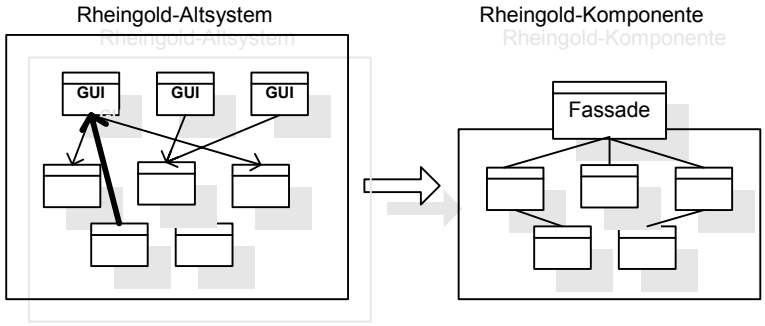
20.3 Umwandeln in Komponenten

Zusammenfassend läßt sich sagen, dass das bestehende System am besten in zwei Schichten aufgeteilt wird. Eine Benutzerschnittstellenschicht, die nicht wiederverwendet werden kann, und eine Anwendungslogikschicht, die mit einer neuen Schnittstelle versehen werden muss, um die gewünschte Funktionalität zu erbringen. Mit dem Werkzeug konnte die Trennung in zwei Schichten nicht automatisch bestimmt werden, die Ergebnisse der Algorithmen im zweiten Versuch führten allerdings dazu, dass schnell klar war, dass eine Zerlegung dieses Teils nicht sinnvoll ist.

Somit braucht man zwei Restrukturierungen, um Rheingold in eine wiederverwertbare Komponente umzuwandeln:

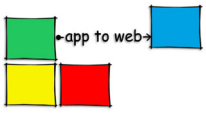
- *GUI-Abhängigkeiten entfernen*: Die gefundenen Abhängigkeiten zu den MFC-Klassen müssen entfernt werden.
- *Neue Schnittstelle bereitstellen*: Eine Fassadenklasse muss hinzugefügt werden, die die gewünschten Methoden zur Berechnung der Fahrtdauer in Abhängigkeit von Schiffstyp und Flussabschnitt bereitstellt.

Abbildung 52:Umwandlung in Komponente



Bei der Untersuchung des Quelltextes der beteiligten Klassen wurde festgestellt, dass die Methoden zur Geschwindigkeitsberechnung eines Schiffes auf zwei Klassen verteilt sind. Nach Rückfragen bei den Entwicklern stellte sich heraus, dass an dieser Stelle eine Umstrukturierung vorgenommen worden war, jedoch nicht alle Methoden zur Geschwindigkeitsberechnung einheitlich in einer Klasse zusammengefasst wurden. Somit bietet es sich an, hier die Restrukturierung *Methode verschieben* einzusetzen. Mit dem Werkzeug konnte dies nicht automatisch bestimmt werden, da die einzigen berücksichtigten Strukturierungseinheiten Klassen sind.

Die vorgeschlagenen Transformationen konnten allerdings bis jetzt noch nicht durchgeführt werden, da für die Sprache C++ kein Werkzeug zur Verfügung steht, dass die Restrukturierungen erleichtert. Sie könnten jedoch jederzeit von Hand ausgeführt werden, sind jedoch aufgrund der fehlenden Werkzeugunterstützung, nur mit erheblichem Aufwand durchzuführen.



21 Adaption von Quelltextkomponenten

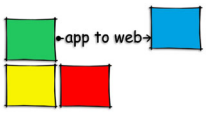
Sollen die wiedergewonnenen Komponenten in der Praxis wiederverwendet werden, muss zuerst geprüft werden, ob die Komponenten die an sie gestellten Anforderungen erfüllen. Diese Thematik wird ausführlich in [Sen02] behandelt, an dieser Stelle werden deshalb nur noch einmal die wichtigsten Erkenntnisse zusammengefasst.

Differenzen zwischen angebotener und angeforderter Spezifikation können folgende Bereiche betreffen:

- **Struktur und Schnittstellen:** Die gewünschte Funktionalität ist in der Komponente anders modelliert, oder sie ist im Bezug auf den Ort (Namensräume, Klassen, Namen, Typen) anders geschnitten.
- **Persistenz und Datenmodell:** Ablegen bzw. Übertragen der Daten wird auf unterschiedliche Art und Weise durchgeführt.
- **Performanceanforderungen:** Algorithmen in der Komponente genügen nicht den Anforderungen, z.B. bezüglich Komplexität oder Speicherbedarf.
- **Kompositions, Protokollanforderungen:** Oft sind die Anforderungen bzgl. Interaktion, Kommunikation und Synchronisation verschieden.
- **Externalisierung des Programmablaufs:** Fehlerbehandlung wird in den beiden Spezifikationen anders vorgesehen, z.B. Rückgabewerte vs. Ausnahmenbehandlung.

Grundsätzlich können zwei Ansätze zur Adaption einer existierenden Komponente unterschieden werden. In der ersten Variante wird zusätzlicher Klebecode generiert, der als Vermittler zwischen Benutzern der Komponente und der Komponente selbst fungiert, in der zweiten Variante wird der Quelltext der Komponente invasiv angepasst. Beide Möglichkeiten haben ihre Vor- und Nachteile. Invasive Adaption ist nur möglich, wenn der Quelltext verfügbar ist, also nicht bei binären Komponenten. Sie ermöglicht tiefgreifende Änderungen im Inneren der Komponente, ist allerdings aufwändiger durchzuführen und hat den Nachteil, dass die Komponente ihre ursprüngliche Gestalt verliert. Bei der Verwendung von Klebecode bleibt die Komponente in ihre ursprünglichen Form erhalten, alle gewünschten Änderungen müssen jedoch außerhalb der Komponente erfolgen und erfordern deshalb mehr zusätzlichen Quelltext.

Die Adaption kann manuell oder werkzeuggestützt erfolgen, eine halbautomatische Vorgehensweise senkt die Fehleranfälligkeit jedoch beträchtlich. Zur Adaption verwendbare Ansätze sind Restrukturierungen [Opd92] [Fow99], Aspektorientiertes



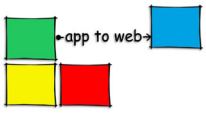
Programmieren [KLM+97] und Inject/J [GK01]. Inject/J bietet gemessen an den unterstützten Adaptionen den größten Funktionsumfang. Es umfasst alle Möglichkeiten, die Restrukturierungen und Aspektorientiertes Programmieren bieten, und erlaubt darüber hinaus die Formulierung eigener, komplexer Adaptionen.

22 Werkzeugunterstützung

Zur Gewinnung von Komponenten aus Altsystemen können an vielen Stellen Werkzeuge eingesetzt werden, die den Entwicklern diese Aufgabe leichter machen. Zur Unterstützung der Aktivität Finden von Komponentenkandidaten, wurde am FZI das Werkzeug JAMES entwickelt [Tri01]. Bei der Evaluation der Komponentenkandidaten und der Ableitung nötiger Transformationen ist eine Visualisierung hilfreich. Hierzu wurde ein Werkzeug namens Echidna [Mar02] geschaffen. Zur automatischen Ausführung der Restrukturierungen kann zum einen das am FZI entstandene Werkzeug Inject/J [GK01] benutzt werden, das während des Projektes App2Web weiterentwickelt wurde. Ein anderes Werkzeug zur Ausführung von Programmtransformationen ist das *Design Maintenance System* von Semantic Design [SD], das am IESE zum Einsatz kam. Die genannten Werkzeuge werden in diesem Kapitel beschrieben, um einen Eindruck ihres Funktionsumfanges zu vermitteln.

22.1 JAMES

James ist ein Werkzeug zur automatischen Extraktion von Komponenten in Java-Systemen, und wurde im Verlauf einer Diplomarbeit [Tri01] am Forschungszentrum Informatik entwickelt. Ziel des Werkzeuges ist es, mit verschiedenen Clusteringalgorithmen [BG02] Komponentenkandidaten zu finden. Ein Ablauf von James besteht aus folgenden vier Abschnitten:



- **Faktenextraktion:** Zuerst müssen die Entitäten und Relation aus dem Quelltext extrahiert werden, auf denen die Clusteringalgorithmen arbeiten. Hierzu wird die frei verfügbare Bibliothek Recoder [LH01] benutzt. James benutzt als Entitäten nur Klassen eines Systems, und verschiedene Beziehungen zwischen diesen Klassen wie z.B. Vererbungsbeziehungen, Variablenzugriffe, etc. .
- **Vereinfachung:** Zwischen zwei Klassen können mehrere verschiedene Beziehungen existieren, z.B. eine Vererbungs- und eine Aufrufbeziehung. In diesem Schritt werden die mehrfachen Beziehungen durch eine einzige, gewichtete Beziehung ersetzt. Wie stark die einzelnen Beziehungstypen in das Gesamtgewicht der Kante einfließen, kann pro Typ parametrisiert werden.
- **Clustering:** Nachdem das Modell aufbereitet wurde, kann in diesem Schritt ein Clusteringalgorithmus durchgeführt werden. James unterstützt sowohl hierarchische als auch nichthierarchische Algorithmen.
- **Analyse:** Insbesondere das Ergebnis eines hierarchischen Algorithmus bedarf einer Analyse, da von Hand in der durch den Algorithmus aufgebauten Hierarchie entschieden werden muss, wo die Komponentengrenzen zu ziehen sind. Die hierarchischen Algorithmen liefern zwar Gruppen von Klassen, also Kandidaten, jedoch muss auch hier noch einmal überprüft werden, ob die vorgeschlagene Zerlegung sinnvoll ist.

James befindet sich zur Zeit noch in einem prototypischen Stadium. Seine Benutzung erfordert detaillierte Kenntnisse über die internen Abläufe und die verwendeten Formate. Jedoch ist es eines der wenigen Werkzeuge, die Komponentenkandidaten in objektorientierten Systemen - zur Zeit werden Java und C++ unterstützt- suchen und finden können. Sein Einsatz ist einer manuellen Vorgehensweise vorzuziehen.

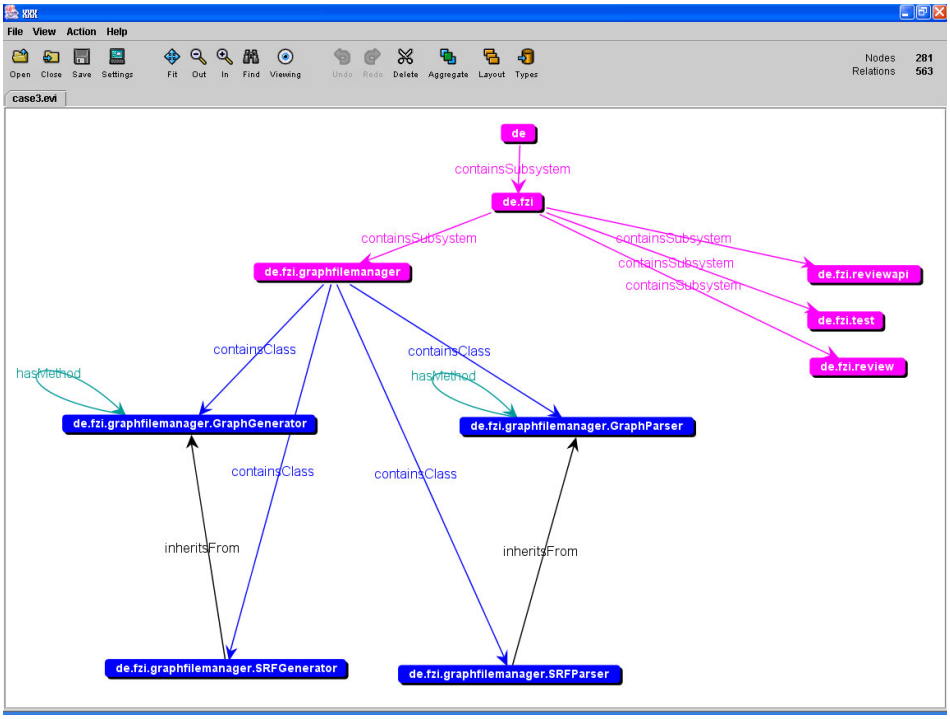
22.2 Echidna

Mit Echidna ist ein Werkzeug entstanden, mit dem Softwaresysteme geeignet visualisiert werden können. In Echidna sind die in [ABGS01] beschriebene Mechanismen implementiert, die ein Arbeiten mit sehr großen Graphen erst ermöglichen:

- Filterung: Alle Knoten und Kanten sind typisiert. Es ist interaktiv möglich, die aktuelle Menge an Knoten- und Kantentypen festzulegen. Die ausgeblendeten Kanten und Knoten werden aber nicht wirklich gelöscht, sondern nur unsichtbar gemacht, so dass sie bei Bedarf wieder angezeigt werden können.
- Gruppierung: Gruppierung bedeutet eine Menge von Entitäten niedriger Abstraktionsebene durch Entitäten einer höheren Abstraktionsebene zu ersetzen, und die Abhängigkeiten der unteren Ebene auf die höhere Ebene zu übertragen. Echidna ermöglicht die interaktive Gruppierungen von Methoden- auf Klassen-, Klassen- auf Subsystem- und Subsystem- auf Subsystemebene. Dabei muss das Abstraktionsniveau nicht überall gleich sein, sondern es kann beliebig variiert werden. Dies

erlaubt es zum Beispiel, einen Teil des Systems auf Methodenebene, und andere Teile, die weniger wichtig sind, auf Subsystemebene zu betrachten.

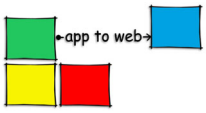
Abbildung 53: Visualisierung eines Systems auf Subsystem- und Klassenebene



Echidna wurde komplett in Java entwickelt, und ist somit plattformunabhängig einsetzbar. Momentan kommt es zusammen mit den Goose-Tools [CBS01] zum Einsatz, weshalb bis jetzt auch nur ein proprietäres Format zum Lesen der Graphen unterstützt wird. Es besteht aber die Möglichkeit, die Graphen in einem Vektorformat namens SVG [W3C] zu exportieren. Der einzige Nachteil von Echidna ist die momentan noch fehlende Unterstützung automatischer Layouts. Diese soll in den nächsten Monaten integriert werden.

22.3 Inject/J

Inject/J [GK01] ist ein Werkzeug zur skriptgesteuerten invasiven Softwareadaption für Java, welches am Forschungszentrum Informatik (FZI) entwickelt wird. Viele der in Kapitel 4 beschriebenen Transformationen lassen sich mit seiner Hilfe werkzeuggestützt durchführen. Mit dem Werkzeug Inject/J wird eine Skriptsprache zur



Beschreibung der Adaptionen bereitgestellt, die auf folgenden vier Konzepten basiert:

1. **Namensraum:** Der Namensraum umfasst alle Pakete und Klassen eines Systems, die während der Adaption berücksichtigt werden sollen.

2. **Webepunkte**: Die Webepunkte sind die Stellen im System, welche durch Inject/J transformiert werden können. Sie bilden somit das Metamodell von Inject/J. Zusätzlich zu den implizit definierten Webepunkten wie Klassen, Methoden, Zugriffen etc. existieren in Inject/J noch explizite Webepunkte. Dies sind speziell markierte Bereiche im Quelltext des Systems, welche durch das Metamodell von Inject/J ansonsten nicht beschrieben werden können. Webepunkte werden durch einen Namen identifiziert. Diese entsprechen den im Quelltext benutzten Identifikatoren, wie zum Beispiel Klassennamen, Methodensignaturen oder Variablennamen. Abbildung 54 zeigt einige Beispiele solcher Webepunktidentifikatoren. Darüberhinaus besitzen Webepunkte eine Reihe von bereitgestellten oder berechneten Attributen, wie z.B. die Sichtbarkeit einer Klasse.

Abbildung 54:Webepunkt-Identifikatoren in Inject/J

class `'fzi.injectj.Main'`	Identifikator für Klasse
method `'start(java.lang.Object)'`	Identifikator für Methode
access `'peek()'` **to class** `'java.util.Stack'`	Identifikator eines Methodenzugriffs
access `'stack'`	Identifikator eines Variablenzugriffs
explicit `'init'`	Identifikatoren eines expliziten Webepunkts

3. **Navigation:** Mit Hilfe der in der Skriptsprache enthaltenen Navigationsanweisungen ist es möglich, die von einer Adaption betroffenen Webepunkte aufzusuchen. Dabei werden zwei Arten von Navigation unterschieden: hierarchische Navigation und direkte Navigation. Die hierarchische Navigation erfolgt dabei entlang der Strukturierungshierarchie von Java, das heißt von einer Klasse zu einer Methode, von dort zu einem Zugriff etc. Zu diesem Zweck werden die Webepunktidentifikatoren mit Navigationsmustern verglichen. Diese Navigationsmuster sind im Wesentlichen reguläre Ausdrücke in Verbindung mit Quantoren. Mit Hilfe der hierarchischen Navigation werden die einzelnen Webepunkte eindeutig identifiziert. Auf ein mit hierarchischer Navigation besuchten Webepunkt kann zu einem späteren Zeitpunkt unter Zuhilfenahme eines Verweises direkt zugegriffen werden. In diesem Fall spricht man von direkter Navigation.

4. **Transformation:** Inject/J unterstütz eine Reihe von Basistransformationen, aus denen sich komplexere Transformationen zusammensetzen lassen. Die Basis-

transformationen werden in der Inject/J-Terminologie Webeoperationen genannt. Inject/J unterstützt dabei, je nach Webepunkt, folgende Arten von Webeoperationen:

- **`before/after`** fügt neuen Code so ein, dass er vor beziehungsweise nach dem momentanen Webepunkt ausgeführt wird.
- **`add`** fügt Elemente in eine Menge ein. Neben neuen Klassen kann dies zum Beispiel eine neue Methode oder ein neues Attribut in der Merkmalsmenge einer Klasse sein. Durch entsprechende Ausprägungen der add-Anweisung ist es aber zum Beispiel auch möglich, eine neue Ausnahme in die Liste der geworfenen Ausnahmen einer Methode einzufügen.
- **`change`** ändert die Eigenschaften eines Webepunkts, wie zum Beispiel die Modifizierer einer Klasse oder einer Methode.
- **`replace`** ersetzt den bisherigen Quelltext eines Webepunkts durch ein neues Quelltext-Fragment.
- **`delete`** entfernt den momentanen Webepunkt.

Neben den bereits beschriebenen Anweisungen bietet Inject/J noch Anweisungen zur bedingten Ausführung, Listeniteration und Benutzerinteraktion. Auf diese wird an dieser Stelle nicht weiter eingegangen, detaillierte Informationen über die Skriptsprache sind unter [GK01] zu finden. Das Werkzeug unterliegt folgendem Korrektheitsbegriff: war ein System vor der Transformation übersetzbar, so ist es dies nach der Transformation ebenfalls. Im derzeitigen Entwicklungsstand kann dies allerdings nicht in allen Fällen garantiert werden, da noch

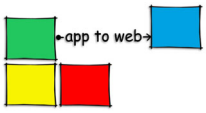
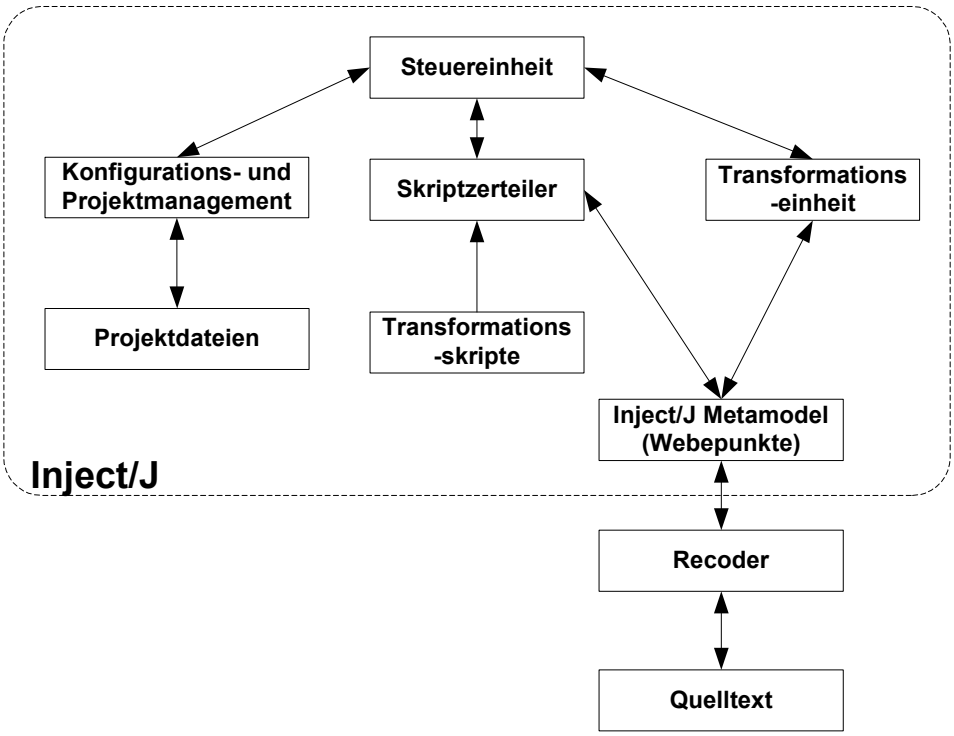
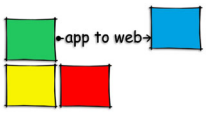


Abbildung 55:Aufbau des Werkzeuges Inject/J



nicht alle nötigen Vorbedingungen akkurat geprüft werden können. Andererseits sind viele Transformationen nicht durchführbar, weil die geprüften Vorbedingungen zu restriktiv sind. Der grobe Aufbau des Werkzeugs ist in Abbildung 55 dargestellt. Inject/J benutzt die Meta-programmier- Bibliothek Recoder [LH01], um den Quelltext des zu adaptierenden Systems einzulesen. Recoder stellt dabei Programme als Strukturbaum dar, wobei dieser direkt manipulierbar ist. Mit Hilfe von Recoder ist es darüberhinaus möglich, diesen veränderten Strukturbaum wieder als Quelltext zurückzuschreiben, wobei die Formatierung erhalten bleibt. Über diesen Strukturbaum legt Inject/J eine weitere Abstraktionsschicht, die Webepunkte. Die Steuereinheit veranlasst den Aufbau dieser Abstraktionsebene, wohingegen die Transformationseinheit die Transformationen darüber ausführt. Da ein Durchlauf von Inject/J eine Vielzahl von Einstellungen erfordert, wie zum Beispiel benutzte Quelldateien, benutzte Transformationsskripte oder die Klassen des Namensraums, existiert ein separates Konfigurations- und Projektmanagement, welches diese Einstellungen verwaltet und gegebenenfalls in Projektdateien ablegen kann. Die Transformationsskripte werden nicht direkt interpretiert, sondern durch einen Zerteiler vorverarbeitet, wobei gleichzeitig die syntaktische Korrektheit des Skripts geprüft wird. Das Zusammenspiel aller Teile wird schließlich durch die Steuereinheit koordiniert.

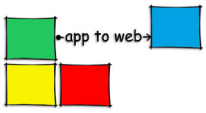


22.4 DMS

Industrielle Softwaresysteme haben oft Ausmaße von mehreren hundertausend oder sogar mehreren Millionen Zeilen Code, was, objektorientierte Programmierung vorausgesetzt, tausende von Klassen in ebensovielen Dateien bedeutet. Refactorings zur systematischen Transformation dieser Softwaresysteme sind also manuell kaum auf zuverlässige Art und Weise durchzuführen, da die Struktur der Software, selbst bei sorgfältiger Dokumentierung, unüberschaubar ist. Somit stellt sich die Frage wie Refactorings implementiert werden können. Prinzipiell ist die Implementierung von Refactorings mit jedem Tool möglich, das den Zugriff auf den Syntaxbaum eines geparsten Programms erlaubt, um diesen zu analysieren und zu transformieren. Die Komplexität der meisten Refactoring-Algorithmen lässt allerdings gewisse Hilfsmittel, wie eine Bibliothek zum Bearbeiten des Syntaxbaums, sinnvoll erscheinen, um eine fehlerfreie und überschaubare Implementierung zu gewährleisten. Ein kommerzielles Toolset, das viele dieser Hilfsmittel zur Verfügung stellt, ist das DMS (Design Maintenance System). Im Folgenden wird das DMS und die Fähigkeiten seiner verschiedenen Werkzeuge vorgestellt; insbesondere im Hinblick auf die Implementierung von Refactorings.

DMS wurde von der Firma Semantic Designs (Austin, Texas) unter der Federführung von Dr. Ira Baxter entwickelt und wird stetig weiterentwickelt und ausgebaut. Es handelt sich hierbei um eine Sammlung von verschiedenen Tools für Compilerbau und Programmanalyse, die mittels zugehöriger APIs (Application Programming Interface) und einer zum DMS gehörenden funktionalen Programmiersprache (PARLANSE) gesteuert werden können. Ferner ermöglicht PARLANSE die Interaktion und den Datenaustausch zwischen den verschiedenen Tools. Die wichtigste Eigenschaft von PARLANSE ist jedoch die Möglichkeit zur Parallelprogrammierung und somit zur parallelen Verarbeitung von Syntaxbäumen großer Softwaresysteme. Im einzelnen stehen zur Zeit folgende Tools, in DMS auch Domains genannt, zur Verfügung:

- **DMS Lexical Domain:** Ein Tool zur Generierung von Lexern, dessen Funktionsweise und Syntax üblichen Tools wie zum Beispiel Flex entspricht.
- **DMS Syntactical Domain:** Ein Parsergenerator, der im Gegensatz zu üblichen Tools, wie Bison, auch Parser für LR(1)- Grammatiken, einer Oberklasse der üblichen LALR(1)-Grammatiken, erzeugen kann. Ansonsten ist auch dieses Tool in seiner Funktionsweise und seiner Syntax den üblichen Tools sehr ähnlich. Jedoch ist die automatische Generierung von abstrakten Syntaxbäumen möglich. Dies geschieht mittels der Eliminierung von Kettenableitungen, der Entfernung von Terminalknoten ohne angehängten Wert und weiterer möglicher Optimierungen. Auf den resultieren abstrakten Syntaxbaum kann mittels eines API zugegriffen werden, so dass der Syntaxbaum verändert werden kann. Diese Fähigkeit ist für die Implementierung von Refactorings entscheidend. Daher wird in den folgenden Abschnitten genauer darauf eingegangen.



- **DMS Box Pretty Printer Domain:** Hierbei handelt es sich um ein Tool zur Generierung von Pretty Printern zum Zweck der automatischen Formatierung von Programmtexten, einem sogenannten Pretty Printer, mit dessen Hilfe man z.B. die Einrückung von untergeordneten Blocks, wie Schleifen oder Verzweigungen, automatisieren kann.
- **DMS Attribute Grammar Domain:** Ein Tool zum Auswerten von Attributgrammatiken (in diesem Fall S-Attributierung). Besonderheiten im Vergleich zu herkömmlichen Tools, wie Bison, sind die Fähigkeit zum Durchlaufen mehrer Durchgänge (Passes) und die Möglichkeit zum Aufruf von in PARLANSE geschriebenen Funktionen, um Berechnungen mit den Attributen durchzuführen.
- **DMS Parlanse Domain:** Hierbei handelt es sich um einen Compiler für die schon zuvor erwähnte Programmiersprache PARLANSE. PARLANSE ermöglicht die parallele Bearbeitung mehrer Syntaxbäume, was DMS hervorragend für die Transformation großer Softwaresysteme qualifiziert. Charakteristisch für PARLANSE ist, dass es eine stark typisierte funktionale Sprache ist, deren Syntax an Common Lisp erinnert. Der Sprachumfang ermöglicht Modularisierung, Synchronisation mittels Semaphoren, sowie Zugriff auf Systemfunktionen. Geplant ist ferner eine XML-Schnittstelle zur Erleichterung des Datenaustauschs mit anderen Softwaresystemen. Der Compiler erzeugt Bytecode, der auf einer virtuellen Maschine verarbeitet wird, die dafür sorgt, dass Multiprozessorsysteme bei der parallelen Ausführung optimal genutzt werden können. Alle APIs der anderen Domains sind in PARLANSE geschrieben und können aus PARLANSE-Programmen heraus aufgerufen werden.
- **DMS Rule Specification Domain:** Diese Domain erlaubt es Pattern im Syntaxbaum zu definieren und diese durch andere definierte Patterns zu ersetzen. Da es sich hierbei um ein Tool handelt, das für die Implementierung von Refactorings sehr nützlich ist, wird im Folgenden detailliert darauf eingegangen.

Bei der Implementierung von Refactorings ist es im ersten Schritt notwendig einen Lexer und einen Parser für die entsprechende Sprache zu erzeugen. Für die meisten gängigen Programmiersprachen sind die dafür notwendigen Skripte von Semantic Designs erhältlich, so dass sich diese Arbeit einsparen lässt. Da alle Refactoring-Algorithmen umfangreiche semantische Informationen benötigen, ist es unabdingbar eine Attributgrammatik und eine Symboltabelle für die jeweilige Programmiersprache zu erstellen. Die Attributgrammatik lässt sich mit Hilfe der DMS Syntactical Domain (s.o.) erstellen und für die Symboltabelle ist im DMS schon eine entsprechende Datenstruktur vorgesehen, die eine komfortable Implementierung ermöglicht. Nach Beendigung dieser Vorarbeiten kann die Implementierung des eigentlichen Refactoring-Algorithmus beginnen. Dabei stehen grundsätzlich zwei Vorgehensweisen zur Verfügung, die allerdings fast immer in hybrider Form angewandt werden sollten, um eine effiziente und leicht verständliche Implementierung zu erreichen. Die beiden Vorgehensweisen sind:

- **Prozedurale Vorgehensweise:** Die Transformationen werden durch Durchlaufen des Syntaxbaums durchgeführt. Beim Durchlaufen werden zuerst die für die Korrektheit der Transformation nötigen Vorbedingungen überprüft, danach wird der Syntaxbaum durch löschen, anhängen und abändern von Teilbäumen mit Hilfe der API transformiert und anschließend von einem Unparser wieder ausgegeben. Diese Vorgehensweise ist intuitiv verständlich, allerdings wird die Implementierung bei komplizierten Refactorings schnell unverständlich und die entsprechenden Programme zu groß und zu verschachtelt, da viele Fallunterscheidungen vorkommen, um alle möglichen Konstellationen des Syntaxbaums zu betrachten. Im Folgenden wird der prozedurale Ansatz anhand von Beispielen illustriert.
- **Pattern-basierte Vorgehensweise:** Bei dieser Vorgehensweise werden Pattern im Syntaxbaum definiert, also bestimmte syntaktische Strukturen festgelegt, die durch automatisierte Funktionen im DMS im Syntaxbaum gesucht werden können. Diese Pattern können dann im Falle eines Auftretens durch andere Pattern ersetzt werden. Der Vorteil dieser Vorgehensweise ist die leicht verständliche Implementierung der Transformation, allerdings ist diese Art von Programmierung im allgemeinen etwas ungewohnt. Ferner kommt dieser Ansatz nicht ohne die prozedurale Vorgehensweise aus, da die für die Korrektheit der Refactoring notwendigen Vorbedingungen, nicht mit Hilfe der DMS Rule Specification Domain überprüft werden können. Auch dieser Ansatz wird im Folgenden anhand von Beispielen näher erläutert.

Betrachten wir zunächst einmal ein einfaches Beispiel für einen prozeduralen Durchlauf durch den Syntaxbaum eines Java-Programms. Dabei wird ein Teilbaum durch einen anderen Teilbaum, der zuvor erzeugt wurde, ersetzt. Bevor dies geschehen kann, muss jedoch zunächst einmal die richtige Stelle im Syntaxbaum des Java-Programms gefunden werden. Zu diesem Zweck stellt die API eine Funktion Namens **`AST:ScanNodes`** zur Verfügung, die einen übergebenen Syntaxbaum so lange durchläuft, bis ein Knoten gefunden wurde, der eine bestimmte Eigenschaft erfüllt, die mit Hilfe einer Funktion mit Rückgabetyp Boolean spezifiziert werden kann (im Beispiel: **`function visit`**). An der gefundenen Stelle im Syntaxbaum, markiert durch die Variable **`marker`**, wird dann ein Knoten mit einem String-Literal mit dem Wert `name` durch einen Knoten mit einem String-Literal mit dem Wert `Vorname` ersetzt. Die Ersetzung erfolgt mit Hilfe der API-Funktion **`AST:ReplaceNthChild`**. Auf die komplette Erläuterung des Beispiels wird aus Platzgründen verzichtet, wobei sicherlich grundlegende Kenntnisse über funktionale Programmierung für das Verständnis des Beispiels notwendig sind.

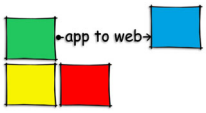


Abbildung 56:Beispiel für eine prozedurale Transformation

```
1: (define global
2:   (module
3:     (public)
4:       [marker AST:Node]
5:       [id string]
6:   )module
7: )define
8:
9: (define visit
10:  (lambda
11:    (function boolean AST:Node)
12:      (value
13:        (local
14:          (;;
15:            (ifthen (== (AST:GetNodetype ?) 354)
16:              (;;
17:                (ifthen (== (@ (AST:GetString ?)) string)
18:                  (;;
19:                    (= global:marker ?)
20:                    (return ~f)
21:                  );;
22:                )ifthen
23:              );;
24:            )ifthen
25:          );;
26:        )local
27:        ~t
28:      )value
29:   )lambda
30: )define
31:
32: (define scannodes_example
33:   (action
34:    (procedure
35:      (structure
36:        [graph AST:Forest]
37:        [domain_representation AST:Representation]
38:        [syntax_tree AST:Node]
39:        [helpnode AST:Node]
40       )structure
41:    )procedure
42:    (local
43:     (;;
44:       (include `../../../Parser/Source/ASTRepresentationControls.par')
45:     );;
46:     (;;
47:        (= global:id `name')
48:        (AST:ScanNodes syntax_tree visit)
49:        (= helpnode (AST:CreateNodeWithString
50:        (AST:GetForestRepresentationInstance graph domain_representation)
51:        354 `Vorname' AST:DefaultFormat))
52:        (AST:ReplaceNthChild (AST:GetParent global:marker) 1 helpnode)
53:     );;
54:   )local
55:  )action
56: )define
```

Zu beachten ist, dass bei der Erzeugung des neuen Knoten **`helpnode`** durch die Funktion **`AST:CreateNodeWithString`**, eine automatische Regelnummerie-

rung von DMS benutzt wird. Die Zahl 354 steht in diesem Fall für einen Knoten, der der identifier-Regel aus der Java-Grammatik zugeordnet ist.

Im Beispiel wird ersichtlich, dass diese Vorgehensweise schon bei relativ einfachen Ersetzungen zu sehr unübersichtlichen Programmen führen kann. Auf diese Art der Programmierung kann aber nicht völlig verzichtet werden, da die Vorbedingungen für die Refactorings nicht mit Hilfe der Patterns und Ersetzungsregeln überprüft werden können. Dies werden wir im nächsten Abschnitt anhand eines Beispiels sehen.

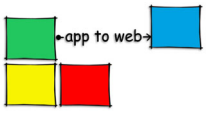
Das folgende Beispiel zeigt, wie man mit Hilfe von Patterns und Ersetzungsregeln, Syntaxbäume transformieren kann. Eine durch Kommata getrennte Aufzählung von Bezeichnern in einer Variablendeklaration wird in einzelne Variablendeklarationen für die Bezeichner aufgespalten. Also zum Beispiel wird **`public int k,l,b;`** zu **`public int k; public int l; public int b;.`**

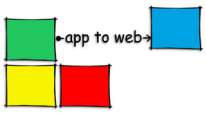
Abbildung 57:Beispiel für eine patternbasierteTransformation

```
rule
breakup_list(type:type,vdl:variable_declarator_list,vd:variable_declarator):
field_declaration -> class_body_declarations =
"public \type \vdl , \vd ; " -> "public \type \vd; public \type \vdl ;"
```

`Breakup_list` ist der Bezeichner dieser Ersetzungsregel. Die Parameter, die an die Regel übergeben werden, sind die getypten Teilbäume, die in dem Pattern, das im Syntaxbaum gefunden werden soll, vorkommen können. Die Parameterliste wird gefolgt von der Angabe des Typs des Syntaxbaums auf den die Ersetzungsregel angewendet werden kann (**`field_declaration`**) und den Typ des Syntaxbaums, der nach der Regelanwendung zurückgegeben wird (**`class_body_declarations`**). Im Rumpf der Regel wird auf der linken Seite des Pfeiles das Pattern, das gesucht werden soll und auf der rechten Seite des Pfeiles, das Pattern, das eingesetzt werden soll, angegeben. In diesem Fall wird eine Variablendeklaration mit dem Modifier **`public`** aufgespalten, wobei sich auf beiden Seiten die Variablen aus der Parameterliste wiederfinden.

Nun kann man durch solche Ersetzungsregeln keine semantischen Informationen aus dem Syntaxbaum gewinnen. Hierzu muss stets der prozedurale Ansatz verwendet werden. Die Interaktion zwischen PARLANSE-Programmen und dem Aufruf der Ersetzungsregeln ist wiederum durch eine API sichergestellt. Nur die Verwendung beider Vorgehensweisen stellt eine effiziente Implementierung der Refactoring sicher.





23 Zusammenfassung und Ausblick

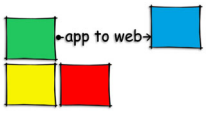
In diesem Dokument wurde der Prozess beschrieben, wie man aus Altsystemen gewonnene Komponentenkandidaten in Komponenten umwandeln kann. Dazu wurde zuerst der in diesem Dokument verwendete Begriff der Komponente definiert. Der Schwerpunkt dieses Dokumentes lag auf der Beschreibung von Gründen einen Komponentenkandidaten nachträglich zu modifizieren, und von Restrukturierungen, mit denen der Quelltext eines Systems entsprechend verändert werden kann. Es wurden drei Gründe identifiziert, einen Komponentenkandidaten oder dessen Quelltext anzupassen:

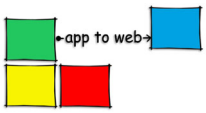
- Die automatische Partitionierung in Komponentenkandidaten entspricht nicht der Vorstellung des Entwicklers.
- Die *requires*- und *provides-Schnittstellen* sind nicht in geeigneter Form vorhanden.
- Der Komponentenkandidat soll an eine bestehende Komponenteninfrastruktur angepasst werden.

Die in diesem Dokument beschriebenen Restrukturierungen betreffen alle drei Bereiche. Sie können zwar nicht alle möglicherweise auftretenden Fälle abdecken, bieten aber Lösungen für immer wiederkehrende Probleme. Die Vorgehensweise zur Komponentifizierung wurde anhand einer realen Fallstudie beschrieben und es wurden mehrere Werkzeuge vorgestellt, die den Benutzer bei der Komponentifizierung unterstützen.

Der hier beschriebene Prozess kann noch in folgenden Bereichen verbessert werden:

- Noch können nicht alle beschriebenen Restrukturierungen werkzeuggestützt erfolgen. Die vorhandenen Werkzeuge sollten entsprechend erweitert werden. Zusätzlich sollten noch weitere allgemein einsetzbare Restrukturierungen gesammelt werden.
- Der Prozess zur Transformation wird noch nicht zusammenhängend durch Werkzeuge abgedeckt. Die einsetzbaren Werkzeuge zum Verständnis des Quelltextes, zur Berechnung der Komponentenkandidaten und zur Restrukturierung des Codes sollten ineinander verzahnt werden, und insbesondere mit einer gemeinsamen Visualisierung versehen werden, um die Durchführung der Transformation für den Benutzer so einfach wie möglich zu machen.





24 Literaturverzeichnis

[AAB+02] Michalis Anastasopoulos, Christoph Andriessens, Holger Bär, Christian Bunse, Jean-Francois Girard, Isabel John, Dirk Muthig, and Tim Romberg. Überblick über den Stand der Technik. Technical report, Applications2Web, December 2002.

[ABGS01] Christoph Andriessens, Markus Bauer, Jean-Francois Girard, and Olaf Seng. Redokumentation von Altsystemen. Technical report, Applications2Web, 2001.

[ACCL01] Lerina Aversano, Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Migrating Legacy Systems to the Web: an Experience Report. In *Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[Aea01] Colin Atkinson and Joachim Bayer et al. *Component-based Product Engineering with UML*. Addision Wesley, 2001.

[Bay01] Joachim Bayer. Wiederverwendung und Integration re-engineerter Komponenten. Technical report, Applications2Web, 2001.

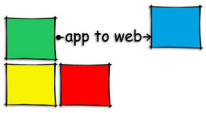
[BG02] Markus Bauer and Jean-Francois Girard. Identifikation von Komponentenkandidaten in Altsystemen. Technical report, Applications2Web, 2002.

[CBS01] Oliver Ciupke, Markus Bauer, and Olaf Seng. GOOSE - A tool set for analysing the design of object-oriented software systems. http://www.fzi.de/prost/tools/goose/, 2001.

[com] Compiere - Smart Open Source ERP + CRM Business Solution for the Small-Medium Enterprise. http://www.compiere.org.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[FS99] Martin Fowler and Kendall Scott. *UML Distilled*. Addison Wesley, 1999.



[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GK01] Thomas Genssler and Volker Kuttruff. Werkzeuggestützte Softwareadaption mit Inject/J. Technical report, 3. Workshop Reengineering, Bad Honnef, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akcsit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[Kut02] Volker Kuttruff. Ein Modell für invasive Softwareadaption. Master's thesis, Universität Karlsruhe, 2002.

[LH01] Andreas Ludwig and Dirk Heuzeroth. Recoder homepage. http://recoder.sourceforge.net/, 2001.

[Lot00] Tony Loton. Make an EJB from any Java Class with Java Reflection. *JavaWorld*, 2000.

[Mar02] Thomas Marz. Visualisierung von Softwaresystemen. Studienarbeit - Universität Karlsruhe, 2002.

[MH00] Richard Monson-Haefel. *Enterprise Java Beans*. O'Reilly, 2000.

[OMG] OMG. The OMG's CORBA Website. http://www.corba.org.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

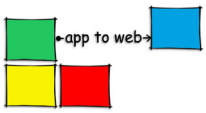
[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[SD] Inc. Semantic Designs. Semantic Designs, Inc. http://www.semdesigns.com.

[Sen02] Olaf Seng. Wiederverwendung objektorienterten Quelltexts. Technical report, Applications2Web, 2002.

[SGCH01] K. Sullivan, W. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE*, 2001.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.



[Tri01] Adrian Trifu. Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems. Master's thesis, Universität Karlsruhe, 2001.

[W3C] W3C. Scalable Vector Graphics (SVG) 1.0 Specification. http://www.w3.org/TR/SVG.

[W3C02a] W3C. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP, 2002.

[W3C02b] W3C. Web Service Definition Language (WSDL). http://www.w3.org/TR/wsdl, 2002.