# Von der Problemerkennung zur Problembehebung: 12 Jahre Softwaresanierung am FZI

Volker Kuttruff, Mircea Trifu und Peter Szulman
FZI Forschungszentrum Informatik, Karlsruhe, Germany
{kuttruff, mtrifu, szulman}@fzi.de

**Abstract:** Softwareentwicklung ist ein evolutionärer Prozess. Die Wartung und Weiterentwicklung existierender Softwaresysteme erfordert aufgrund der Größe heutiger Systeme systematische und automatisierbare Methoden. Die Erforschung solcher Methoden ist ein Schwerpunkt der Gruppe Programmstrukturen des FZI Forschungszentrum Informatik. Im folgenden Beitrag werden wir einen Überblick über unsere Forschungsarbeiten der letzten zwölf Jahre in diesem Bereich geben. Diese Arbeiten umfassen das ganze Spektrum der Softwaresanierung, angefangen bei Techniken des Reverse-Engineering und der Qualitätsbewertung über die Ableitung geeigneter Maßnahmen zur Überbrückung der Lücke zwischen Problemerkennung und Problembehebung bis hin zur automatisierten Reorganisation existierender Softwaresysteme.

## 1 Überblick

Softwareentwicklung ist ein evolutionärer und kein linearer Prozess, der dem sprichwörtlichen Wasserfall folgt. Vorhandene Software muss an neue Anforderungen angepasst oder in neue Systeme integriert werden. In der Gesamtsicht verteilen sich daher die weitaus größten Kosten auf die Softwareevolution. Nach dem Durchbruch der objektorientierten Sprachen in den 1990ern trat die Frage nach der Sanierung objektorientierter Systeme zunehmend in den Fokus sowohl wissenschaftlicher als auch industrieller Betrachtungen. Ab Mitte der 1990er war Softwaresanierung auch innerhalb des Bereichs Programmstrukturen bzw. innerhalb des jetzigen Nachfolgebereichs Software Engineering am FZI Forschungszentrum Informatik eines der zentralen Forschungsthemen.

Die wissenschaftlichen Arbeiten in diesem Forschungthema wurden in zahlreichen Forschungsprojekten vorangetrieben. Im durch die EU im Rahmen des Esprit-Programms geförderten Projektes FAMOOS (Laufzeit 1996-1999) [BBC+99] wurde der Grundstein für die automatische Erkennung von Problemstrukturen in objektorientierten Softwaresystemen gelegt. Erste Ansätze zur automatischen Reorganisation solcher Problemstrukturen wurden ebenfalls in FAMOOS entwickelt und im EU-Projekt TROOP (Laufzeit 1998-1999) weitergeführt. Die Anwendung und Verfeinerung der entwickelten Techniken zum Reengineering erfolgte im Rahmen des durch das BMWi geförderten InnoNet-Projektes APP2WEB (Laufzeit 2000-2002, `http://app2web.fzi.de`). Ziel dieses Projektes war die Bereitstellung von Techniken zur Aufbereitung bestehender Altanwendungen, um letztere für den Einsatz im WWW vorzubereiten. Mit der konsequenten Integration und Weiterentwicklung der bis dahin entstandenen Ansätze sowie einer softwaretechnischen

Stabilisierung der prototypisch erstellten Werkzeuge wurde im BMBF-Projekt COMPOBENCH (Laufzeit 2001-2004, `http://compobench.fzi.de`) begonnen. Diese Arbeiten wurden anschließend im ebenfalls vom BMBF geförderten Projekt QBENCH (Laufzeit 2003-2006, `http://www.qbench.de`) weitergeführt. Neben der Verbesserung der automatischen Erkennung von Problemstrukturen sowie der Behebung dieser Probleme durch automatisierte Reorganisationen wurde in diesen Projekten insbesondere auch die Fragestellung bearbeitet, wie man von Problemstrukturen zu geeigneten Lösungsstrukturen bzw. Reorganisationsoperationen gelangt, d. h. wie die Lücke zwischen Problemerkennung und Problembehebung überbrückt bzw. verringert werden kann. Abbildung 1 zeigt die zeitliche Anordnung der vorgestellten Projekte sowie die in diesem Beitrag vorgestellten Werkzeuge des FZI.

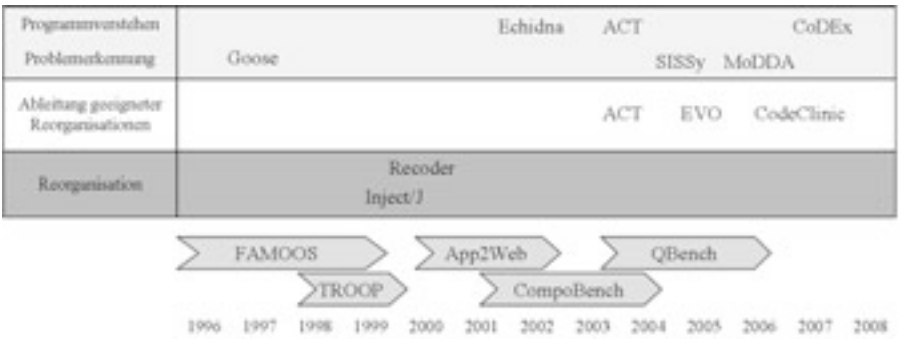


Abbildung 1: Überblick über Projekte und Werkzeuge des FZI zum Thema Reengineering

Nicht zuletzt aufgrund unserer Wurzeln, welche im Übersetzerbau zu finden sind, basieren unsere Arbeiten meist auf dem Quelltext eines existierenden Systems. Dies hängt auch damit zusammen, dass für existierende Systeme im Allgemeinen nur der Quelltext das einzige konsistente Modell des Systems darstellt. Weiterführende Entwurfsdokumente und Dokumentationen werden, sofern sie überhaupt vorhanden sind, im Laufe der Weiterentwicklung fast immer nicht ausreichend gepflegt. Aufgrund der Größe heutiger Systeme von mehreren Millionen Zeilen Quelltext ist eine Automatisierung der von uns bearbeiteten Reengineering-Themenbereiche Problemerkennung, Reorganisation von Software, Ableitung geeigneter Reorganisationsoperationen sowie Qualitätssicherung unabdingbar. In diesem Beitrag werden wir zeigen, welche automatisierten Ansätze wir in den letzten zwölf Jahren erarbeitet haben.

Die verbleibenden Teile dieses Beitrags gliedern sich wie folgt. Im nächsten Abschnitt 2 zeigen wir, wie die für weitere Analysen und Reengineeringschritte benötigten Informationen aus existierenden Softwaresystemen gewonnen werden können. Abschnitt 3 stellt unsere Arbeiten im Bereich der automatisierten Reorganisation von Softwaresystemen vor. Mit der Erkennung von Problemen im Rahmen einer entwicklungsbegleitenden Qualitätssicherung beschäftigt sich Abschnitt 4. Unsere Arbeiten zur Überbrückung der Lücke zwischen Problemerkennung und Problembehebung werden in Abschnitt 5 vorgestellt. Schließlich wird im letzten Abschnitt 6 noch eine kurze Zusammenfassung gegeben.

# 2 Reverse Engineering

## 2.1 Faktenextraktion

Die Faktenextraktion ist die wichtigste Basistechnik für die im Reengineering anfallenden Aktivitäten, sei es das Programmverstehen, automatisierte Qualitätssicherung oder die werkzeuggestützte Problembehebung. Mit der Hilfe der Faktenextraktion kann der Quelltext eines Softwaresystems in ein Systemmodell überführt werden, das dessen Artefakte (Klassen, Methoden, Parameter, usw.) und deren Beziehungen (Methodenaufrufe, Vererbungsbeziehungen, usw.) enthält. Die Faktenextraktion bedient sich der aus Übersetzungsvorgängen bekannten Schritte des Übersetzerfrontends: lexikalische, syntaktische und semantische Analyse. Obwohl die Theorie der Faktenextraktion sauber definiert ist, ist der Weg von der Theorie zu in der industriellen Praxis einsetzbaren Werkzeugen sehr aufwendig. Die Faktenextraktoren mussten so konzipiert und implementiert werden, dass sie mit sehr großen Systemen auch dann noch umgehen können, falls diese aus mehreren Millionen Zeilen Quelltext bestehen. Daher wurde bei den Faktenextraktoren besonderes Augenmerk auf die Aspekte Skalierbarkeit und Robustheit gelegt. Die Implementierung von stabilen und skalierbaren Faktenextraktoren stellte keinen zu unterschätzenden Aufwand dar: Die heute stabil laufenden Java, C++ und Delphi Faktenextraktoren wurden im Laufe mehrerer Jahre entwickelt und extrahieren Informationen auf verschiedenen Abstraktionsniveaus, angefangen bei grobgranularen Einheiten wie Klassen bis hin zu feingranularen Strukturen wie abstrakte Kontrollflussgraphen.

## 2.2 Programmverstehen

Das Programmverstehen stellt einen wesentlichen Teil der Wartungsaktivitäten dar. Außer der im nächsten Abschnitt vorgestellten Visualisierung entwickelten wir Ansätze, die durch die entsprechende Aufbereitung einer gegebenen Softwarestruktur zu einem verbesserten Programmverstehen beitragen.

Das Werkzeug ACT [Bau06] wurde primär mit dem Ziel entwickelt, das Verständnis eines Softwaresystem zu erleichtern. Der verfolgte Ansatz kombiniert Mustersuchen mit Ballungsanalysen um strukturprägende Systembausteine zu identifizieren und diese anschließend so zu gruppieren, dass eine möglichst gute Unterteilung in Teilsysteme entsteht. Die günstige Zerlegung des Systems bietet eine übersichtliche Sicht auf die Systemstruktur und fördert dadurch das Programmverstehen. Da der Ansatz auch zur Überbrückung der Lücke zwischen Problemerkennung und Problembehebung genutzt werden kann, wird er in Abschnitt 5.1 nochmals genauer betrachtet.

Ein weiterer Ansatz, der das Softwareverständnis unterstützt wurde in [Tri08] vorgestellt. Dieser Ansatz benutzt Datenflussinformationen zur Erkennung funktionaler Belange (engl. concerns) in objektorientierter Software. Ein funktionaler Belang ist als eine datenorientierte Sicht definiert und enthält die notwendige Funktionalität zur Berechnung einer vorgegebener Menge verwandter Ausgaben. Diese Ausgaben heißen Informationssenken

und können sowohl vom Benutzer spezifiziert als auch automatisch erkannt werden. Die Belangerkennung erfolgt vollautomatisch durch die Erweiterung der Informationssenken entlang den eingehenden Datenflusspfaden. Implementiert auf Basis von RECODER (siehe Abschnitt 3.1), bietet das Werkzeug CODEX eine Plattform zur Erkennung und Analyse solcher Belange an.

### 2.3 Softwarevisualisierung

Die Einsatzmöglichkeiten von Softwarevisualisierungen beim Reengineering sind sehr vielfältig. Größtenteils setzen wir die verschiedenen Visualisierungstechniken zur Unterstützung des Entwicklers beim Programmverstehen und bei der Problemerkennung ein. Grundlage für die Darstellung ist ein Graph, der aus dem Systemmodell gewonnen wird.

Die Visualisierung von Software zur Unterstützung des Programmverstehens stellte eine Herausforderung dar. Schon für kleine Systeme wird die Darstellung der vollständigen Struktur viel zu unübersichtlich, um von Nutzen zu sein. Die Lösung hierfür ist es, dem Betrachter abstrakte Sichten zur Verfügung zu stellen. Auf Basis von Graphenhomomorphismen wurden die Operationen Aggregation und Selektion definiert, die als Ergebnisse abstrahierte Sichten auf Systemmodelle liefern. Durch diese Sichtenbildung kann man eine Struktur so aufbereiten, dass von nicht interessierenden Aspekten abstrahiert wird [Ciu02]. Das am FZI entwickelte Werkzeug ECHIDNA dient zur Visualisierung großer Softwaresysteme und stellt neben der Möglichkeit der Sichtenbildung eine Vielzahl verschiedener Layout-Algorithmen zur Aufbereitung der Graphdarstellung zur Verfügung.

Visualisierungen setzen wir nicht nur zur Unterstützung des Programmverstehens ein, sondern auch für die Problemerkennung. Das QBENCH-Projekt verfolgte den Ansatz einer semi-automatischen Problemerkennung. Die dabei eingesetzten Visualisierungen illustrieren identifizierte Anomalien bzw. Problemstellen. Die graphische Darstellung soll zum einen das Verständnis der konkreten Probleminstanz verbessern bzw. erleichtern. Dies kann z. B. dadurch erreicht werden, dass die Vernetzung der betroffenen Artefakte visualisiert wird. Zum anderen kann durch geeignete Visualisierung die Priorisierung der Probleminstanzen bezüglich der Problemschwere erleichtert werden.

## 3 Reorganisation

Die Reorganisation existierender Software kann aus vielen Gründen notwendig sein, sei es, um eine Software vor einer Weiterentwicklung zu sanieren, oder um neue Anforderungen umzusetzen. Historisch betrachtet hat sich das FZI zunächst mit der Sanierung spezieller Entwurfsprobleme beschäftigt. Erste Arbeiten hierzu gehen auf das EU-Projekt FAMOOS zurück, wo untersucht wurde, wie ausgewählte Entwurfsstrukturen transformativ in andere Strukturen überführt werden können [GS99]. Ebenfalls in dieser Zeit wurden Ansätze zur Betrachtung von Entwurfsmustern als Operatoren entwickelt [GSMZ98]. Ein solcher Entwurfsmusteroperator beschreibt, wie eine gegebene Ausgangsstruktur mit Hil-

fe einer Sequenz einfacher Restrukturierungen in eine Struktur überführt werden kann, welche ein konkretes Entwurfsmuster implementiert [Zim97]. Dieses Konzept wurde im EU-Projekt TROOP weiterentwickelt und schließlich in die damalige Version des UML-Modellierungswerkzeugs TOGETHER/J integriert.

Bis zu diesem Zeitpunkt wurden die erarbeiteten Konzepte mangels geeigneter Infrastrukturen meist nur prototypisch umgesetzt. Um die Konzepte verallgemeinern zu können und um den Entwicklungsaufwand für derartige automatisierte Reorganisationsoperationen zu reduzieren, wurde zusammen mit unserem Partnerinstitut, dem Institut für Datenorganisation und Programmstrukturen der Universität Karlsruhe, eine gemeinsame stabile Infrastruktur zu automatisierten Reorganisation von Quelltext entwickelt. Dies führte zur Entwicklung des Rahmenwerks RECODER [Lud02] und dem darauf aufsetzenden Werkzeug INJECT/J [Gen04], wobei die Entwicklung insbesondere in den Projekten TROOP und COMPOBENCH vorangetrieben wurde. Auf diese Werkzeuge wird in den beiden folgenden Abschnitten genauer eingegangen.

### 3.1 RECODER

RECODER (`http://recoder.sf.net`) ist ein Rahmenwerk zur Quellcode-Metaprogrammierung. Hierfür stellt RECODER Möglichkeiten zur Analyse und Transformation von JAVA-Quellen bereit. Das System erlaubt das Einlesen von JAVA-Quellcode, welches in Form eines attributierten Strukturbaums dargestellt wird. Weiterhin stellt RECODER Funktionen zur Manipulation dieses Strukturbaums zur Verfügung, ebenso wie eine Infrastruktur, die damit verbundene Aktivitäten wie Transaktionen und Rücksetzen von Änderungen unterstützt. Nach der Transformation des attributierten Strukturbaums können die ursprünglichen Quellen modulo der Änderungen wieder generiert werden, wobei die ursprüngliche Formatierung des Quelltexts inklusive Kommentare erhalten bleibt. Abbildung 2 zeigt dieses Vorgehen nochmals.

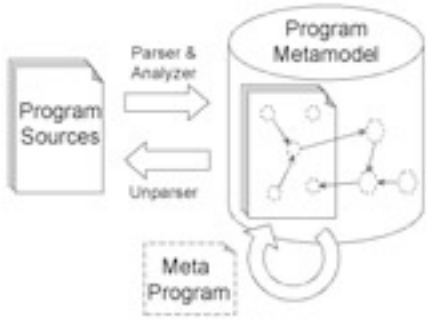


Abbildung 2: Prozess der Metaprogrammierung mit RECODER

RECODER stellt bis auf wenige Ausnahmen ein vollständiges Übersetzer-Frontend dar. Neben einer lexikalischen und syntaktischen Analyse werden zusätzlich noch Names-,

Typ- und Querverweisanalysen unterstützt. Letztere sind dabei soweit wie möglich inkrementell, d. h. lokale Änderungen des attributierten Strukturbaums führen nicht notwendigerweise wieder zu einer Neuberechnung aller Attribute. Die Ergebnisse der Namens-, Typ- und Querverweisanalysen werden innerhalb von RECODER durch sog. *Services* zur Verfügung gestellt. Ebenfalls als Service ist ein Änderungsvermittler umgesetzt, welchem Änderungen am Strukturbaum mitgeteilt werden müssen. Aus diesen Informationen werden notwendige inkrementelle Analysen sowie die zum Rücksetzen einer Sequenz von Änderungen notwendigen Schritte automatisch abgeleitet.

Das Modell, welches RECODER zu Grunde liegt, ist aus dem Sprachmodell von JAVA abgeleitet, d. h. es werden alle Sprachelemente unterstützt und abgebildet, angefangen bei Klassendeklarationen bis hin zu Ausdrücken und einzelnen Operatoren. Wie bereits erwähnt, werden im Gegensatz zu einem reinen Übersetzer-Frontend auch Kommentare berücksichtigt und durch das Modell unterstützt. Das Modell spezifiziert darüber hinaus Primitive zur lokalen Manipulation eines konkreten Strukturbaums. Diese Primitive sind das Einfügen neuer Teilbäume, das Entfernen existierender Teilbäume und als Zusammenfassung dieser Primitive das Ersetzen eines existierenden Teilbaums durch einen anderen Teilbaum. Mit Hilfe dieser Primitiven lassen sich prinzipiell alle Änderungen am Quelltext eines Programms durchführen, die auch durch manuelle Änderungen durchführbar sind. Die hierzu notwendige Sequenz von lokalen Änderungen muss bei Nutzung von RECODER in Form eines JAVA-Programms erfolgen. Im Gegensatz zu reinen Graphersetzungssystemen kann die Steuerung dieses Metaprogramms hierbei auf Basis der durch die Services bereitgestellten Analyseergebnisse erfolgen.

RECODER skaliert gut für große Systeme. Der Ressourcenverbrauch ist in etwa mit dem entsprechender Übersetzer zu vergleichen. RECODER bildet für zahlreiche weitere Reengineering-Werkzeuge des FZI die benötigte Schnittstelle zum Quelltext eines existierenden Systems, z. B. für die JAVA-Faktenextraktoren von GOOSE und SISSY. Andere Werkzeuge, wie das im nächsten Abschnitt vorgestellte INJECT/J, nutzen darüber hinaus auch die Möglichkeit zur programmgesteuerten Manipulation von Quelltext.

### 3.2 INJECT/J

INJECT/J (`http://injectj.fzi.de`) ist eine domänenspezifische, operationale, dynamisch typisierte Skriptsprache sowie ein Werkzeug zur invasiven Softwarekomposition und -adaption. Ziel der Sprache und des Werkzeugs ist die allgemeine Adaption von JAVA-Programmen. Es werden Möglichkeiten angeboten, um die Analysen und Transformationen von RECODER zu steuern. Hierzu stellt INJECT/J die folgenden grundlegenden Konzepte zur Verfügung:

**Navigation** im INJECT/J Adaptionsmodell. Dieses Adaptionsmodell stellt im Vergleich zum feingranularen RECODER-Modell eine vereinfachte Sicht auf das zu adaptierende Programm dar, da insbesondere feingranulare Elemente wie Ausdrücke und Operatoren vor dem Nutzer verborgen werden. Die Navigation innerhalb des Adaptionsmodells erfolgt mit Hilfe von Namensmustern und Erkennungsmustern. Letztere sind Graphmuster,

welche unterschiedliche Modellelemente basierend auf strukturellen und quantitativen Eigenschaften wie z. B. Metrikwerten selektieren. Namensmuster werden benutzt, um benamte Modellelemente mit Hilfe regulärer Ausdrücke über diesen Namen zu selektieren. Zum Beispiel selektiert das Namensmuster `classes(mypackage.*)` alle Klassen innerhalb des Pakets `mypackage`, während `method(**.*(int))` alle Methoden mit einem Parameter vom Typ `int` auswählt.

**Transformationen** werden benutzt, um das Adaptionsmodell zu manipulieren. INJECT/J stellt hierzu sog. *semantische Transformationen* zur Verfügung. Im Gegensatz zu rein syntaktischen Transformationen, wie sie RECODER bietet, fassen semantische Transformationen Gruppen von semantisch zusammengehörenden Operationen zusammen und verbergen diese vor einer einfachen Schnittstelle. Falls notwendig werden automatisch weitere Transformationen wie das Abflachen von Ausdrücken durchgeführt. Semantische Transformationen garantieren mindestens die Korrektheit bezüglich der Syntax und der statischen Semantik der Programmiersprache. Diese und weitere Eigenschaften werden durch Vor- und Nachbedingungen überwacht. INJECT/J bietet eine ganze Reihe solch semantischer Transformationen in Form einer Bibliothek, wobei diese Transformationen einfache Restrukturierungen (z. B. `rename`) bis hin zu aus dem aspektorientierten Programmieren bekannte Operationen (z. B. `beforeAccess`, `afterFailure`) umfassen.

**Kontrollstrukturen** wie Schleifen und Verzweigungen werden benutzt, um die Transformation zu steuern. Die Bedingungen dieser Kontrollstrukturen können dabei Ergebnisse von Modellanfragen nutzen. Letztere sind seiteneffektfreie Prädikate und Funktionen über dem Adaptionsmodell.

**Interaktion** mit dem Nutzer kann erfolgen, um eine Transformation interaktiv zu parametrisieren. Ein einfaches Beispiel hierfür ist die Frage nach einem Namen für eine neue Klasse oder temporäre Variable.

Mit Hilfe dieser Konzepte und der ausdrucksstarken Skriptsprache lassen sich komplexe Transformationen einfach spezifizieren. Wiederverwendbare Transformationen lassen sich einfach in das System integrieren. INJECT/J bietet einen Erweiterungsmechanismus, um neue semantische Transformationen und Modellanfragen bereitzustellen. Diese können entweder auf Basis der INJECT/J-Skriptsprache oder direkt in Java implementiert werden, wobei letztere direkten Zugriff auf das RECODER-Modell haben.

# 4 Qualitätsicherung

Die ersten Schritte im Bereich der technischen Qualitätssicherung haben ihre Wurzeln im EU-Projekt FAMOOS, in dem zum ersten Mal am FZI die Sanierung großer objektorientierter Systeme erforscht wurde. Wichtige Teile davon sind die Gewinnung und Untersuchung von Softwarestrukturen. Der Zweck der Untersuchung war zuerst, das Softwareverständnis zu erleichtern, aber basierend auf den Ergebnissen dieses Projekts entstand eine Reihe von Folgearbeiten im Themenbereich automatische Erkennung von Strukturproblemen in Softwaresystemen [Ciu02]. Die identifizierten Strukturprobleme sind zum Teil aus der Literatur gesammelt, z. B. die bekannten Bad-Smells von Fowler [Fow99]

oder die aus [BMM98] bekannten Anti-Patterns. Zum anderen Teil wurden sie als Verletzungen anerkannter Entwurfsprinzipien [Rie96] definiert. Die Umsetzung der Problemerkennung erfolgt mit Hilfe von so genannten Erkennungsstrategien [Mar01], die eine semi-formale Definition einer heuristischen Regel basierend auf Softwaremaßen erfasst. Erkennungsstrategien waren zuerst als Prolog-Regel im Werkzeug GOOSE formalisiert. GOOSE wurde für die Untersuchung objektorientierter Systeme gebaut und war in der Lage, Quellcode aus unterschiedlichen Sprachen in einem einfachen abstrakten Modell abzubilden und in einer Prolog-Fakten-Datenbank zu speichern. Die tatsächliche Erkennung erfolgt in GOOSE durch die von der Prolog-Inferenzmaschine durchgeführte Mustersuche.

Weil Strukturprobleme einen negativen Einfluss auf die innere Qualität eines Softwaresystems haben, kam die Idee, diese Qualität anhand von gefundenen Strukturproblemen zu bewerten. Je mehr Strukturprobleme ein System hat, desto schlechter ist seine innere Qualität. Innere Qualität bezieht sich auf die technische Qualität des Systems und die Qualität des Entwurfs. In [Mar02] wurde ein neues Qualitätsmodell eingeführt, das die klassische Verfeinerung des Qualitätsbegriffs nach ISO9126 mit komplexen Problemmustern verbindet. Gegenüber den alten FCM-Qualitätsmodellen, die die abstrakten Qualitätseigenschaften direkt mit Softwaremaßen verbinden, hat ein solcher Ansatz den Vorteil, dass er die Verbesserung der Qualität vereinfacht. Die Strukturprobleme sind viel aussagekräftiger als die reinen Softwaremaße und können schon Hinweise über sinnvolle Behebungsmaßnahmen geben, die zu einer Qualitätssteigerung führen.

Das BMBF-Forschungsprojekt QBENCH ist einen Schritt weiter gegangen und hatte sich als Ziel gesetzt, einen durchgängigen werkzeuggestützten Qualitätssicherungsprozess zu definieren, der sowohl die Problemerkennung als auch die Problembehebung abdeckt. Im Rahmen dieses Projekts wurden die Erkennungsstrategien verbessert und zum industriellen Einsatz gebracht. Das in QBENCH entstandene Werkzeug SISSY ist heute ein eigenständiges Open-Source-Projekt und wird in vielen Softwareunternehmen regelmäßig eingesetzt. Auf dieses Werkzeug wird im folgenden Abschnitt 4.1 genauer eingegangen.

Aus QBENCH haben sich mehrere Ideen entwickelt. Eine davon ist die Qualitätssicherung modellgetrieben entwickelter Softwaresysteme, die in [And07] adressiert wurde. Die größte Herausforderung hier war die Lokalisierung der Strukturprobleme auf der richtigen Modellebene und die Nachvollziehbarkeit der Verbindungen der Modellelemente zwischen den Modellebenen. Die von SISSY bekannten Problemmuster wurden angepasst und neue Strukturprobleme für modellgetrieben entwickelte Software wurden definiert. Der Ansatz wurde in Form einer Erweiterung von SISSY namens MODDA implementiert.

### 4.1 SISSY

Unser aktuelles Werkzeug zur strukturellen Untersuchung von Softwaresystemen (SISSY) ist eine Open-Source-Plattform zur Erkennung von Problemmusterinstanzen. SISSY ist nicht nur eine skalierbare Weiterentwicklung von GOOSE, sondern ein ganz neuer Entwurf. Das Metamodell hinter SISSY ist viel detaillierter und ermöglicht viel komplexere sprachunabhängige Analysen.

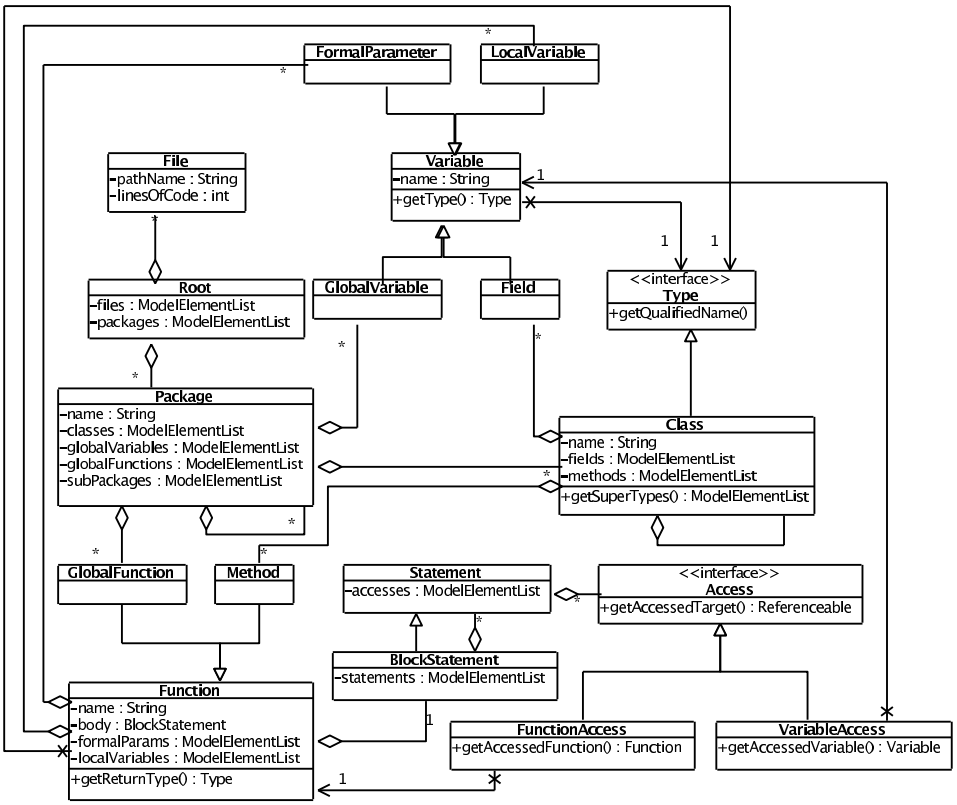


Abbildung 3: Das Metamodell des Werkzeugs SISSY

Abbildung 3 stellt die wichtigsten Modellelemente des Metamodells dar. Alle Modellelemente sind in einem Aggregationsbaum angeordnet, dessen Wurzel die Klasse Root ist. Die pro Metamodellinstanz einzige Instanz dieser Klasse fungiert als Einstiegspunkt bezüglich der Navigation in diesem Modell. Sie bündelt in Form eines Containers alle freien Strukturierungsentitäten wie Pakete und Dateien. Obwohl Dateien keine Sprachkonstrukte sind, werden sie explizit durch die Klasse File im Modell vertreten, um die Verbindung der Elemente zum Dateisystem einfach zurückverfolgen zu können. Dafür hat jedes Modellelement, das innerhalb einer Datei enthalten sein kann, eine Verbindung zum enthaltenden File-Objekt. Die Klasse Package modelliert einen Namensraum mit Sichtbarkeiten ähnlich zum Paketkonstrukt aus Java, das andere Namensräume, Klassen, globale Variablen bzw. globale Funktionen enthalten kann. Klassen werden im Modell durch die Klasse Class modelliert, die als Untertyp des allgemeineren Typs Type definiert wird. Eine Klasse hat einen Namen, kann Attribute und Methoden enthalten und befindet sich in verschiedenen Vererbungsbeziehungen mit anderen Klassen. Die gemeinsame Funktionalität aus Method und GlobalFunction wurde in der Klasse Function ausfaktorisiert. Eine Funktionsdefinition spezifiziert den Namen der Funktion, die formalen Parameter, den Rückgabetyp und, im Fall einer konkreten Funktion, den Funktionsrumpf und die

lokalen Variablen. Wie im Fall der Klasse Function wurde die gemeinsame Funktionalität aus Field, FormalParameter, LocalVariable und GlobalVariable in die Klasse Variable ausfaktorisiert. Alle Variablen haben einen Namen und einen deklarierten Typ. Der Funktionsrumpf enthält Anweisungen, modelliert durch die Klasse Statement, die wiederum Zugriffe enthält. Die Klasse Access und ihre Unterklassen modellieren die verschiedenen Arten von Zugriffen, die in einer OO-Sprache spezifiziert werden können, wie z. B. Zugriffe auf Funktionen oder Variablen.

Der Quelltext eines Softwaresystems kann mit Hilfe von Faktenextraktoren für drei verbreitete objektorientierte Sprachen (Java, C++ und Delphi) in das Modell von SISSY überführt werden, wobei die konkreten Sprachkonstrukte auf die abstrakten sprachübergreifenden Modellelemente des Metamodells abgebildet werden.

SISSY kann sowohl einfache Metriken wie z. B. Anzahl der Quelltextzeilen, Klassen, Methoden, Attribute als auch komplexere Softwaremetriken wie z. B. Vererbungstiefe, zyklomatische Komplexität, Kopplung und Kohäsion ermitteln und mit Hilfe von Heuristiken nach strukturellen Problemen bzw. Problemmustern suchen. SISSY bietet allerdings noch viel mehr an. Es implementiert die Basistechnologie für modellbasierte Transformationssimulationen, die für Auswirkungsanalyse, Problembehebung und strukturelle Optimierungen benutzt werden kann.

Um die Vorteile einer Standardsprache wie SQL auszunutzen, lässt sich das oben vorgestellte Metamodell in eine SQL-Datenbank exportieren, deren Schema speziell dafür entwickelt wurde. Hierdurch lassen sich die Analysen bzw. die Suche nach Problemmustern einfach und effizient als SQL-Anfragen implementieren. SISSY enthält SQL-Anfragen für 52 Problemmuster, die die gesamte Bandbreite von Bad-Smells und Entwurfsprinzipien von der Architekturebene bis zur Implementierungsebene abdecken.

# 5 Überbrückung der Lücke zwischen Problemerkennung und Problembehebung

Während die Erkennung von Problemen in Softwaresystemen sowie die Behebung der Probleme durch Reorganisation dieser Systeme mit den vorgestellten Ansätzen und Werkzeugen bereits zu großen Teilen automatisierbar sind, gilt dies für die Überbrückung der Lücke, d. h. die Bereitstellung von Lösungen für identifizierte Probleme, nicht. Auch wenn für einige einfache Fälle eine eindeutige Abbildung von Problemstruktur zu Lösungsstruktur angegeben werden kann, so erscheint es aus heutiger Sicht illusorisch, diese Lücke allgemeingültig vollständig automatisiert zu überbrücken. Dies liegt letztlich daran, dass eine Problemstruktur letztlich nur *symptomatisch* ist: sie zeigt das Vorhandensein eines Problems an, aber nicht, *warum* das Problem existiert. Ohne genauere Betrachtung des Grundes einer Problemstruktur kann keine eindeutige Lösungsstruktur vorgeschlagen werden. Der Grund kann allerdings von vielen Parametern wie zum Beispiel Projektkontext und -anforderungen abhängen.

Die endgültige Entscheidung, welche Lösungsstruktur am geeignetsten ist, muss daher vom Softwareingenieur getroffen werden. Aber auch falls diese Abbildung vollständig au-

tomatisierbar wäre, so zeigt die Erfahrung, dass ein Softwareingenieur einem vollständig automatisierten Prozess eher ablehnend gegenübersteht, da er hierdurch die Kontrolle über Änderungen am Quelltext des Systems verlieren würde.

Auch wenn die Überbrückung der Lücke nicht vollständig automatisierbar erscheint, so kann ein Softwareingenieur durch Einsatz geeigneter Verfahren dennoch unterstützt werden, indem automatisch geeignete Lösungen vorgeschlagen werden, welche der Softwareingenieur als letzte Instanz nur noch bestätigen oder ablehnen muss. Hierdurch lässt sich die Lücke zwar nicht vollständig schließen, aber doch signifikant verringern. Im Folgenden zeigen wir daher Ansätze und Werkzeuge, welche in den letzten Jahren am FZI entstanden sind, um diese Lücke zu verringern. Hierbei wurden zwei Stoßrichtungen verfolgt: optimierende und analytische Ansätze. In den Abschnitten 5.1 und 5.2 werden zwei optimierende Ansätze vorgestellt, welche basierend auf Metriken und Entwurfsheuristiken versuchen, bessere Systemzerlegungen zu bestimmen, in denen die ursprünglichen Probleme nicht mehr vorhanden sind. Im Abschnitt 5.3 wird ein analytischer Ansatz vorgestellt, der versucht, Probleme genauer zu analysieren und darauf aufbauend geeignete Lösungen zu bestimmen.

## 5.1 ACT

Mit Hilfe des Werkzeugs ACT lassen sich Systemstrukturen aus der Implementierung objektorientierter Softwaresysteme gewinnen. Hierzu bedient sich ACT eines *hybriden* Verfahrens, welcher Mustersuchen und Ballungsanalysen kombiniert [Bau06].

Ausgangspunkt des Verfahrens ist ein Modell eines Softwaresystems, welches mit Hilfe von Faktenextraktoren aus dem Quelltext gewonnen wird. Dieses Modell bildet zentrale Elemente wie Klassen und Methoden sowie die Beziehungen zwischen diesen Elementen ab. In einem ersten Schritt wird nach bestimmten Mustern innerhalb des Modells gesucht, welche eine bestimmte Bedeutung für die Architektur des Softwaresystems besitzen. Anschließend werden die Elemente des Systems mit Hilfe von Algorithmen zur Ballungsanalyse so gruppiert, dass die Elemente eines Systems möglichst gut gekapselte Teilsysteme bilden. Hierbei werden die Rollen der einzelnen Elemente für die Architektur mit ausgewertet. Abbildung 4 zeigt das Verfahren im Überblick.

ACT wurde primär mit dem Ziel entwickelt, das Verständnis eines Softwaresystems zu verbessern. Trotzdem lässt sich das Verfahren auch dazu nutzen, die Lücke zwischen Problemerkennung und Problembehebung zu verringern. Die Zerlegung des Systems erfolgt auf Grundlage von Heuristiken, welche einen guten Entwurf beschreiben. Unterscheidet sich die durch ACT durchgeführte Zerlegung des Systems signifikant von der tatsächlich sichtbaren Zerlegung (z. B. Zerlegung in Pakete), so deutet dies auf ein mögliches Problem hin. Durch die vorgeschlagene Zerlegung liefert ACT eine geeignetere Zielstruktur, wobei sich die notwendigen Reorganisationsoperationen durch Vergleich der tatsächlich vorgefundenen mit der vorgeschlagenen Zerlegung ableiten lassen.

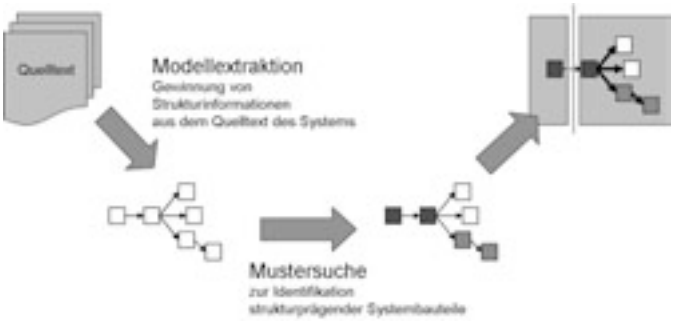


Abbildung 4: Systemzerlegung durch Mustersuche und Ballungsanalyse

## 5.2 EVO

Das in [SSB06] beschriebene Verfahren zur suchbasierten Behebung von Problemmusterinstanzen ist ein Weg, um den Softwareingenieur bei der Aufgabe der Bestimmung konkreter Reorganisationen zu unterstützen. Es greift Konzepte von ACT auf und entwickelt sie im Hinblick auf die Reduktion der Lücke zwischen Problemerkennung und Problembehebung weiter. Das Verfahren bestimmt Restrukturierungen, die eine existierende Systemstruktur in eine verhaltensgleiche Struktur mit weniger Problemmusterinstanzen überführen. Es arbeitet nicht direkt auf dem Quelltext eines Systems, sondern simuliert die Restrukturierungen auf einem abstrakten Modell des Quelltexts. Somit verbleibt die endgültige Entscheidung, ob eine Restrukturierung ausgeführt werden soll, beim Softwareingenieur.

Die Entwurfsheuristiken zur Bewertung der strukturellen Qualität werden als Zielfunktion formuliert. Mit ihr werden folgende drei Entwurfsprinzipien erfasst: Geheimnisprinzip und Schnittstellen, Kopplung und Kohäsion sowie Komplexität. Diese Prinzipien sind nicht direkt messbar, wie üblich wird ihre Abwesenheit mit Hilfe von Quelltextmetriken und Graphmustersuchen zur Identifikation von Strukturproblemen bestimmt. Die Zielfunktion bildet die strukturelle Qualität eines Systems auf eine reelle Zahl ab, indem sie die Ergebnisse der Metriken und die Anzahl der gefundenen Strukturprobleme auf die Größe des Systems normiert, gewichtet und anschließend linear kombiniert.

Der im Werkzeug EVO verfolgte Ansatz betrachtet das Problem als Optimierungsaufgabe, die mit einem evolutionären Algorithmus bearbeitet wird. Zur Simulation von Quelltextrestrukturierungen unterstützt das Verfahren insgesamt 10 Modellrestrukturierungen, wie zum Beispiel das Verschieben, Aufteilen und Verschmelzen von Klassen sowie das Verschieben von Methoden und Attributen. Auf einer Population von Modellen werden zufallsgesteuert Modellrestrukturierungen ausgeführt (*Mutation* eines Modells) oder die bisherigen Modellrestrukturierungen zweier Modelle geeignet miteinander kombiniert (*Rekombination*). Die Ergebnisse werden anschließend anhand der Werte der Zielfunktion verglichen. Eine Auswahl der Population für die nächste Iteration des evolutionären Algorithmus erfolgt durch eine Turnierselektion. Abbildung 5 zeigt das Verfahren im Überblick.

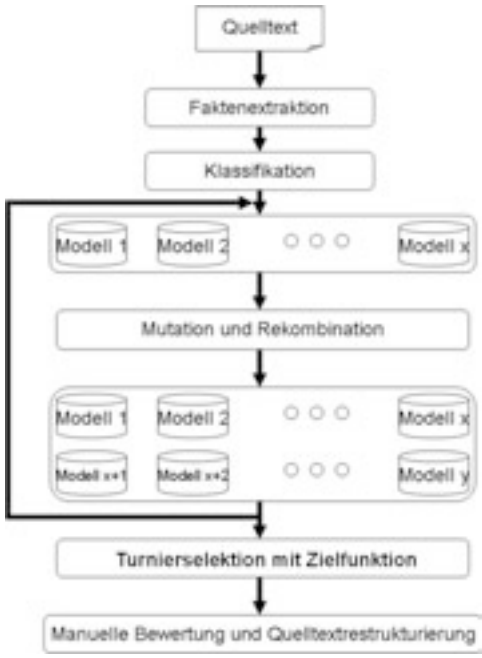


Abbildung 5: Ablauf des suchbasierten Verfahrens mit EVO

### 5.3 CODECLINIC

Ein weiterer Ansatz zur Verringerung der Lücke zwischen Problemerkennung und Problembehebung liegt dem Werkzeug CODECLINIC [TR07] zu Grunde. Die zentrale Idee dieses Ansatzes ist die Wiederherstellung einer Verbindung zwischen einer symptomatischen Problemstruktur und der Ursache dieses Problems. Die verfolgte Annahme ist, dass falls die Ursache einer Problemstruktur nur genau genug ermittelt werden kann, sich die Menge der ursprünglichen potentiellen Lösungen des Problems auf ein Element einschränken lässt.

Um von einer Problemstruktur zur Ursache des Problems zu gelangen, wird neben dem eigentlichen Problem noch nach weiteren *Indikatoren* gesucht, welche auf die eigentliche Entwurfsabsicht und damit auch auf die Ursache schließen lassen. Indikatoren sind hierbei zunächst keine problematischen Strukturen, sondern nur (strukturelle) Eigenschaften des Systems, welche auf eine bestimmte Entwurfsabsicht schließen lassen. Die Erkennung und Behebung von Problemen erfolgt in CODECLINIC aufbauend auf sog. Restrukturierungsmustern. Diese fassen die folgenden Beschreibungen zusammen: Entwurfsproblem, Indikatoren, Checkliste zur Bestätigung der Zielstruktur sowie eine das Entwurfsproblem behebende Zielstruktur. CODECLINIC wurde als Eclipse-Plugin umgesetzt. Es bedient sich dabei eines Katalogs von Restrukturierungsmustern.

# 6 Zusammenfassung

In den letzten zwölf Jahren habe wir uns in verschiedensten Projekten mit verschiedensten Facetten des Software-Reengineering befasst. In dieser Zeit sind zahlreiche Ansätze und Werkzeuge entstanden, welche den Prozess vom Verstehen der Software, der Erkennung von Problemstrukturen über die Ableitung geeigneter Lösungen bis hin zur automatisierten Reorganisation der Software abdecken.

Insbesondere die Ansätze zur Ermittlung von Problemstrukturen (GOOSE, SISSY) wurden bereits mehrfach an großen realen Softwaresystemen mit Erfolg eingesetzt. Während anfangs die Skepsis gegenüber automatischen Analysen groß war, so ist dies heute nicht mehr der Fall bzw. automatisierte Analysen werden gefordert. Es zeigt sich, dass die wissenschaftlichen Arbeiten hier der industrielle Praxis meist einen Schritt voraus sind. Daher ist anzunehmen, dass in Zukunft auch die aktuellen wissenschaftlichen Arbeiten zur automatisierten Reorganisation sowie zur Ableitung geeigneter Reorganisationen für die Praxis relevant werden. Nicht zuletzt der zunehmende Einsatz von Restrukturierungen in modernen Entwicklungsumgebungen lässt dies vermuten.

Eine weitere Erkenntnis aus der betrachteten Zeit ist, dass aus wissenschaftlicher Sicht uninteressante, da ausreichend erforschte Themengebiete wie die Konstruktion von Parsern bzw. Faktenextraktoren im Umfeld des Reengineering durchaus noch große Probleme bei der technischen Umsetzung bereiten können. Der Aufwand für diese Umsetzungen, welche mit industriellem Code zurecht kommen, darf nicht unterschätzt werden, insbesondere für Sprachen, für die keine genauen oder mehrere unterschiedliche Spezifikationen vorliegen. Die Größe industrieller Softwaresysteme erfordert darüber hinaus geeignete Modelle, um die für globale Analysen notwendige Datenmengen verarbeiten zu können.

Aktuelle Trends in der Softwareentwicklung erfordern zukünftig eine Anpassung und Weiterentwicklung unserer bisherigen Reengineering-Methoden. Ein Trend ist hierbei die zunehmende Entkopplung der Bausteine eines Softwaresystems mit Hilfe reflektiver Techniken (z. B. wie innerhalb des Eclipse-Frameworks) sowie entsprechender Softwarearchitekturen (z. B. SOA). Die bisher aus dem Quelltext abgeleiteten Abhängigkeiten sind somit zunehmend nicht mehr ausreichend, es müssen weitere Artefakte mit berücksichtigt werden, welche unter Umständen in völlig unterschiedlichen Sprachen wie z. B. XML vorliegen. Nur bei Berücksichtigung dieser weiter gehenden Abhängigkeiten sind auch in Zukunft die darauf aufbauenden Reengineering-Techniken sinnvoll anwendbar. Ein weiterer zu beobachtender Trend in der Softwareentwicklung insbesondere bei großen Systemen ist der Einzatz modellgetriebener Technologien. Erste Ansätze zur Bewertung von mit solchen Technologien erstellten Softwaresystemen wurden bereits entwickelt und werden in Zukunft eine immer wichtigere Rolle bei den Reengineering-Aktivitäten des FZI einnehmen.

# Literatur


[And07] Christoph Andriessens. *Werkzeuggestützte Identifikation von Strukturproblemen bei modellgetriebener Softwareentwicklung*. Dissertation, Universität Karlsruhe (TH), 2007.

[Bau06] Markus Bauer. *Strukturuntersuchung großer Softwaresysteme*. Dissertation, Universität Karlsruhe (TH), 2006.

[BBC+99] Holger Bär, Markus Bauer, Oliver Ciupke, Serge Demeyer, Stephane Ducasse, Michele Lanza, Radu Marinescu, Robb Nebbe, Oscar Nierstrasz, Michael Przybilski, Tamar Richner, Matthias Rieger, Claudio Riva, Anne-Marie Sassen, Benedikt Schulz, Patrick Steyaert, Sander Tichelaar und Joachim Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook, 1999.

[BMM98] William J. Brown, Raphael C. Malveau und Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[Ciu02] Oliver Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen*. Dissertation, Universität Karlsruhe (TH), 2002.

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

[Gen04] Thomas Genssler. *Werkzeuggestützte Adaption objektorientierter Programme*. Dissertation, Universität Karlsruhe (TH), 2004.

[GS99] Thomas Genssler und Benedikt Schulz. Transforming Inheritance into Composition - A Reengineering Pattern. In *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing*, 1999.

[GSMZ98] Thomas Genssler, Benedikt Schulz, Berthold Mohr und Walter Zimmer. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Proceedings of the 27th TOOLS conference*, 1998.

[Lud02] Andreas Ludwig. *Automatische Transformation großer Softwaresysteme*. Dissertation, Universität Karlsruhe (TH), 2002.

[Mar01] Radu Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS)*, 2001.

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Dissertation, Politehnica University of Timişoara, 2002.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Professional, 1996.

[SSB06] Olaf Seng, Johannes Stammel und David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference, Seattle*. ACM Press, 2006.

[TR07] Adrian Trifu und Urs Reupke. Towards Automated Restructuring of Object Oriented Systems. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*, 2007.

[Tri08] Mircea Trifu. Using Dataflow Information for Concern Identification in Object-Oriented Software Systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2008.

[Zim97] Walter Zimmer. *Frameworks und Entwurfsmuster*. Dissertation, Universität Karlsruhe (TH), 1997.