UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Kombination von Clustering- und musterbasierten Reverse-Engineering-Verfahren

Master-Arbeit

im Rahmen des Studiengangs Informatik
zur Erlangung des Grades
Master of Science

von
Oleg Travkin
Ginsterweg 1
33813 Oerlinghausen

vorgelegt bei
Jun.-Prof. Dr.-Ing. Steffen Becker
und
Prof. Dr. Wilhelm Schäfer

Paderborn, April 2011

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Inhalt

Abbildungsverzeichnis

1 Einleitung

Software wird heutzutage immer seltener von Grund auf neu entwickelt. Meist muss existierende Software lediglich an geänderte Anforderungen angepasst oder um neue Funktionalität erweitert werden. Die Dokumentation der Software wird dabei oft aus Zeit- und Kostengründen nicht mitgepflegt und veraltet daher mit der Zeit. Um Anpassungen an Software durchzuführen, deren Dokumentation veraltet ist, muss die Software erst mit Hilfe von *Reverse-Engineering*-Verfahren analysiert und verstanden werden.

Als Reverse Engineering bezeichnet man die Rückgewinnung von Entwurfsinformationen aus der Implementierung eines Softwaresystems. Von *Forward Engineering* spricht man hingegen bei einer Neuentwicklung oder bei Anpassungen eines bestehenden Softwaresystems an neue Anforderungen. Der Begriff *Reengineering* bezeichnet eine Kombination von Forward und Reverse Engineering [DDN08].

Die Größe von Softwaresystemen nimmt zu, wie am Umfang der gängigen Betriebssysteme deutlich wird. Schätzungen nach kann die Wartung eines Softwaresystems bis zu 80% aller Kosten im Lebenszyklus eines Systems ausmachen [Som92]. Die Komplexität großer Systeme macht daher effiziente Reengineering-Verfahren erforderlich. Auf Grund der zunehmenden Bedeutung des Reengineerings in der aktuellen Softwareentwicklung wurden in den letzten Jahren zahlreiche Techniken und Verfahren vorgestellt, die beim Verstehen eines Altsystems helfen sollen. Man kann die meisten dieser Verfahren in zwei Kategorien einteilen: **i)** die Clustering-basierten und **ii)** die musterbasierten Reverse-Engineering-Verfahren [Sar03].

1.1 Clustering-basiertes Reverse Engineering

Bei der Wartung von umfangreichen Softwaresystemen ist es wichtig, sich zunächst einen Überblick zu verschaffen. Die Softwarearchitektur eines Systems ist eine Sicht auf das System [Kos05], welche die Struktur des Systems abstrakt und modular darstellt und daher diesen Zweck erfüllt.

Clustering-basierte Reverse-Engineering-Verfahren versuchen, die Softwarearchitektur von Softwaresystemen zu rekonstruieren. Grundelemente der Implementierung eines Systems (üblicherweise Klassen) werden dabei mit Hilfe von Softwaremetriken, wie zum Beispiel *Coupling* oder *Stability* [Mar94], analysiert. Anhand der ermittelten Werte für die verwendeten Softwaremetriken werden die Grundelemente in Softwarekomponenten gruppiert und durch Konnektoren, welche gegenseitige Beziehungen repräsentieren, miteinander verbunden. Das Ergebnis einer Clustering-basierten Analyse ist eine Menge von Softwarekomponenten

und Konnektoren, die Teilsysteme oder Module des analysierten Systems, sowie deren Beziehungen zueinander repräsentieren [BT04].

1.2 Musterbasiertes Reverse Engineering

Eine der Herausforderungen eines Reengineers ist es, den Zweck von Teilen eines Softwaresystems zu verstehen, bevor Änderungen an diesem Teil durchgeführt werden können. Bei der Entwicklung von Software tauchen bestimmte Herausforderungen wiederholt auf, sodass auch deren Lösungen strukturelle Ähnlichkeiten aufweisen und durch Muster beschrieben werden können. Die *Entwurfsmuster* von Gamma et al. [GHJV95] beschreiben bewährte Lösungen solcher Herausforderungen als Muster und dokumentieren unter anderem ihren Zweck.

Musterbasierte Reverse-Engineering-Verfahren versuchen, Anwendungen von zuvor beschriebenen Mustern in der Implementierung eines Softwaresystems zu identifizieren. Eine Reengineer kann zu einer erkannten Anwendung eines bestimmten Musters in dessen Dokumentation nachschlagen, um den Zweck der Anwendung zu erfahren und damit den Teil der Software, in der das Muster angewendet wurde, besser zu verstehen. Das musterbasierte Reverse Engineering kann aber auch zur Erkennung von Schwachstellen eines Softwaresystems genutzt werden. Schwachstellen werden zum Beispiel durch *Antimuster* [BMMM98] und *Bad Smells* [Fow99] beschrieben. Sie wirken sich negativ auf die Software aus, indem sie zum Beispiel den Aufwand von Wartungsarbeiten erhöhen und sollten daher vermieden werden. Die Dokumentation von Schwachstellen enthält unter anderem Maßnahmen zu deren Beseitigung und kann einem Reengineer daher bei der Wartung von Software helfen.

1.3 Motivation

Ein Reengineer muss ein Softwaresystem erst verstehen bevor er es anpassen kann. Dabei hat er die Wahl zwischen Clustering-basierten und musterbasierten Reverse-Engineering-Ansätzen zur Analyse des Systems. Die Abbildung 1.1 visualisiert die Anwendung der beiden Reverse-Engineering-Ansätze. Links wird die Clustering-basierte Analyse eines Softwaresystems unter Verwendung von Softwaremetriken dargestellt. Rechts wird die musterbasierte Analyse dargestellt, die in einem System nach Vorkommen von zuvor spezifizierten Mustern sucht.

Um einen Überblick über ein System zu erhalten, eignet sich die Clustering-basierte Analyse, da diese versucht, die Softwarearchitektur des analysierten Systems zu rekonstruieren. Die Softwarearchitektur gibt Aufschluss über die Aufteilung und Zusammensetzung des Systems in Komponenten. Weil die Clustering-basierte Analyse auf der Berechnung von Softwaremetriken basiert und die Berechnung von Metrikwerten in den meisten Fällen einfach ist, kann die Analyse selbst für große Softwaresysteme in kurzer Zeit durchgeführt werden [KDW+10].

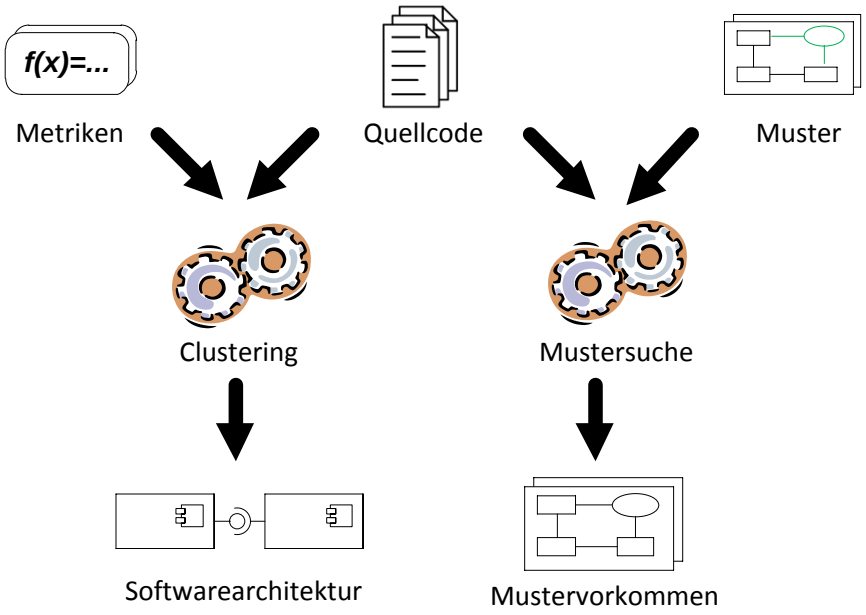

Abbildung 1.1: Anwendung des Clustering- und musterbasierten Reverse Engineerings

Die Ergebnisse der Clustering-basierten Analyse können jedoch von Schwachstellen in dem analysierten Softwaresystem beeinflusst werden. Schwachstellen können zum Beispiel die Kopplung in einem System erhöhen. Da die Kopplung durch Softwaremetriken in dem Clustering-basierten Ansatz erfasst wird, üben Schwachstellen damit Einfluss auf die Werte von Softwaremetriken und damit auch auf die rekonstruierte Softwarearchitektur aus. Ein weiterer Nachteil des Clustering-basierten Reverse Engineerings ist, dass die Softwarearchitektur zwar die Struktur eines Systems widergibt, aber nicht den Zweck der darin enthaltenen Komponenten. Der Zweck einer Komponente muss daher von einem Reengineer manuell erschlossen werden.

Um einen detaillierteren Einblick in ein Softwaresystem zu erhalten, eignet sich ein musterbasierter Reverse-Engineering-Ansatz, da Muster meist einen niedrigeren Abstraktionsgrad als Softwarekomponenten haben. Die Mustersuche kann zur Erkennung von Schwachstellen, welche die Softwarearchitektur beeinflussen, eingesetzt werden. Die Grundannahme dieser Arbeit ist, dass die Beseitigung von Schwachstellen zu einer Verbesserung der Softwarearchitektur führt, weil der Einfluss von Schwachstellen auf die Softwarearchitektur durch ihre Beseitigung entfernt wird. Darüber hinaus kann die Dokumentation von Mustern Aufschluss über den Zweck ihrer Verwendung geben und dadurch beim Verstehen eines analysierten Systems helfen.

Weil bei einer Mustersuche detailliertere Informationen, wie der Kontrollfluss eines Programms, berücksichtigt werden, erfordert eine musterbasierte Analyse deutlich mehr Zeit als eine Clustering-basierte Analyse des gleichen Systems. Wegen des meist niedrigen Abstraktionsgrades von Mustern können musterbasierte Analysen auch in kleinen Softwaresystemen viele Mustervorkommen aufdecken. Die Menge der Mustervorkommen kann jedoch mit zunehmender Größe eines analysierten Softwaresystems zunehmend unpraktikabel werden, da jedes Mustervorkommen von einem Reengineer begutachtet und der Zusammenhang zu anderen Mustervorkommen manuell erschlossen werden muss. Eine Analyse einzelner Teile des Systems wird dabei durch Mustervorkommen aus anderen Teilen des Systems erschwert.

1.4 Lösungsansatz

Um mögliche Synergieeffekte einer Kombination des Clustering- und musterbasierten Reverse Engineerings zu nutzen, sollte im Rahmen dieser Arbeit ein integrierter Reverse-Engineering-Prozess entstehen, der die Ansätze der beiden Reverse-Engineering-Verfahren kombiniert und erstmals in den Arbeiten [TDB11, DB11] vorgestellt wird. Der Fokus dieser Arbeit liegt dabei auf der Identifikation von Schwachstellen in der Softwarearchitektur des Systems und damit auch in dem System selbst. Ein Überblick über den angestrebten Prozess wird in Abbildung 1.2 dargestellt.

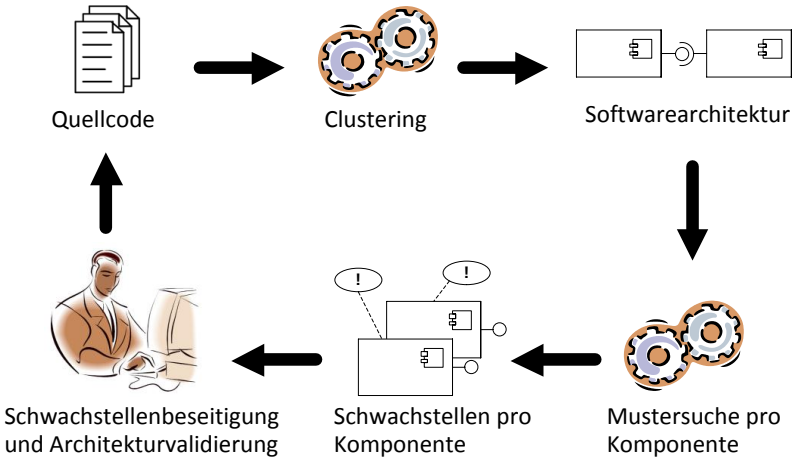

Abbildung 1.2: Anwendung des Clustering- und musterbasierten Reverse Engineerings in Kombination

Die Clustering-basierte Analyse wird dabei, wie zuvor, zur Rekonstruktion der Softwarearchitektur eines analysierten Softwaresystems verwendet. Um genauere

Informationen zu einzelnen Komponenten, die durch das Clustering bereitgestellt werden, zu erhalten, soll ein Reengineer eine oder mehrere Komponenten mit Hilfe der Mustersuche analysieren können. Die Anwendung der Mustersuche soll Schwachstellen in einzelnen Komponenten aufdecken und einen Reengineer damit auf Möglichkeiten zur Verbesserung des analysierten Softwaresystems und dessen Softwarearchitektur hinweisen. Nach Beseitigung der Schwachstellen kann der Reengineer eine neue Iteration des hier vorgestellten Reverse-Engineering-Prozesses einleiten, um zu validieren, wie sich die Architektur des analysierten Systems durch die Beseitigung der Schwachstellen verändert hat.

Die Anwendung des musterbasierten Reverse Engineerings auf einzelne Komponenten verspricht zwei Vorteile gegenüber einer herkömmlichen Mustersuche, die ein Softwaresystem als Ganzes analysiert. Zum einen sind Mustervorkommen auf die analysierten Komponenten beschränkt, wodurch sie den Fokus des Reengineers berücksichtigen und dadurch weniger Aufwand bei der Auswertung der Analyseergebnisse entsteht. Zum anderen ist zu erwarten, dass eine Analyse eines Teils des Systems deutlich weniger Zeit beansprucht, als es für das gesamte System der Fall wäre. Als Konsequenz dieser beiden Vorteile wird der musterbasierte Reverse-Engineering-Ansatz in der in dieser Arbeit vorgeschlagenen Kombination mit einem Clustering-basierten Ansatz auch für zunehmend große Softwaresysteme praktikabel.

Neben der Erkennung von Schwachstellen bietet der Ansatz auch die Möglichkeit zur Erkennung von Entwurfsmustern. Diese Richtung wird jedoch im Rahmen dieser Arbeit nicht weiter verfolgt.

1.5 Struktur der Arbeit

Im folgenden Kapitel werden die zum Verständnis dieser Arbeit benötigten Grundlagen vermittelt. Die Problemstellung dieser Arbeit wird in Kapitel 3 tiefergehend erläutert. Konzeptionelle und technische Herausforderungen dieser Arbeit werden dort zusammengefasst. In Kapitel 4 wird zunächst der durch diese Arbeit entstandene Reverse-Engineering-Prozess genauer erläutert. In Abschnitt 4.2 wird beschrieben, wie eine im Rahmen einer Clustering-basierten Analyse rekonstruierte Softwarearchitektur bei der Mustersuche berücksichtigt werden kann und wie die Mustersuche dabei auf einzelne Komponenten beschränkt wird. Muster zur Erkennung von Schwachstellen, die sowohl die Softwarearchitektur als auch die Implementierung eines Systems berücksichtigen und in dieser Arbeit erarbeitet wurden, werden in Abschnitt 4.3 vorgestellt. In Abschnitt 4.4 werden die Einschränkungen des kombinierten Ansatzes beschrieben. Kapitel 5 liefert Details über die Umsetzung der in Kapitel 4 vorgestellten Konzepte. Praktische Erfahrungen mit dem kombinierten Reverse-Engineering-Ansatz werden in Kapitel 6 an einem Softwaresystem-Beispiel dokumentiert. Die Abgrenzung zu verwandten Arbeiten wird in Kapitel 7 erläutert. Abschließend liefert das Kapitel 8 eine Zusammenfassung und einen Ausblick auf mögliche Arbeiten in Folge dieser Arbeit.

2 Grundlagen und Begriffe

Dieses Kapitel erläutert wichtige Grundlagen, sowie Begriffe, die zum Verständnis dieser Arbeit benötigt werden. In dem folgenden Abschnitt 2.1 werden zunächst einige Begriffe erklärt. In dem Abschnitt 2.2 werden die Datenmodelle GAST, Service Architecture und Source Code Decorator vorgestellt. Einen Überblick über den Clustering-basierten Reverse-Engineering-Ansatz gibt der Abschnitt 2.3. Der musterbasierte Reverse-Engineering-Ansatz wird hingegen in Abschnitt 2.4 vorgestellt.

2.1 Begriffserklärungen

Im Folgenden werden einige Begriffe vorgestellt, die im weiteren Verlauf dieser Arbeit mit dem hier erläuterten Hintergrund verwendet werden.

Konzeptionelle Architektur Als konzeptionelle Architektur wird diejenige Architekur bezeichnet, die von Softwareentwicklern während der Entwicklung von Software beabsichtigt und entworfen wurde. In einer gepflegten Softwaredokumentation wird meist die konzeptionelle Architektur eines Softwaresystems dokumentiert. Die konzeptionelle Architektur wird oft auch als ideelle, intendierte oder auch als logische Architektur bezeichnet. [DP09]

Konkrete Architektur Die konkrete Architektur eines Softwaresystems ist diejenige, die sich aus dem Quellcode eines Softwaresystems ableiten lässt, also aus der Umsetzung einer konzeptionellen Architektur. Man spricht daher auch von der implementierten, realisierten oder physischen Architektur. [DP09] Diese kann von der konzeptionellen Architektur auf Grund von wiederkehrender Wartung oder dem Hinzufügen neuer im ursprünglichen Entwurf nicht vorgesehener Funktionalität und den daraus resultierenden Veränderungen an der Implementierung abweichen.

Musterfund Ein Musterfund ist ein identifiziertes Mustervorkommen. Die Identifikation kann beispielsweise im Rahmen einer Mustersuche erfolgen. Die Ergebnisse einer Mustersuche können *True-* und *False-Positives* enthalten. *False-Positives* sind falsche Mustervorkommen. Sie stimmen zwar strukturell mit dem zu Grunde liegenden Muster überein, sind aber dennoch keine Anwendung des Musters. *True-Positives* sind dagegen tatsächliche Mustervorkommen des zu Grunde liegenden Musters. Ein Musterfund ist ein *True-Positive* und muss als solcher bestätigt werden.

Mustervorkommen Ein Mustervorkommen ist das Vorkommen eines Teilgraphen in einem Wirtsgraph, wobei der Teilgraph alle strukturellen Eigenschaften mit dem zu Grunde liegenden Muster teilt. Da es sich bei dem Wirtsgraph um einen Objektgraph handelt und dieser attributiert und typisiert ist, müssen die Objekttypen und ihre Attribute mit denen des Musters übereinstimmen, so fern sie durch das Muster beschrieben werden.

Schwachstelle Schwachstellen sind in der Regel Teile einer Software, die mit unerwünschten Nebeneffekten einhergehen und daher oft erhöhten Aufwand bei der Wartung der Software verursachen. In der Literatur werden Schwachstellen meist als Bad Smells [Fow99] oder Antimuster [BMMM98] bezeichnet. Während Bad Smells schlechte Code-Eigenschaften der Software beschreiben, existieren Antimuster neben Code auch für Softwarearchitekturen und dem Projektmanagement. Dies ist jedoch kein konzeptioneller Unterschied, sondern liegt am Fokus der Autoren.

Im weiteren Verlauf dieser Arbeit wird der Begriff Schwachstelle verwendet, um Teile aus der Implementierung einer bestimmten Software zu charakterisieren, die sich negativ auf deren Softwarearchitektur auswirken.

2.2 Datenmodelle

Im Rahmen dieser Arbeit werden einige Datenmodelle immer wieder verwendet und sind deshalb für das Verständnis der vorgestellten Konzepte unerlässlich. Dieser Abschnitt liefert einen Überblick über die wichtigsten verwendeten Datenmodelle. Im folgenden Abschnitt wird zunächst das GAST-Metamodell erläutert, welches Implementierungen von Softwaresystemen als abstrakten Syntaxbaum repräsentiert und auf dessen Basis Analysen durchgeführt werden. In Abschnitt 2.2.2 wird das SAMM vorgestellt, das zur Modellierung von Softwarearchitekturen benötigt wird. Anschließend wird in Abschnitt 2.2.3 das Source-Code-Decorator-Meta-Model vorgestellt, dessen Instanzen die Korrespondenz zwischen Instanzen des GAST-Metamodells und des SAMM repräsentieren.

2.2.1 Generalisierter abstrakter Syntaxbaum - GAST

Ein abstrakter Syntaxbaum (AST) dient der Repräsentation von Quellcode eines Programms in einem Modell und definiert eine abstrakte Syntax. Der generalisierte abstrakte Syntaxbaum (engl. *generalized abstract syntax tree (GAST)*) ist eine spezielle Ausprägung eines abstrakten Syntaxbaums und wurde im Rahmen des QBench-Projekts am FZI Karlsruhe entwickelt [QBe10]. Das GAST-Metamodell wurde mit dem Ziel entwickelt, die Programmiersprachen Java, C++ und Delphi zu unterstützen. Daher generalisiert das Metamodell einige Sprachkonstrukte, die sprachspezifisch sind (z.B. Schnittstellen in Java) zu Konstrukten, welche mit Hilfe von Objekt-orientierten Sprachen im Allgemeinen ausgedrückt werden können.

Ein GAST-Modell wird sowohl von dem Reverse-Engineering-Werkzeugen SoMoX als auch in Folge dieser Arbeit von Reclipse als Eingabe verwendet. Um Quellcode einzulesen und eine Instanz des GAST-Metamodells zu erzeugen, wird das Werkzeug SISSy verwendet [ABM+06, SSM06, TS04].

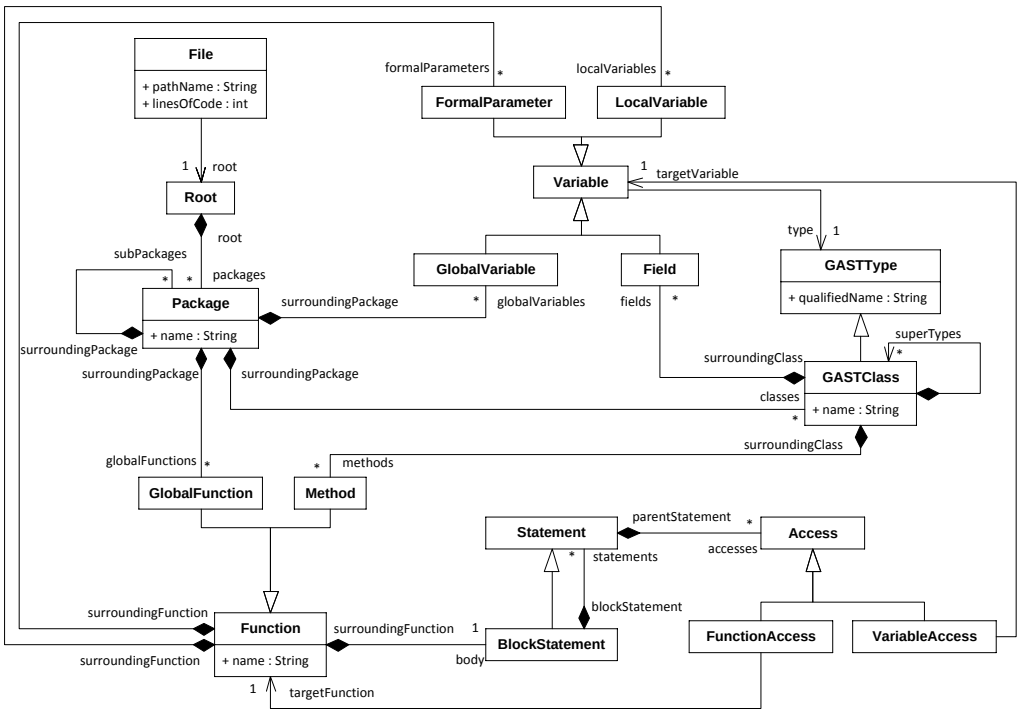

Abbildung 2.1: Ausschnitt aus dem GAST-Metamodell (aktualisiert [QBe10])

Insbesondere für Reclipse werden Musterspezifikationen auf Basis eines abstrakten Syntaxbaums benötigt. Eine Musterspezifikation in Reclipse modelliert einen Ausschnitt einer möglichen Instanz eines abstrakten Syntaxbaums. Im Folgenden werden daher die wichtigsten Elemente des GAST-Metamodells und ihre Bedeutung erläutert. Dabei wurden insbesondere einige Spezialisierungen zu Gunsten der Übersichtlichkeit ausgelassen. Die Abbildung 2.1 zeigt einen Ausschnitt der im GAST-Metamodell existierenden Klassen als Klassendiagramm.

Root Das Root-Element ist die Wurzel eines GAST-Modells. Über Kompositionsbeziehungen sind von dem Root-Element alle anderen Elemente erreichbar.

File Elemente eines abstrakten Syntaxbaums, wie zum Beispiel Klassen können Dateien zugeordnet werden. Um diese Zuordnung festzuhalten, wird das File-Element verwendet. Es hält neben einem String, der den Pfad der repräsentierten Datei darstellt, auch Referenzen auf Klassen oder Packages, die der Datei zugeordnet werden.

Package Ähnlich zu den Packages in Java, stellt das Package-Element des GAST-Metamodells einen Namensraum mit bestimmten Sichtbarkeiten dar. Ein Package kann weitere Package-Elemente, Klassen, globale Variablen oder Funktionen enthalten.

GASTType Das GASTType-Element repräsentiert einen beliebigen Datentyp in dem Modell. Dies können primitive Datentypen oder auch Klassen sein. Im Attribut qualifiedName wird der voll qualifizierte Name des Typs abgelegt und kann zur eindeutigen Identifikation des Typs verwendet werden.

GASTClass Klassen werden im GAST über das Element GASTClass repräsentiert und spezialisieren das Element GASTType. Neben den geerbten Typ-Eigenschaften referenziert ein GASTClass-Element die von ihm definierten Methoden, Attribute und gegebenenfalls innere Klassen. Eine Klasse kann außerdem einem Package zugeordnet werden.

Function Das Element Function ist die Oberklasse für jegliche Operationen, die ausgeführt werden können. Neben einem Attribut für den Namen, hat ein Function-Objekt Referenzen auf eine Menge von lokalen Variablen und formale Parameter. Der Rückgabetyp einer Function wird über eine Spezialisierung(DeclarationTypeAccess) von Access festgelegt und konnte in der Abbildung aus Platzgründen nicht abgebildet werden. Ein Function-Objekt enthält immer ein *BlockStatement*, das wiederum weitere Statement-Elemente enthalten kann.

GlobalFunction Das GlobalFunction-Element realisiert global zugängliche Operationen, also Operationen, die keiner Klasse angehören und daher lediglich einem Namensraum zugeordnet werden können, der durch das Package repräsentiert wird. C-Funktionen werden beispielsweise als GlobalFunction repräsentiert.

Method Operationen, die in Klassen definiert werden, werden durch Method-Elemente modelliert. Das Method-Element ist eine Spezialisierung des Elements Function.

Variable Das Variable-Element ist die Oberklasse für jegliche Art von Variablen. Eine Variable hat stets einen Namen und einen Typ.

LocalVariable Lokale Variablen, also Variablen, die in einer beliebigen Operation definiert werden, werden durch LocalVariable-Elemente im Modell repräsentiert.

FormalParameter Das Element FormalParameter repräsentiert eine Parametervariable einer Operation. Ein FormalParameter-Element ist stets an ein Function-Element gebunden.

GlobalVariable Ein GlobalVariable-Element steht für eine in einem bestimmten Namensraum global zugängliche Variable, wobei der Namensraum durch das Package-Element festgelegt wird.

Field Klassen- und Objektvariablen werden im Modell durch das Field-Element modelliert. Ein Field-Objekt ist daher stets über eine Kompositionsbeziehung mit einem GASTClass-Objekt verbunden.

Statement Das Element Statement modelliert verschiedene Anweisungen, die eine Operation enthalten kann. Die unterschiedlichen Arten von Anweisungen werden durch Spezialisierungen von Statement wie zum Beispiel BlockStatement repräsentiert. Ein Statement-Element kann eine Menge von Access-Elementen enthalten, die wiederum Zugriffe auf bestimmte Variablen und Operationen innerhalb der vom Statement repräsentierten Anweisungen modellieren.

BlockStatement Das BlockStatement ist ein spezielles Statement, das andere Statements enthalten kann. Es wird als Wurzel aller Statements in einem Function-Element verwendet.

Access Zugriffe auf bestimmte Variablen oder Operationen werden im GAST-Modell über Spezialisierungen des Access-Elements modelliert. Ein Access-Element ist stets in einem Statement enthalten.

FunctionAccess Ein Zugriff auf eine Operation wird über das Element FunctionAccess modelliert. Ein FunctionAccess-Objekt referenziert das Function-Objekt, das für die Operation steht, auf die zugegriffen wird.

VariableAccess Ein Zugriff auf eine Variable wird über das Element VariableAccess modelliert. Ein VariableAccess-Objekt referenziert das Variable-Objekt, das für die Variable steht, auf die zugegriffen wird.

2.2.2 Service Architecture Meta-Model - SAMM

Im Rahmen dieser Arbeit wird das Clustering-basierte Werkzeug SoMoX genutzt, um die Softwarearchitektur eines Softwaresystems zu rekonstruieren. SoMoX nutzt als Metamodell für die Softwarearchitektur von Systemen das *Service Architecture Meta-Model* (SAMM).

Das SAMM dient neben SoMoX auch der integrierten Reengineering-Umgebung Q-Impress [qim10] als Metamodell, der auch SoMoX (siehe Abschnitt 2.3) angehört. Die Umgebung Q-Impress muss neben der rekonstruierten Softwarearchitekturen noch weitere Daten in einem Modell halten, die ebenfalls durch das SAMM repräsentiert werden. In diesem Abschnitt wird lediglich der Teil des SAMM erläutert, der für die statische Analyse mit SoMoX verwendet wird und damit zur Spezifikation einer Softwarearchitektur nötig ist. Eine detaillierte Beschreibung des SAMM ist in [SAM11] zu finden.

Im weiteren Verlauf dieser Arbeit beziehen sich Musterspezifikationen des musterbasierten Reverse-Engineering-Ansatzes Reclipse (siehe Abschnitt 2.4) unter anderem auch auf die Softwarearchitektur eines Systems. Um solche Musterspezifikationen leichter nachvollziehen zu können, wird im Folgenden der für die Softwarearchitektur benötigte Teil des SAMM erläutert. Ein Teil des SAMM, der zur Spezifikation von Softwarearchitekturen benötigt wird, wird in Abbildung 2.2 dargestellt.

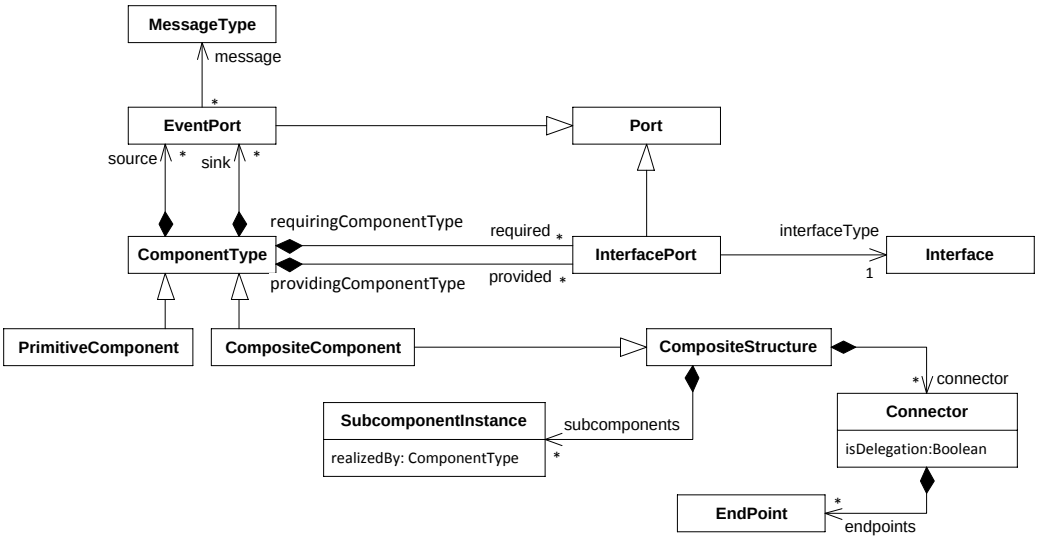

Abbildung 2.2: Zur Spezifikation von Softwarearchitekturen verwendeter Teil aus dem SAMM

ComponentType repräsentiert eine Komponente und ist die abstrakte Oberklasse von PrimitiveComponent und CompositeComponent. Sie implementiert daher die gemeinsamen Eigenschaften der beiden Klassen. Über die Assoziationen zu InterfacePort werden die benötigten (required) und zu Verfügung gestellten (provided) Schnittstellen der Komponente modelliert. Bei den durch InterfacePort repräsentierten Schnittstellen handelt es sich um Funktionalität durch beispeilweise Methoden, die bereitgestellt oder benötigt wird. Datenaustausch zwischen Komponenten wird durch die Klasse EventPort modelliert, wobei die Rolle source die Quell- und die Rolle sink die Zielkomponente repräsentiert.

PrimitiveComponent ist eine atomare Komponente und hat daher keine Subkomponenten. Eine PrimitiveComponent hat damit den niedrigste Abstraktionsgrad unter den Komponenten in einer Softwarearchitektur.

CompositeComponent repräsentiert eine Komponente, die aus Subkomponenten besteht. Subkomponenten können sowohl Elemente vom Typ PrimitiveComponent als auch CompositeComponent sein. Die Zusammensetzung einer Komponente wird über die geerbten Assoziationen subcomponents und connector der Oberklasse CompositeStructure modelliert.

InterfacePort repräsentiert eine Instanz von einem Interface und damit Funktionalität, die von einer Komponente benötigt oder zur Verfügung gestellt wird und durch Methodenaufrufe genutzt werden kann. InterfacePorts bilden damit einen Teil der Schnittstelle einer Komponente.

Interface repräsentiert den nach außen sichtbaren Teil einer Komponente. Dieser Teil wird durch eine Menge von Operationen repräsentiert, die von anderen Komponenten aufgerufen werden können. Ein Interface dient als Typ für InterfacePort-Elemente.

CompositeStructure ist eine abstrakte Klasse, von der CompositeComponent erbt. Sie repräsentiert die Kompositionsbeziehungen von einer CompositeStructure zu ihren Subkomponenten. Über die Assoziation connector werden Connector-Elemente referenziert, welche Delegationen von den Ports der Subkomponenten zu den Ports einer CompositeComponent repräsentieren. Die Subkomponenten der CompositeStructure werden über die Assoziation subcomponents referenziert und durch SubcomponentInstance repräsentiert.

SubcomponentInstance ist eine Subkomponente in einer CompositeStructure. Die Subkomponente repräsentiert dabei eine Instanz einer bestimmten Komponente, die durch das Attribut realizedBy vom Typ ComponentType festgelegt wird.

Connector repräsentiert eine Verbindung zwischen den Ports von Komponenten, die zur Kommunikation genutzt wird. Das abgeleitete Attribut isDelegation bestimmt, ob es sich bei dem Connector um eine Delegation zwischen einer Subkomponente und der Komponente handelt, der die Subkomponente angehört. Ein Connector kann anstatt einer Delegation auch ein Assembly repräsentieren, wobei die kommunizierenden Komponenten sich in diesem Fall in der selben Hierarchieebene befinden.

EndPoint repräsentiert eine Verknüpfung von einem Connector zu einem bestimmten Port einer Komponente. In einem Connector werden die miteinander kommunizierenden Ports über verschiedene EndPoints festgelegt.

MessageType repräsentiert einen Typ von Nachrichten, die zum Datenaustausch zwischen Komponenten genutzt werden. Ein MessageType wird über eine Menge von Parametern definiert, die in einer Nachricht enthalten sein müssen.

Port wird von InterfacePort und EventPort spezialisiert und repräsentiert eine der Schnittstellen einer Komponente.

EventPort repräsentiert eine nachrichtenbasierte Schnittstelle einer Komponente. Der zur Kommunikation erlaubte Nachrichtentyp wird über das Attribut message vom Typ MessageType festgelegt.

2.2.3 Source Code Decorator Meta-Model

Das Clustering-basierte SoMoX rekonstruiert die Softwarearchitektur eines komponentenbasierten Systems, wobei die Softwarearchitektur eine Instanz des SAMM bildet (siehe Abschnitt 2.2.2). Softwarearchitektur-Elemente aus dem SAMM referenzieren jedoch keine Elemente aus dem GAST-Modell (siehe Abschnitt 2.2.1), durch die sie implementiert werden. Ohne weitere Maßnahmen würden die Informationen über die Zuordnung von Elementen aus der Softwarearchitektur zu ihren implementierenden Elementen in einem GAST-Modell nach der Durchführung eines Clusterings verloren gehen. Diese Zuordnung von bestimmten Modellelementen zueinander wird im Folgenden auch als Korrespondenz bezeichnet.

Um diese Informationen nicht zu verlieren, wird ein Metamodell benötigt, welches die Korrespondenz von bestimmten Softwarearchitektur-Elementen zu ihren implementierenden Elementen aus dem GAST spezifiziert. Zu diesem Zweck wird in SoMoX das *Source Code Decorator Meta-Model* (SCDMM) genutzt. Die für das Verständnis dieser Arbeit wichtigsten Elemente des SCDMM sind in Abbildung 2.3 dargestellt und werden im Folgenden einzeln erläutert.

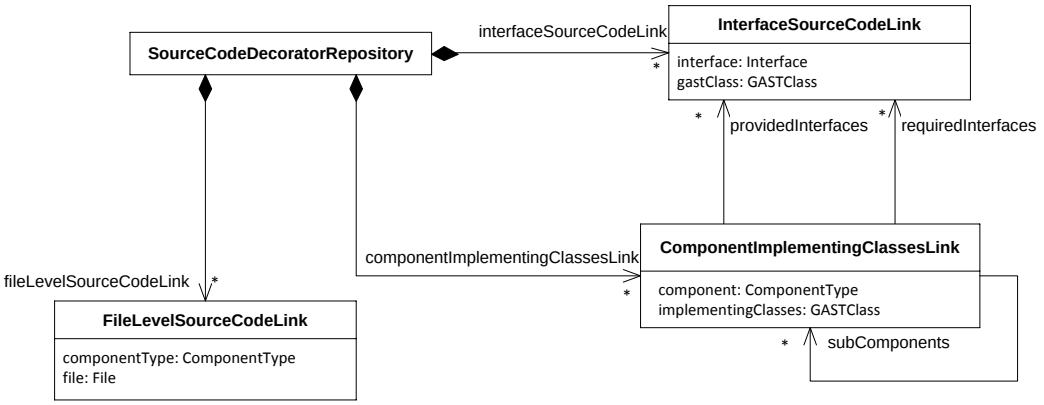

Abbildung 2.3: Ausschnitt aus dem Source-Code-Decorator-Metamodell

SourceCodeDecoratorRepository dient als Wurzel des Modells und enthält alle anderen Modellelemente. Daher hat SourceCodeDecoratorRepository eine Kompositionsbeziehung zu allen Elementen im SCDMM.

ComponentImplementingClassesLink repräsentiert eine Korrespondenz einer bestimmten Komponente aus dem SAMM zu einer Menge von Klassen aus

dem GAST. Die Komponente wird über das Attribut component vom Typ ComponentType referenziert. Die Klassen, welche die Komponente implementieren, werden über das Attribut implementingClasses vom Typ GASTClass referenziert. Die Eigenschaft, dass Komponenten aus mehreren Subkomponenten zusammengesetzt sein können, wird über die n-äre Assoziation subComponents repräsentiert, die von ComponentImplementingClassesLink zu sich selbst führt. Über die Assoziationen providedInterfaces und requiredInterfaces werden InterfaceSourceCodeLinks referenziert, welche die Korrespondenz von Komponentenschnittstellen zu den die Schnittstellen implementierenden GASTClass-Elementen repräsentieren.

InterfaceSourceCodeLink repräsentiert die Korrespondenz zwischen einer Komponentenschnittstelle im SAMM und einer die Schnittstelle definierenden Klasse bzw. einem Interface im GAST. Die Komponentenschnittstelle wird über das Attribut interface vom Typ Interface referenziert. Über das Attribut gastClass vom Typ GASTClass wird eine Klasse oder ein Interface aus dem GAST referenziert. Ein InterfaceSourceCodeLink ist über die Assoziationen providedInterfaces oder requiredInterfaces einem ComponentImplementingClassesLink zugeordnet, welches die Korrespondenz von einer Komponente zu ihren implementierenden Klassen repräsentiert.

FileLevelSourceCodeLink repräsentiert die Korrespondenz einer Komponente im SAMM zu einer Datei oder einem Ordner im Dateisystem des analysierten Systems, welche im GAST von File repräsentiert wird. Die Komponente wird über das Attribut componentType vom Typ ComponentType referenziert. Über das Attribut file vom Typ File wird ein Ordner oder eine Datei referenziert.

2.3 Clustering-basiertes Reverse Engineering

Die Wartung und Weiterentwicklung großer Softwaresysteme erfordert Kenntnisse über die Softwarearchitektur des gewarteten Systems. In einer Softwarearchitektur werde Teile eines Systems durch Komponenten repräsentiert. Die Unterteilung verschafft Entwicklern einen Überblick und hilft ihnen beim Verstehen des Systems.

Um die Softwarearchitektur eines Systems zu rekonstruieren, messen Clustering-basierte Ansätze die Werte bestimmter Metriken, von denen angenommen wird, dass sie in direktem Zusammenhang mit der Softwarearchitektur des Systems stehen. Anhand der Metrikwerte, sowie einiger manuell festgelegter Grenzwerte, wird das System in Komponenten aufgeteilt. Für miteinander kommunizierende Komponenten werden Schnittstellen ermittelt und die an der Kommunikation beteiligten Komponenten über Konnektoren verbunden. Die Softwarearchitektur ergibt sich anschließend aus der Komponentenaufteilung und den dazu gehörenden Schnittstellen, sowie den Konnektoren.

Im Rahmen dieser Arbeit wird das am FZI Karlsruhe entwickelte Clustering-basierte Werkzeug SoMoX (*Software Model Extractor*) [Kla09, Kro10] als Stellvertreter für Clustering-basierte Reverse-Engineering-Ansätze verwendet.

Im weiteren Verlauf dieses Abschnitts wird zunächst ein Überblick über den Clustering-basierten Reverse-Engineering-Ansatz gegeben. In Abschnitt 2.3.2 werden die für diese Arbeit bedeutendsten Metriken und Strategien erläutert, anhand derer das Clustering durchgeführt wird. Abschließend werden in Abschnitt 2.3.3 die Grenzen des Clustering-basierten Reverse Engineerings diskutiert.

2.3.1 Überblick über den Clustering-Prozess

Der Clustering-Ansatz von SoMoX orientiert sich an der starken Komponentendefinition von Szyperski [Szy02]. Diese setzt im Wesentlichen voraus, dass für alle Komponenten Schnittstellen definiert werden, die von den Komponenten implementiert werden. Kommunikation darf in einem solchen System allein über festgelegte Schnittstellen stattfinden, es sei denn, die beteiligten Kommunikationspartner gehören ein und der selben Komponente an.

Softwaresysteme können anhand verschiedener Eigenschaften wie zum Beispiel der Kopplung oder der Paketzugehörigkeit in Komponenten zerlegt werden. Solche Eigenschaften dienen in einem Clustering-basierten Reverse-Engineering-Ansatz zur Definition von Metriken. Ein Metrikwert repräsentiert wie stark eine bestimmte Eigenschaft erfüllt ist. Bestimmte Metriken haben von System zu System unterschiedlich starken Einfluss auf eine Clustering-basierte Analyse. Ihr Einfluss kann daher von einem Reengineer durch Angabe von Metrikgewichten konfiguriert werden. Ein hohes Gewicht einer Metrik hat zur Folge, dass die von der Metrik gemessene Eigenschaft des Systems höheren Einfluss auf das Clustering ausübt. Ein geringes Gewicht führt zu geringem Einfluss.

Die Abbildung 2.4 gibt eine Übersicht über den Clustering-basierten Analyseprozess von SoMoX. Im ersten Schritt wird der Quellcode des zu analysierenden Systems eingelesen. Dazu wird das Parser-Werkzeug SISSy verwendet [ABM+06, SSM06, TS04]. Das Ergebnis dieses Schritts ist ein generalisierter abstrakter Syntaxbaum (GAST), wie er in Abschnitt 2.2.1 beschrieben wird. Dieser dient als Eingabe für die Clustering-basierte Analyse. Die eigentliche Analyse läuft iterativ und startet vor der ersten Iteration mit der Erkennung von primitiven Komponenten. Primitive Komponenten werden in der Regel durch eine einzelne Klassen implementiert. In einigen Fällen, wie beispielsweise bei inneren Klassen, fassen primitive Komponenten aber auch mehrere Klassen zusammen.

Vor jeder Iteration werden Paare von bis dato gefundenen Komponenten gebildet. Diese Paare dienen als Kandidaten für weitere, abstraktere Komponenten. Eine Iteration beginnt mit der Evaluierung von Basismetriken für jeden Komponentenkandidaten. Die Werte der Basismetriken dienen in einem Folgeschritt zur Berechnung von weiteren, komplexeren Metrikwerten, welche auf den Basismetrikwerten aufbauen.

Die gewonnenen Metrikwerte werden anschließend anhand der vom Reengineer

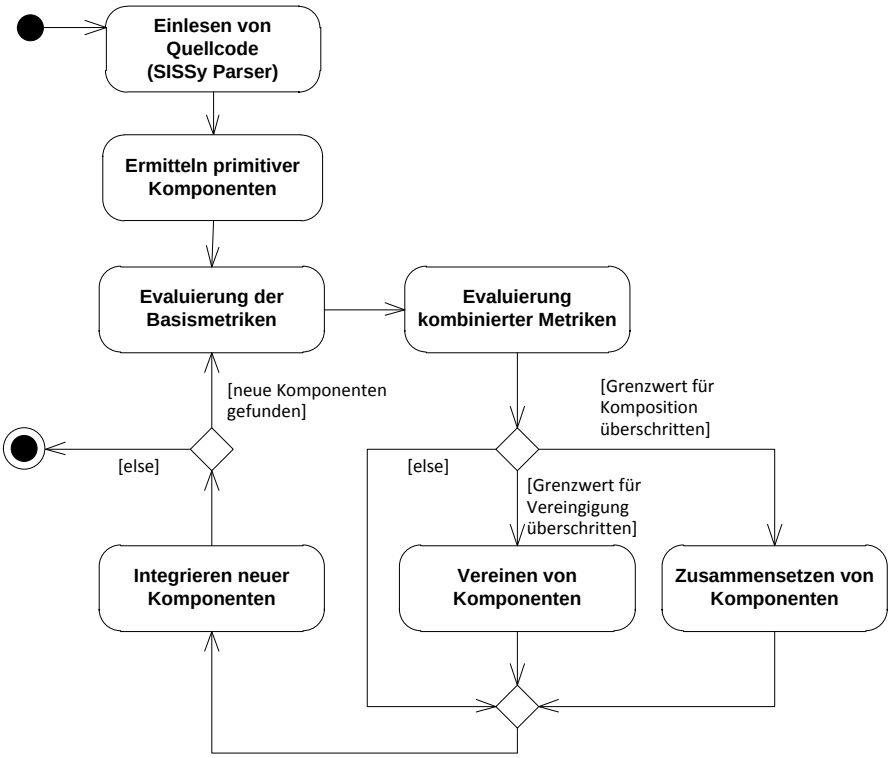

Abbildung 2.4: Ein Überblick über den Clustering-Prozess in SoMoX

gewählten Metrikgewichte gewichtet, um zu bestimmen, ob ein Komponentenkandidat zu einer neuen Komponente vereint, über Komposition zusammengesetzt oder verworfen wird. Dazu werden einzelne Metrikwerte aufsummiert und es wird geprüft, ob der für die Komposition oder das Vereinen benötigte Grenzwert überschritten wird. Ist das nicht der Fall, wird der Kandidat verworfen. Bei dem Vereinen eines Komponentenkandidaten entsteht eine neue Komponente, wobei die zwei Komponenten, welche das Kandidatenpaar gebildet haben, verworfen werden. Dieser Vorgang wird auch als *Merging* bezeichnet. Wird für einen Komponentenkandidaten entschieden, dass dieser zusammengesetzt werden soll, so bleiben die Komponenten, welche das Kandidatenpaar bilden, erhalten. Die neu entstehende Komponente erhält damit eine Kompositionsbeziehung zu den durch das Kandidatenpaar vertretenen Komponenten. Dies wird als *Composition* bezeichnet.

Anschließend werden neu entstandene Komponenten in die Ergebnis-Softwarearchitektur integriert. Dabei werden die Schnittstellen von Komponenten neu bestimmt bzw. aktualisiert und die Komponenten anhand der Schnittstellen miteinander über Konnektoren verbunden.

Wurden in einer Iteration neue Komponenten erkannt, so wird eine weitere Iteration durchgeführt. Die Analyse ist abgeschlossen wenn in einer Iteration keine neuen Komponenten mehr gefunden werden.

2.3.2 Strategien und Metriken

Komponentenbasierte Softwaresysteme unterscheiden sich nicht nur von System zu System in dem Stil ihrer Implementierung, sondern auch innerhalb eines Systems. Es werden daher verschiedene Strategien benötigt, um auf die Besonderheiten des jeweiligen Implementierungsstils einzugehen und anhand dieser das Clustering des Systems vorzunehmen.

Während der Analyse trifft eine Strategie Entscheidungen, ob beispielsweise ein Komponentenkandidat zu einer neuen Komponente vereint bzw. zusammengesetzt werden soll. Um solche Entscheidungen treffen zu können und gleichzeitig flexibel in der Formulierung von solchen Entscheidungskriterien zu sein, sieht der SoMoX-Ansatz vor, dass bei der Formulierung einer Strategie bereits bestehende Strategien wiederverwendet werden können. Auf unterster Ebene werden Strategien auf Basis von Metriken formuliert. Während einer Analyse werden dann die Werte der zu Grunde liegenden Metriken ausgewertet und durch die Strategie zu einer neuen Wertung verarbeitet.

Da Strategien auf einer Reihe von Metriken beruhen, werden im Folgenden zunächst die für diese Arbeit wichtigsten von SoMoX implementierten Metriken vorgestellt. Anschließend werden einige darauf aufbauenden Strategien für die Erkennung von Komponenten erläutert. Für weitere Strategien und Metriken, sowie Details dazu, wird auf die Dissertation von Klaus Krogmann verwiesen [Kro10]. Die im Folgenden genannten Definitionen zur Berechnung der Strategie- und Metrikwerte stammen ebenfalls aus seiner Arbeit.

2.3.2.1 Metriken in SoMoX

Eine Möglichkeit Eigenschaften eines Softwaresystems zu repräsentieren und zu formalisieren sind Metriken. Eine Metrik beschreibt durch ihren Wert, wie stark ein Systemelement ein Kriterium erfüllt. Je höher ein Metrikwert, desto stärker trifft das Kriterium zu. Alle von SoMoX implementierten Metriken haben einen Wertebereich von 0 bis 1, wobei der Wert 1 bedeutet, dass das durch die Metrik formalisierte Kriterium voll zutrifft und 0, dass dieses überhaupt nicht zutrifft.

Im Gegensatz zu objektorientierten Metriken, die sich meist auf einzelne Klassen oder Objekte beziehen, müssen die Metriken des SoMoX-Ansatzes mit Mengen von Klassen umgehen können, da Softwarekomponenten durch mehrere Klassen implementiert werden. Außerdem fehlen der Objektorientierung Konzepte zur Definition von Komponenten und ihrer Schnittstellen, weshalb objektorientierte Metriken nur eingeschränkt für die Analyse von Komponenten wiederverwendet werden können [CKK01]. Metriken wie beispielsweise die Kopplung wurden daher in dem SoMoX-Ansatz adaptiert. Die Kopplung wird in SoMoX zwischen zwei

Klassenmengen bestimmt, anstatt zwischen zwei Klassen, wie es in der objektorientierten Variante der Fall ist.

Im Folgenden werden die beiden Metriken *Coupling* und *InterfaceViolation* vorgestellt, da diese zwei der wichtigsten Indikatoren für das Clustering eines komponentenbasierten Systems sind. Anschließend wird in dem darauf folgenden Abschnitt erläutert, wie die Metriken in Strategien zur Bestimmung von Komponenten eingesetzt werden.

Coupling Die Kopplung zwischen zwei Komponenten wird anhand der gegenseitigen Zugriffe gemessen. Sei A die Menge der Klassen einer Komponente und B die Menge der Klassen einer anderen Komponente. Sei außerdem $R(A, B)$ die Anzahl der Zugriffe von den Klassen aus A auf Klassen aus der Menge B. Dann ist die Kopplung zwischen diesen beiden Komponenten wie folgt definiert

$$Coupling(A, B) := \frac{R(A, B)}{R(A, all)} = \frac{InterneZugriffe}{ExterneZugriffe}$$

wobei *all* die Menge aller Klassen des Systems ist. Die Kopplung zwischen zwei Komponenten ist nicht kommutativ.

InterfaceViolations Verletzungen von Schnittstellen zwischen zwei Komponenten werden über die Metrik *InterfaceViolations* gemessen. Dabei werden Zugriffe gezählt, welche aus einer Komponente in eine andere Komponente führen und dabei auf einen konkreten Typ zugreifen, anstatt eine Schnittstelle zu nutzen. Die Anzahl der Zugriffe wird zu der Anzahl aller Zugriffe ins Verhältnis gesetzt. Der Metrikwert ist maximal, wenn alle Zugriffe über konkrete Typen stattfinden. Sei $RI(A, B)$ die Anzahl der Zugriffe aus der Klassenmenge A in die Klassenmenge B, wobei die Zugriffe Schnittstellen umgehen, dann ist der Wert der Metrik

$$InterfaceViolations(A, B) := \frac{RI(A, B)}{R(A, all)}$$

$R(A, all)$ ist wie bereits bei der Metrik *Coupling* die Anzahl aller Zugriffe auf Klassen außerhalb der Menge A. Die Metrik ist nicht kommutativ.

2.3.2.2 Strategien zur Komponentenerkennung

Strategien in SoMoX entscheiden darüber, ob zwei Komponenten zu einer neuen Komponente vereint bzw. zusammengesetzt werden oder ob zum Beispiel ein Interface aus der Implementierung einer Komponente Teil ihrer Schnittstellendefinition sein muss. Diese Entscheidungen werden auf Basis von Metrikwerten getroffen. Eine Strategie wird durch eine Berechnungsvorschrift definiert. Eine Entscheidung wird getroffen, indem ermittelt wird, ob ein zuvor festgelegter Grenzwert überschritten wird. Ähnlich zu den Metriken können Strategien andere Strategien wie-

derverwenden, um komplexe Zusammenhänge zwischen Eigenschaften eines Systems widerzuspiegeln.

Im Folgenden werden zunächst die Strategien *InterfaceAdherence* und *InterfaceBypassing* vorgestellt, um zu erläutern, wie Metriken in eine Strategie einfließen. Anschließend werden die Strategien *ComponentMerge* und *ComponentComposite* näher beleuchtet, da diese über die Komponentenzusammensetzung in einer Softwarearchitektur entscheiden und dabei andere Strategien wiederverwenden.

Die Strategien *ConsistentNaming, HierarchyMapping, AbstractConreteBalance* und *SubsystemComponent* werden hier aus Platzgründen nicht näher erläutert, können aber in der Dissertation von Krogmann [Kro10] nachgeschlagen werden.

InterfaceBypassing Die Strategie *InterfaceBypassing* entscheidet darüber, ob Schnittstellen zwischen zwei Komponenten verletzt werden und dient damit als Indikator dafür, dass Komponenten vereint werden müssen. Im Gegensatz zu der Metrik *InterfaceViolation* berücksichtigt die Strategie auch die Kopplung zwischen zwei Komponenten mit Hilfe der Metrik *Coupling*. Die Idee ist dabei, dass Verletzungen von Schnittstellen zwischen zwei Komponenten nur dann im Rahmen eines Clusterings berücksichtigt werden sollen, wenn ein gewisser Grad an Kopplung zwischen den Klassen beider Komponenten herrscht. Die Strategie *InterfaceBypassing* ist wie folgt definiert

$$InterfaceBypassing(A,B) :=$$

$$\begin{cases} max(IV(A,B), IV(B,A)) & \text{if } max(Coupling(A,B), Coupling(B,A)) > \epsilon \\ 0 & \text{else} \end{cases}$$

Der Wert der Strategie ergibt sich aus dem Maximum der beiden *InterfaceViolation*-Werte ($IV(A,B)$ und $IV(B,A)$) zwischen den Komponenten, wenn ein Mindestschwellwert ϵ an Kopplung zwischen den Klassen der beiden Komponenten überschritten wurde. Andernfalls wird die Strategie mit dem Wert 0 ausgewertet.

InterfaceAdherence Um festzustellen, ob vorhandene Schnittstellen zweier Komponenten eingehalten werden, wird die Strategie *InterfaceAdherence* verwendet. Die Einhaltung von Schnittstellen ist ein Indiz dafür, dass Komponenten in Kompositionsbeziehung zueinander stehen. Die Strategie stützt sich in ihrer Berechnung auf die Metriken *InterfaceViolation* und *Coupling*. Die Berechnung ist wie folgt definiert

$$InterfaceAdherence(A,B) :=$$

$$\begin{cases} 1 - max(IV(A,B), IV(B,A)) & \text{if } max(Coupling(A,B), Coupling(B,A)) > \epsilon \\ 0 & \text{else} \end{cases}$$

Der Wert der *InterfaceViolation*-Metrik wird von 1 abgezogen. Der Wert der

Strategie ist daher maximal, wenn der Wert der *InterfaceViolation*-Metrik minimal ist. Da der Wert der *InterfaceViolation*-Metrik auch minimal ist, wenn keine gegenseitigen Schnittstellen verwendet werden, wird dieser nur dann verwendet, wenn ein Schwellwert ϵ für die Kopplung (*Coupling*) zwischen den beiden Komponenten überschritten wird. Die Summe der beiden Werte von *InterfaceAdherence* und *InterfaceBypassing* ergibt immer 1, wenn Kopplung zwischen zwei Komponenten herrscht. Ist der Wert der einen Strategie minimal, so ist der Wert der anderen Strategie maximal und umgekehrt.

ComponentMerge Um zu entscheiden, ob zwei Komponenten zu einer neuen Komponente vereint werden, wird die Strategie *ComponentMerge* ausgewertet. Die Auswertung der Strategie stützt sich auf Werte anderer Strategien, wie die *InterfaceBypassing*-Strategie, die als Indikator für das Vereinen zweier Komponenten gewertet werden. Die Werte der verwendeten Strategien werden mit zuvor von einem Reengineer angegebenen Gewichten w_i gewichtet, aufsummiert und durch die Anzahl aller verwendeten Strategien dividiert. Die Berechnung ist folgt definiert

$$
\begin{aligned}
ComponentMerge(A,B) := \; & (w_1 * InterfaceBypassing(A,B) + \\
& w_2 * ConsistentNaming(A,B) + \\
& w_3 * AbstractConcreteBalance(A,B) + \\
& w_4 * HierarchyMapping(A,B)) \\
& /4
\end{aligned}
$$

Neben der Verletzung von Schnittstellen berücksichtigt die Strategie bei ihrer Auswertung auch Namensähnlichkeiten von Klassen (*ConsistentNaming*), das Verhältnis von abstrakten zu konkreten Implementierungselementen (*AbstractConcreteBalance*) und die Paket- bzw. Ordnerstruktur (*HierarchyMapping*) in der Implementierung.

ComponentComposition Ob zwei Komponenten in Kompositionsbeziehung zueinander stehen und daher im Rahmen des Clusterings zu einer neuen Komponente zusammengesetzt werden sollen, wird mit Hilfe der Strategie *ComponentComposition* entschieden. Diese stützt sich bei ihrer Auswertung auf andere Strategien, die als ein Indikator für eine solche Beziehung erachtet werden. Der Unterschied zu der Strategie *ComponentMerge* besteht zum einen darin, dass bei einer Kompositionsbeziehung die Einhaltung von vorhandenen Schnittstellen gefordert wird. Daher wird bei der Berechnung eines Wert der Strategie *ComponentComposition* anstatt der Strategie *InterfaceBypassing*, die Strategie *InterfaceAdherence* verwendet, welche die Einhaltung von Schnittstellen repräsentiert. Zum anderen kann eine der beiden Komponenten ein Subsystem der anderen Komponente bilden. Dies wird durch eine weitere Strategie *SubsystemComponent* als Indikator für eine Kompositionsbeziehung zwischen zwei Komponenten berücksichtigt.

$$ComponentComposition(A,B) := \begin{array}{l}(w_1 * InterfaceAdherence(A,B)+ \\ w_2 * ConsistentNaming(A,B)+ \\ w_3 * AbstractConcreteBalance(A,B)+ \\ w_4 * HierarchyMapping(A,B)) \\ w_5 * SubsystemComponent(A,B)) \\ /5\end{array}$$

Die Strategien *ConsistentNaming, AbstractConcreteBalance* und *HierarchyMapping* werden in beiden Strategien *ComponentMerge* und *ComponentComposition* als ein Indikator für eine Komponente gewertet. Ob zwei Komponenten im Rahmen eines Clusterings zu einer neuen Komponente vereint oder zusammengesetzt werden, wird daher maßgeblich über die Strategien *InterfaceAdherence, InterfaceBypassing* und *SubsystemComponent* entschieden. Da *InterfaceAdherence* und *InterfaceBypassing* sich beide direkt auf die Metriken *InterfaceViolation* und *Coupling* beziehen, sind diese die entscheidenden Metriken in einem Softwaresystem.

2.3.3 Grenzen des Clustering-basierten Reverse Engineerings

Der Clustering-basierte Reverse-Engineering-Ansatz unterliegt einigen Einschränkungen. Die wichtigsten Einschränkungen des Ansatzes werden in diesem Abschnitt erläutert.

Beschränkung auf komponentenbasierte Softwaresysteme Clustering-basierte Reverse-Engineering-Ansätze rekonstruieren die Softwarearchitektur eines Systems. Zwar hat jedes Softwaresystem eine Softwarearchitektur [BT04], doch ist der Clustering-basierte Ansatz von SoMoX auf die Analyse von zumindest teilweise komponentenbasiert entwickelten Softwaresystemen beschränkt. Der Ansatz beruht auf der strengen Komponentendefinition von Szyperski [Szy02], wonach die Schnittstellen von Komponenten getrennt von ihrer Implementierung definiert werden müssen. Andernfalls ist der Clustering-basierte-Ansatz nicht sinnvoll anwendbar, da grundlegende Annahmen über die Eigenschaften eines komponentenbasierten Softwaresystems, die durch die Metriken erfasst werden, in einem solchen System nicht zutreffen. Im Extremfall besteht die Softwarearchitektur dann entweder nur aus einer Komponente oder sie repräsentiert jede Klasse des Systems durch eine Komponente. Beide Varianten helfen einem Reengineer jedoch nicht dabei, das System zu verstehen.

Eingeschränkter Erkenntnisgewinn Die Softwarearchitektur eines Systems, die mit Hilfe eines Clustering-basierten Reverse-Engineering-Ansatzes rekonstruiert wird, kann einem Reengineer beim Verstehen des analysierten System helfen, indem es ihm einen Überblick über die Komponenten des Systems verschafft. Jedoch bleibt dabei der Zweck einer Komponente verborgen und muss manuell oder durch

Einsatz anderer Ansätze, wie dem musterbasierten Reverse Engineering, ermittelt werden. In großen Systemen, in denen eine Komponente umfangreiche Teile der Implementierung eines Systems repräsentieren kann, kann dies sehr aufwändig sein.

Abstraktion von Software-Eigenschaften durch Metriken Metriken repräsentieren Eigenschaften von Software in einem numerischen Wert und abstrahieren diese daher stark. Bei der Abstraktion solcher Eigenschaften gehen teilweise wichtige Details über die Softwarearchitektur verloren, wie zum Beispiel bei der *InterfaceViolation*-Metrik. Diese setzt die Anzahl der Zugriffe zwischen zwei Komponenten zu der Anzahl aller Zugriffe einer Komponente ins Verhältnis. Details über die Art der Zugriffe (Methodenaufruf, Zugriff auf Variablen oder Vererbung) gehen bei der Abstraktion verloren. Der Kontrollfluss in einem Softwaresystem kann auf Grund seiner Komplexität nur schwer oder gar nicht durch Metriken erfasst werden. Solche Informationen können jedoch Aufschluss über mögliche Schwachstellen in einem System und dessen Softwarearchitektur geben.

Anfälligkeit für Schwachstellen Eine grundlegende Beobachtung dieser Arbeit ist, dass Schwachstellen in einem System Einfluss auf dessen Softwarearchitektur ausüben können. Dadurch üben Schwachstellen auch auf das Clustering Einfluss aus. Eine detailliertere Beschreibung dieses Problems ist in Kapitel 3 zu finden.

2.4 Musterbasiertes Reverse Engineering

Beim Design von Software tauchen bestimmte Herausforderungen immer wieder auf, wie beispielsweise das Warten auf ein Ereignis, um anschließend darauf zu reagieren. Solche Herausforderungen müssen nicht jedes Mal neu gelöst werden, da oft bereits bewährte Lösungen in Form von Entwurfsmustern existieren [GHJV95]. Im Falle des Beispiels wäre das von Gamma et al. vorgestellte Observer-Muster eine typische Lösung für die Herausforderung. Eine solche Lösung kann an die eigenen Anforderungen angepasst werden, was den Zeit- und Kostenaufwand der Softwareentwicklung gegenüber einer Neuentwicklung gering hält.

Die wiederholte Verwendung von bewährten Lösungen bildet die Grundidee für das musterbasierte Reverse Engineering. Der charakteristische Teil einer Lösung wird durch ein Muster spezifiziert, so dass anschließend in der Implementierung beliebiger Softwaresysteme nach Vorkommen des Musters gesucht werden kann. Wird ein Vorkommen eines bestimmten Musters identifiziert, so handelt es sich bei dem Mustervorkommen mit großer Wahrscheinlichkeit um eine Anwendung des Entwurfsmusters. Zu einem Entwurfsmuster kann der Reengineer in dessen Dokumentation nachschlagen, um den Zweck des Entwurfsmusters zu erfahren und damit auch des Teils der Software, der durch die Anwendung des Entwurfsmusters implementiert wird. Derartige Informationen können dabei helfen, ein System besser zu verstehen.

Antimuster [BMMM98] beschreiben ebenfalls typische und oft verwendete Lösungen für wiederkehrende Herausforderungen beim Entwickeln von Software. Im Gegensatz zu den Entwurfsmustern sind sie jedoch schlechte Lösungen, da sie sich beispielsweise durch unnötige Kopplung negativ auf ein Softwaresystem auswirken können und damit den Aufwand von Wartungsarbeiten des System erhöhen können. Neben den Entwurfsmustervorkommen kann auch die Erkennung von Antimustervorkommen das Ziel des musterbasierten Reverse Engineerings sein, um Teile der Software zu identifizieren, die überarbeitet werden sollten.

An der Universität Paderborn wird im Fachgebiet Softwaretechnik seit Jahren das musterbasierte Reverse-Engineering-Werkzeug Reclipse entwickelt [NSW+02, Nie04, Tra06, Wen07, DMT10]. In diesem Abschnitt werden alle Grundlagen des musterbasierten Reverse Engineerings am Beispiel von Reclipse dargestellt.

Im folgenden Abschnitt 2.4.1 wird erläutert, wie Muster in Reclipse spezifiziert werden. In Abschnitt 2.4.2 wird der Prozess der Mustersuche vorgestellt. Dabei werden Mustervorkommen zu den zuvor spezifizierten Mustern in einem Softwaresystem gesucht. Anschließend werden Einschränkungen des musterbasierten Reverse Engineerings in Abschnitt 2.4.3 vorgestellt.

2.4.1 Spezifikation von Mustern

Entwurfsmuster, wie zum Beispiel *Singleton* von Gamma et al. [GHJV95], werden meist informell beschrieben, wobei die Beschreibung oft ein Klassendiagramm als Beispiel für das Muster enthält. Abbildung 2.5 zeigt ein solches Beispiel für das Singleton-Entwurfsmuster in einem Klassendiagramm.

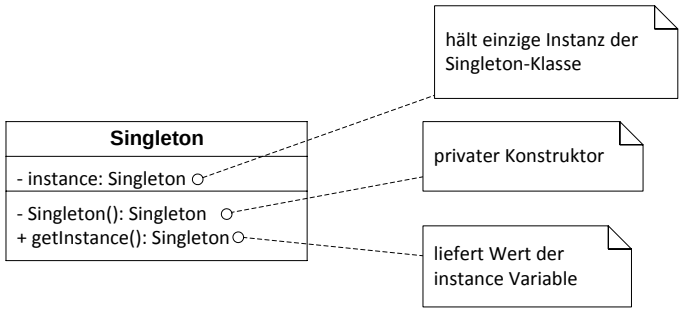

Abbildung 2.5: Klassendiagramm des Singleton-Entwurfsmusters [GHJV95]

Das Singleton-Muster findet Anwendung, wenn es von einer Klasse nur eine Instanz zur Laufzeit eines Softwaresystems geben soll. Um dies zu gewährleisten, erhält die Klasse, die das Singleton-Muster implementiert, eine statische, öffentlich zugängliche Zugriffsmethode, welche die einzige Instanz der Klasse zurückliefert.

Damit die Instanziierung der Singleton-Klasse aus anderen Klassen unterbunden wird und nur noch innerhalb der Singleton-Klasse möglich ist, wird die Sichtbarkeit ihres Konstruktors auf private gesetzt. Zusätzlich erhält die Singleton-Klasse eine Klassenvariable vom Typ der Singleton-Klasse, welche ebenfalls die Sichtbarkeit private hat. Diese Klassenvariable wird beim ersten Aufruf der öffentlichen Zugriffsmethode durch den privaten Konstruktor initialisiert und hält von da an die Referenz auf die einzige Instanz der Klasse, so dass die öffentliche Zugriffsmethode von da an nur noch den Wert der Klassenvariable zurückliefert. Da die Instanziierung der Singleton-Klasse nur in der öffentlichen Zugriffsmethode implementiert ist und diese die Instanziierung nur einmal in dem Lebenszyklus der Software vornimmt, ist somit sichergestellt, dass nur eine Instanz der Singleton-Klasse existiert.

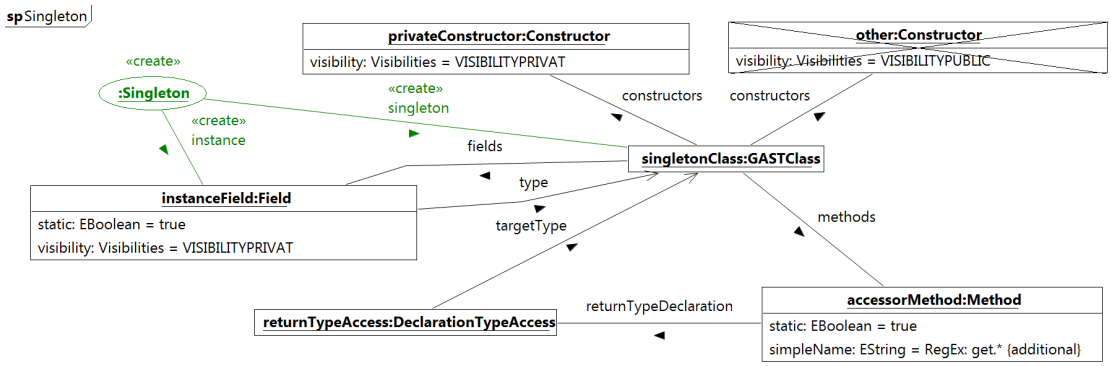

Abbildung 2.6: Musterspezifikation des Singleton-Musters

Muster werden in Reclipse als typisierte und gegebenenfalls attributierte Graphen formalisiert, um anschließend mittels Graphmatching nach einem Mustergraphen in einem Wirtsgraphen, der die Implementierung eines Softwaresystems repräsentiert, zu suchen. Dazu wird ein Metamodell benötigt, welches die Typen und Attribute des Graphen definiert und die Implementierung eines Softwaresystems repräsentieren kann. Ein abstrakter Syntaxbaum, wie der in Abschnitt 2.2.1 vorgestellte GAST, erfüllt diese Voraussetzungen. Im Folgenden wird daher die Musterspezifikation am Beispiel des Singleton-Entwurfsmusters auf Basis des GAST-Metamodells vorgestellt. Abbildung 2.6 zeigt die Musterspezifikation des Singleton-Entwurfsmusters in Reclipse.

Der Graph einer Musterspezifikation kann in vier Teile unterteilt werden, die verschiedene Bedeutungen für das Matching haben. Dabei ist ein Teil des Graphen notwendig (1). Dieser beschreibt den Kern des Musters. Außerdem können Teile spezifiziert werden, die zusätzlich zu dem notwendigen Teil existieren können (2) oder für das erfolgreiche Matching nicht existieren dürfen (3). Ein anderer Teil wird erzeugt (4), wenn das Matching erfolgreich war.

Notwendiger Teil Der schwarze Teilgraph der Musterspezifikation beschreibt den Teil des Musters, der existieren muss, damit das Matching erfolgreich sein kann. Während der Mustersuche wird nach Teilgraphen im Wirtsgraphen gesucht, die isomorph zu dem schwarzen Objektgraphen im Muster sind und dessen Bedingungen für die Typen der Knoten und Kanten und deren Attribute erfüllen. Für das Singleton-Entwurfsmuster wird daher zunächst ein Objekt singletonClass vom Typ GASTClass benötigt, das die Singleton-Klasse repräsentiert. Ein Objekt instanceField vom Typ Field repräsentiert die Klassenvariable, welche die Instanz der Singleton-Klasse hält und mit dem Objekt singletonClass über eine Referenz vom Typ type verbunden ist. Eine Referenz fields von singletonClass zu instanceField beschreibt, dass das instanceField ein Attribut der durch singletonClass repräsentierten Klasse ist. Der Typ der Klassenvariable und der Singleton-Klasse muss gleich sein. Für das Objekt instanceField existieren Attributbedingungen, die besagen, dass das Attribut static den Wert true und das Attribut visibility den Wert VISIBILITYPRIVAT haben muss, um der Beschreibung des Singleton-Entwurfsmusters zu entsprechen. Da der Konstruktor einer Singleton-Klasse eine eingeschränkte Sichtbarkeit haben muss, referenziert das singletonClass-Objekt ein Objekt privateConstructor vom Typ Constructor über die Kante constructors. Das privateConstructor-Objekt hat eine Attributbedingung für die eingeschränkte Sichtbarkeit.

Um die Musterspezifikation zu vervollständigen und die öffentliche Zugriffsmethode der Singleton-Klasse in die Spezifikation mit aufzunehmen, wurden die Objekte accessorMethod vom Typ Method und returnTypeAccess vom Typ DeclarationTypeAccess angelegt. Das Objekt accessorMethod repräsentiert die statische, öffentliche Zugriffsmethode, weshalb es eine Attributbedingung für das Attribut static mit dem Wert true hat. Das Objekt returnTypeAccess repräsentiert den Rückgabetyp der Methode accessorMethod. Über die Kante targetType wird das Objekt singletonClass als Rückgabetyp referenziert.

Zusätzlicher Teil Eine Musterspezifikation kann einen zusätzlichen Teil spezifizieren, der zu einem Muster gehören kann, aber nicht zwingend notwendig ist. Ein solcher Teil wird als additional markiert. Zusätzliche Objekte oder Annotationen werden gestrichelt dargestellt. Zusätzliche Teile einer Musterspezifikation helfen bei der Bewertung von Mustern, um *True-Positives* von *False-Positives* zu unterscheiden [Wen07, Tra06].

Die *Singleton*-Musterspezifikation enthält lediglich eine Attributbedingung in Form eines regulären Ausdrucks für den Namen der öffentlichen Zugriffsmethode, die additional ist. Der reguläre Ausdruck beschreibt einen Namen, der mit dem Präfix "get" anfangen muss.

Nicht erlaubter Teil Eine Musterspezifikation kann Teile beschreiben, die für ein erfolgreiches Matching nicht gefunden werden dürfen. Solche Teile werden als negative markiert. Negative Objekte oder Kanten werden durchgestrichen darge-

stellt.

Um zu spezifizieren, dass eine Singleton-Klasse keinen öffentlichen Konstruktor hat, wurde ein negatives Objekt other vom Typ Constructor mit einer Attributbedingung für das Attribut visibility und dem Wert VISIBILITYPUBLIC angelegt. Das Objekt singletonClass ist über die Referenz constructors mit dem negativen Objekt verbunden und darf daher keinen Konstruktor mit den beschriebenen Eigenschaften haben.

Erzeugter Teil Der grüne Teil einer Musterspezifikation ist eine Annotation (Ellipse), die einen Musterfund markiert und erzeugt wird, wenn der notwendige Teil in einem Wirtsgraph gefunden wurde und sichergestellt ist, dass der negative Teil nicht existiert. Grüne Elemente werden deshalb auch mit dem Label ≪create≫ markiert. Zu jedem Muster existiert ein eigener Annotationstyp, der die Rollen von Objekten in dem Muster festlegt, wobei die Rollen durch die grünen Kanten und einen Rollennamen definiert werden. Im Beispiel ist die Annotation vom Typ Singleton und definiert die Rollen singleton und instance.

Zusätzliche Konstrukte der Musterspezifikation Neben den hier vorgestellten Konstrukten zur Spezifikation eines Musters, existieren noch einige mehr. Die wichtigsten zum Verständnis dieser Arbeit benötigten Konstrukte werden im Folgenden vorgestellt. Weitere Details zur Musterspezifikation, -Suche und deren Bewertung können in den Arbeiten [Nie04, Wen07, Tra06] nachgelesen werden.

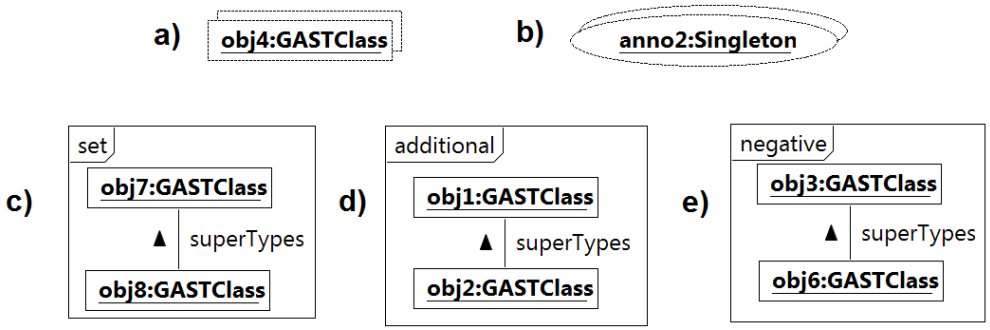

Abbildung 2.7: Notation weiterer Konstrukte der Musterspezifikation in Reclipse

Bei der Spezifikation von Mustern ist es oft notwendig, bestimmte Mengen von Objekten gleichen Typs repräsentieren zu können. Das Entwurfsmuster *Observer* [GHJV95] beschreibt zum Beispiel eine Subject-Klasse, auf die mehrere konkrete Implementierungen der abstrakten Observer-Klassen hören, um auf deren Ereignisse zu reagieren. Die Anzahl der Observer kann von Fall zu Fall einer Anwendung des Musters variieren. Um möglichst viele Varianten des Musters durch eine Musterspezifikation abzudecken, kann man in diesem Fall das Objekt, welches die Spezialisierungen der Observer-Klasse repräsentiert, als Menge deklarie-

ren. Die Mustersuche in Reclipse annotiert daraufhin Anwendungen des *Observer*-Entwurfsmusters mit beliebig vielen Spezialisierungen der Observer-Klasse.

Eine Objektmenge wird als zwei übereinander liegende Objekte in der Musterspezifikation visualisiert, wie in Abbildung 2.7 a) dargestellt.

Auch Annotationsmengen können aus den gleichen Gründen bei der Spezifikation von Mustern hilfreich sein. Sie werden entsprechend als zwei übereinander liegende Annotationen dargestellt. Ein Beispiel für eine Menge von Singleton-Annotationen wird in Abbildung 2.7 b) dargestellt.

Oft reicht es nicht aus, einzelne Objekte als Objektmenge zu deklarieren, da ein ganzer Teilgraph eines Musters mehrfach vorkommen kann. Um einen solchen Teilgraph in einer Musterspezifikation als mehrfach vorkommenden Teilgraph zu deklarieren, werden daher Set-Fragmente verwendet. Diese können einen Teilgraph einer Musterspezifikation enthalten und repräsentieren dessen mehrfaches Vorkommen in der Musterspezifikation. Die Notation des Set-Fragments wird in der Abbildung 2.7 c) dargestellt.

Neben Set-Fragmenten existieren außerdem Additional- und Negative-Fragmente, die einen Teil der Musterspezifikation als additional bzw. negative kennzeichnen. Die Bedeutung von Teilen, die als additional bzw. negative markiert sind, wurde in den vorhergehenden Absätzen erläutert. Ein Beispiel zu der Notation von solchen Fragmenten ist in der Abbildung 2.7 d) und e) zu finden.

2.4.2 Suche nach Mustervorkommen

Nachdem ein Musterkatalog[1] spezifiziert wurde, müssen einige Arbeitsschritte durchlaufen werden, um Vorkommen der spezifizierten Muster in einem Softwaresystem identifizieren zu können. Die Mustersuche in Reclipse benötigt zwei Eingaben. Zum einen wird der Musterkatalog benötigt. Zum anderen wird ein AST benötigt, der als Wirtsgraph in der Suche nach Mustervorkommen verwendet wird. Der AST repräsentiert die Implementierung eines Softwaresystems und wird von einem Parser bereitgestellt.

Abbildung 2.8 gibt einen Überblick über den Reverse-Engineering-Prozess in Reclipse nach dem ein Musterkatalog erstellt und ein zu analysierendes Softwaresystem mit Hilfe eines Parsers eingelesen wurde. In Reclipse wird für jede Musterspezifikation in einem Musterkatalog ein musterspezifischer Suchalgorithmus generiert, der das Matching des Musters implementiert. Der Suchalgorithmus wird mit Hilfe von Story-Diagrammen [Zün01] formalisiert. Details zu der Generierung der Suchalgorithmen sind in den Arbeiten [Nie04, Wen07, Foc10] zu finden.

Der Inferenz-Mechanismus von Reclipse steuert die Mustersuche und verwaltet die Suchergebnisse. Die Inferenz bestimmt den Kontext für die Anwendung eines Suchalgorithmus. Der Kontext ist in diesem Zusammenhang ein Objekt im Wirtsgraph, das den gleichen Typ hat, wie ein Objekt mit einer Rolle in einer Musterspezifikation.

[1] Ein Musterkatalog ist eine Sammlung einzelner Musterspezifikationen.

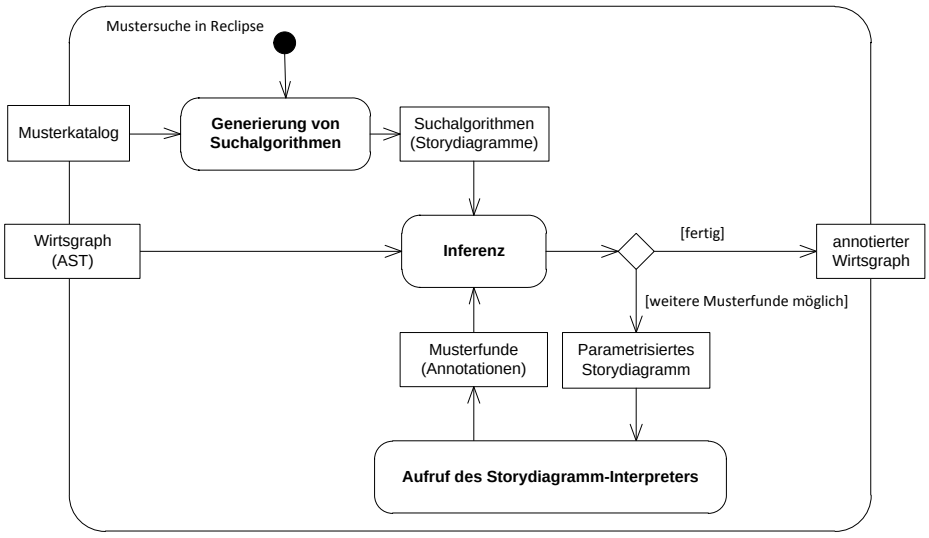

Abbildung 2.8: Visualisierung der nötigen Arbeitsschritte im Rahmen einer Mustersuche in Reclipse als Aktivitätendiagramm

Das Matching des Musters findet durch die Anwendung des entsprechenden Suchalgorithmus statt. Dazu werden die Story-Diagramme, die den Suchalgorithmus implementieren, durch den zuvor bestimmten Kontext parametrisiert und mit Hilfe des Story-Diagramm-Interpreters [GHS09] interpretiert. Wird dabei ein Mustervorkommen in dem Wirtsgraphen erkannt, so wird der Teil des Wirtsgraphen, der zu dem Mustervorkommen gehört, annotiert und als Ergebnis zurückgeliefert. Details zur Einbindung des Story-Diagramm-Interpreters im Rahmen der Inferenz in Reclipse sind in der Masterarbeit von Fockel [Foc10] zu finden. Zuvor musste ausführbarer Java-Code aus den Story-Diagrammen generiert werden [FNTZ00, GSR05] und auf den Wirtsgraphen angewendet werden.

Die Mustersuche in Reclipse wird beendet, wenn keine weiteren Mustervorkommen gefunden werden können. Dies ist der Fall, wenn kein Kontext mehr gefunden werden kann, der nicht bereits auf Mustervorkommen analysiert wurde. Das Ergebnis der Mustersuche ist ein annotierter Wirtsgraph.

2.4.3 Grenzen des musterbasierten Reverse Engineerings

Der musterbasierte Reverse-Engineering-Ansatz unterliegt einigen Einschränkungen. Die wichtigsten Einschränkungen des Ansatzes werden in diesem Abschnitt erläutert.

Eingeschränkte Anwendbarkeit durch hohen Ressourcenbedarf Die Mustersuche wird durch das Subgraph-Isomorphie-Problem beschrieben, wobei ein Mus-

ter den gesuchten Graph und die untersuchte Software den Wirtsgraph bildet. Dabei gilt es zu entscheiden, ob der Wirtsgraph einen zum gesuchten Graph isomorphen Subgraph enthält. Das Subgraph-Isomorphie-Problem ist im Allgemeinen NP-vollständig, kann aber auch unter bestimmten Bedingungen in polynomieller Zeit entschieden werden [Epp99]. Die graphbasierte Mustersuche ist daher mit hohem Bedarf an Speicher und CPU-Leistung bzw. -Zeit verbunden. Für große Softwaresysteme ist der musterbasierte Reverse-Engineering-Ansatz daher nicht praktikabel, da eine Analyse zu viel Zeit in Anspruch nimmt oder gar nicht erst möglich ist.

Eingeschränkte Anwendbarkeit durch zu viele Musterfunde Eine Herausforderung des musterbasierten Reverse-Engineering-Ansatzes ist die mit der Größe eines analysierten Systems steigende Anzahl von Musterfunden. Grund für die hohe Anzahl von Musterfunden ist neben dem Umfang eines analysierten Systems das geringe Abstraktionslevel von Mustern. Bei steigender Anzahl von Musterfunden steigt auch der Aufwand einer Auswertung der Analyseergebnisse, da der Zusammenhang einzelner Musterfunde von einem Reengineer manuell erschlossen werden muss.

Musterspezifikation bestimmt Qualität der Ergebnisse Die Ergebnisse des musterbasierten Reverse Engineerings werden maßgeblich durch die Spezifikation eines Musters beeinflusst. Ist die Spezifikation zu ungenau, weil zu wenige Eigenschaften eines Musters spezifiziert wurden, so ist mit einem erhöhten Maß an *False-Positives* unter den Mustervorkommen zu rechnen. Enthält eine Musterspezifikation hingegen viele Eigenschaften eines Musters, so kann es sein, dass Implementierungsvarianten des Musters nicht erkannt werden und für den Reengineer verborgen bleiben.

3 Problemstellung und Lösungsidee

Die meisten der existierenden Reverse-Engineering-Ansätze lassen sich in die Kategorien *Clustering-basiert* und *musterbasiert* einordnen [Sar03]. Diese unterliegen jedoch einigen Einschränkungen. Die Grenzen des Clustering-basierten Reverse Engineerings wurden in Abschnitt 2.3.3 erläutert, die des musterbasierten Reverse Engineerings wurden in Abschnitt 2.4.3 beschrieben.

Dieses Kapitel erläutert in dem folgenden Abschnitt 3.1 die Einschränkungen des Clustering-basierten Verfahrens, die im Rahmen dieser Arbeit durch die Kombination mit einem musterbasierten Verfahren reduziert werden. Eine kurze Beschreibung der in dieser Arbeit verfolgten Lösungsidee für das Problem folgt in Abschnitt 3.2. Abschließend werden die besonderen konzeptionellen Herausforderungen in Abschnitt 3.3 und die technischen Herausforderungen in Abschnitt 3.4 zusammengefasst.

3.1 Problemstellung

Clustering-basierte Reverse-Engineering-Ansätze rekonstruieren die konkrete Softwarearchitektur [DP09] aus der Implementierung eines untersuchten Softwaresystems. Dabei werden die für das System gemessenen Metrikwerte zur Herleitung von Komponenten und Konnektoren genutzt. Für das Clustering eines Systems können Metriken wie beispielsweise *Coupling* und *Cohesion* [Mar94] oder auch komplexere Metriken wie *Slice Layer Architecture Quality* (SLAQ) [CKK08] verwendet werden.

Ein hoher Metrikwert für Kopplung zwischen zwei Klassen erhöht beispielsweise die Wahrscheinlichkeit, dass diese gemeinsam in einer Komponente gruppiert werden. Klassen mit geringer Kopplung werden hingegen mit hoher Wahrscheinlichkeit verschiedenen Komponenten zugeordnet.

In einem nach Szyperski [Szy02] komponentenbasiert entwickelten System sollte jede Softwarekomponente ihre verwendeten und bereitgestellten Schnittstellen definieren. Kommunikation mit anderen Komponenten in Form von Methodenaufrufen sollte nur über deren Schnittstellen geschehen, um die Austauschbarkeit von Komponenten zu gewährleisten. Dies steht im Einklang mit dem Entwurfsprinzip "Program to an interface, not an implementation" von Gamma et al. [GHJV95]. Lediglich Klassen, die einer gemeinsamen Komponente angehören, dürfen in einem solchen System miteinander kommunizieren, ohne ihre Schnittstellen zu verwenden. Aufrufe von Methoden, die nicht Teil einer Schnittstelle sind, erhöhen die

Kopplung zwischen den beteiligten Klassen. Kopplung ist daher ein wichtiger Indikator für die Gruppierung von mehreren Klassen in einer Komponente.

Da die Implementierung eines Systems die Grundlage der Messungen von Metrikwerten ist, kann das Reverse Engineering mit einem Clustering-basierten Ansatz immer nur die konkrete Softwarearchitektur des Systems rekonstruieren. Softwaresysteme unterliegen jedoch ständiger Wartung und werden im Laufe der Zeit immer wieder verändert und um neuen Funktionalität erweitert. Auf Grund von mangelndem Verständnis des zu ändernden Systems können Änderungen an dessen Implementierung zu Schwachstellen im System führen [Par94]. Schwachstellen und Erweiterungen verändern die Softwarearchitektur eines Systems. Wenn die konzeptionelle Softwarearchitektur beispielsweise in der Design-Dokumentation nicht angepasst wird, entsteht eine Abweichung der konkreten von der konzeptionellen Softwarearchitektur.

```
interface IA {
  m1();
}

interface IB {
  m2();
}

class A implements IA {
  IB ib;
  m1(){
    B b = (B) ib;
    b.m3();
  }
}

class B implements IB {
  m2() {…}
  m3() {…}
}
```

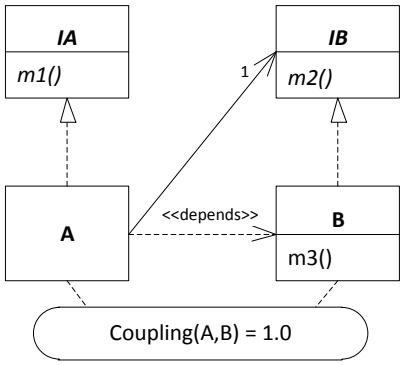

Abbildung 3.1: Beispiel: Code und Klassendiagramm zu einer Interface Violation

Schwachstellen erhöhen häufig die Kopplung im System. Dies steht im Gegensatz zu dem Bemühen eines Softwarearchitekten um schwache Kopplung und eine modulare Architektur. Um zu zeigen, wie eine Schwachstelle die Kopplung erhöht und damit eine Clustering-basierte Analyse beeinflussen kann, wird in Abbildung 3.1 eine Interface Violation als Beispiel für eine Schwachstelle dargestellt. Die

Abbildung enthält Quellcode, sowie das den Quellcode repräsentierende Klassendiagramm.

Die Klassen A und B implementieren jeweils ihre Interfaces IA bzw. IB. Die Klasse B implementiert außerdem eine Methode m3(), die nicht in ihrem Interface enthalten ist. In der Klasse A wird die Klasse B über ihr Interface IB referenziert. Klasse A ruft jedoch die Methode m3() von B auf. Dazu wird die Referenz ib vom Typ IB in den konkreten Typ B umgewandelt (Type Cast), da sonst die Methode m3() für die Klasse A nicht sichtbar ist. Damit umgeht Klasse A das von B bereitgestellte Interface IB und verletzt diese Schnittstelle. Die Interface Violation führt zu einer hohen Kopplung zwischen A und B, wie in Abbildung 3.1 dargestellt. In diesem Beispiel ist der Wert der Metrik $Coupling(A, B) = 1$ und damit maximal. Die Berechnung von Metrikwerten für Kopplung wird in Abschnitt 2.3.2 beschrieben.

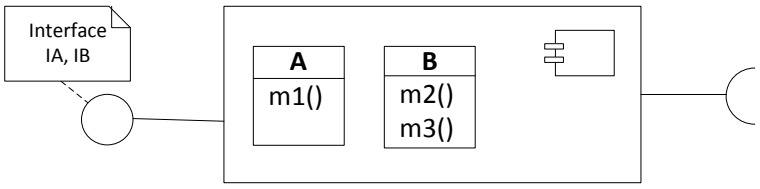

Abbildung 3.2: Konkrete Softwarearchitektur des Beispiels

Die starke Kopplung zwischen A und B bewirkt damit, dass A und B während einer Clustering-basierten Analyse einer Komponente zugeordnet werden, wie es in Abbildung 3.2 dargestellt ist.

Die Interface Violation im Beispiel ist leicht zu beseitigen, indem die Methode m3() dem Interface IB hinzugefügt wird und die Typumwandlung in Klasse A entfernt wird. Der entsprechende Code ist in Abbildung 3.3 dargestellt. Klasse A verliert dadurch die Abhängigkeitsbeziehung zu Klasse B, die im Klassendiagramm in Abbildung 3.1 zu sehen ist. Das Entfernen dieser Abhängigkeit bewirkt, dass der Wert der Kopplung zwischen A und B nun $Coupling(A, B) = 0$ ist. Ein Clustering-basierter Reverse-Engineering-Ansatz hat damit zumindest hinsichtlich der Kopplung keinen Grund mehr, die beiden Klassen einer gemeinsamen Komponente zuzuordnen und würde die konzeptionelle Architektur, die in Abbildung 3.4 dargestellt ist, ableiten.

Das Beispiel zeigt, dass Schwachstellen in einem Softwaresystem einen negativen Einfluss auf die Metrikwerte des Systems haben und damit auch das Clustering durch Schwachstellen beeinflusst wird. Die konkrete Softwarearchitektur des Systems kann dadurch von der konzeptionellen Architektur abweichen, da lediglich die konkrete Architektur inklusive der im System vorkommenden Schwachstellen

```
interface IA {
  m1();
}

class A implements IA {
  IB ib;
  m1(){
    ib.m3();
  }
}
```

```
interface IB {
  m2();
  m3();
}

class B implements IB {
  m2() {…}
  m3() {…}
}
```

Abbildung 3.3: Korrektur der Interface Violation

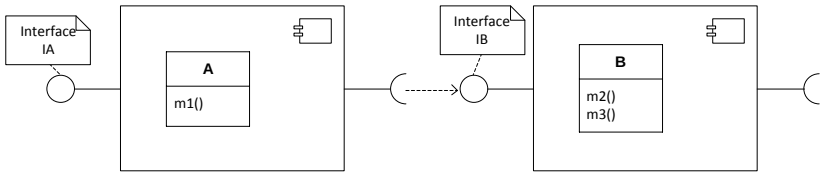

Abbildung 3.4: Konzeptionelle Softwarearchitektur für das Beispiel

rekonstruiert werden kann. Daher ist die grundlegende Hypothese dieser Arbeit, dass die Beseitigung von Schwachstellen in der Implementierung des Systems zu einem höheren Grad an Modularisierung des Systems und zu einer Annäherung von konzeptioneller und konkreter Softwarearchitektur führt. Die Identifikation von derartigen Schwachstellen in der Architektur eines Systems liefert einem Reengineer somit wichtige Informationen darüber, welche Teile des von ihm untersuchten Systems er im Rahmen des Reengineerings überarbeiten sollte und wie er das tun muss.

3.2 Lösungsidee

Die Softwarearchitektur eines Systems bietet zwar einen Überblick über das analysierte System, liefert jedoch keine direkte Aussage darüber, ob Anpassungen nötig sind. Schwachstellen im System, die beispielsweise durch wiederkehrende Erweiterungen entstanden sind, sind ein Indiz dafür, dass eine Abweichung zwischen konzeptioneller und konkreter Softwarearchitektur vorliegt. Damit sind sie auch ein direkter Hinweis darauf, wie und wo im System etwas von einem Reengineer

angepasst werden muss. Zur Identifikation derartiger Schwachstellen kann ein musterbasierter Reverse-Engineering-Ansatz (siehe Abschnitt 2.4) genutzt werden. Dieser kann gezielt nach Mustern in der Implementierung eines Systems suchen, welche für Schwachstellen typisch sind.

Der im Rahmen dieser Arbeit verfolgte Ansatz ist die Kombination von Clustering- und musterbasiertem Reverse Engineering. Dabei soll der Clustering-basierte Ansatz zunächst eine Softwarearchitektur für ein zu untersuchendes Softwaresystem rekonstruieren. Der für den musterbasierten Reverse-Engineering-Ansatz verwendete abstrakte Syntaxbaum (AST) des Systems soll um die so gewonnenen Informationen über die Softwarearchitektur angereichert werden. Anschließend soll die Mustersuche, Schwachstellen in der Softwarearchitektur identifizieren und dadurch das Clustering-basierte Reverse Engineering unterstützen.

Die Mustersuche ist auf Grund ihrer Schwächen (siehe Abs. 2.4.3) beim Reverse Engineering für große Systeme nicht immer möglich oder auch nicht sinnvoll, da sie zu viele Ergebnisse liefern würde. Im Rahmen dieser Arbeit wird daher die Idee verfolgt, die Mustersuche lediglich für einzelne erkannte Komponenten oder gegebenenfalls eine kleine Auswahl an Komponenten anzuwenden. Damit wird zum einen die Laufzeit der Mustersuche gering gehalten, da nur ein Teil des Systems analysiert werden muss, und zum anderen werden Mustervorkommen ausgeblendet, die sich über das gesamte System verteilen oder anderen Komponenten zuzuordnen sind.

Eine detailliertere Erläuterung dazu, wie die beiden Reverse-Engineering-Ansätze kombiniert werden, ist in Kapitel 4 zu finden. Die Details zur Umsetzung werden in Kapitel 5 erläutert.

3.3 Konzeptionelle Herausforderungen

Im Folgenden werden die wichtigsten konzeptionellen Herausforderungen vorgestellt, die im Rahmen dieser Arbeit bearbeitet werden.

Neuer Analyseprozess Eine Herausforderung ist der Entwurf eines neuen Prozesses, der die Reverse-Engineering-Prozesse der bisher einzeln angewandten Clustering- und musterbasierten Ansätze in einem vereint. Dabei soll die Softwarearchitektur als Ergebnis des Clustering-basierten Ansatzes bei der Suche nach Mustern ausgenutzt werden. Insbesondere muss geklärt werden, in welcher Form der Datenaustausch zwischen den beiden Verfahren geschieht.

Im Rahmen des kombinierten Ansatzes soll die Mustersuche Schwachstellen in der Softwarearchitektur eines Systems identifizieren. Dazu werden besondere Muster benötigt. Im Gegensatz zu den üblichen objektorientierten Entwurfs- oder Antimustern [GHJV95, BMMM98] handelt es sich dabei um Muster, welche sowohl Elemente des ASTs als auch Elemente der Softwarearchitektur enthalten. Eine Vererbungsbeziehung zwischen Klassen verschiedener Komponenten sollte zum

Beispiel vermieden werden, da sie eine Implementierungsabhängigkeit zwischen den beteiligten Komponenten erzeugt, obwohl die Komponenten möglicherweise sonst in keiner Beziehung zueinander stehen. Um die Komponentenzugehörigkeit zu berücksichtigen, muss ein Muster sich daher auch auf die Softwarearchitektur beziehen können. Damit ergeben sich zwei weitere Herausforderungen für diese Arbeit.

Vereinigung der Modelle Zum einen müssen AST und Softwarearchitektur in einem gemeinsamen Modell vereint werden, damit Muster basierend auf Informationen aus beiden Modellen spezifiziert werden können und der Mustersuche ein zusammenhängender Wirtsgraph zur Verfügung gestellt werden kann (vergl. Abs. 2.4).

Erarbeitung von Mustern Zum anderen müssen Schwachstellenmuster erarbeitet werden, um einen Katalog von Musterspezifikationen zu schaffen, die nicht nur Schwachstellen in der Implementierung, sondern auch in der Softwarearchitektur des Systems aufdecken können. Die von diesen Mustern dokumentierten Schwachstellen müssen daher einen Einfluss auf die Architektur eines Systems ausüben.

Anpassung der Mustersuche Eine weitere Aufgabe als Teil der Kombination liegt in der Beschränkung der Laufzeit des musterbasierten Ansatzes. Hier gilt es insbesondere ein Konzept zu schaffen, wie die Mustersuche auf einen bestimmten Teil des ASTs als Wirtsgraph beschränkt und damit die Laufzeit gering gehalten werden kann.

Validierung Um den im Rahmen dieser Arbeit verfolgten Ansatz auf seine Tauglichkeit zu überprüfen, werden Praxiserfahrungen benötigt. Daher sollen mit dem kombinierten Reverse-Engineering-Ansatz eines oder mehrere Systeme analysiert werden. Als Beispiel wird im Rahmen dieser Arbeit das komponentenbasierte CoCoME [coc10] verwendet, da dessen Architektur gut dokumentiert ist und zum Vergleich mit den Analyseergebnissen genutzt werden kann.

3.4 Technische Herausforderungen

Für einige Herausforderungen dieser Arbeit gilt es eine technische Lösung zu finden. Die wichtigsten davon werden im Folgenden vorgestellt. Die Herausforderungen resultieren aus einer Umstellung auf neue Metamodelle für die Musterspezifikation und die aus einem Musterkatalog generierten Story-Diagramme [DHL+11]. Zum Startzeitpunkt dieser Arbeit befand sich Reclipse in einer Umstellung auf EMF-basierte Metamodelle [EMF10]. Es existierte bereits eine EMF-basierte Implementierung eines Musterspezifikations-Editors.

Generierung von Story-Diagrammen Bisher war es für die Mustersuche nötig, Java-Code, der die Algorithmen zur Mustersuche implementiert, aus Story-Diagrammen zu generieren und auszuführen. Mit dem neuen Story-Diagramm-Metamodell muss auch die Generierung der Story-Diagramme überarbeitet werden.

Nutzung des Interpreters Im weiteren Verlauf der Umstellung muss die Mustersuche auf den am HPI in Potsdam entwickelten Ansatz zur Interpretation von Story-Diagrammen umgestellt werden [GHS09]. Durch dessen Nutzung wird die Generierung von ausführbarem Code überflüssig, wodurch ein Arbeitsschritt im Gesamtprozess wegfällt. Der Story-Diagramm-Interpreter wird parallel zu dieser Masterarbeit auf das neue Story-Diagramm-Metamodell umgestellt. Dabei ist damit zu rechnen, dass sich dessen Schnittstellen ändern und damit deren Verwendung in Reclipse ebenfalls angepasst werden muss.

Anpassungen an der Reclipse-Inferenz Die Veränderungen am Story-Diagramm-Metamodell betreffen neben der Generierung von Story-Diagrammen aus einem Musterkatalog auch deren Ausführung im Rahmen der Reclipse-Inferenz. Die Inferenz muss an das neue Story-Diagramm-Metamodell angepasst werden. Das betrifft insbesondere die Strategien zur Mustersuche, welche die Reihenfolge bestimmen, in der nach einzelnen Mustern gesucht wird. Daneben ist aber auch die Persistierung und Darstellung von Ergebnissen der Mustersuche betroffen und muss überarbeitet werden.

4 Kombinierter Reverse-Engineering-Ansatz

Schwachstellen erhöhen oft die Kopplung in einem Softwaresystem. Kopplung ist ein wichtiger Indikator für das Clustering eines Systems und ist damit maßgeblich für dessen Analyseergebnis in Form einer Softwarearchitektur. Die Korrektur von Schwachstellen verändert daher die Softwarearchitektur eines Systems. Da Schwachstellen Teil der konkreten Architektur sind, aber nicht zur konzeptionellen Architektur gehören, herrscht Grund zur Annahme, dass ihre Korrektur eine Annäherung von konkreter und konzeptioneller Architektur eines Systems bewirkt.

Der folgende Abschnitt 4.1 liefert einen Überblick darüber, wie der Clustering- und der musterbasierte Ansatz im Rahmen dieser Arbeit zu einem neuen Reverse-Engineering-Prozess kombiniert werden. In Abschnitt 4.2 wird vorgestellt, wie die Modelle von Softwarearchitektur und AST vereinigt werden, um die Softwarearchitektur während der Mustersuche zu berücksichtigen und die Mustersuche auf einzelne Komponenten einschränken zu können. Im darauf folgenden Abschnitt 4.3 werden einige im Rahmen dieser Arbeit erarbeitete Muster zur Erkennung von Schwachstellen in der Softwarearchitektur eines Systems vorgestellt. Abschließend folgt in Abschnitt 4.4 eine Diskussion über die Einschränkungen des kombinierten Reverse-Engineering-Ansatzes.

4.1 Überblick

In diesem Abschnitt wird vorgestellt, wie der Clustering-basierte SoMoX-Ansatz und der musterbasierte Reclipse-Ansatz kombiniert werden, um das Reverse Engineering eines komponentenbasierten Softwaresystems zu verbessern. Dabei wird die von dem Clustering-basierten Ansatz gewonnene Softwarearchitektur von dem musterbasierten Ansatz auf Schwachstellen analysiert. Der Reengineer hat daraufhin die Möglichkeit, die gefundenen, potentiellen Schwachstellen genauer zu betrachten und gegebenenfalls zu korrigieren. Anschließend können die so gewonnenen Komponenten einzeln untersucht werden und zum Beispiel Entwurfsmustervorkommen gesucht werden. Der Fokus dieser Arbeit liegt jedoch auf der Rückgewinnung der Softwarearchitektur und der Aufdeckung von Schwachstellenvorkommen.

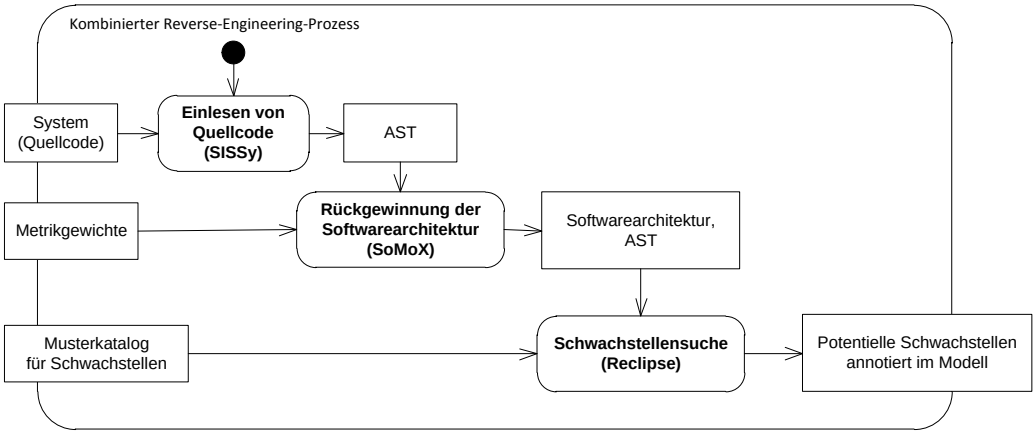

Abbildung 4.1: Ein Überblick über den kombinierten Reverse-Engineering-Prozess

4.1.1 Kombinierter Analyseprozess

Einen Überblick darüber, wie die beiden Ansätze kombiniert werden, liefert das Aktivitätendiagramm in Abbildung 4.1. Der Parser SISSy wird eingesetzt, um den Quellcode eines zu untersuchenden Systems einzulesen. Es entsteht ein generalisierter abstrakter Syntaxbaum (GAST, siehe Abschnitt 2.2.1). Dieser wird von SoMoX verwendet, um die Softwarearchitektur des analysierten Systems zu rekonstruieren. SoMoX benötigt dazu einige Metrikgewichte, welche von einem Reengineer vor der Analyse an das aktuell analysierte System angepasst werden müssen, um charakteristische Eigenschaften des Systems in der Analyse zu berücksichtigen [BHT+10]. Anhand der gemessenen Metrikwerte und der Metrikgewichte wird das Softwaresystem durch das Clustering von SoMoX in Komponenten unterteilt, welche samt ihrer Schnittstellen und Beziehungen zueinander die Softwarearchitektur des analysierten Systems bilden.

Ursprünglich ist für die Mustersuche in Reclipse vorgesehen, dass der gesamte abstrakte Syntaxbaum eines Systems auf Mustervorkommen durchsucht wird [Nie04]. Zur Beschleunigung der Analyse soll dies in der Kombination von Clustering und Mustersuche vermieden werden. Durch das Clustering liegt bereits eine Softwarearchitektur vor. Diese soll verwendet werden, um die Mustersuche auf einzelne Komponenten einzuschränken und damit den musterbasierten Reverse-Engineering-Ansatz auch in großen Systemen anwenden zu können.

Um nach Mustern in einem System zu suchen, die auf Schwachstellen in der Softwarearchitektur hindeuten, müssen die spezifizierten Muster eines Musterkatalogs sich auf die Softwarearchitektur beziehen können. Dazu wird ein Modell benötigt, welches sowohl die Softwarearchitektur, als auch den abstrakten Syntaxgraphen modelliert und in einem gemeinsamen Modell vereint. Eine Kombination der Modelle ist ohne weiteres möglich, da der musterbasierte Ansatz zwar metho-

disch vorsieht, dass ein abstrakter Syntaxbaum auf Mustervorkommen durchsucht wird, jedoch lediglich einen attributierten, typisierten und zusammenhängenden Wirtsgraph erfordert. Wie das Softwarearchitektur- und AST-Modell zur Musterspezifikation und -suche kombiniert werden, wird in Abschnitt 4.2 erläutert. Der Abschnitt 4.3 stellt einige Muster vor, die zur Erkennung von potentiellen Schwachstellen in der Softwarearchitektur eines komponentenbasierten Softwaresystems dienen.

4.1.2 Musterbasierte Schwachstellensuche

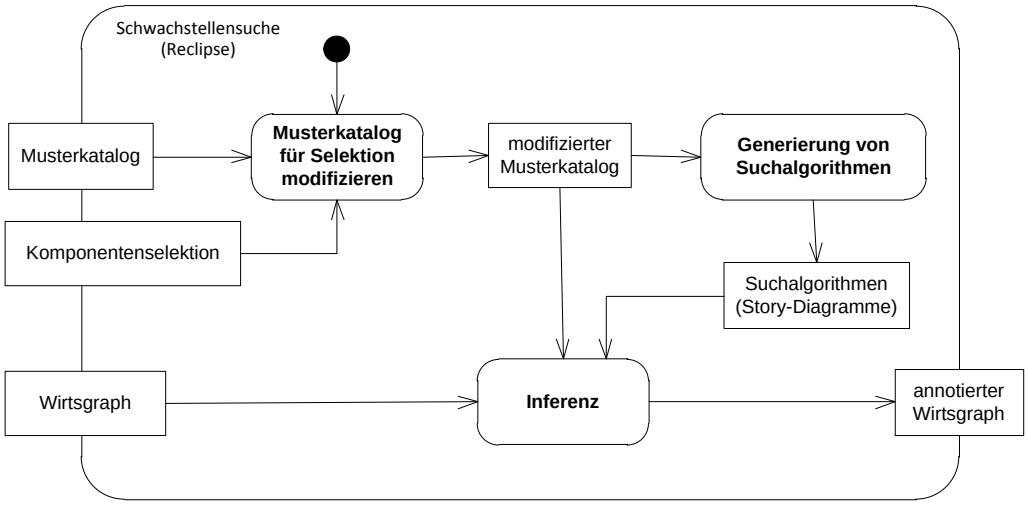

Abbildung 4.2: Die Schwachstellensuche mit Reclipse unter Berücksichtigung der Softwarearchitektur

Die Komponenten, die auf Schwachstellen hin untersucht werden sollen, müssen von einem Reengineer vor der Mustersuche selektiert werden. In Abbildung 4.2 ist ein Aktivitätendiagramm abgebildet, dass die musterbasierte Schwachstellensuche mit Reclipse unter Berücksichtigung der Softwarearchitektur darstellt. Bevor die Schwachstellensuche beginnen kann, wird ein Musterkatalog mit den Musterspezifikationen für potentielle Schwachstellen benötigt. Daneben sind die Selektion von den zu untersuchenden Komponenten (Auswahl aller Komponenten auch möglich) und der Wirtsgraph erforderlich, in dem nach Mustervorkommen gesucht werden soll.

Um zu gewährleisten, dass nur die selektierten Komponenten auf Mustervorkommen durchsucht werden, werden dem Musterkatalog einige Bedingungen hinzugefügt. Die Bedingungen bewirken, dass Teile des Systems während der Mustersuche ausgeblendet werden. Wie die Bedingungen erzeugt werden und wie die

Mustersuche dadurch eingeschränkt wird, wird in Abschnitt 4.2.4 erläutert.

Für die Suche nach Mustervorkommen werden Suchalgorithmen benötigt, welche die Suche im Wirtsgraph durchführen und Musterfunde annotieren. Die Suchalgorithmen werden in Form von Story-Diagrammen [Zün01, DHL+11] aus den modifizierten Musterspezifikationen des Musterkatalogs generiert. Details zu der Generierung von Story-Diagrammen und zu den im Rahmen dieser Arbeit nötigen Anpassungen an diesem Prozessschritt werden in Abschnitt 5.2 erläutert.

Anschließend werden die Suchalgorithmen durch die Inferenz auf den Wirtsgraphen angewandt, wodurch Mustervorkommen und damit potentielle Schwachstellen annotiert werden.

4.2 Integration von AST und Softwarearchitektur

Im Rahmen der musterbasierten Schwachstellensuche wird ein gemeinsames Modell für den AST und die Softwarearchitektur benötigt. Dafür gibt es im Wesentlichen zwei Gründe:

1. a) Musterspezifikationen, die potentielle Schwachstellen in der Softwarearchitektur aufdecken sollen, müssen sich auch auf Teile aus der Softwarearchitektur beziehen und Bedingungen an diese ausdrücken können.

   b) Während einer Mustersuche findet ein Graphmatching in einem zusammenhängenden Wirtsgraphen statt (siehe Abs. 2.4). Um eine Bedingung an die Softwarearchitektur prüfen zu können, muss der Wirtsgraph das Modell der Softwarearchitektur enthalten.

2. Die Mustersuche soll im Rahmen des hier vorgestellten Ansatzes auf einzelne Komponenten oder eine Kombination davon beschränkt werden können, um die Mustersuche auch für große Systeme anwenden zu können. Der AST bildet stets das gesamte zu analysierenden Softwaresystem ab. Um nicht den gesamten AST zu analysieren, muss im Laufe der Mustersuche für einzelne Elemente aus dem AST geprüft werden, welcher Komponente sie angehören. Gehört das AST-Element nicht der gerade untersuchten Komponente an, so soll die Mustersuche diese nicht weiter berücksichtigen.

Im Folgenden wird zunächst erläutert, welche Möglichkeiten existieren, die beiden Modelle für AST und Softwarearchitektur zu vereinen. Dem folgt die Begründung der Entscheidung für das Source-Code-Decorator-Metamodell. In Abschnitt 4.2.3 werden einige Muster vorgestellt, die sowohl die Spezifikation von Schwachstellenmustern erleichtern sollen, als auch die Möglichkeit eröffnen, die Mustersuche auf einzelne Komponenten einzuschränken. Anschließend folgt ein Abschnitt darüber, wie die Mustersuche auf einzelne Komponenten beschränkt wird.

4.2.1 Möglichkeiten zur Mustersuche und -spezifikation unter Berücksichtigung der Softwarearchitektur

Es gibt mehrere Möglichkeiten, die Softwarearchitektur für die Mustersuche bereitzustellen. Eine Möglichkeit ist die Definition von speziellen Annotationen für Softwarearchitektur-Elemente wie Komponenten, die im AST-Modell nicht enthalten sind. Diese speziellen Annotationen könnten bei der Musterspezifikation verwendet werden, wenn beispielsweise der Bezug eines AST-Elements zu einer Komponente von Bedeutung ist. Derartige Annotationen könnten im Rahmen eines Vorverarbeitungsschritts im AST angehängt werden. Diese Möglichkeit stellt jedoch eine eher technische als konzeptionelle Lösung dar und Bedarf vieler Anpassungen an dem Vorverarbeitungsschritt, sollte sich das Modell der Softwarearchitektur in Zukunft ändern.

Geschickter als die Definition von speziellen Annotationen und das Implementieren eines weiteren Vorverarbeitungsschritts, ist die direkte Verwendung des Softwarearchitektur-Modells. Damit ist es möglich, Muster zu spezifizieren, welche nicht nur die Struktur der Implementierung im AST berücksichtigen, sondern auch beliebige Strukturen der Softwarearchitektur. Da damit lediglich die Musterspezifikationen von bestimmten Modellen für AST und Softwarearchitektur abhängen, aber nicht die Implementierung des hier vorgestellten Ansatzes, können in Zukunft auch andere Softwarearchitektur-Modelle verwendet werden. Der musterbasierte Reverse-Engineering-Ansatz von Reclipse könnte damit in späteren Arbeiten auch mit anderen Clustering-basierten Ansätzen kombiniert werden.

4.2.2 Verwendung des Source-Code-Decorator-Metamodells

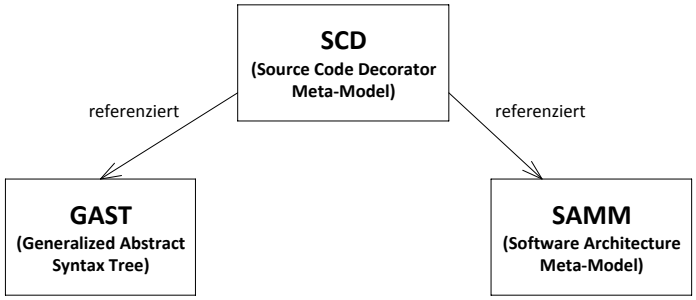

Abbildung 4.3: Zusammenhang der Modelle SAMM, GAST und SCD

Die Mustersuche in Reclipse erfordert einen zusammenhängenden Wirtsgraph. Das Softwarearchitektur-Metamodell SAMM in SoMoX, dessen Instanzen Softwarearchitekturen beschreiben, hängt jedoch nicht direkt mit dem GAST zusammen. Der Zusammenhang der beiden Modelle wird in SoMoX über ein drittes

Modell, das Source-Code-Decorator-Metamodell, hergestellt, welches die Korrespondenz zwischen GAST und SAMM modelliert. Die Abbildung 4.3 visualisiert den Zusammenhang der Modelle, wobei die Pfeilrichtung eine Referenz auf das entsprechende Modell symbolisiert. GAST und SAMM sind mit Hilfe des Korrespondenzmodells ein zusammenhängender Graph und erfüllen damit ein notwendiges Kriterium der Mustersuche an den Wirtsgraph. Gemeinsam mit den SAMM und GAST-Metamodell dient das Source-Code-Decorator-Metamodell als grundlegendes Modell zur Spezifikation und Suche von Mustern in dem hier vorgestellten Ansatz.

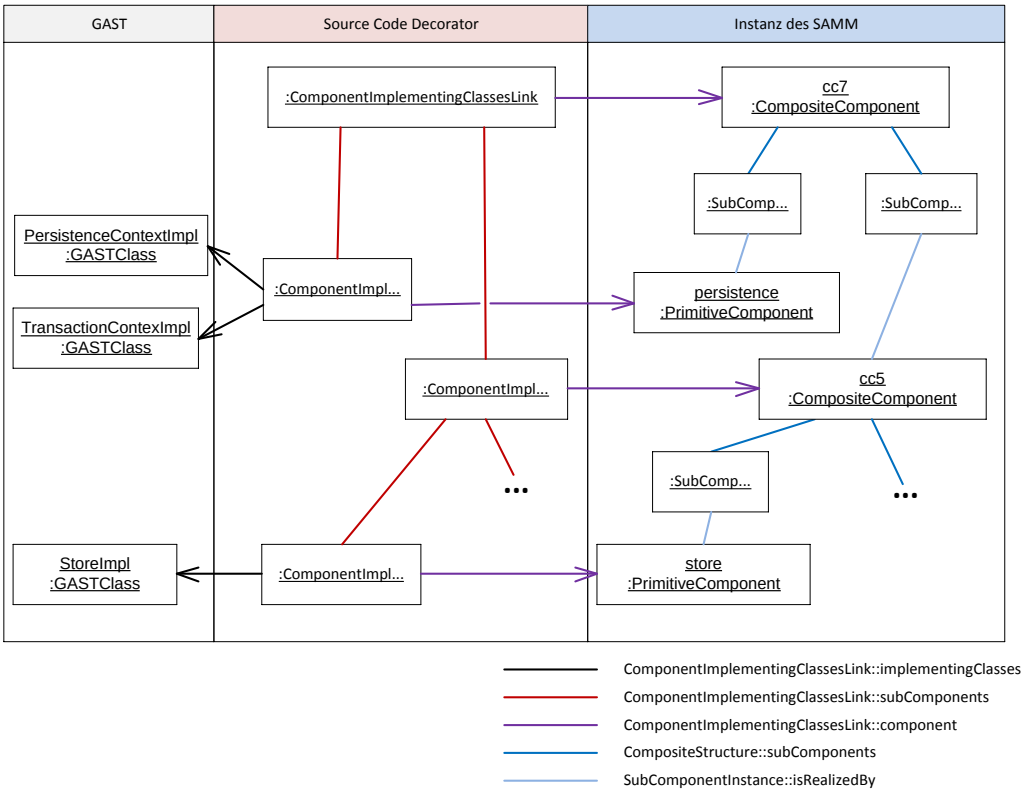

Abbildung 4.4: Korrespondenz zwischen GAST und SAMM-Instanz

Die Abbildung 4.4 stellt einen Ausschnitt eines Objektgraphen als Beispiel für ein Analyseergebnis mit dem Clustering-basierten SoMoX-Ansatz dar. Auf der linken Seite sind Modellelemente aus dem GAST zu sehen (*GASTClass*). Auf der rechten Seite sind die dazu gehörigen Modellelemente aus der SAMM-Instanz abgebildet (*CompositeComponent* und *PrimitiveComponent*), welche die Softwarearchitektur des analysierten Systems repräsentiert. In der Mitte halten die Objekte vom Typ *ComponentImplementingClassesLink* aus dem Source-Code-Decorator-

Metamodell die Korrespondenz zwischen den Komponenten aus der Softwarearchitektur auf der rechten Seite und ihren implementierenden Klassen aus dem GAST auf der linken Seite.

Mit der Kombination der drei Modelle als gemeinsames Metamodell für den Wirtsgraphen können Muster spezifiziert und gesucht werden, die nicht nur bestimmte Strukturen im GAST, sondern auch in der Softwarearchitektur eines Systems aufdecken können. Bei der Spezifikation von Mustern können damit Informationen genutzt werden, die einen höheren Abstraktionsgrad als der abstrakte Syntaxbaum (GAST) haben.

Komponenten als Teil der Softwarearchitektur eines Softwaresystems können hierarchisch sein. Dies kann die Spezifikation von Mustern erschweren. In Abbildung 4.4 besteht die Komponente *cc7* aus zwei weiteren Komponenten, *persistence* und *cc5*. Jede Komponente wird durch mindestens eine Klasse im GAST implementiert, entweder direkt oder über Subkomponenten. Würde man prüfen wollen, ob eine Komponente die Schnittstelle der Komponente *cc7* verletzt, so müsste dies für alle Klassen im GAST in der Abbildung 4.4 geprüft werden, da sie alle als Teil einer Subkomponente zu der Komponente *cc7* gehören und damit auch Teil ihrer Schnittstelle sein können. Da solche Zusammenhänge auf Grund einer Vielfalt von möglichen Ausprägungen nicht direkt als Objektstruktur spezifiziert werden können, werden einige Hilfsmuster benötigt. Diese werden im folgenden Kapitel erläutert.

4.2.3 Korrespondenzmuster zwischen SAMM und GAST

Die Kombination der drei Modelle Service Architecture, GAST und Source Code Decorator bringt zwei Probleme mit sich, welche die Musterspezifikation und Mustersuche betreffen.

1. Hierarchische Komponenten sind Komponenten, die selbst (hierarchische) Komponenten enthalten können. Jede Komponente wird durch eine oder mehrere Klassen implementiert. Hierarchische Komponenten werden jedoch oft auf mehreren Hierarchieebenen durch Subkomponenten implementiert. Um in einer Musterspezifikation die Korrespondenz einer Klasse zu einer Komponente zu spezifizieren, reicht es daher nicht immer aus, lediglich das Korrespondenzelement aus dem Source-Code-Decorator-Metamodell zu verwenden. Gegebenenfalls kann ein komplexer Objektgraph von Nöten sein, der die Hierarchie einer Komponente berücksichtigt. Die Spezifikation von Mustern, wie die einer Interface Violation, wird dadurch erschwert.

2. Die Korrespondenz zwischen SAMM und GAST wird nur unidirektional festgehalten. Lediglich den Objekten vom Typ *ComponentImplementingClasses-Link* aus dem Source-Code-Decorator-Metamodell sind sowohl die Komponenten als auch deren Klassen bekannt. Damit ist es beispielsweise nicht möglich, zu bestimmen, welcher Komponente eine Klasse angehört, ohne

dabei über alle *ComponentImplementingClassesLink*-Objekte zu iterieren, da eine Komponente und Klasse ihr Korrespondenzelement nicht referenziert. Die Laufzeit einer Mustersuche wird davon negativ beeinflusst, da die Korrespondenz einer Klasse zu einer Komponente für die in Abschnitt 4.3 vorgestellten Schwachstellenmuster von zentraler Bedeutung ist.

Um diesen Problemen entgegen zu wirken, wurden im Rahmen dieser Arbeit einige Hilfsmuster spezifiziert, welche die Spezifikation von Mustern zur Erkennung von Schwachstellen und anderen Softwarearchitektur-bezogenen Mustern erleichtern sollen. Die in diesem Abschnitt beschriebenen Musterspezifikationen bilden die Grundlage für weitere Musterspezifikationen. Das bedeutet, dass andere Musterspezifikationen von den hier vorgestellten Musterspezifikationen abhängen. Da während der Mustersuche die Abhängigkeiten zwischen den Musterspezifikationen berücksichtigt werden, werden zuerst Mustervorkommen zu den im Folgenden vorgestellten Musterspezifikationen gesucht, bevor nach anderen Mustervorkommen gesucht wird. Die Suche nach anderen Mustervorkommen kann sich dabei auf bereits identifizierte Korrespondenzen zwischen Komponenten und Klassen stützen, wodurch die Laufzeit der Mustersuche nicht unnötig durch wiederholte Matchings gleicher oder ähnlicher Muster erhöht wird.

4.2.3.1 Idee der Korrespondenzmuster

Eine wesentliche Idee der im Folgenden vorgestellten Muster ist es, weiteren Musterspezifikationen eine einheitliche Möglichkeit bereitzustellen, wie die Korrespondenz zwischen Komponenten und ihren implementierenden Klassen als Teil einer Musterspezifikation formalisiert werden kann. An dem Beispiel-Objektgraph zu einem Analyseergebnis mit SoMoX in Abbildung 4.4 ist zu erkennen, dass einige Komponenten direkt korrespondierende Klassen besitzen, durch die sie implementiert werden. Andere Komponenten werden durch Subkomponenten implementiert und haben daher keine direkt korrespondierenden Klassen.

Die Abbildung 4.5 zeigt am Beispiel aus der Abbildung 4.4, wie die Korrespondenz für Komponenten, die durch Subkomponenten implementiert werden, nach einer Mustersuche mit den in diesem Abschnitt vorgestellten Korrespondenzmustern hergestellt werden soll. Die Komponente cc7 auf der rechten Seite wird in der Abbildung in der Rolle component referenziert. Auf der linken Seite werden alle korrespondierenden Klassen, die direkt oder indirekt in der Komponente cc7 enthalten sind, in der Rolle classes referenziert. Die Annotation Component hält dabei die Referenzen auf alle beteiligten Objekte.

In weiteren Musterspezifikationen kann so die Annotation Component samt der Rollen classes und component wiederverwendet werden, um die Korrespondenz von Komponenten zu ihren Klassen zu spezifizieren. Um dies zu ermöglichen, wird zunächst eine Musterspezifikation benötigt, welche die Rollen classes und component definiert. Die dafür benötigte Musterspezifikation *Component* ist in

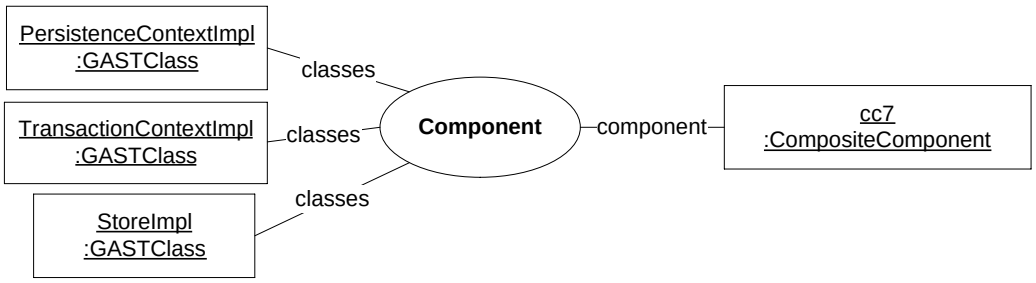

Abbildung 4.5: Korrespondenz der Klassen zur Komponente cc7 aus dem Beispiel in Abbildung 4.4

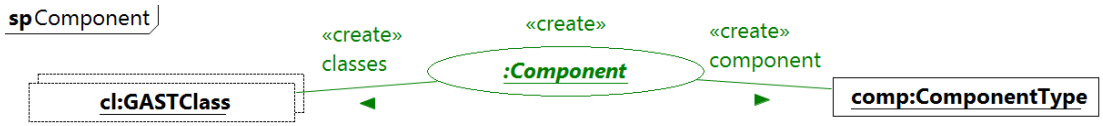

Abbildung 4.6: Das abstrakte Muster repräsentiert die Korrespondenz zwischen einer Komponente und einer Menge von Klassen

Abbildung 4.6 dargestellt. Die Musterspezifikation ist abstrakt[1] und definiert keinerlei Beziehungen zwischen den Klassen aus dem GAST und einer Komponente aus dem SAMM. Da dabei die Fälle *direkte* und *indirekte* Korrespondenz unterschieden werden müssen, werden die Korrespondenzbeziehungen im Folgenden in zwei Spezialisierungen dieser Musterspezifikation spezifiziert.

4.2.3.2 Direkte Korrespondenz von Komponenten und Klassen

Um eine direkte Korrespondenz von einer Komponente zu einer Menge von Klassen zu spezifizieren, reicht zunächst ein simples Muster aus. In Abbildung 4.7 ist die entsprechende Musterspezifikation abgebildet. Das Muster wird durch einen Objektgraph mit drei Rollen formalisiert. Ein Objekt vom Typ ComponentImplementingClassesLink in der Rolle link referenziert ein Objekt vom Typ ComponentType in der Rolle component und eine Objektmenge vom Typ GASTClass in der Rolle classes. Der Typ ComponentType ist die Oberklasse von PrimitiveComponent und CompositeComponent. Daher gilt dieses Muster für beide Typen von Komponenten. Die Rolle classes muss als Objektmenge spezifiziert werden, da eine Komponente durch mehrere Klassen implementiert werden kann, wie das Beispiel in Abbildung 4.4 zeigt.

[1] Abstrakte Muster definieren eine Art Schnittstelle für andere Muster.

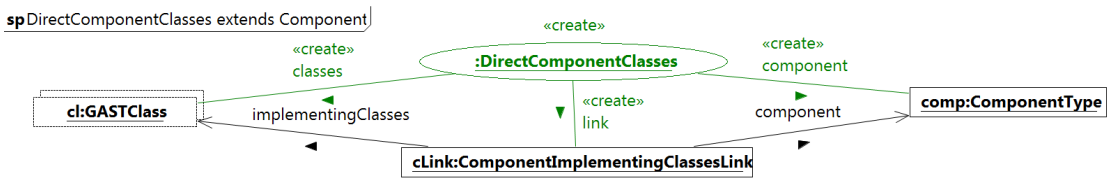

Abbildung 4.7: Musterspezifikation der direkten Korrespondenz zwischen Komponente und Klassen

Eine Suche nach dem *DirectComponentClasses*-Muster würde in dem Beispiel-Objektgraphen aus Abbildung 4.4 zwei Mustervorkommen entdecken. Die Abbildung 4.8 zeigt den Objektgraphen aus dem Beispiel inklusive der durch die Mustersuche erzeugten Annotationen für die *DirectComponentClasses*-Mustervorkommen. Aus Gründen der Übersichtlichkeit wurde der Objektgraph ausgegraut und es wurden einige der Kanten entfernt. Außerdem wurden die Annotation und ihre Beziehungen zu den Objekten in der Abbildung grün eingefärbt.

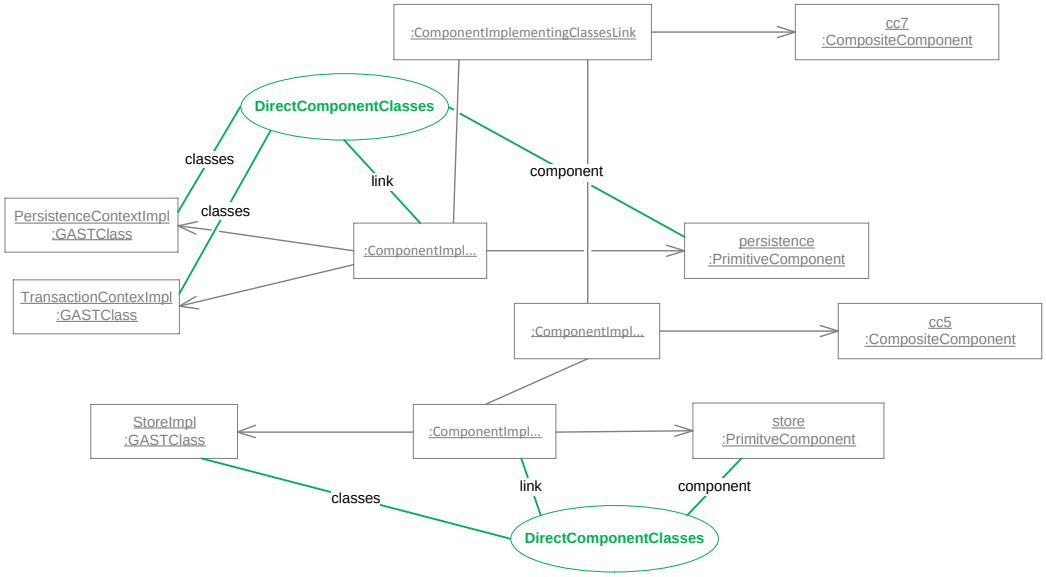

Abbildung 4.8: DirectComponentClasses-Mustervorkommen im Objektgraphen aus dem Beispiel in Abbildung 4.4

Annotiert werden lediglich die Komponenten, die durch PrimitiveComponent-Objekte im SAMM-Modell vertreten sind. Dies ist damit begründet, dass die CompositeComponent-Objekte in dem Beispiel im GAST keine Klassen direkt enthalten und damit das *DirectComponentClasses*-Muster nicht gefunden werden kann. Andernfalls wären auch diese Objekte mit einer entsprechenden Annotation versehen

worden.

An dem Beispiel wird deutlich, dass ein Muster für die direkte Korrespondenz zwischen Komponenten und Klassen nicht ausreicht, da Komponenten (CompositeComponent-Objekte), die durch Subkomponenten implementiert werden, nicht annotiert werden. Bei derartigen Komponenten müssen die Kompositionsbeziehungen zu ihren Subkomponenten berücksichtigt werden, um die Korrespondenz zu ihren implementierenden Klassen zu identifizieren. Dafür werden weitere Hilfsmuster benötigt, die in den folgenden Abschnitten vorgestellt werden.

4.2.3.3 Erkennung von Kompositionsbeziehungen

Um für eine Komponente zu bestimmen, durch welche Klassen sie implementiert wird, muss zunächst bestimmt werden, aus welchen Subkomponenten sie zusammengesetzt ist. Sind alle Subkomponenten einer Komponente identifiziert, so lassen sich anschließend auch deren implementierende Klassen verhältnismäßig einfach bestimmen.

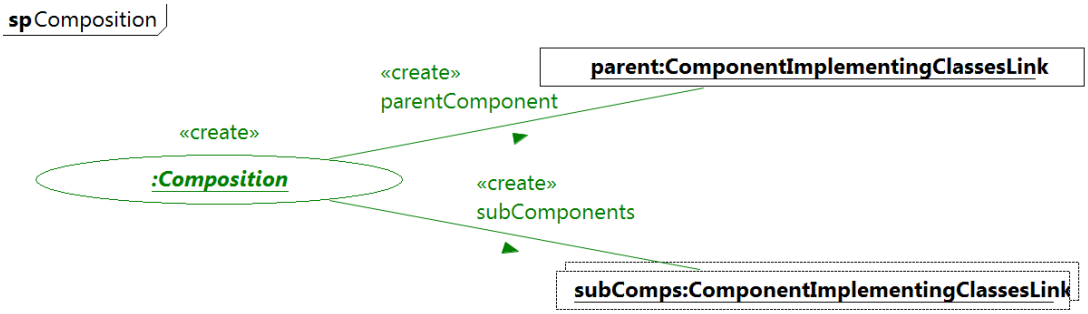

Abbildung 4.9: Abstraktes Muster zur Repräsentation von Kompositionsbeziehungen zwischen Komponenten

Im Gegensatz zu den Kompositionsbeziehungen im Service-Architecture-Metamodell sind die Kompositionsbeziehungen im Source-Code-Decorator-Metamodell über eine subComponents-Assoziation von und zu ComponentImplementingClassesLink modelliert. Um die folgenden Musterspezifikationen möglichst einfach zu halten, beziehen sie sich auf ComponentImplementingClassesLink-Objekte stellvertretend für die Komponente, auf die das ComponentImplementingClassesLink-Objekt verweist.

Eine Subkomponente kann selbst Subkomponenten enthalten. Diese Rekursion kann beliebig tief sein. Da für die oberste Komponente der Rekursion bestimmt werden soll, aus welchen Subkomponenten sie besteht, muss die Kompositionsbeziehung einer Komponente zu ihren Subkomponenten als transitiv betrachtet werden.

Um eine derartige Beziehung mit Hilfe von Musterspezifikationen zu erkennen, werden drei Musterspezifikationen benötigt. Diese sind in ihrer Struktur ähnlich

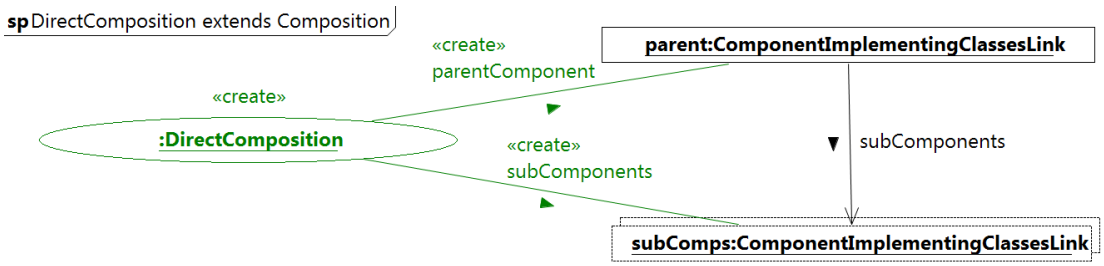

Abbildung 4.10: Musterspezifikation einer direkten Kompositionsbeziehung zwischen mehreren Komponenten

zu den von D. Travkin [Tra06] spezifizierten Mustern zur Erkennung von Vererbungsbeziehungen zwischen Klassen. Dabei wird eine Musterspezifikation *Composition* wie in Abbildung 4.9 benötigt. Diese ist abstrakt und definiert die Rollen parentComponent für die Elternkomponente und die Rolle subComponents für Subkomponenten. Die Musterspezifikation *Composition* wird von zwei weiteren Musterspezifikationen erweitert.

Eine Erweiterung von *Composition* ist die Musterspezifikation *DirectComposition* in Abbildung 4.10. Sie spezifiziert die direkte Kompositionsbeziehung zwischen zwei Komponenten über deren subComponents-Assoziation zu einander.

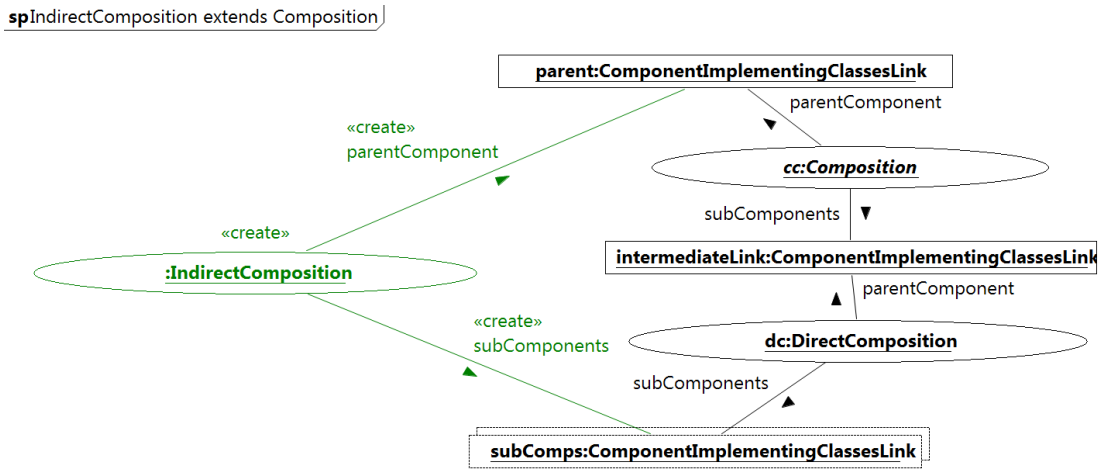

Abbildung 4.11: Komponenten in Kompositionsbeziehung mit mindestens einer Komponete zwischen ihnen

Um beliebig tiefe Rekursionen der Kompositionsbeziehungen durch die Muster abzudecken, wird eine weitere Musterspezifikation *IndirectComposition* benötigt, die in Abbildung 4.11 abgebildet ist. Die Musterspezifikation enthält eine Annotation vom Typ DirectComposition und eine Annotation vom Typ Composition, die beide eine bereits erkannte Kompositionsbeziehung repräsentieren. Da die Mus-

terspezifikation *IndirectComposition* ebenfalls *Composition* erweitert, steht die Composition-Annotation stellvertretend für eine direkte oder indirekte Kompositionsbeziehung, analog zur Vererbung und Polymorphie in objektorientierten Programmiersprachen. Damit werden Komponenten mit einer Annotation vom Typ IndirectComposition annotiert, wenn sie transitiv über mindestens eine weitere Komponente zueinander in Kompositionsbeziehung stehen.

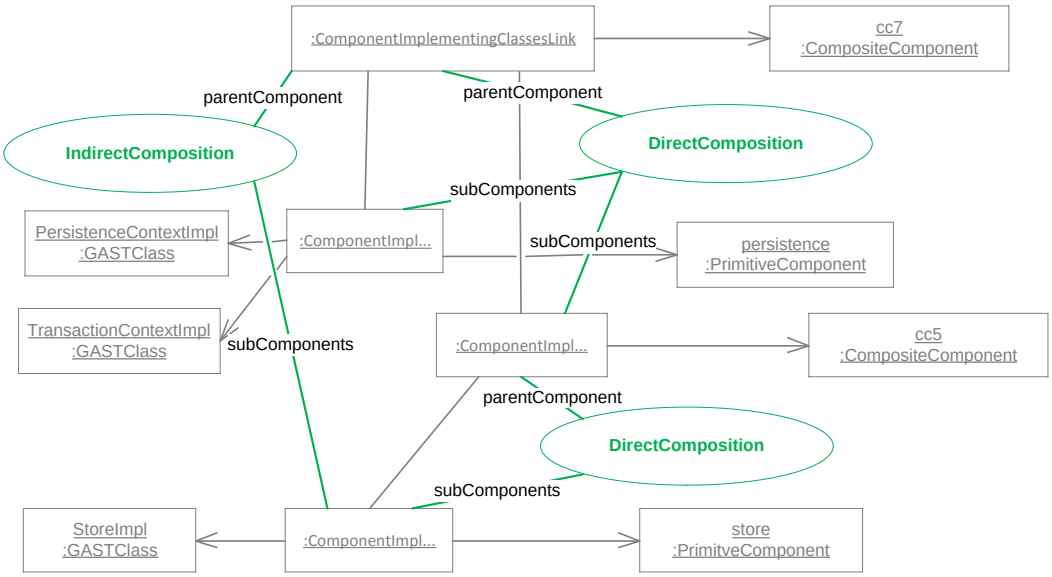

Abbildung 4.12: Annotationen für Kompositionsbeziehungen der Komponenten aus dem Beispiel in Abbildung 4.4

Abbildung 4.12 zeigt die Musterfunde, die durch die Musterspezifikation *DirectComposition* und *IndirectComposition* während einer Mustersuche in dem Beispiel aus Abbildung 4.4 annotiert werden. Das ComponentImplementingClassesLink-Objekt, welches die Komponente cc7 referenziert, wurde zweimal annotiert. Einmal durch die DirectComposition-Annotation, welche die Kompositionsbeziehung zu den beiden direkten Subkomponenten persistence und cc5 repräsentiert und einmal durch eine IndirectComposition-Annotation, welche die indirekte Kompositionsbeziehung zu der store-Komponente repräsentiert.

Durch die zwei Musterspezifikationen *DirectComposition* und *IndirectComposition* werden während einer Mustersuche alle Subkomponenten einer Komponente, direkt oder indirekt, annotiert. Da nun über die *Composition*-erweiternden Musterspezifikationen alle Subkomponenten bekannt sind, müssen lediglich deren korrespondierende Klassen eingesammelt werden. Die dafür benötigte Musterspezifikation wird im folgenden Abschnitt vorgestellt.

4.2.3.4 Indirekte Korrespondenz von Komponenten und Klassen

Um die Klassen im GAST zu identifizieren, welche mit einer bestimmten Komponente über ihre Subkomponenten indirekt korrespondieren, wird eine weitere Musterspezifikation *IndirectComponentClasses* benötigt. Im vorhergehenden Abschnitt wurden die Musterspezifikationen erläutert, die nötig sind, um die Kompositionsbeziehungen zwischen einzelnen Komponenten zu identifizieren. Diese Musterspezifikationen werden in dem folgenden Muster wiederverwendet, um alle implementierenden Klassen der Subkomponenten einer bestimmten Komponente zusammenzufassen und zu annotieren.

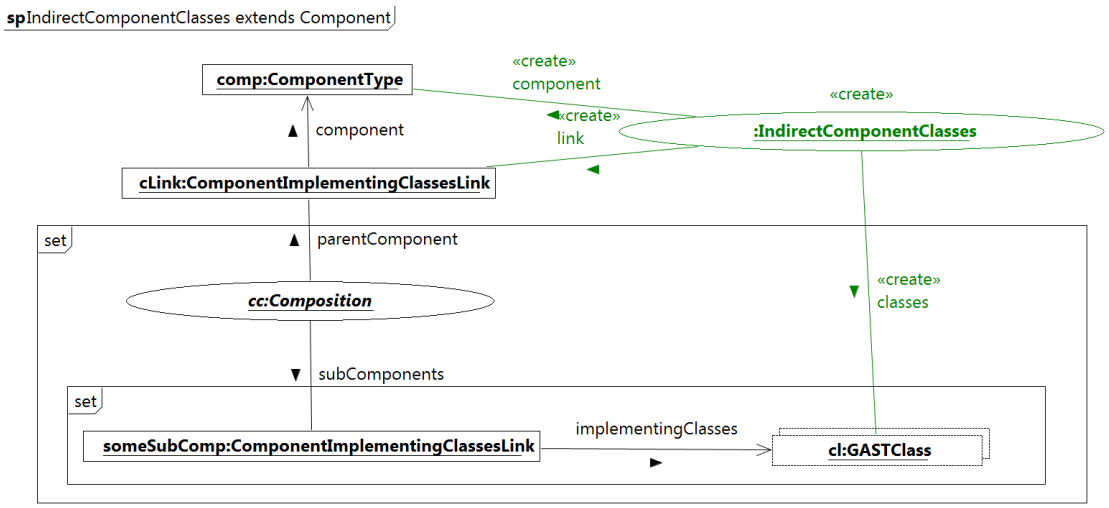

Abbildung 4.13: Musterspezifikation für indirekt korrespondierende Komponenten und Klassen

Die in Abbildung 4.13 dargestellte Musterspezifikation *IndirectComponentClasses* erweitert die Musterspezifikation *Component* (vergl. Abb. 4.6, S. 46). Damit beschreibt auch dieses Muster die Korrespondenz zwischen einer Komponente in der Rolle component aus der SAMM-Instanz und einer Menge von Klassen in der Rolle classes aus dem GAST. Die Korrespondenz einer Subkomponente someSubComp zu ihren implementierenden Klassen, repräsentiert durch die Objektmenge cl, wird durch die Referenz implementingClasses modelliert. Da eine Komponente mehrere Subkomponenten enthalten kann, wird dieser Teil der Musterspezifikation als Menge modelliert, erkennbar an dem set oben links im Kasten.

Subkomponenten müssen in Kompositionsbeziehung zu ihrer Elternkomponente stehen. Dies wird durch die Annotation cc vom Typ Composition ausgedrückt, welche die Rollen parentComponent und subComponents bindet. Durch die Verwendung des Annotationstypen Composition werden die beiden Fälle direkter und indirekter Komposition zusammengefasst. Da durch die Muster *DirectComposition* und *IndirectComposition* mehrere Annotationen an eine Komponente angehängt

werden können, muss auch dieser Teil der Musterspezifikation durch eine Menge modelliert werden, welche die Menge der Subkomponenten und ihrer implementierenden Klassen enthält.

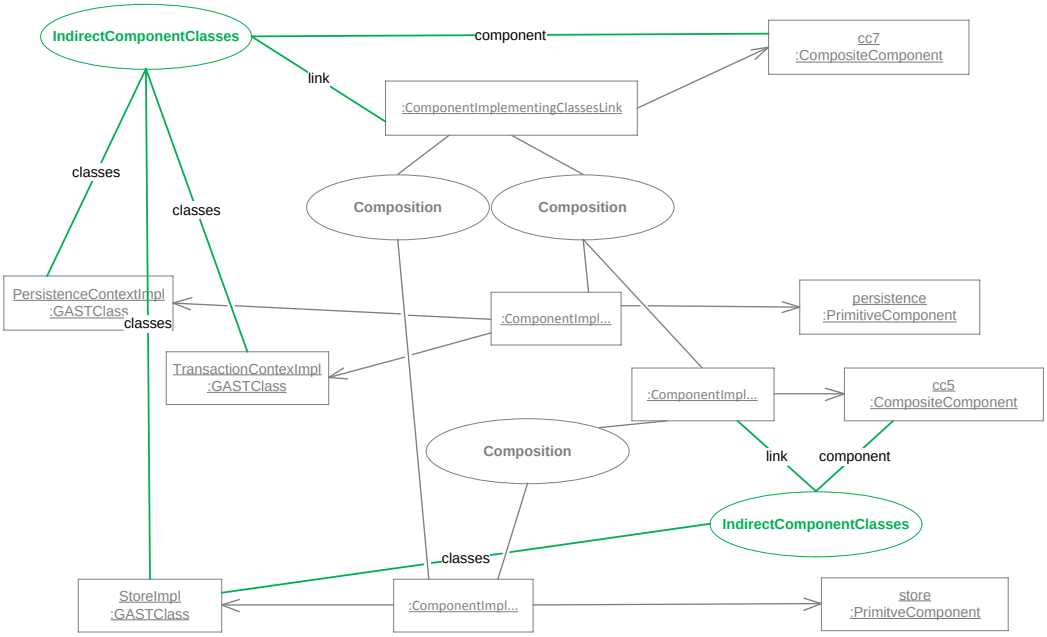

Abbildung 4.14: Annotierte Komponenten und Klassen, welche diese implementieren, aus dem Beispiel in Abbildung 4.4

In Abbildung 4.14 wurde ein Ausschnitt des annotierten Wirtsgraphen aus dem Beispiel in Abbildung 4.4 nach einer Mustersuche mit den hier vorgestellten Musterspezifikationen dargestellt. Dabei wurden die Annotationen für die Kompositionsbeziehungen, die bereits in Abbildung 4.12 dargestellt wurden, durch den allgemeineren Annotationstyp Composition ersetzt, um das Matching des Musters *IndirectComponentClasses* verständlicher zu gestalten. Es wurden zwei neue Annotationen von Typ IndirectComponentClasses hinzugefügt. Zum einen wurde die Komponente cc5 annotiert, weil sie über ein ComponentImplementingClassesLink-Objekt mit einer Composition-Annotation annotiert wurde, welche die Komponente store referenziert und diese durch die Klasse StoreImpl implementiert wird. Zum anderen wurde die Komponente cc7 annotiert. Diese wurde zweimal mit einer Composition-Annotation versehen, die durch das Matching von einer direkten und einer indirekten Kompositionsbeziehung erzeugt wurden. Jede Composition-Annotation referenziert eine Menge von Komponenten, welche wiederum durch eine Menge von Klassen implementiert werden und in der Rolle classes in der Annotation IndirectComponentClasses annotiert wurden.

Nun sind alle Klassen aus dem GAST, welche die Komponente cc7 und cc5

implementieren, in der Rolle classes von einer Annotation vom Typ IndirectComponentClasses annotiert und können beim Matching von anderen Mustern wiederverwendet werden, wenn die Korrespondenz von Komponenten zu Klassen von Bedeutung ist. Da *DirectComponentClasses* und *IndirectComponentClasses* von dem Muster *Component* erben, kann in anderen Musterspezifikationen die Component-Annotation genutzt werden, um sowohl direkte als auch indirekte Korrespondenz zu spezifizieren, so wie in Abbildung 4.5 auf Seite 47 dargestellt.

4.2.4 Inferenz-Beschränkung auf einzelne Komponenten

Um die Mustersuche auch auf große Softwaresysteme mit mehreren hundert tausend Zeilen Code anwenden zu können, muss sie beschränkt werden. Eine vollständige Suche nach Mustervorkommen in derart großen Systemen ist wegen dem immensen Ressourcenbedarf der Inferenz nicht praktikabel und würde außerdem zu lange dauern. Im Rahmen des hier vorgestellten Ansatzes kann die Inferenz von einem Reengineer auf eine oder mehrere Komponenten eingeschränkt werden, indem die zu analysierenden Komponenten vor dem Beginn der Inferenz selektiert werden.

In einem Musterkatalog existieren Abhängigkeiten zwischen den darin enthaltenen Musterspezifikationen, sodass während der Inferenz einige Muster erst gefunden werden können, wenn bestimmte andere Muster bereits gefunden wurden. Diese Abhängigkeitsbeziehungen zwischen den Musterspezifikationen sollen in diesem Ansatz in Verbindung mit dem aus der Selektion von Komponenten abgeleiteten Bedingungen dazu genutzt werden, die Inferenz auf die selektierten Komponenten einzuschränken.

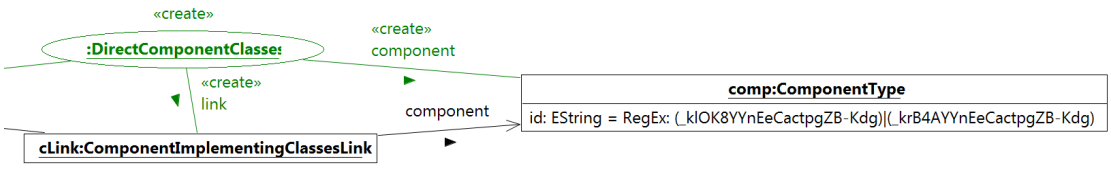

Abbildung 4.15: Generierte Attributbedingung in der Musterspezifikation *DirectComponentClasses*

Die Bedingungen werden dabei aus dem id-Attribut der selektierten Komponenten abgeleitet und den Musterspezifikationen *DirectComponentClasses* und *IndirectComponentClasses* aus dem vorhergehenden Abschnitt 4.2.3 als Attributbedingung an das comp-Objekt vom Typ ComponentType hinzugefügt. Die Attributbedingung wird als regulärer Ausdruck realisiert, wobei die Werte des id-Attributs der selektierten Komponenten mit dem ODER-Operator | in einem Ausdruck verknüpft werden. Ein Beispiel dazu mit zwei selektierten Komponenten und daher mit einem regulären Ausdruck, der zwei Werte erlaubt, ist in Abbildung 4.15 dargestellt.

Mit der Attributbedingung für das id-Attribut der Komponenten können die Muster *DirectComponentClasses* und *IndirectComponentClasses* während der Inferenz nur noch im Zusammenhang mit den selektierten Komponenten erkannt werden, da diese als einzige den regulären Ausdruck für das id-Attribut erfüllen können. Andere Musterspezifikationen eines Musterkatalogs, die von einer der beiden Musterspezifikationen abhängen, können damit ebenfalls nur noch in Zusammenhang mit einer der selektierten Komponenten gefunden werden. Das selbe gilt auch für Musterspezifikationen, die von *Component* abhängen, da es durch *DirectComponentClasses* und *IndirectComponentClasses* erweitert wird.

Eine Notwendigkeit bei dieser Vorgehensweise ist, dass alle Musterspezifikationen eines Musterkatalogs direkt oder indirekt von der *Component*- oder einer *Component*-erweiternden Musterspezifikation abhängen. Ist eine Musterspezifikation unabhängig von anderen, so greifen die generierten Attributbedingungen nicht. Das hat zur Folge, dass ein derartiges Muster im gesamten Wirtsgraph gesucht wird und nicht nur in dem Teil, der einer der selektierten Komponenten zugeordnet werden kann.

Eine Abhängigkeit zu *Component* wird beispielsweise dadurch erzielt, dass eine Component-Annotation in einer Musterspezifikation verwendet wird. Da der Typ GASTClass in den meisten Musterspezifikationen vorkommt, reicht es aus, eine Component-Annotation an die GASTClass über die Rolle classes zu hängen, was lediglich die Korrespondenz der Klasse zu einer Komponente widerspiegelt.

Die im folgenden Abschnitt vorgestellten Musterspezifikationen zur Erkennung von Schwachstellen in einer Softwarearchitektur hängen alle von dem *Component*-Muster ab. Sie sind daher auch ein Beispiel für die hier vorgeschlagene Vorgehensweise bei der Spezifikation von Mustern, um die Inferenz auf einzelne Komponenten zu beschränken.

4.3 Muster zur Erkennung von Schwachstellen

In den vorhergehenden Abschnitten 4.1 und 4.2 wurde vorgestellt, wie die Clustering- und musterbasierten Reverse-Engineering-Verfahren im Rahmen des hier vorgestellten Ansatzes kombiniert werden. Ein Ergebnis dieser Kombination ist, dass der Wirtsgraph für die Mustersuche eine Instanz der vereinten Metamodelle von GAST, Service Architecture und Source Code Decorator ist und damit sowohl die Implementierung, als auch die Architektur eines Softwaresystems in einem Modell vereint. Dies eröffnet die Möglichkeit, nach Mustern zu suchen, die sich auf die Architektur eines Systems beziehen und allein mit dem GAST eines Systems bisher nicht möglich oder nur umständlich zu spezifizieren waren. Im Rahmen dieser Arbeit wurden einige Musterspezifikationen erarbeitet, die sich zur Beschreibung von Schwachstellen neben dem GAST auch auf die Softwarearchitektur beziehen. Diese werden in diesem Abschnitt vorgestellt.

Ein grundlegendes Ziel dieser Arbeit ist es, die Rekonstruktion der Softwarearchitektur eines Systems mit dem Clustering-basierten Ansatz durch den mu-

sterbasierten Ansatz zu unterstützen. Dabei wird die Annahme verfolgt, dass die Beseitigung von Schwachstellen in einem System die konkrete Softwarearchitektur des Systems an dessen konzeptionelle Softwarearchitektur annähert. Deshalb wurden ausschließlich Muster zur Erkennung von potentiellen Schwachstellen eines Systems erarbeitet. Solche Schwachstellen können anschließend von einem Reengineer begutachtet und gegebenenfalls beseitigt werden.

Zwei der wichtigsten Metriken im Rahmen einer Clustering-basierten Analyse sind *Coupling* und *InterfaceViolation* (siehe Abschnitt 2.3). Sie bilden die Grundlage der meisten anderen Strategien und Metriken und haben daher einen großen Einfluss auf das Clustering und damit auch auf die resultierende Softwarearchitektur eines Systems. Um Schwachstellen eines Systems zu erkennen, die sich negativ auf dessen Softwarearchitektur auswirken, liegt der Fokus der im Folgenden vorgestellten Muster daher auf Strukturen, welche die Metrikwerte für *Coupling* und *InterfaceViolation* stark beeinflussen.

4.3.1 Interface Violation

Eine Interface Violation liegt dann vor, wenn vorhandene Schnittstellen umgangen werden. Ein Beispiel für eine Interface Violation wurde in Abschnitt 3.1 vorgestellt. Oft werden während Wartungsarbeiten an Softwaresystemen bestehenden Klassen neue Methoden oder Attribute hinzugefügt. Dabei kann es vorkommen, dass aus den Änderungen resultierende nötige Anpassungen an Schnittstellen der überarbeiteten Klassen vergessen oder übersehen werden. Die neuen Methoden und Attribute werden anschließend von anderen Klassen aus verwendet und da die Pflege der Schnittstelle vernachlässigt wurde, werden oft nach und nach direkte Referenzen auf die implementierenden Klassen eingefügt, damit die neue Funktionalität genutzt werden kann.

Interface Violations werden während des Clusterings durch die Metrik *InterfaceViolation* gemessen. Da bei dem Vorkommen einer Interface Violation eine Klasse direkt referenziert wird, anstatt ihr Interface zu verwenden, wird durch eine Interface Violation die Kopplung und daher auch die Werte der Metrik *Coupling* zwischen den beteiligten Klassen erhöht. Daher neigt das Clustering dazu, die beteiligten Klassen bzw. Komponenten, zwischen denen Interface Violations existieren, einer Komponente zuzuordnen, obwohl diese möglicherweise konzeptionell unterschiedlichen Komponenten angehören sollten.

Im Folgenden werden zwei Varianten der Interface Violation unterschieden. Zum einen kann eine Schnittstelle verletzt werden, indem Methoden einer Klasse, die nicht Teil ihrer Schnittstelle sind, von anderen Klassen verwendet werden. Zum anderen kann eine Schnittstelle verletzt werden, indem eine Klasse auf eine Variable einer anderen Klasse zugreift.

4.3.1.1 Illegaler Aufruf einer Methode

Die erste Variante der Interface Violation ist der illegale Aufruf einer Methode. Die grundlegende Annahme bei diesem Schwachstellenmuster ist, dass eine Methode durch eine Schnittstelle deklariert werden muss, wie es in einem komponentenbasierten Softwaresystem bei Kommunikation zwischen verschiedenen Komponenten der Fall sein sollte.

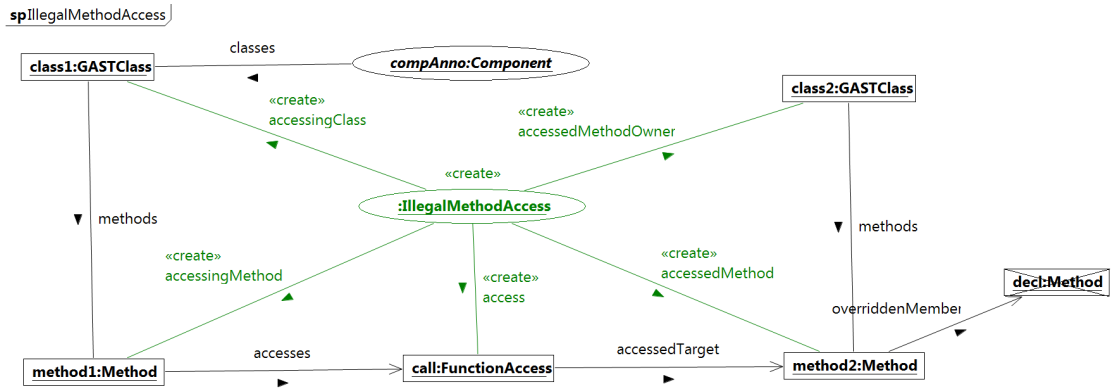

Abbildung 4.16: Musterspezifikation einer Interface Violation durch Methodenzugriff

Das Schwachstellenmuster wird in Abbildung 4.16 dargestellt. Es spezifiziert eine Klasse class1 in der Rolle accessingClass, die einer der von einem Reengineer selektierten Komponenten angehört, repräsentiert durch die Annotation compAnno vom Typ Component. Die GASTClass class1 enthält eine Methode method1 in der Rolle accessingMethod. Die Methode method1 ruft eine andere Methode method2 in der Rolle accessedMethod auf. Der Zugriff auf die Methode method2 ist durch ein Objekt vom Typ FunctionAccess modelliert, welches die Rolle access hat. Die Klasse class2, welche die Methode method2 implementiert, spielt die Rolle accessedMethodOwner in der Musterspezifikation. Für die Musterspezifikation ist es wichtig, dass die Methode method2 keiner Schnittstelle angehört, da es sich sonst um keine Schnittstellenverletzung handelt. Dies wird durch die Assoziation overriddenMember ausgedrückt, die von method2 auf ein negatives Objekt decl vom Typ Method verweist. Für ein erfolgreiches Matching des Musters darf damit die Methode method2 keine Methoden aus Oberklassen überschreiben und keine Methoden von Interfaces implementieren.

Mögliche Maßnahmen zur Korrektur Bei Mustervorkommen von *IllegalMethodCall* existieren zwei Varianten, die danach unterschieden werden können, ob die Klasse class2 ebenfalls zu der durch die Annotation compAnno repräsentierten Komponente angehört, oder ob es sich dabei um eine Klasse aus einer anderen Komponente handelt. Diese Fallunterscheidung wird in der Musterspezifikation

von *IllegalMethodCall* vernachlässigt, um hier keine zwei nahezu identische Musterspezifikationen abbilden zu müssen.

Im ersten Fall handelt es sich um eine Interface Violation innerhalb der selben Komponente. Diese ist somit nicht zwingend eine Schwachstelle, da innerhalb von Komponenten auch Kommunikation ohne die Nutzung von Schnittstellen erlaubt ist. Das Clustering kann in diesem Fall aber auch zwei konzeptionell verschiedene Komponenten zu einer Komponente in der Softwarearchitektur vereint haben. Dies geschieht beispielsweise, wenn mehrere Interface Violations zwischen den beteiligten Komponenten vorhanden sind und damit ein bestimmter Anteil der Zugriffe nicht über die vorhandenen Schnittstellen geschieht. In diesem Fall handelt es sich bei dem Mustervorkommen von *IllegalMethodCall* um eine tatsächliche Schwachstelle in dem System, welche die Softwarearchitektur negativ beeinflusst und daher von einem Reengineer beseitigt werden sollte.

Handelt es sich bei der Klasse class2 in dem Muster *IllegalMethocCall* um eine Klasse aus einer anderen Komponente als die durch die Annotation compAnno repräsentierte Komponente, dann handelt es sich bei dem Mustervorkommen von *IllegalMethodCall* tatsächlich um eine Schwachstelle. In diesem Fall wurden vorhandene Schnittstellen zwischen zwei verschiedenen Komponenten verletzt. Die Auswirkungen der Interface Violation auf das Clustering und damit auch auf die Softwarearchitektur waren in diesem Fall jedoch nicht gravierend genug, um die beiden Komponenten zu einer neuen zu vereinen. Dennoch handelt es sich in diesem Fall um eine Interface Violation und damit um eine Schwachstelle im System, weshalb sie beseitigt werden sollte.

4.3.1.2 Illegaler Zugriff auf eine Variable

Die zweite Variante einer Interface Violation ist der Zugriff auf eine Variable einer fremden Klasse. Da in komponentenbasierten Systemen die Kommunikationen zwischen Komponenten über vordefinierte Schnittstellen geschehen sollte und die Kommunikation über das Auslesen oder Manipulieren von Variablen nicht dazu zählt, ist dies eine Schnittstellenverletzung. Neben der erhöhten Kopplung durch den direkten Zugriff wird diese Art von Interface Violation auch als *Bad Practice* in objektorientierten Sprachen angesehen, weil der Zustand eines Objekts von anderen Objekten geändert wird, ohne dass das Objekt die Änderungen seines Zustands mitbekommen und angemessen reagieren kann.

Die Musterspezifikation in Abbildung 4.17 ähnelt der Musterspezifikation von *IllegalMethodCall* in Abbildung 4.16. Dabei repräsentiert die Annotation vom Typ Component ebenfalls eine der für die Analyse selektierten Komponenten, wodurch die Klasse class1 vom Typ GASTClass in der Rolle accessingClass auf die implementierenden Klassen dieser Komponenten eingeschränkt wird. Die Klasse class1 muss eine Methode method1 implementieren, welche einen Zugriff auf eine fremde Variable enthält. Der Zugriff auf eine Variable wird durch das Objekt fAccess vom Typ VariableAccess modelliert, welches von der Methode method1 referenziert wird. Das Objekt fAccess referenziert ein Objekt aField in der Rolle accessedField

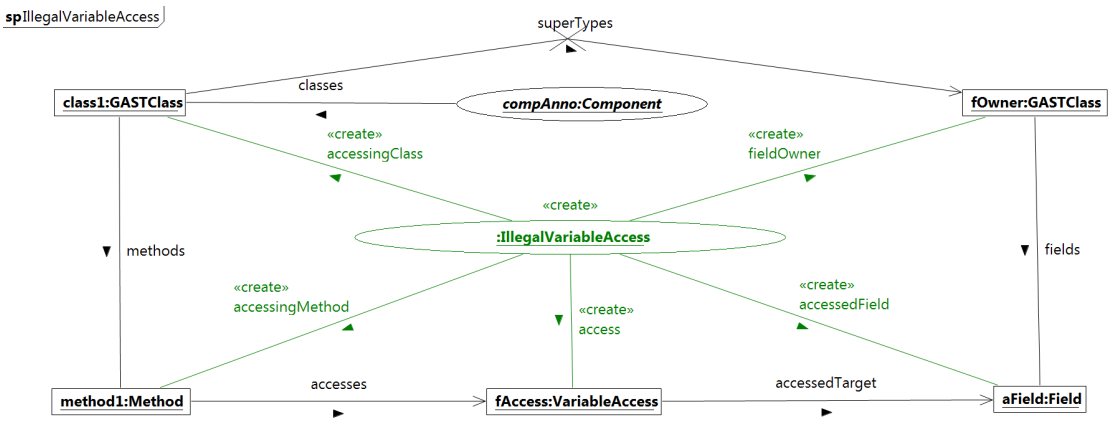

Abbildung 4.17: Musterspezifikation einer Interface Violation durch Zugriff auf eine fremde Variable

über die Referenz accessedTarget. Das Objekt aField vom Typ Field modelliert daher eine Variable, auf die innerhalb der Methode method1 zugegriffen wird. Um zu spezifizieren, dass diese Variable einer anderen Klasse angehört als derjenigen, die den Zugriff durchführt, wurde ein weiteres Objekt fOwner vom Typ GASTClass in der Rolle fieldOwner angelegt. Dieses referenziert das Objekt aField über die Assoziation fields und modelliert damit die Zugehörigkeit der Variable, auf die zugegriffen wird, zu einer anderen Klasse. Eine negative Kante superTypes von class1 zu fOwner spezifiziert, dass die Klasse fOwner keine Oberklasse von class1 sein darf, da es sich in diesem Fall bei der Variable um ein geerbtes Attribut handeln würde.

Mögliche Maßnahmen zur Korrektur Wie schon bei *IllegalMethodCall* existieren auch bei *IllegalVariableAccess* zwei Varianten von Mustervorkommen, je nach Komponentenzugehörigkeit der Klasse fOwner. Gehört die Variable, die durch das Objekt aField repräsentiert wird, einer Klasse an, die zu einer anderen Komponente gehört, als die durch die Component-Annotation repräsentierte Komponente, dann stellt ein Mustervorkommen von IllegalVariableAccess eine tatsächliche Schwachstelle dar. Es gelten die selben Argumente wie schon bei *IllegalMethodCall*.

Gehören die Klassen class1 und fOwner derselben Komponente an, dann muss von einem Reengineer differenziert werden, ob das Clustering zwei konzeptionell verschiedene Komponenten zu einer neuen vereint hat oder die Komponente tatsächlich Teil der konzeptionellen Softwarearchitektur ist.

Unabhängig von der Fallunterscheidung ist es jedoch sinnvoll, ein Vorkommen von *IllegalVariableAccess* zu beseitigen, da Objekte ihre Zustände nicht gegenseitig ändern können sollten, ohne dass das betroffene Objekt die Änderung mitbekommt. Eine Möglichkeit, diese Schwachstelle zu beseitigen, ist die Einschränkung

der Sichtbarkeit der Variable verbunden mit dem Hinzufügen von get- und set-Methoden, welche den Zugriff und die Manipulation der Variable ermöglichen.

4.3.2 Unauthorized Call

Das Schwachstellenmuster *Unauthorized Call* dient der Erkennung von Kommunikation zwischen Komponenten des analysierten Systems. Die Kommunikation findet dabei über Schnittstellen statt, die der kommunizierenden Komponente nicht zur Verfügung gestellt werden. Damit verstößt die Implementierung gegen die in der Softwarearchitektur als *required* festgelegten Schnittstellen. Aus der Implementierung des Systems allein ist eine solche Schwachstelle nicht zu erkennen, da die Schnittstellen einer Komponente nur in der Softwarearchitektur festgelegt werden.

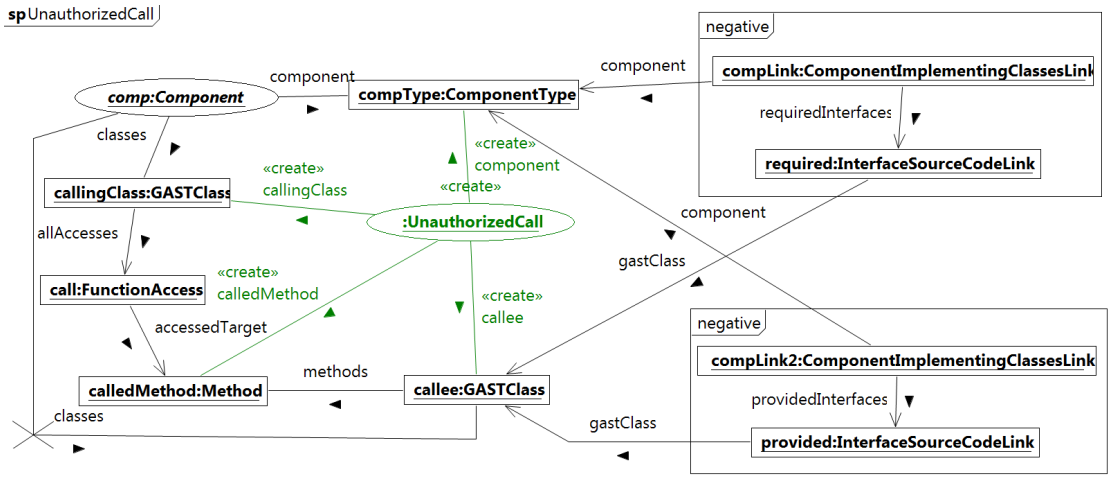

Abbildung 4.18: Musterspezifikation zur Erkennung von Kommunikation über eine nicht erlaubte Schnittstelle

Die Abbildung 4.18 zeigt die Musterspezifikation für die Schwachstelle *Unauthorized Call*. Die Component-Annotation comp referenziert eine zugehörige Klasse callingClass aus einem GAST-Modell, welche einen Zugriff auf eine Methode einer anderen Klasse implementiert. Der Zugriff wird durch das Objekt call vom Typ FunctionAccess modelliert, welches über die Assoziation accessedTarget ein Objekt calledMethod vom Typ Method referenziert. Das Objekt calledMethod wird von einer anderen Klasse callee vom Typ GASTClass implementiert.

Um zu spezifizieren, dass callee nicht zu den benötigten Schnittstellen der durch callingClass implementierten Komponente gehört, muss eine entsprechende SAMM-Instanz in einem negativen Fragment modelliert werden. Die von der Annotation comp annotierte Komponente compType in der SAMM-Instanz wird von dem Objekt compLink aus dem Source-Code-Decorator-Modell referenziert. Das Objekt compLink referenziert ein weiteres Objekt required aus dem Source-Code-

Decorator-Modell vom Typ InterfaceSourceCodeLink über die Assoziation requiredInterfaces. Ein solches Objekt hält eine Referenz auf ein Objekt vom Typ GASTClass, welches eine Schnittstelle der analysierten Komponente repräsentiert. Eine Kante von dem in einem negativen Fragment liegenden Objekt required zu callee über die Assoziation gastClass spezifiziert, dass callee nicht zu den benötigten Schnittstellen der analysierten Komponente gehören darf. Ein weiteres negatives Fragment verhindert das erfolgreiche Matching des Musters, wenn es sich um Kommunikation mit den eigenen Schnittstellen handelt. Das Fragment unterscheidet sich von dem anderen negativen Fragment lediglich in dem Link provided, über den bereitgestellte Schnittstellen der Komponente compType erreicht werden. Eine negative Kante von der Annotation comp zu callee spezifiziert, dass es sich bei der Kommunikation von callingClass mit callee um Kommunikation zwischen zwei verschiedenen Komponenten handelt.

Mögliche Maßnahmen zur Korrektur Um eine Schwachstelle *Unauthorized Call* zu beseitigen, kann entweder die Definition der Schnittstellen einer Komponente überarbeitet werden oder der unerlaubte Aufruf der Methode entfernt werden, die nicht zu den Schnittstellen der Komponente gehört. Im ersten Fall reicht es ein Interface, das die unerlaubte Methode deklariert, zu den benötigten Schnittstellen der betreffenden Komponenten hinzuzufügen, da Schnittstellendefinition einer Komponente und deren Implementierung dadurch wieder in Einklang gebracht werden. Im zweiten Fall muss von einem Reengineer überprüft werden, welche Konsequenzen durch die Entfernung des Methodenaufrufs für die Implementierung des Systems entstehen. Gegebenenfalls müssen umfassende Anpassungen an der Implementierung in Folge der Entfernung des Methodenaufrufs vorgenommen werden.

4.3.3 Inheritance between Components

Vererbungsbeziehungen zwischen Klassen bilden Abhängigkeiten in der Implementierung und verursachen Kopplung zwischen Unter- und Oberklasse. Wenn sich die Oberklasse ändert, muss die Unterklasse an die Änderung angepasst werden. Derartige Abhängigkeiten dürfen zwischen Klassen, die unterschiedlichen Komponenten zugeordnet sind, nicht vorkommen. Eine Komponente muss nach Szyperski [Szy02] von der Implementierung anderer Komponenten, zu denen sie nicht in Kompositionsbeziehung steht, unabhängig entwickelt werden.

Eine Musterspezifikation zur Erkennung von Vererbung über Komponentengrenzen hinweg wird in Abbildung 4.19 dargestellt. Darin enthalten sind zwei Objekte, subClass und superClass, vom Typ GASTClass, die über die Assoziation superTypes verbunden sind und damit zwei Klassen in Vererbungsbeziehung zueinander repräsentieren. Die Component-Annotation referenziert eine davon über die Rolle classes. Eine negative Kante für die Rolle classes zu dem anderen Objekt superClass bedeutet, dass diese nicht zu den implementierenden Klassen der annotierten Komponente gehören darf und damit einer anderen Komponente angehören

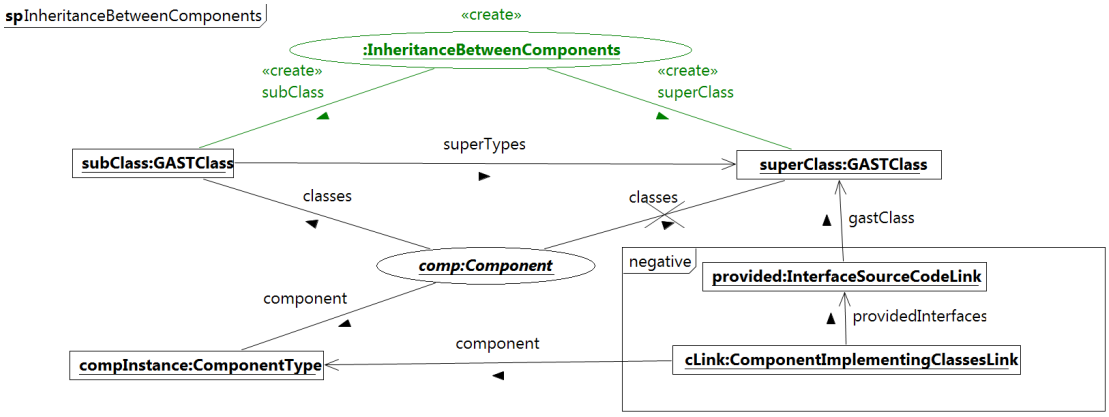

Abbildung 4.19: Musterspezifikation zur Erkennung von Vererbung über Komponentengrenzen hinweg

muss. Da die Component-Annotation lediglich Klassen, aber nicht die Schnittstellen einer Komponenten über den classes-Link referenziert, wird noch ein negatives Fragment benötigt, um zu spezifizieren, dass das Objekt superClass keine Schnittstelle der annotierten Komponente sein darf. Das negative Fragment enthält die beiden Objekte provided vom Typ InterfaceSourceCodeLink und cLink vom Typ ComponentImplementingClassesLink. Damit darf es für das erfolgreiche Matching des Musters kein provided-Objekt geben, welches das Objekt superClass referenziert und über ein Objekt cLink mit der annotierten Komponente compInstance verbunden ist.

Eine *Inheritance-between-Components*-Schwachstelle kann unterschiedlich starke Auswirkungen auf das Clustering bzw. die Softwarearchitektur haben, da die Kopplung zwischen Ober- und Unterklasse davon abhängt, wie viele der geerbten Attribute und Methoden in der Unterklasse verwendet werden. Je mehr Funktionalität eine Unterklasse von der Oberklasse nutzt, desto höher ist die Kopplung und damit auch die Wahrscheinlichkeit, dass beide Klassen im Rahmen eines Clusterings der selben Komponente zugeordnet werden. Das Schwachstellenmuster in Abbildung 4.19 beschreibt einen Fall, in dem das Clustering trotz Kopplung zwischen den beteiligten Klassen, die Klassen unterschiedlichen Komponenten zugeordnet hat. Für Fälle mit höheren Werten für die Kopplung und der daraus resultierenden Zuordnung beider Klassen einer gemeinsamen Komponente existiert noch kein geeignetes Schwachstellenmuster und bleibt zukünftigen Arbeiten überlassen.

Mögliche Maßnahmen zur Korrektur Der Aufwand für die Beseitigung der Schwachstelle kann von Fall zu Fall stark schwanken. In Fällen, in denen umfangreiche Funktionalität geerbt wird, ist es sinnvoll, die Zusammensetzung der Komponenten zu überdenken. Fowler schlägt das Refactoring "*Replace Inheritan-*

ce with Delegation" für *Bad Smells* vor, in denen Vererbung ungünstig eingesetzt wurde [Fow99]. Eine Variante dieses Refactorings in komponentenbasierten Systemen wäre, die geerbte Funktionalität durch Subkomponenten implementieren zu lassen und als Service bereitzustellen, der von anderen Komponenten genutzt werden kann.

In anderen Fällen, in denen es sich um einen geringen Umfang an geerbter Funktionalität handelt, muss differenziert werden, ob das Entfernen der Vererbungsbeziehung und eine Kopie der Funktionalität aus der Oberklasse in die Unterklasse nicht sinnvoller ist als eine Implementierungsabhängigkeit zwischen zwei konzeptionell verschiedenen Komponenten. Einen höheren Grad an Modularisierung des analysierten Systems würde man sich in diesem Fall durch in geringem Maße redundanten Code erkaufen.

4.3.4 Undercover Transfer Object

Transferobjekte werden in komponentenbasierten Systemen zum Datenaustausch genutzt. Die Daten eines Transferobjekts werden von einer Komponente gesetzt, die das Transferobjekt anschließend an eine andere Komponente zur Weiterverarbeitung versendet. Da Transferobjekte lediglich Datenbehälter sind, sind sie kein Teil der Softwarearchitektur. Dennoch führen Transferobjekte zu Kopplung zwischen Klassen, die das Versenden oder Empfangen eines Transferobjekts implementieren und üben dadurch Einfluss auf das Clustering aus.

Wegen ihrer Rolle als Datenbehälter werden Transferobjekte üblicherweise vor dem Clustering herausgefiltert. Oft werden die Klassen von Transferobjekten bereits während der Implementierung eines Systems als solches gekennzeichnet. In CoCoME [coc10] zum Beispiel geschieht dies durch ein Suffix "TO" in der Namensgebung der Klassen, was eine Filterung anhand von Namenskonventionen vor einem Clustering erlaubt. In anderen Fällen, in denen keine derartigen Namenskonventionen vorhanden sind, müssen die strukturellen Eigenschaften einer Transferobjekt-Klasse als Kriterium für die Filterung dienen.

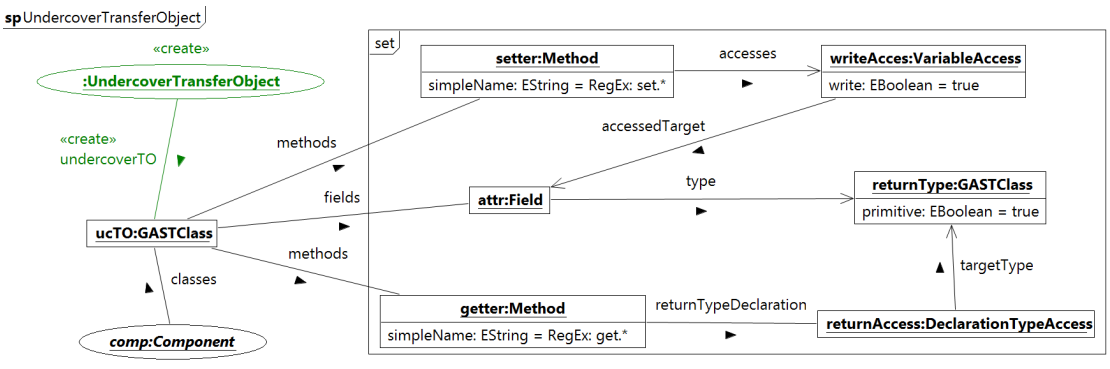

Abbildung 4.20: Musterspezifikation zur Erkennung von Transferobjekten

Ein Transferobjekt hat ähnliche Eigenschaften wie der *Bad Smell Data Class* nach Fowler [Fow99]. Im Wesentlichen hat eine solche Klasse meist primitive Attribute, sowie Getter- und Setter-Methoden für ihre Attribute. Komplexeres Verhalten wird von einem Transferobjekt nicht implementiert.

In Abbildung 4.20 wird die im Rahmen dieser Arbeit verwendete Musterspezifikation für die Schwachstelle *Undercover Transfer Object* dargestellt. Das Objekt attr vom Typ Field repräsentiert ein Attribut der Klasse ucTO. Eine Getter-Methode für das Attribut attr wird durch das Objekt getter vom Typ Method spezifiziert. Eine Attributbedingung für den Namen der Methode mit Hilfe eines regulären Ausdrucks lässt ein Matching nur dann zu, wenn der Name mit einem Präfix "get" anfängt. Der Rückgabetyp einer Getter-Methode muss derselbe sein, wie der Typ des Attributs. Um diese Eigenschaft zu spezifizieren, referenziert die Getter-Methode ein Objekt vom Typ DeclarationTypeAcces über die Assoziation returnTypeDeclaration, welches durch das selbe GASTClass-Objekt getypt ist, wie auch das Attribut attr. Die Setter-Methode wird durch ein Objekt setter vom Typ Method mit einer entsprechenden Attributbedingung für das Präfix des Namens repräsentiert. Das Objekt setter muss einen Schreibzugriff auf das Attribut attr enthalten, was durch ein Objekt vom Typ VariableAccess modelliert wird, in dem das Attribut write den Wert true hat. Da es mehrere Attribute geben kann, für die es Getter- und Setter-Methoden gibt, muss dieser Teilgraph der Musterspezifikation in einem Set-Fragment liegen.

Mögliche Maßnahmen zur Korrektur Eine durch das Muster *Undercover Transfer Object* erkannte Transferobjekt-Klasse kann nach ihrer Identifikation manuell in die Liste der zu filternden Klassen im Rahmen eines Clusterings eingetragen werden, wodurch sie das Clustering nicht weiter beeinflussen kann. Alternativ dazu, könnten identifizierte Transferobjekte mit Hilfe von Namenskonventionen als solche gekennzeichnet werden, sodass sie während des Clusterings automatisiert anhand der Namen gefiltert werden können, anstatt sie einzeln in eine Liste eintragen zu müssen.

4.3.5 Verwendung von Nicht-Transferobjekten zur Kommunikation

Die Komponenten eines Systems dürfen nur über Transferobjekte oder die Werte primitiver Datentypen miteinander kommunizieren. Das Versenden von Referenzen auf Objekte, die Teile einer Komponente implementieren und deshalb keine Transferobjekte sind, verursacht unnötige Kopplung auf der Empfängerseite zu der Versendenden Komponente. Obwohl ein Empfänger lediglich Zugang zu einigen Daten benötigt, erhält er mit der Referenz auf ein Objekt aus einer anderen Komponente Zugang zur Funktionalität, die konzeptionell nicht für ihn vorgesehen ist. Die Verwendung dieser Funktionalität würde zwar auch durch die beiden Muster für *Interface Violations* erkannt werden, doch erlaubt das hier vorgestellte

Muster die Identifikation von Stellen in der Implementierung, die eine Verletzung von Schnittstellen möglich machen, weil die Verwendung von Transferobjekten in der Implementierung nicht konsequent durchgesetzt wurde.

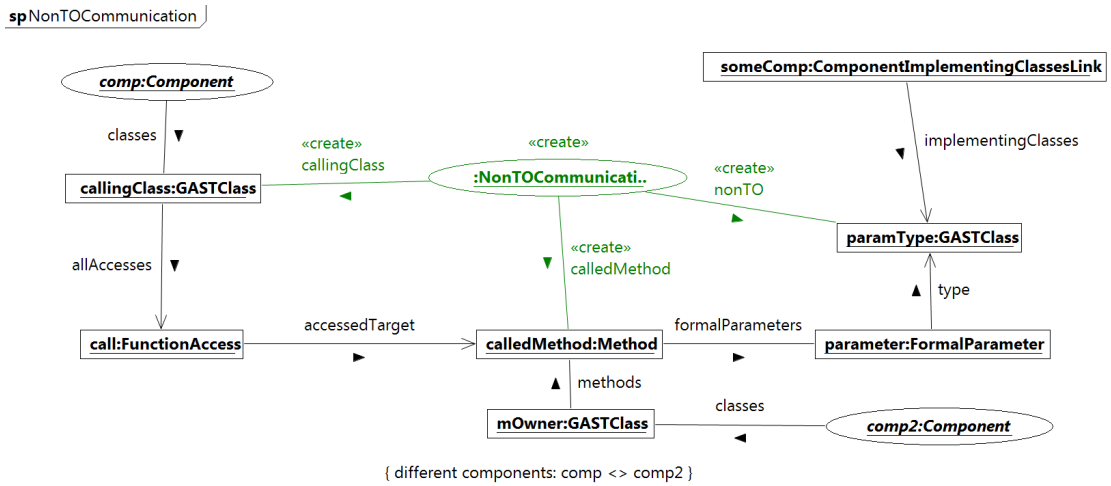

Abbildung 4.21: Musterspezifikation zur Erkennung von Kommunikation über nicht-Transferobjekte

Die Musterspezifikation *NonTOCommunication* zur Erkennung von Kommunikation, die nicht über Transferobjekte führt, wird in Abbildung 4.21 dargestellt. Das Muster beschreibt einen Aufruf einer Methode zwischen zwei Klassen verschiedener Komponenten, wobei ein Parameter nicht durch ein Transferobjekt typisiert ist. Die aufrufende Klasse ist durch das Objekt callingClass vom Typ GASTClass spezifiziert und wird von der Component-Annotation comp referenziert. Die Klasse, deren Methode aufgerufen wird, wird von dem Objekt mOwner repräsentiert, welches von einer anderen Component-Annotation comp2 referenziert wird. Der OCL-Ausdruck "comp <> comp2" beschreibt, dass comp und comp2 unterschiedliche Komponenten sein müssen. Die Zugehörigkeit der Klassen zu verschiedenen Komponenten ist von besonderer Bedeutung, da Methodenaufrufe mit Parametern, die keine Transferobjekte sind, der Konfiguration einer Komponente dienen könnten. Derartige Methodenaufrufe sind innerhalb einer Komponente erlaubt und sollten daher nicht als potentielle Schwachstelle annotiert werden.

Der Methodenaufruf wird durch die beiden Objekte call und calledMethod spezifiziert, die von den beteiligten Klassen über die entsprechenden Assoziationen referenziert werden. Die Methode calledMethod hat einen Parameter, der durch das Objekt parameter vom Typ FormalParameter repräsentiert wird. Der Typ des Parameters wird über die Assoziation type zu einem GASTClass-Objekt festgelegt. Um die Musterspezifikation auf Fälle einzuschränken, in denen der Parameter durch kein Transferobjekt typisiert ist, wurde eine Referenz von einem ComponentImplementingClassesLink zum GASTClass-Objekt hinzugefügt, welches den Typ des

Parameters repräsentiert. Ein GASTClass-Objekt, das von einem ComponentImplementingClassesLink-Objekt referenziert wird, repräsentiert eine Klasse, die einen Teil einer Komponente implementiert und deshalb kein Transferobjekt sein kann. Eine Ausnahme dafür sind Transferobjekte, die nicht vor dem Clustering gefiltert wurden. Diese würden jedoch durch das Muster *UndercoverTransferObject* erkannt werden.

Im Source-Code-Decorator-Modell referenzieren zwar Objekte vom Typ ComponentImplementingClassesLink Klassen, die eine Komponente implementieren, jedoch können diese Klassen durch beispielsweise abstrakte Klassen oder Interfaces weitere Obertypen haben. Diese Typen können ebenfalls der Typ eines Methodenparameters sein und wegen der Polymorphie in objektorientierten Programmiersprachen dennoch die Verwendung der von ComponentImplementingClassesLink referenzierten Klassen ermöglichen. Interfaces oder abstrakte Klassen werden nicht von ComponentImplementingClassesLink-Objekten referenziert, weshalb die Musterspezifikation *NonTOCommunication* nicht alle Fälle von Kommunikation über Nicht-Transferobjekte abdeckt.

Um auch diese Fälle zu berücksichtigen, wird eine weitere Musterspezifikation *NonTOCommunication2* benötigt. Diese unterscheidet sich von *NonTOCommunication* lediglich darin, dass der Typ des Parameters von einem InterfaceSourceCodeLink-Objekt referenziert wird. Im Source-Code-Decorator-Modell halten InterfaceSourceCodeLink-Objekte die Korrespondenz zwischen einer Komponentenschnittstelle im SAMM und den Interfaces oder abstrakten Klassen im GAST. Eine Abbildung, in der die Musterspezifikation *NonTOCommunication2* dargestellt ist, ist im Anhang B.1 zu finden.

4.4 Einschränkungen des kombinierten Ansatzes

Die Kombination von Clustering- und musterbasiertem Reverse Engineering, wie sie im Rahmen dieser Arbeit vorgestellt wird, unterliegt einigen Einschränkungen. Dieser Abschnitt erläutert die wichtigsten dieser Einschränkungen.

Statische Analyse von Softwaresystemen Das Clustering- und das musterbasierte Reverse Engineering sind beides statische Analyseansätze, weswegen der im Rahmen dieser Arbeit verfolgte kombinierte Ansatz das Laufzeitverhalten eines Systems nicht berücksichtigen kann.

Beschränkung auf Java, C++ und Delphi Der Parser SISSy, der den Quellcode eines Systems in eine GAST-Instanz einliest, unterstützt aktuell die Programmiersprachen Java, C++ und Delphi. Deshalb können momentan nur Systeme analysiert werden, die in diesen Programmiersprachen implementiert wurden. Konzeptionell sind aber auch andere objektorientierte Programmiersprachen möglich, sofern sie in eine Instanz des GAST-Modells eingelesen werden können.

Beschränkung auf komponentenbasiert entwickelte Softwaresysteme Ferner unterliegt der Clustering-basierte Ansatz der Einschränkung, dass ein analysiertes System komponentenbasiert sein muss, da ansonsten keine dem System zu Grunde liegende Softwarearchitektur existiert, die rekonstruiert werden kann. Diese Einschränkung gilt auch für die Kombination der beiden Ansätze, da die durch das Clustering rekonstruierte Softwarearchitektur im Rahmen des musterbasierten Reverse Engineerings wiederverwendet wird.

Abhängigkeit der Mustersuche von den Clustering-Ergebnissen Die Ergebnisse der Mustersuche können mit der durch das Clustering rekonstruierten Softwarearchitektur für ein System stark variieren. Da die Mustersuche auf einzelne Komponenten aus der Softwarearchitektur angewendet wird, sind die Musterfunde spezifisch für eine oder mehrere analysierte Komponenten. Ändert sich die Zusammensetzung von Komponenten auf Grund eines Clustering-Durchlaufs mit beispielsweise veränderten Metrikgewichten, so werden auch andere Muster im Rahmen einer anschließenden Mustersuche gefunden.

Die musterbasierte Suche nach Schwachstellen in einem analysierten System sollte daher erst dann durchgeführt werden, wenn durch Veränderung der Metrikgewichte keine Verbesserung der durch das Clustering rekonstruierten Softwarearchitektur erreicht werden kann. Andernfalls besteht die Gefahr, dass ein System auf Grund der Beseitigung von vermeintlichen Schwachstellen, die einer ungünstigen Wahl von Metrikgewichten zuzuordnen sind, mit neuen Schwachstellen versehen wird. Die Softwarearchitektur eines analysierten Systems könnte sich dadurch noch weiter von der konzeptionellen Architektur entfernen und damit verschlechtern, anstatt sich zu verbessern.

Ausblenden von Mustervorkommen Der im Rahmen dieser Arbeit erarbeitete Reverse-Engineering-Ansatz analysiert während der Mustersuche einzelne Komponenten bzw. eine Kombination von Komponenten eines Softwaresystems. Dadurch werden Teile des Systems im Rahmen einer Mustersuche ausgeblendet. Potentielle Schwachstellen oder andere Mustervorkommen, die sich in ihrer Struktur über das gesamte System erstrecken, bleiben auf diese Weise verborgen.

Durch Sammeln von weiteren praktischen Erfahrungen mit dem kombinierten Reverse-Engineering-Ansatz muss in zukünftigen Arbeiten geklärt werden, ob durch das Ausblenden von Teilen eines Systems für einen Reengineer bedeutende Informationen verloren gehen und wie dem gegebenenfalls entgegen gewirkt werden kann.

Exemplarischer Musterkatalog zur Schwachstellenerkennung In Abschnitt 4.3 wurden einige Muster vorgestellt, die typisch für Schwachstellen in einem komponentenbasierten System sind. Die Schwachstellenmuster bilden jedoch nur einen Teil der möglichen Schwachstellen ab. Weitere Arbeiten sind nötig, in denen Strukturen von Schwachstellen in einem komponentenbasierten System analysiert und

in weitere Schwachstellenmuster überführt werden.

Obwohl im Rahmen dieser Arbeit ausschließlich Schwachstellenmuster erarbeitet wurden, um damit negative Eigenschaften eines Systems aufzudecken, ist der kombinierte Reverse-Engineering-Ansatz keineswegs darauf beschränkt. Es besteht auch die Möglichkeit, nach typischen Entwurfsmustern in einem komponentenbasierten System zu suchen und damit genauere Entwurfsinformationen zu einzelnen Komponenten zu erhalten als allein durch die Softwarearchitektur ersichtlich ist. Dies übersteigt jedoch den Rahmen dieser Arbeit und wird zukünftigen Arbeiten überlassen.

5 Umsetzung

Um die in Kapitel 4 vorgestellten Konzepte umzusetzen, mussten einige technische Anpassungen an Reclipse vorgenommen werden. Diese werden im Folgenden erläutert. Einer der Beiträge dieser Arbeit ist die Umstellung der Generierung von Suchalgorithmen in Reclipse auf das neue Metamodell für Story-Diagramme [DHL+11]. Infolgedessen musste auch die Inferenz angepasst werden, welche die Suchalgorithmen im Analyseprozess von Reclipse wiederverwendet. Außerdem war eine Erweiterung der Inferenz notwendig, um die in Abschnitt 4.2.4 vorgestellte Beschränkung der Inferenz auf einzelne Komponenten umzusetzen.

Der folgende Abschnitt liefert einen Überblick über die Komponenten, die zur Implementierung des kombinierten Reverse-Engineering-Ansatzes benötigt werden. Abschnitt 5.2 beschreibt die wichtigsten Anpassungen, die an der Generierung der Suchalgorithmen erfolgt sind. Anschließend werden die wichtigsten Anpassungen und Erweiterungen der Inferenz in Abschnitt 5.3 erläutert.

5.1 Architektur der Werkzeugumgebung

Dieser Abschnitt gibt einen Überblick über die Softwarekomponenten, die zur Umsetzung des in Kapitel 4 vorgestellten Reverse-Engineering-Ansatzes verwendet werden. Die Abbildung 5.1 visualisiert die verwendeten Komponenten und ihre Abhängigkeiten zueinander. Direkte Abhängigkeiten zu Komponenten wurden dabei in der Darstellung ausgeblendet, wenn eine transitive Abhängigkeit besteht. Die Komponenten wurden als Eclipse[1] Plug-ins umgesetzt. Eine Auflistung der Plug-ins, welche die abgebildeten Komponenten implementieren, ist im Anhang E zu finden.

Zur Umsetzung des in dieser Arbeit beschriebenen, kombinierten Reverse-Engineering-Ansatzes wurden die vier Werkzeuge SISSy [TS04], SoMoX [Kro10], Story-Diagramm-Interpreter [GHS09] und Reclipse [NSW+02] verwendet, deren Komponenten im Folgenden erläutert werden. Anschließend folgt ein Überblick über angepasste und neu entwickelte Komponenten.

SISSy Der SISSy Parser (Nr. 5 in Abb. 5.1) wird zum Einlesen eines Softwaresystems genutzt. Das eingelesene System wird dabei als Instanz des GAST-Metamodells (1) zur Verfügung gestellt.

[1] www.eclipse.org

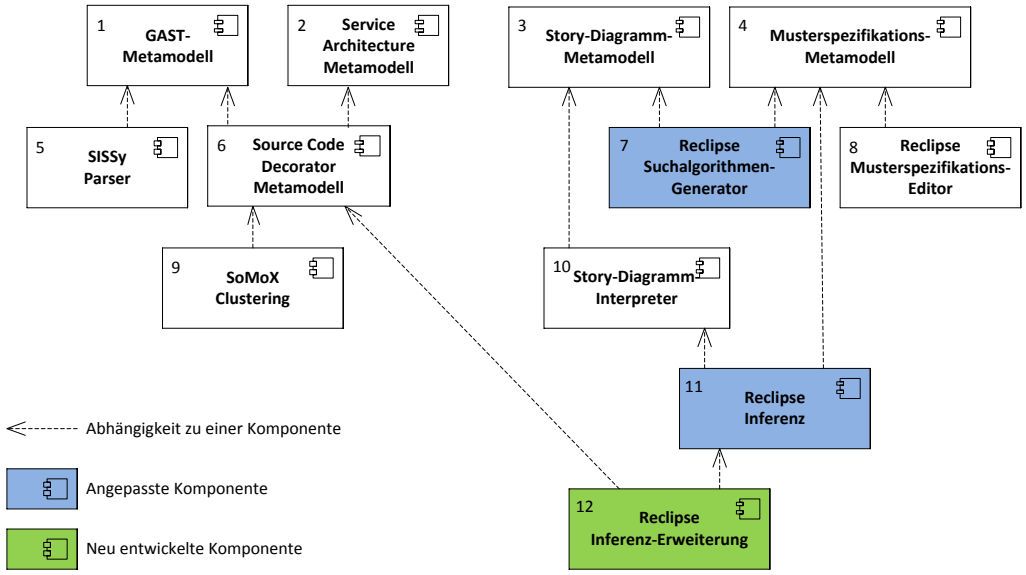

Abbildung 5.1: Übersicht der beteiligten Komponenten zur Implementierung des kombinierten Reverse-Engineering-Ansatzes

SoMoX Das Clustering-basierte Reverse Engineering wird von SoMoX (9) implementiert und wird auf einer Instanz des GAST-Metamodells (1) ausgeführt. Als Ergebnis des Clusterings wird die rekonstruierte Softwarearchitektur als Instanz des Service-Architecture-Metamodells (2) bereitgestellt. Um die Zuordnung von Komponenten aus der SAMM-Instanz zu deren Implementierung im GAST-Modell zu repräsentieren, wird eine Instanz des Source-Code-Decorator-Metamodells (6) erzeugt.

Story-Diagramm-Interpreter Der Story-Diagramm-Interpreter (10) führt Story-Diagramme aus, die auf Basis des Story-Diagramm-Metamodells (3) spezifiziert wurden.

Reclipse Der Musterspezifikations-Editor (8) in Reclipse erzeugt Instanzen des Musterspezifikations-Metamodells (4). Der Suchalgorithmen-Generator (7) verwendet diese, um Instanzen des Story-Diagramm-Metamodells (3) zu generieren. In der Inferenz wird der Story-Diagramm-Interpreter (10) dazu verwendet, um die generierten Suchalgorithmen auszuführen. Für die Bewertung von Mustervorkommen wird deren Musterspezifikation herangezogen, weshalb eine Abhängigkeit von der Inferenz zum Musterspezifikations-Metamodell (4) besteht. Da in Reclipse verschiedene AST-Metamodelle durch eine Extension in Eclipse definiert werden können, existiert keine direkte Abhängigkeit zum GAST-Metamodell (1).

Anpassungen und Neuentwicklungen Wie bereits in Abschnitt 3.4 beschrieben, befindet sich Reclipse in einer Umstellung von Metamodellen, die auf Fujaba [Fuj10] basieren, auf EMF-basierte Metamodelle. Im Zuge der Umstellung wurde ein neues Metamodell für Story-Diagramme (3) entwickelt [DHL+11]. Eine Umstellung des Story-Diagramm-Interpreters auf das neue Metamodell erfolgt parallel zu dieser Arbeit. Ein EMF-basierter Musterspezifikations-Editor war zu Beginn dieser Arbeit bereits prototypisch implementiert.

Im Rahmen dieser Arbeit wurden die Generierung der Suchalgorithmen (7), sowie die Inferenz in Reclipse (11) auf die neuen Metamodelle für Story-Diagramme und die Musterspezifikation umgestellt. Eine Erweiterung der Inferenz (12), welche die Selektion der zu analysierenden Komponenten (vergl. Abs. 4.2.4) in dem kombinierten Reverse-Engineering-Ansatz ermöglicht, wurde neu entwickelt.

Der folgende Abschnitt 5.2 beschreibt die wichtigsten Anpassungen, die an dem Suchalgorithmen-Generator erfolgt sind. Anschließend folgt ein Abschnitt über die Anpassungen und Erweiterungen an der Inferenz in Reclipse.

5.2 Anpassung der Story-Diagramm-Generierung

Im Rahmen der Masterarbeit von Fockel [Foc10] wurde erstmals der Story-Diagramm-Interpreter [GHS09] für die Inferenz in Reclipse eingesetzt. Die Verwendung des Story-Diagramm-Interpreters birgt den Vorteil, dass für Story-Diagramme, die in Reclipse die Suchalgorithmen für einzelne Muster implementieren, kein ausführbarer Java-Code [FNTZ00, GSR05] mehr generiert werden muss. Dies vereinfacht den Reverse-Engineering-Prozess in Reclipse. In seiner Umsetzung verwendet Fockel jedoch Musterspezifikationen auf Basis eines mittlerweile veralteten Metamodells und generiert aus diesen die Suchalgorithmen. Außerdem wurde das in seiner Umsetzung verwendete Metamodell für die Story-Diagramme im Rahmen der Reclipse-Umstellung auf EMF-basierte Modelle umfassend überarbeitet [DHL+11]. Damit sind sowohl Quell- als auch Zielmodell des Schritts zur Generierung der Mustersuchalgorithmen nicht mehr aktuell und wurden daher im Rahmen dieser Arbeit durch die neuen Metamodelle ersetzt.

Der folgende Abschnitt beschreibt die Abbildung von dem alten Story-Diagramm-Metamodell auf das neue Metamodell. Anschließend folgt in Abschnitt 5.2.2 eine Erläuterung von Refactorings, die am Suchalgorithmen-Generator durchgeführt wurden.

5.2.1 Abbildung der Story-Diagramm-Metamodelle von Alt auf Neu

Um die Generierung der Suchalgorithmen in Reclipse von dem Fujaba-basierten Story-Diagramm-Metamodell auf das aktuellere, EMF-basierte Story-Diagramm-Metamodell umzustellen, mussten zunächst Elemente der beiden Metamodelle identifiziert werden, die auf einander abgebildet werden können. Anschließend

wurden diese in der Implementierung des Suchalgorithmus-Generators ersetzt. Für Modellelemente, für die keine direkte Abbildung möglich ist, musste die Implementierung des Generators auf vergleichbare Konstrukte des neuen Metamodells angepasst werden.

Im Folgenden wird die Abbildung in zwei Abschnitten vorgestellt. Im ersten Abschnitt wird die Abbildung des Teils der Metamodelle beschrieben, der zur Modellierung von Aktivitäten benötigt wird. Der zweite Abschnitt befasst sich mit der Abbildung des Teils, der zur Modellierung von Story Patterns verwendet wird. Neben den Teilen für Aktivitäten und Story Patterns enthält das neue Story-Diagramm-Metamodell auch einen Teil zur Modellierung von einer Vielzahl von Ausdrücken, um beispielsweise Bedingungen in Aktivitäten und Story Patterns zu spezifizieren. Auf einen eigenen Abschnitt über die Abbildung der Ausdrücke wird hier auf Grund des Umfangs des neuen Story-Diagramm-Metamodells für diesen Teil verzichtet und auf das Papier [DHL+11] verwiesen.

5.2.1.1 Abbildung von Aktivitäten

Das neue Story-Diagramm-Metamodell wurde in Bezug auf die Modellierung von Aktivitäten nur geringfügig verändert. Einen Ausschnitt der beiden Metamodelle für Aktivitäten ist in Abbildung 5.2 dargestellt. Elemente mit gleicher Semantik wurden in den Klassendiagrammen gleich angeordnet, damit Gemeinsamkeiten der beiden Metamodelle leichter erkennbar sind.

UMLActivityDiagram und Activity repräsentieren beide eine Aktivität. In Fujaba referenziert die Klasse UMLActivityDiagram Elemente, die zur Spezifikation der Aktivität benötigt werden und von UMLDiagramItem erben, über die geerbte Assoziation elements. Die Klasse Activity im neuen Story-Diagramm-Metamodell referenziert ihre Elemente über getrennte Assoziationen. Knoten, die durch ActivityNode repräsentiert werden, werden über die Assoziation ownedActivityNodes referenziert. Kanten werden durch ActivityEdge repräsentiert und von Activity über die Assoziation ownedActivityEdges referenziert.

In Fujaba werden Knoten durch ein Element vom Typ UMLActivity repräsentiert. Kanten werden hingegen durch UMLTransition repräsentiert. Die Namen der Assoziationen zur Modellierung von eingehenden und ausgehenden Kanten eines Knotens wurden von entry und exit im Fujaba-basierten Metamodell auf incoming und outgoing im neuen Metamodell geändert.

Ein Knoten einer Aktivität wird in beiden Metamodellen durch diverse Spezialisierungen der Klasse UMLActivity, respektive ActivityNode, realisiert. Die Klasse UMLStoryActivity, respektive StoryNode, repräsentiert einen Knoten, der ein Story Pattern enthält. Im Gegensatz zum alten Metamodell wird in dem neuen Story-Diagramm-Metamodell zwischen Knoten unterschieden, deren Story Pattern lediglich ein Matching repräsentiert (MatchingStoryNode) und Knoten, deren Story Pattern neben dem Matching auch Veränderungen am Wirtsgraphen vornimmt (ModifyingStoryNode).

Das neue Story-Diagramm-Metamodell enthält eine weitere Spezialisierung Ac-

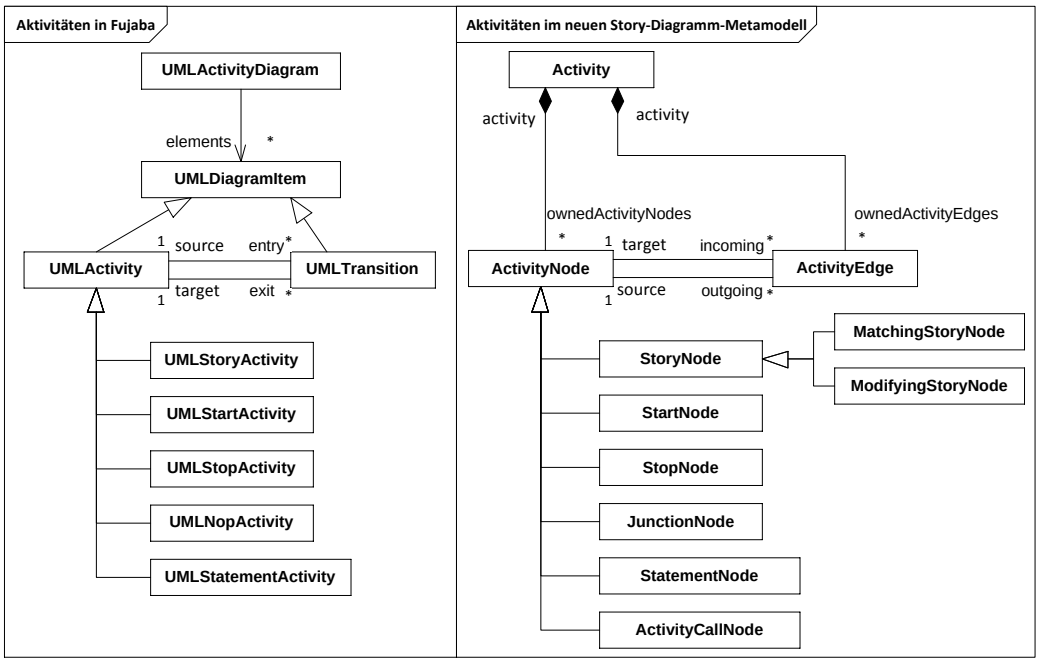

Abbildung 5.2: Metamodellausschnitt zur Modellierung von Aktivitäten

tivityCallNode von ActivityNode, zu der in dem alten Metamodell kein Element mit gleicher Semantik existiert. Die Klasse ActivityCallNode repräsentiert einen Aufruf eines anderen Story-Diagramms. Da Story-Diagramme im alten Metamodell die Semantik einer Methode in einer Klasse spezifizieren, ist dort der Aufruf eines anderen Story-Diagramms gleichbedeutend mit einem Aufruf einer Methode. Ein Aufruf einer Methode kann im alten Metamodell innerhalb eines Story Patterns mit Hilfe von Java-Code und in einem UMLCollabStat-Element spezifiziert werden. Verwendungen von UMLCollabStat zum Aufruf einer Methode wurden bei der Anpassung des Suchalgorithmen-Generators durch Aufrufe von Story-Diagrammen (ActivityCallNode) ersetzt. Ein UMLCollabStat-Element kann jedoch beliebigen Java-Code enthalten und beispielsweise Bedingungen formulieren, die während der Ausführung eines Suchalgorithmus gelten müssen. Derartiger Java-Code wurde durch gleichwertige Ausdrücke mit Hilfe des neuen Story-Diagramm-Metamodells oder von OCL [OMG06] formalisiert und in der Generierung ersetzt.

5.2.1.2 Abbildung von Story Patterns

Der Teil des neuen Story-Diagramm-Metamodells zur Modellierung von Story Patterns wurde gegenüber dem vorherigen Metamodell umfassend verändert. Einen Ausschnitt der beiden Metamodelle für Story Patterns wurde in Abbildung 5.3

dargestellt.

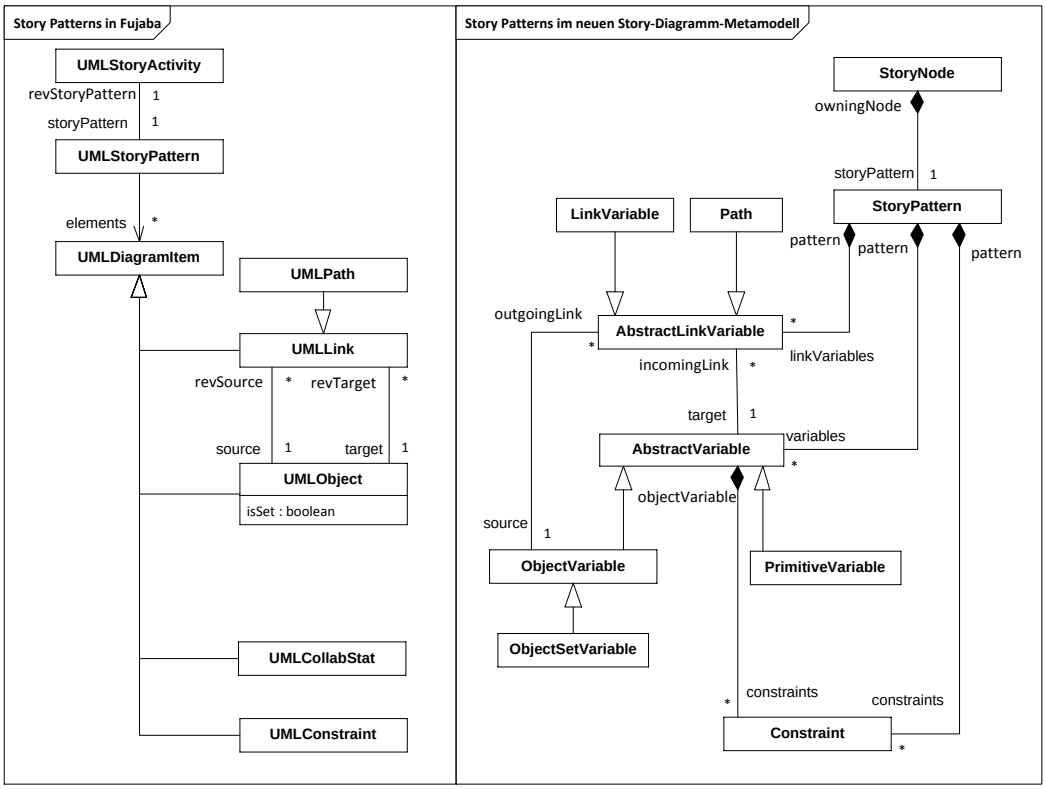

Abbildung 5.3: Metamodellausschnitt zur Modellierung von Story Pattern

Ein Story Pattern wird in Fujaba als UMLStoryPattern spezifiziert und an einen Knoten UMLStoryActivity in einer Aktivität angehängt. Im neuen Story-Diagramm-Metamodell wird dafür ein StoryPattern spezifiziert und an einen StoryNode angehängt. Wie bei den Aktivitäten in Fujaba referenziert ein UMLStoryPattern die Elemente (Spezialisierungen von UMLDiagramItem), die zur Spezifikation eines Story Patterns benötigt werden, über eine geerbte Assoziation elements. Ein StoryPattern im neuen Metamodell referenziert Kindelemente über die Assoziationen linkVariables, variables und constraints in den unterschiedlichen Rollen getrennt voneinander.

Die Klasse UMLObject repräsentiert im alten Metamodell eine Variable in einem Story Pattern. Die Namensgebung ist dabei trügerisch, da die Variable nicht nur den Wert eines Objekts, sondern auch den Wert eines primitiven Datentyps repräsentieren kann. Das neue Metamodell sieht zur Repräsentation von Variablen die Klasse AbstractVariable vor. Die Spezialisierungen ObjectVariable und PrimitiveVariable von der Klasse AbstractVariable repräsentieren entsprechend eine Ob-

jektvariable bzw. eine primitive Variable. Objektmengen gleichen Typs werden in Fujaba als UMLObject mit dem Wert true für das Attribut isSet vom Typ boolean repräsentiert. Das neue Story-Diagramm-Metamodell sieht dafür eine Spezialisierung ObjectSetVariable der Klasse ObjectVariable vor.

Links zwischen Objekten werden im alten Metamodell durch die Klasse UMLLink repräsentiert. Eine source- und target-Assoziation von UMLLink verweist auf UMLObject und repräsentiert die Quelle bzw. das Ziel einer Referenz. UMLPath erbt von UMLLink und repräsentiert einen Pfad zwischen zwei Objekten, der gegebenenfalls über andere Objekte und ihre Links führt. Im neuen Story-Diagramm-Metamodell werden Pfade (Path) und einzelne Links (LinkVariable) als Spezialisierungen von AbstractLinkVariable spezifiziert. Da eine Referenz nur ein Objekt und keinen primitiven Wert als Quelle haben kann, führt die Assoziation source von AbstractLinkVariable zu ObjectVariable. Im Gegenzug ermöglicht das neue Metamodell das Referenzieren von primitiven Werten, weswegen die Assoziation target auf AbstractVariable verweist.

Die Klassen UMLConstraint und UMLCollabStat im alten Metamodell haben gemeinsam, dass beide durch Java-Code ausgedrückt werden. Im Gegensatz zu UMLCollabStat, welches beliebigen Java-Code enthalten kann, repräsentiert die Klasse UMLConstraint eine boolsche Bedingung, weshalb ihr Java-Code einen boolschen Java-Ausdruck enthalten muss. Der Java-Code von UMLCollabStat-Elementen wurde in der Generierung von Suchalgorithmen durch ActivityCallNode-Elemente ersetzt, wenn es sich um einen Aufruf einer Methode handelt. Boolsche Java-Ausdrücke, die zuvor durch UMLConstraint oder UMLCollabStat spezifiziert wurden, wurden im Suchalgorithmen-Generator durch Constraint-Elemente aus dem neuen Story-Diagramm-Metamodell ersetzt. Ein Constraint repräsentiert ebenfalls eine boolsche Bedingung und kann durch einen OCL-Ausdruck oder mit Hilfe des Teils aus dem neuen Story-Diagramm-Metamodell für Ausdrücke spezifiziert werden [DHL+11].

5.2.2 Struktur des angepassten Generators

Im Zuge der Anpassungen des Suchalgorithmen-Generators wurde die Struktur des Generierungsmechanismus in Reclipse überarbeitet. Das vorherige Design war spezifisch für die Generierung von Code. Zudem erlaubten die in dem Design vorgesehenen Schnittstellen lediglich Musterspezifikationen auf Basis der alten Metamodelle. Die Abbildung 5.4 visualisiert die Struktur des Generierungsmechanismus in Reclipse, sowie die Erweiterung für die Generierung der Suchalgorithmen, als Klassendiagramm.

5.2.2.1 Allgemeiner Generierungsmechanismus

Die am Generierungsmechanismus von Reclipse beteiligten Klassen werden in der Abbildung in dem Fragment org.reclipse.generator dargestellt. Die zentrale Rolle spielt darin die Klasse AbstractGenerator, die zusammen mit dem Interface IGene-

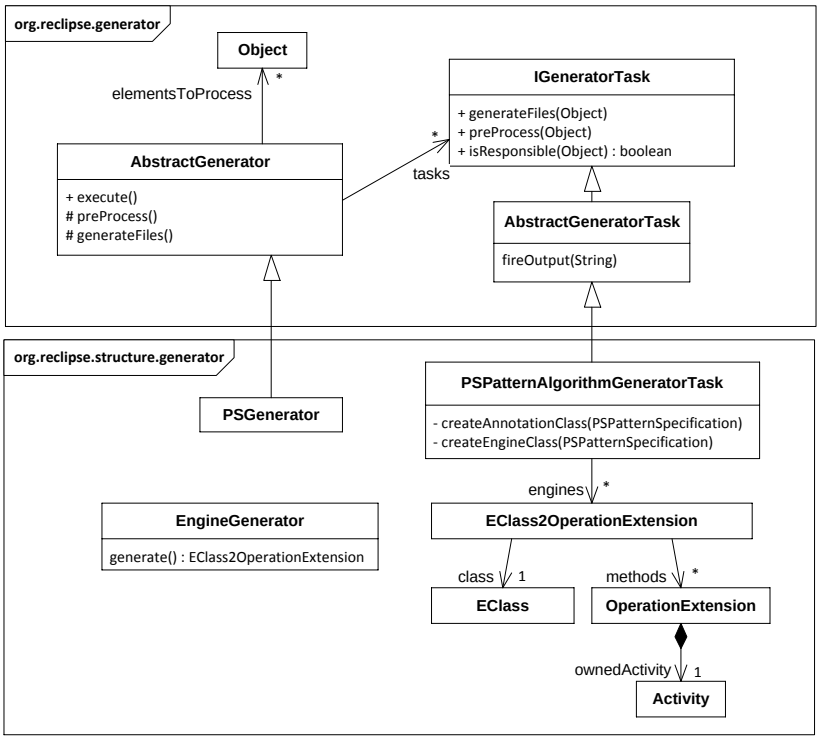

Abbildung 5.4: Ausschnitt eines Klassendiagramms des Generierungsmechanismus in Reclipse und dessen Erweiterung für die Story-Diagramm-Generierung

ratorTask das Entwurfsmuster *Chain of Responsibility* [GHJV95] implementiert. Bei der Klasse AbstractGenerator werden Elemente registriert, die das Interface IGeneratorTask implementieren. IGeneratorTask repräsentiert eine Aufgabe, in Folge deren Ausführung etwas generiert wird. Außerdem werden bei der Klasse AbstractGenerator Elemente registriert, für die etwas generiert werden muss. Diese werden über die Assoziation elementsToProcess über den allgemeinen Typ Object referenziert.

In der Methode execute() der Klasse AbstractGenerator wird zunächst die Methode preProcess() und anschließend generateFiles() aufgerufen. Die beiden Methoden preProcess() und generateFiles() delegieren den Aufruf für jedes registrierte Objekt, das über die Assoziation elementsToProcess referenziert und als Parameter übergeben wird, an die registrierten IGeneratorTask-Elemente, die dafür zuständig sind. Die Zuständigkeit wird zuvor über einen Aufruf von isResponsible(Object) auf einem IGeneratorTask-Element geprüft.

Die Klasse AbstractGeneratorTask implementiert das Interface IGeneratorTask und realisiert eine Anbindung an eine Benutzerschnittstelle, um beispielsweise Konsolenausgaben während des Generierungsprozesses über einen Aufruf der Me-

thode fireOutput(String) zu ermöglichen.

Da Elemente, für die etwas generiert werden muss, im gesamten Entwurf über ihren den Typ Object verwendet werden, entfallen jegliche Anforderungen an ein Metamodell der Elemente. Damit kann der Generierungsmechanismus von Reclipse für verschiedene Aufgaben, die eine Generierung erfordern, wiederverwendet werden.

5.2.2.2 Erweiterung für die Generierung von Suchalgorithmen

Der Suchalgorithmen-Generator ist eine Erweiterung des in dem vorhergehenden Abschnitt vorgestellten Generierungsmechanismus. Die Erweiterung wurde in Abbildung 5.4 in dem Fragment org.reclipse.structure.generator dargestellt. Die Klasse PSGenerator erweitert die Klasse AbstractGenerator, indem sie die generierten Modelle in einem Ecore-Modell persistiert. Die Klasse PSPatternAlgorithmGeneratorTask spezialisiert die Klasse AbstractGeneratorTask und implementiert die Generierung von Annotations- und Engine-Klassen. Der Suchalgorithmus einer Engine-Klasse erzeugt bei einem Mustervorkommen eine Instanz der Annotationsklasse, die für das Muster generiert wurde und verbindet diese mit den Objekten des Mustervorkommens, die eine im Muster spezifizierte Rolle übernehmen.

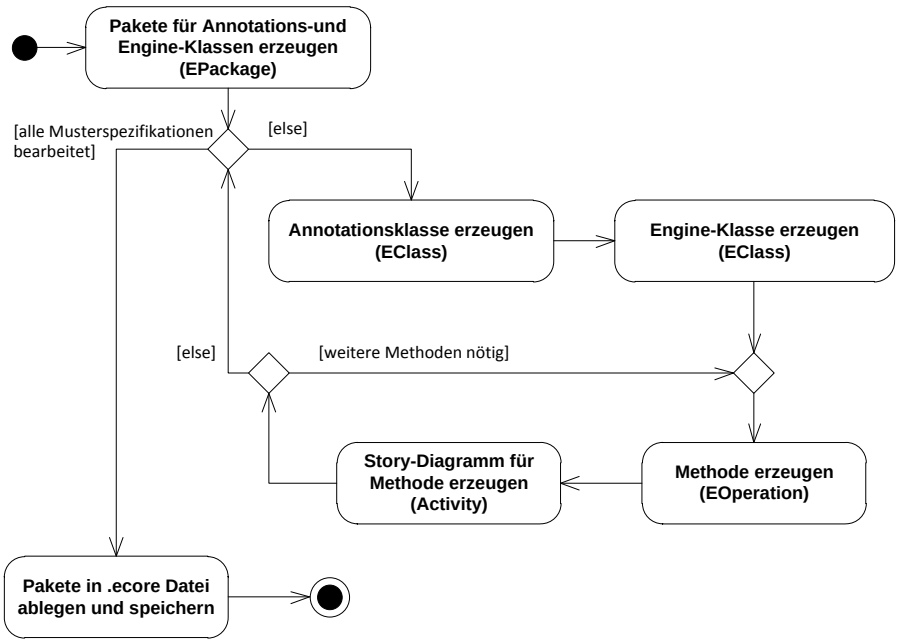

Abbildung 5.5: Ablauf der Generierung von Suchalgorithmen als Aktivitätendiagramm (angelehnt an Ablauf in [Foc10], S. 77)

Der Ablauf der Generierung von Suchalgorithmen wird in Abbildung 5.5 als Ak-

tivitätendiagramm dargestellt. Eine Abbildung des Metamodells für Annotations- und Engine-Klassen ist in Anhang D zu finden. Die Generierung beginnt mit der Erzeugung jeweils eines Pakets (EPackage) für Annotations- und Engine-Klassen. Die folgenden Schritte werden anschließend für jede Musterspezifikation in einem Musterkatalog durchgeführt.

Eine Annotationsklasse wird für jede Musterspezifikation durch einen Aufruf der Methode createAnnotationClass(PSPatternSpecification) in PSPatternAlgorithmGeneratorTask generiert. Die Annotationsklasse wird nach der Musterspezifikation mit einem Suffix „Annotation“ benannt und erbt von der Klasse ASGAnnotation (siehe Anhang D), definiert aber keine neuen Attribute oder Methoden.

Danach wird eine Engine-Klasse durch einen Aufruf der Methode createEngineClass(PSPatternSpecification) generiert und erhält nach und nach Methoden (EOperation), die mit Hilfe von Story-Diagrammen (Activity) den Suchalgorithmus für das Muster implementieren. Eine generierte Engine-Klasse erbt von der Klasse AnnotationEngine (siehe Anhang D).

Die Generierung der Methoden und Story-Diagramme wird in der Klasse EngineGenerator gekapselt und durch den Aufruf der Methode generate() gestartet. Für jede Engine-Klasse wird eine neue Instanz der Klasse EngineGenerator innerhalb des Aufrufs von createEngineClass(PSPatternSpecification) in der Klasse PSPatternAlgorithmGeneratorTask erzeugt und mit dem Muster (PSPatternSpecification) parametrisiert.

Der Aufruf liefert eine Instanz der Klasse EClass2OperationExtension zurück. Die Klasse EClass2OperationExtension vereinfacht den Zugriff auf den Typ der Engine-Klasse, der durch EClass repräsentiert ist und auf die dafür generierten Story-Diagramme (Activity), die von einer Erweiterung einer EOperation durch die Klasse OperationExtension repräsentiert werden.

Nachdem alle Musterspezifikationen bearbeitet wurden, werden die zu Beginn erzeugten Pakete (EPackage) mit den darin enthaltenen Annotations- und Engine-Klassen in einer .ecore-Datei persistiert.

Die Generierung von Story-Diagrammen, die durch den Aufruf der Methode generate() in der Klasse EngineGenerator gekapselt wird, findet über Delegation an weitere Klassen statt. Diese kapseln die Generierung einzelner für den Suchalgorithmus notwendiger Methoden und deren Story-Diagramme, welche die Semantik der Methoden festlegen. In diesem Teil des Suchalgorithmen-Generators wurden im Rahmen dieser Arbeit lediglich die Verwendung der Fujaba-basierten Metamodelle für Musterspezifikation und Story-Diagramme durch die neuen EMF-basierten Metamodelle ersetzt. Da die Abbildung der Metamodelle in Abschnitt 5.2.1 beschrieben wurde, wird an dieser Stelle auf einen detaillierteren Einblick in die Implementierung verzichtet und auf die Arbeiten [Tra06, Wen07, Foc10] verwiesen.

5.3 Anpassung der Inferenz

Im Rahmen dieser Arbeit mussten zwei Aufgaben den Inferenz-Mechanismus betreffend bearbeitet werden. Zum einen musste der Inferenz-Mechanismus auf die neuen Metamodelle umgestellt und eine Anbindung des Story-Diagramm-Interpreters realisiert werden. Diese Anpassungen werden im folgenden Abschnitt erläutert. Zum anderen musste die angepasste Inferenz erweitert werden, um die in Abschnitt 4.2.4 beschriebene Beschränkung der Inferenz auf einzelne Komponenten zu realisieren. Diese Erweiterung wird in Abschnitt 5.3.2 vorgestellt.

5.3.1 Anpassung der Inferenz an die neuen Metamodelle

Die Umstellung von Reclipse auf EMF-basierte Metamodelle, sowie die Anbindung an den Story-Diagramm-Interpreter [GHS09] erforderte eine Anpassung des Inferenz-Mechanismus in Reclipse. Die Abbildung 5.6 visualisiert die wichtigsten Elemente der angepassten Inferenz in einem Klassendiagramm. Die Anbindung der Inferenz an eine Benutzerschnittstelle wurde hier aus Platzgründen vernachlässigt.

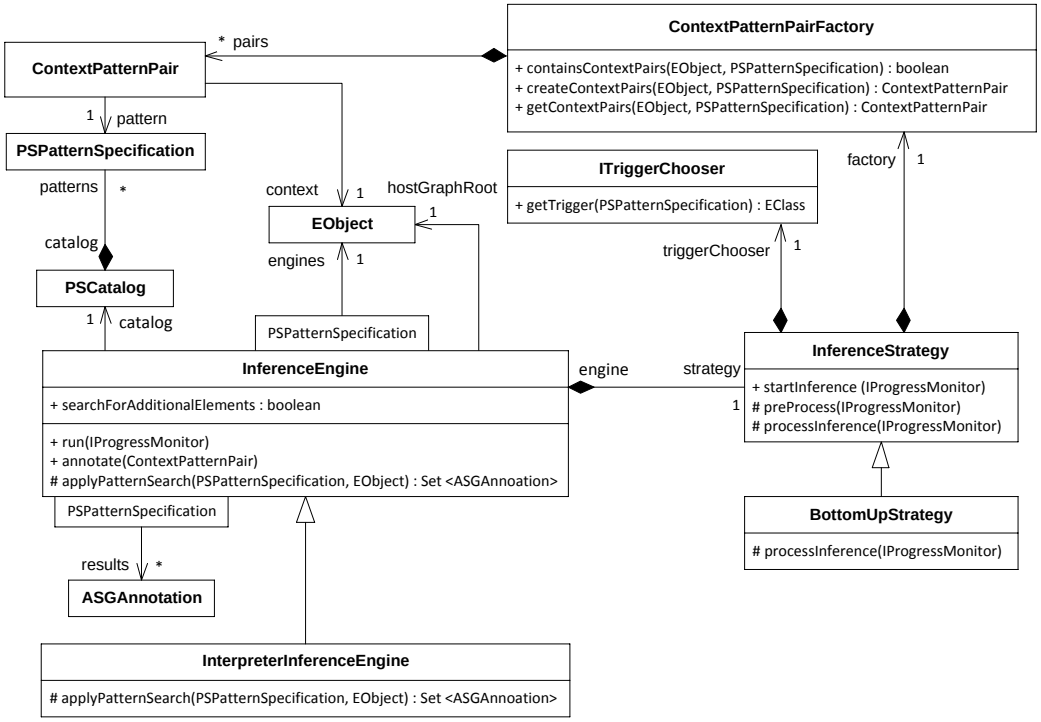

Abbildung 5.6: Ausschnitt eines Klassendiagramms des Inferenz-Mechanismus in Reclipse

Um eine Inferenz zu starten, muss die Klasse InterpreterInferenceEngine, die von

der abstrakten Klasse InferenceEngine erbt, instanziiert und konfiguriert werden. Zu den Konfigurationsparametern zählen ein Musterkatalog, ein Wirtsgraph, die generierten Suchalgorithmen und ein boolscher Parameter searchForAdditionalElements, der entscheidet, ob optionale Elemente des Musters im Rahmen der Mustersuche gesucht werden sollen. Der Musterkatalog wird über die Assoziation catalog der Klasse InferenceEngine zu PSCatalog referenziert. Die InferenceEngine referenziert das Wurzelobjekt eines Wirtsgraphen über die Assoziation hostGraphRoot zu EObject. Die generierten Suchalgorithmen werden über eine qualifizierte Assoziation engines zu EObject referenziert, wobei die Musterspezifikation (PSPatternSpecification) der Schlüssel zu einer Instanz der Engine-Klasse zu der Musterspezifikation ist.

In der Inferenz wurde das Entwurfsmuster *Strategy-Pattern* [GHJV95] angewendet, um den Einsatz verschiedener Strategien zur Implementierung des Ablaufs der Inferenz zu erlauben. Eine Strategie wird durch die abstrakte Klasse InferenceStrategy repräsentiert und stellt eine weitere Möglichkeit dar, die InferenceEngine zu konfigurieren. Spezialisierungen der Klasse InferenceStrategy, wie die Klasse BottomUpStrategy implementieren eine konkrete Strategie, indem sie die Methoden der Oberklasse überschreiben.

Die Klasse InferenceStrategy referenziert die Klasse ContextPatternPairFactory, welche Instanzen der Klasse ContextPatternPair erzeugt und verwaltet. Die Klasse ContextPatternPair repräsentiert ein Paar aus einem Muster und einem Kontextelement aus dem AST, welches den Anfang für ein Matching-Versuch eines bestimmten Musters repräsentiert. Außerdem referenziert die Klasse InferenceStrategy eine Klasse, die das Interface ITriggerChooser implementiert. Das Interface ITriggerChooser dient der Implementierung von Heuristiken, um geeignete Kontextelemente für ein ContextPatternPair und damit für einen Matching-Versuch in einem Muster zu identifizieren. Die Auswahl der Kontextelemente geschieht über ihren Typ. Als Faustregel gilt "je seltener ein bestimmter Typ im AST vorkommt, desto besser", da im Rahmen der Inferenz ein Matching-Versuch für jede Instanz des Typen durchgeführt wird und dadurch weniger Matching-Versuche unternommen werden müssen, um alle Möglichkeiten eines Matchings auszuprobieren.

Ablauf der Inferenz Der Ablauf der Inferenz in Reclipse wird in Abbildung 5.7 als Aktivitätendiagramm dargestellt und im Folgenden genauer erläutert.

Der Inferenz-Mechanismus in Reclipse wird durch einen Aufruf der Methode run(IProgressMonitor) der Klasse InferenceEngine gestartet. Diese führt diverse Initialisierungen, wie das Laden der benötigten Modelle und die Instanziierung der generierten Engine-Klassen durch und ruft anschließend die Methode startInference(IProgressMonitor) der referenzierten InferenceStrategy auf. Daraufhin werden mit Hilfe des ITriggerChoosers Paare von Kontext und Muster (ContextPatternPair) im Wirtsgraph ermittelt und nach und nach abgearbeitet. Die Abarbeitung der gebildeten Paare ist dabei Strategie-spezifisch.

Sobald die Strategie das nächste ContextPatternPair bestimmt hat, für welches

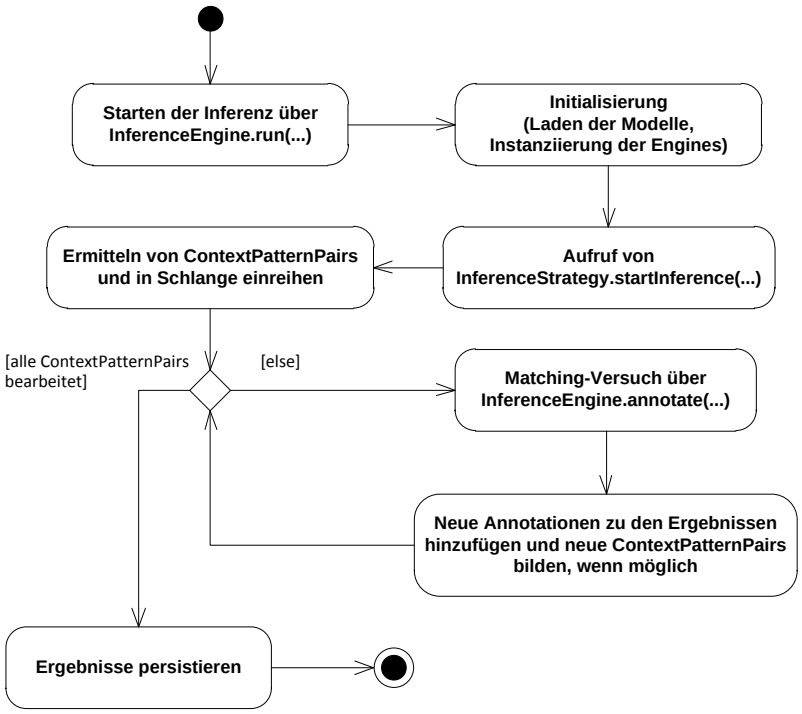

Abbildung 5.7: Ablauf der Inferenz als Aktivitätendiagramm

ein Matching-Versuch unternommen werden soll, ruft sie die Methode annotate(ContextPatternPair) der Klasse InferenceEngine mit dem ContextPatternPair als Parameter auf. Die Methode annotate(ContextPatternPair) leitet den Aufruf an die Methode applyPatternSearch(PSPatternSpecification, EObject) mit dem Muster und dem Kontextelement des ContextPatternPair-Elements als Parameter weiter.

Die Methode applyPatternSearch(PSPatternSpecification, EObject) ist abstrakt und muss von Unterklassen der Klasse InferenceEngine implementiert werden. Der Inferenz-Entwurf sieht vor, das die Methode die Ausführung der Suchalgorithmen kapselt. In der Klasse InterpreterInferenceEngine wird in der Methode applyPatternSearch(PSPatternSpecification, EObject) die Parametrisierung und der Aufruf des Story-Diagramm-Interpreters gekapselt.

Der Aufruf von applyPatternSearch(PSPatternSpecification, EObject) liefert Annotationen (ASGAnnotation) von Mustervorkommen im Wirtsgraph zurück. Diese werden in der Methode annotate(ContextPatternPair), die den Aufruf kapselt, über die qualifizierten Assoziation results mit der Musterspezifikation (PSPatternSpecification) als Schlüssel abgelegt.

Anschließend versucht die Strategie neue ContextPatternPair-Elemente auf Basis der annotierten Mustervorkommen zu ermitteln. Sobald alle ContextPatternPair-

Elemente abgearbeitet wurden, ist die Inferenz beendet und die Ergebnisse werden persistiert.

5.3.2 Inferenzerweiterung für die Analyse einzelner Komponenten

Eine Erweiterung der Inferenz musste im Rahmen dieser Arbeit durchgeführt werden, um die in Abschnitt 4.2.4 erläuterte Einschränkung der Inferenz auf eine manuelle Auswahl von Komponenten der rekonstruierten Softwarearchitektur umzusetzen. Die Auswahl der Komponenten findet über einen Eingabedialog statt. Dieser wird im Folgenden vorgestellt.

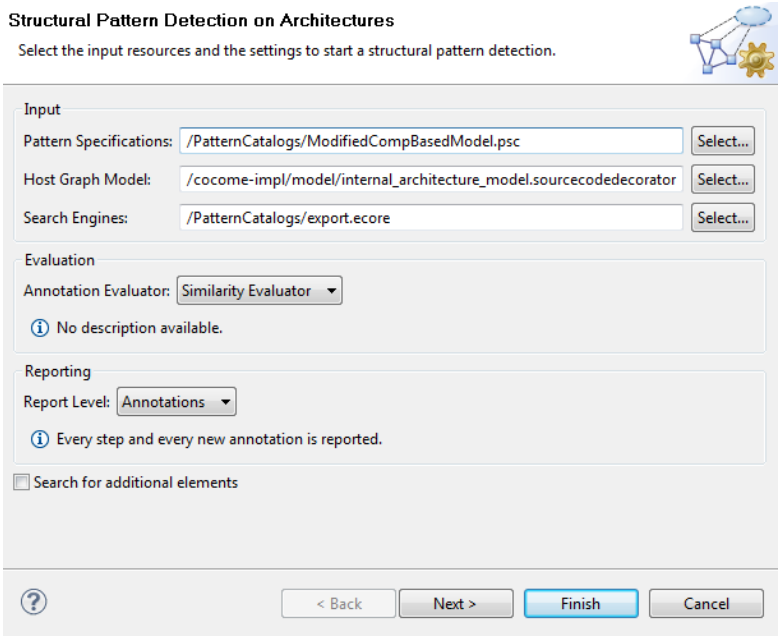

Abbildung 5.8: Eingabedialog der Inferenz: Selektion des Musterkatalogs, Wirtsgraphen und weitere Konfigurationsmöglichkeiten

Der Eingabedialog zum Starten der Inferenz kann über einen Menüeintrag (*Reclipse EMF -> Start Pattern Based Architecture Analysis*) erreicht werden. Die erste Seite des Eingabedialogs wird in Abbildung 5.8 dargestellt. Diese dient der Eingabe von für die Inferenz benötigten Pfaden, unter denen die Modelle des Musterkatalogs (*Pattern Specifications*) und des Wirtsgraphen (*Host Graph Model*) zu finden sind. Eine weiterer Pfad (*Search Engines*) legt fest, wo im Dateisystem die für einen Musterkatalog generierten Suchalgorithmen persistiert werden sollen. Eine Combobox *Annotation Evaluator* ermöglicht die Auswahl verschiedener Strategien zur Bewertung von Mustervorkommen. Eine weitere Combobox *Report Level* legt den Detailgrad von Konsolenausgaben während der Inferenz fest. Durch

die Checkbox *Search for additional elements* kann die Suche nach optionalen Teilen von Mustern im Rahmen der Inferenz deaktiviert werden und die Analyse dadurch gegebenenfalls beschleunigt werden.

Bei der Auswahl des Wirtsgraphen muss ein Source-Code-Decorator-Modell ausgewählt werden, das im Rahmen eines Clusterings in SoMoX erzeugt wird und die Zuordnung zwischen GAST- und SAMM-Modell enthält. Wurde ein Musterkatalog und ein Wirtsgraph ausgewählt, kann über den *Next*-Knopf zur nächsten Dialogseite navigiert werden, wo die Auswahl von zu analysierenden Komponenten der rekonstruierten Architektur vorgenommen werden kann. Die zweite Seite des Eingabedialogs wird in Abbildung 5.9 dargestellt und listet alle in einer rekonstruierten Softwarearchitektur enthaltenen Komponenten. Eine Checkbox links neben jeder Komponente legt fest, ob die Komponente von der Mustersuche berücksichtigt werden soll.

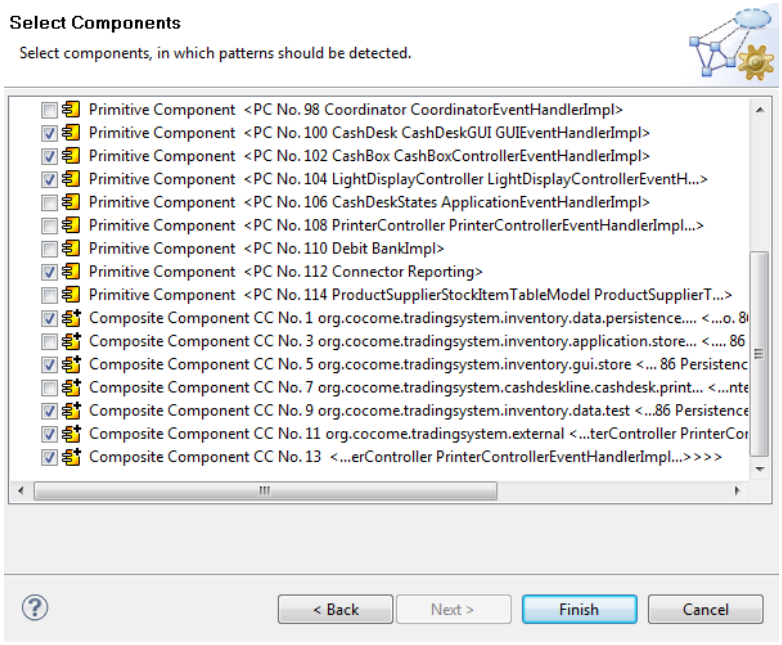

Abbildung 5.9: Eingabedialog der Inferenz: Selektion von zu analysierenden Komponenten

Nach dem die zu analysierenden Komponenten ausgewählt wurden, wird die Konfiguration der Inferenz durch das Drücken des *Finish*-Knopfs beendet. Die id-Attribute der ausgewählten Komponenten werden dann, wie in Abschnitt 4.2.4 beschrieben, dazu verwendet einen regulären Ausdruck zu bilden, der lediglich beim Matching eines id-Werts der ausgewählten Komponenten erfolgreich ausgewertet werden kann. Eine Attributbedingung mit dem regulären Ausdruck wird dann in dem ausgewählten Musterkatalog in den Mustern *DirectComponentClasses* und *IndirectComponentClasses* jeweils dem Objekt comp vom Typ ComponentType hinzugefügt (vergl. Abs. 4.2.3). Anschließend werden die Suchalgorithmen

aus dem modifizierten Musterkatalog generiert und an dem zuvor angegeben Pfad persistiert. Ist dieser Schritt abgeschlossen, wird die Inferenz unter Verwendung der generierten Suchalgorithmen gestartet.

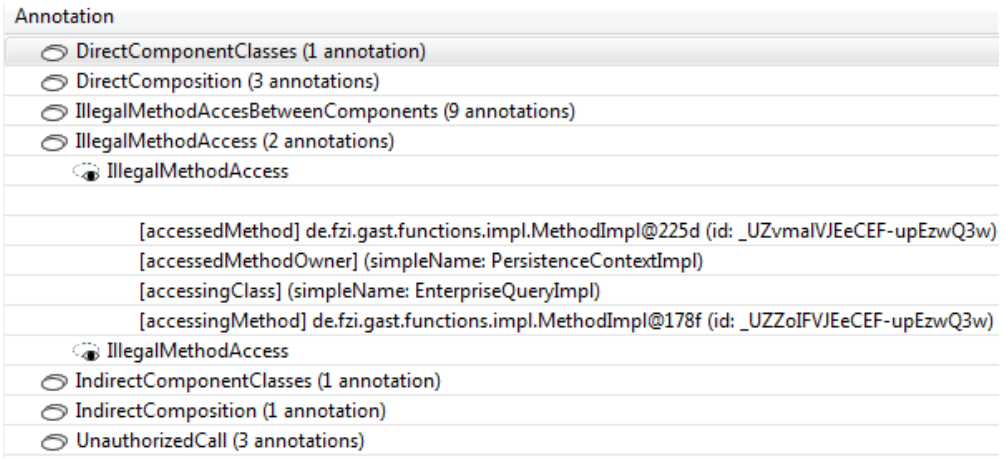

Abbildung 5.10: Beispiel für die Darstellung von Mustervorkommen

Nach Beenden des Inferenz-Durchlaufs werden detektierte Mustervorkommen in einer Tabellenansicht präsentiert. Ein Beispiel für eine solche Ergebnistabelle wurde in Abbildung 5.10 dargestellt. Die Tabelle liefert eine Übersicht und die Anzahl von detektierten Mustervorkommen. Einzelne Mustervorkommen können im Detail betrachtet werden, indem sie aufgeklappt werden. Durch das Aufklappen werden annotierte Objekte eines Mustervorkommens hinter der Rolle, die sie im Muster einnehmen, dargestellt.

6 Gesammelte Erfahrungen

Um die Kombination von Clustering und Mustersuche, die in Kapitel 4 vorgestellt wurde, an einem Beispiel zu validieren und Erfahrungswerte zu sammeln, wurde die Referenzimplementierung des komponentenbasierten Softwaresystems *Common Component Modelling Example* (CoCoME) [coc10] mit dem vorgestellten Ansatz analysiert. Die Ergebnisse der Analyse und die gesammelten Erfahrungen werden in diesem Kapitel dokumentiert.

In dem folgenden Abschnitt wird die Rekonstruktion der Softwarearchitektur von CoCoME erläutert. Die Ergebnisse einer Suche nach den in Abschnitt 4.3 vorgestellten Schwachstellenmustern auf Basis der rekonstruierten Architektur werden in Abschnitt 6.2 erläutert. Wegen technischer Einschränkungen mussten einige der Musterspezifikationen verändert werden, um bessere Ergebnisse zu erzielen. Diese Anpassungen und die in Folge der Anpassungen erzielten Ergebnisse werden in Abschnitt 6.3 vorgestellt. Abschließend werden die gesammelten Erfahrungen mit dem kombinierten Reverse-Engineering-Ansatz in Abschnitt 6.4 diskutiert.

6.1 Clustering von CoCoME

CoCoME ist ein Modellierungsbeispiel für ein verteiltes komponentenbasiertes Shop-System. Eine Implementierung des Shop-Systems mit einem Umfang von ca. 5.000 LOC (Lines of Code) entstand inklusive einer ausführlichen Dokumentation [HKW+08] im Rahmen eines Wettbewerbs[1] als Referenzbeispiel für gute komponentenbasierte Software-Entwicklung.

Die rekonstruierte Softwarearchitektur eines Systems ist Grundlage der Schwachstellensuche in dem kombinierten Reverse-Engineering-Ansatz. Da die rekonstruierte Architektur abhängig von den für das Clustering gewählten Metrikgewichten ist, wurden zwei Clusterings mit unterschiedlichen Metrikgewichten durchgeführt. Bei den für das Clustering verwendeten Metrikgewichten handelt es sich zum einen um empirisch erprobte Metrikgewichte (Clustering 1) [Kro10] und zum anderen um Metrikgewichte, die in dieser Arbeit speziell für das Clustering von CoCoME angepasst wurden (Clustering 2). Um die Ergebnisse reproduzieren zu können, wurde im Anhang C eine Tabelle mit den für das Clustering in SoMoX gewählten Metrikgewichten hinterlegt. Die erste rekonstruierte Softwarearchitektur enthält nur wenige Kompositionsbeziehungen zwischen Komponenten. Die zweite rekonstruierte Architektur enthält lediglich eine Komponente, die in keiner Komposi-

[1] http://www.cocome.org

tionsbeziehung zu einer anderen Komponente steht und ist daher stark hierarchisiert.

Aus Platzgründen kann die rekonstruierte Softwarearchitektur hier nicht komplett, sondern lediglich ein Ausschnitt davon, dargestellt werden. Dieser Ausschnitt wird im Weiteren untersucht, um die in Abschnitt 4.2.4 beschriebene Einschränkung der Inferenz auf einzelne Komponenten zu demonstrieren. Bei dem Entwurf von CoCoME wurde das *Model-View-Controller*-Architekturmuster (MVC) [Ree79] angewendet. Dieses sieht eine Trennung des Systems in die drei Schichten vor. Die *View*-Schicht enthält die Benutzerschnittstelle, die *Controller*-Schicht implementiert die Logik des Systems und die *Model*-Schicht dient der Datenhaltung. Der Ausschnitt der konzeptionellen Architektur von CoCoME in Abbildung 6.1 zeigt die Komponente TradingSystem::Inventory::Data, welche zur *Model*-Schicht gehört und eine Anbindung von CoCoME an eine Datenbank realisiert. In Abbildung 6.2 wird ein Ausschnitt des gleichen Teils der in Clustering 1 rekonstruierten Architektur dargestellt.

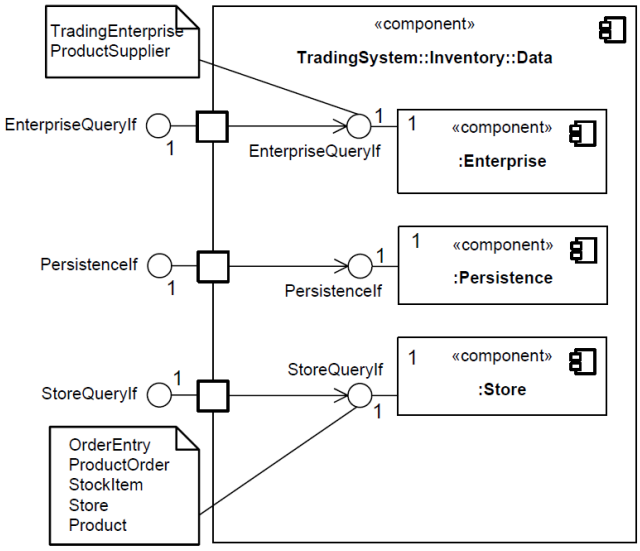

Abbildung 6.1: Ausschnitt der dokumentierten, konzeptionellen Architektur der CoCoME Referenzimplementierung [HKW+08]

Die Komponente TradingSystem::Inventory::Data ist aus den drei Subkomponenten Enterprise, Persistence und Store zusammengesetzt und bietet deren Schnittstellen EnterpriseQueryIf, PersitencesIf und StoryQueryIf nach außen anderen Komponenten an. In den Notizen an den Schnittstellen werden Datentypen gelistet, die bei der Kommunikation über die Schnittstellen verwendet werden. Die in Clustering 1 rekonstruierte Variante in Abbildung 6.2 des gleichen Teils unterscheidet sich sowohl in der Anzahl einzelner Komponenten, als auch in der Art der Zusammensetzung. In der rekonstruierten Architektur wird die konzeptionelle Komponente Persistence durch die zusammengesetzte Komponente CC No1 repräsentiert,

die aus zwei Subkomponenten PersistenceContextImpl und TransactionContextImpl besteht. Die Komponente PersistenceContextImpl verwendet dabei, die Schnittstelle TransactionContext der Komponente TransactionContextImpl. Im Rahmen des Clusterings wird dies als Indiz für eine Kompositionsbeziehung zwischen den beteiligten Komponenten gewertet (vergl. Abs. 2.3.2). Die Komponente CC No3 wurde aus den Komponenten EnterpriseQueryImpl und CC No1 zusammengesetzt. Hier ist die Verwendung der angebotenen Schnittstelle PersistenceContext vermutlich der Grund für die Komposition der beiden Komponenten. Obwohl die Komponente StoreQueryImpl ebenfalls die Schnittstelle PersistenceContext verwendet, wurde im Rahmen des Clusterings keine weitere zusammengesetzte Komponente erzeugt, da mehrere Interface Violations zwischen den Komponenten vorliegen und diese damit vermutlich den Wert der Strategien *InterfaceAdherence* und dadurch auch *ComponentComposition* verringert haben (siehe Abs. 2.3.2).

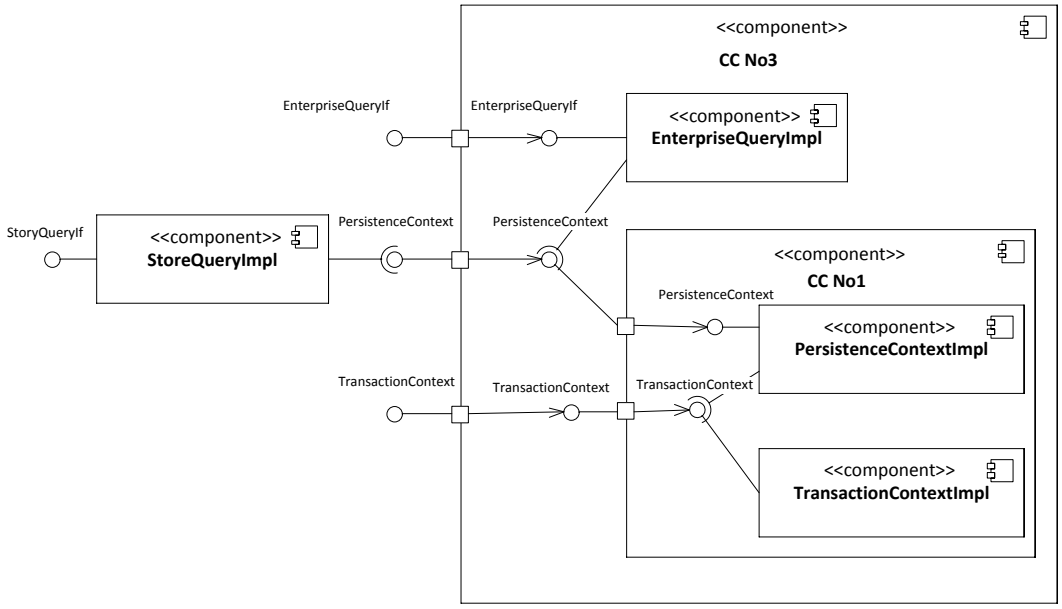

Abbildung 6.2: Ausschnitt der rekonstruierten Architektur der CoCoME Referenzimplementierung (Clustering 1)

Die konzeptionelle Komponente Persistence wird im Quellcode neben den erwähnten Klassen auch durch die Klasse PersistenceImpl und ihr Interface PersistenceIf repräsentiert. Unter der Verwendung der zweiten Konfiguration von SoMoX (siehe Anhang C, Clustering 2) konnte die konzeptionelle Komponente TradingSystem::Inventory::Data nahezu wie in Abbildung 6.1 dargestellt rekonstruiert werden. Lediglich ihre Schnittstelle PersistenceIf und die Klasse PersistenceImpl wurden der Komponente nicht zugeordnet. In keiner der rekonstruierten Architekturen

wurde die Klasse PersistenceImpl und ihr Interface PersitenceIf einer Komponente zugeordnet. Dieses Verhalten konnte im Rahmen der durchgeführten Experimente auch für einige andere Klassen in der Implementierung von CoCoME festgestellt werden. Um den Grund für die unvollständige Zuordnung zu identifizieren, müsste die Berechnung von Metrikwerten in SoMoX genauer untersucht werden, was jedoch im Rahmen dieser Arbeit nicht mehr möglich war. Bei der Suche nach Schwachstellen könnte sich die fehlende Zuordnung von Klassen zu Komponenten insofern negativ auf die Ergebnisse ausgewirkt haben, als dass einige Schwachstellen übersehen wurden.

6.2 Erste Ergebnisse der Mustersuche

In einem ersten Versuch, die in Abschnitt 4.3 beschriebenen Schwachstellenmuster in der ersten Variante der rekonstruierten Softwarearchitektur von CoCoME zu detektieren, wurden die in Tabelle 6.1 gelistete Anzahl von Mustervorkommen ermittelt. Die in Abschnitt 4.2.3 vorgestellten Korrespondenzmuster werden oberhalb der Schwachstellenmuster durch eine doppelte Linie getrennt gelistet.

Musterspezifikation	Clustering 1	Clustering 2
DirectComposition	3	7
IndirectComposition	1	5
DirectComponentClasses	24	19
IndirectComponentClasses	3	7
InheritanceBetweenComponents	72	72
IllegalMethodAccess	496	496
IllegalVariableAccess	105	105
NonToCommunication	-	-
NonToCommunication2	-	-
UnauthorizedCall	612	2222
UndercoverTransferObject	-	-

Tabelle 6.1: Vorkommen der in Kapitel 4 beschriebenen Muster in der Referenzimplementierung von CoCoME

Die Korrespondenzmuster *DirectComposition, IndirectComposition, DirectComponentClasses* und *IndirectComponentClasses* wurden alle entsprechend der rekonstruierten Architektur korrekt detektiert. Für alle detektierten Schwachstellenmuster ist die Anzahl der Mustervorkommen sehr hoch. Eine Unterscheidung nach True und False Positives wurde wegen der hohen Anzahl von Mustervorkommen nur stichprobenartig durchgeführt. Diese ergab, dass nahezu alle der Mustervorkommen als False Positives anzusehen sind. Von einigen Schwachstellenmustern konnten im Rahmen des ersten Analyseversuchs keine Vorkommen detektiert werden, obwohl diese durch manuelle Code-Inspektion nachgewiesen werden konnten.

Die Gründe für diese nicht zufriedenstellenden Ergebnisse sind diverse, teilweise technische Einschränkungen. Um diese zu umgehen, wurden die Schwachstellenmuster verändert. Die einzelnen Einschränkungen und die daraus resultierenden Anpassungen der Musterspezifikationen werden im Folgenden einzeln erläutert. Die angepassten Schwachstellenmuster wurden im Anhang A hinterlegt.

Einträge aus Bibliotheken im GAST-Modell Das GAST-Modell, das durch den Parser SISSy erstellt wird, enthält Einträge für Klassen, die aus Bibliotheken wie zum Beispiel java.awt oder java.io stammen. Speziell die Schwachstellenmuster *IllegalMethodAccess* und *IllegalVariableAccess* wurden unter der Annahme spezifiziert, dass solche Einträge im GAST-Modell gefiltert werden können und daher das GAST-Modell nur Einträge für Klassen enthält, die auch im Fokus der Analyse stehen. Eine Filterung solcher Klassen ist von SISSy nicht vorgesehen und muss daher durch den Einsatz von Attributbedingungen in den Musterspezifikationen vorgenommen werden.

In einem durch SISSy erzeugten GAST-Modell haben Elemente ein Attribut status mit dem Wert NORMAL, wenn es sich um ein Element handelt, das aus Quellcode eingelesen wurde. Elemente aus Bibliotheken erhalten den Wert LIBRARY. In den Schwachstellenmustern wurde daher allen Objekten vom Typ GASTClass eine Attributbedingung mit dem Wert NORMAL für das status-Attribut hinzugefügt. Eine Ausnahme bilden GASTClass-Objekte, die in Beziehung mit Elementen aus dem Source-Code-Decorator-Modell stehen, da diese bereits von SoMoX als Elemente der Softwarearchitektur identifiziert wurden.

Matching von unidirektionalen Links Ein Grund dafür, dass keine Vorkommen der Schwachstellenmuster *NonToCommunication, NonToCommunication2* und im Gegensatz dazu viele von *InheritanceBetweenComponents* und *UnauthorizedCall* erkannt werden, ist, dass der Story-Diagramm-Interpreter das Matching von unidirektionalen Links zum Zeitpunkt der Abgabe dieser Arbeit nur eingeschränkt unterstützt. Ein solcher Link kann im Rahmen eines Matchings mit dem Story-Diagramm-Interpreter nur dann erkannt werden, wenn das Objekt, von dem der Link ausgeht, bereits bekannt ist. Von einem durch den Link referenzierten Objekt aus kann das referenzierende Objekt hingegen nicht erreicht werden. Der Link kann also nicht „rückwärts“ traversiert werden. Ein Muster kann damit nur dann erkannt werden, wenn von einem Objekt, bei dem das Matching beginnt, zu allen anderen Objekten des Musters ein gerichteter Pfad existiert. Alle vier der oben genannten Schwachstellenmuster enthalten mindestens einen unidirektionalen Link zu einem Element, das nur durch das Traversieren eines unidirektionalen Links in umgekehrter Richtung erreicht werden kann und damit diese Bedingung verletzt.

Um dennoch Vorkommen dieser Schwachstellen detektieren zu können, wurden die Schwachstellenmuster derart abgeändert, dass ein solcher gerichteter Pfad zu allen anderen Objekten des Musters zumindest von einem Objekt in der Musterspezifikation existiert. Die beiden Varianten des Schwachstellenmusters *NonTo-*

Communication und *NonToCommunication2* wurden dabei zu einer Musterspezifikation zusammengelegt, da diese sich lediglich in dem Typ eines Objekts unterscheiden (vergl. Abs. 4.3). In beiden Fällen kann das Objekt jedoch auf Grund der Einschränkung des Interpreters im Rahmen eines Matchings nicht erreicht werden, weswegen es in der Musterspezifikation entfernt wurde. Stattdessen wurden Attributbedingungen hinzugefügt, welche die Anzahl von False Positives, die auf Grund des Entfernens des Objekts zu erwarten ist, reduzieren sollen.

Die beiden Muster *InheritanceBetweenComponents* und *UnauthorizedCall* enthalten jeweils ein negatives Fragment, dessen Objekte auf Grund von unidirektionalen Links beim Matching nicht erreicht werden kann. Da damit das Matching eines negativen Fragments fehlschlägt und dies eine Bedingung für ein erfolgreiches Matching des gesamten Musters ist, werden Teile des Wirtsgraphen annotiert, die nicht annotiert werden sollten. Die Muster wurden derart modifiziert, dass der Inhalt eines negativen Fragments erreicht werden kann.

Fallunterscheidung für Interface Violations Die hohe Anzahl der Mustervorkommen von *IllegalMethodAccess* und *IllegalVariableAccess* war Anlass dazu, die Vorkommen danach zu unterscheiden, ob eine Interface-Violation innerhalb einer Komponente oder zwischen zwei unterschiedlichen Komponenten besteht. Von beiden Schwachstellenmustern wurden deswegen zwei Varianten spezifiziert. In dem folgenden Abschnitt repräsentieren die Schwachstellenmuster *IllegalMethodAccess* und *IllegalVariableAccess* daher eine Interface-Violation innerhalb einer Komponente. Eine entsprechende Interface-Violation zwischen zwei verschiedenen Komponenten wird durch die Muster *IllegalMethodAccessBetweenComponents* und *IllegalVariableAccessBetweenComponents* repräsentiert.

Eines der Ziele bei der Suche nach Interface Violations ist es, herauszufinden, ob diese zu einer Vereinigung von möglicherweise konzeptionell verschiedenen Komponenten geführt haben. Das Clustering in SoMoX verwendet die *ComponentMerge*-Strategie jedoch lediglich bei der Ermittlung von primitiven Komponenten. Die Schwachstellenmuster *IllegalMethodAccess* und *IllegalVariableAccess* wurden deswegen auf die Suche in primitiven Komponenten eingeschränkt. Alle anderen Vorkommen von Interface Violations werden durch die Muster *IllegalMethodAccessBetweenComponents* und *IllegalVariableAccessBetweenComponents* abgedeckt.

Interface Violations auf Grund von Enumerations Bei allen Vorkommen von *IllegalVariableAccess* wurde festgestellt, dass es sich dabei um Zugriffe auf Enumeration-Typen handelt. Diese werden in einem GAST-Modell auch durch Elemente vom Typ GASTClass repräsentiert, weshalb die Zugriffe im Rahmen der Mustersuche annotiert wurden. Enumerations sorgen für Kopplung im System und sollten daher nur innerhalb der Komponente verwendet werden, in der sie definiert werden. Wenn sie von mehreren Komponenten verwendet werden, sollten sie vom Clustering ausgeschlossen werden, da ansonsten Metrikwerte für die Kopplung

von Enumerations beeinflusst werden. Um ihre Verwendung innerhalb einer Komponente nicht als Vorkommen von *IllegalVariableAccess* zu annotieren, wurde die Musterspezifikation um eine Attributbedingung zur Filterung solcher Vorkommen erweitert. Alle Enumerations in einem GAST-Modell können von einer gewöhnlichen Klasse lediglich durch das Attribut final unterschieden werden. Im Falle einer Enumeration hat das Attribut den Wert true, weshalb die hinzugefügte Attributbedingung den Wert false verlangt.

6.3 Ergebnisse nach Anpassung der Musterspezifikationen

Die Ergebnisse eines zweiten Analyseversuchs, nachdem die im letzten Abschnitt beschriebenen Anpassungen an den Musterspezifikationen durchgeführt wurden, sind in Tabelle 6.2 gelistet. Eine Analyse der in Abschnitt 6.1 vorgestellten Data-Komponente wurde isoliert von dem Rest des CoCoME-Systems durchgeführt und die Ergebnisse der Tabelle hinzugefügt. Die Anzahl der True Positives wurde bei diesem Analyseversuch durch manuelle Code-Inspektion ausgewertet und jeweils durch einen Schrägstrich getrennt hinter der Anzahl der Mustervorkommen notiert.

	Clustering 1		Clustering 2	
	Data	gesamt	Data	gesamt
Musterspezifikation	P/TP	P/TP	P/TP	P/TP
DirectComposition	3/3	3/3	7/7	7/7
IndirectComposition	1/1	1/1	5/5	5/5
DirectComponentClasses	4/4	24/24	3/3	19/19
IndirectComponentClasses	2/2	3/3	1/1	7/7
InheritanceBetweenComponents	-	-	-	-
IllegalMethodAccess	-	30/-	-	30/-
IllegalMethodAccessBetweenComponents	11/11	16/14	11/11	16/14
IllegalVariableAccess	-	-	-	-
IllegalVariableAccessBetweenComponents	-	52/52	-	32/32
NonToCommunication	-	2/2	-	23/23
UnauthorizedCall	4/2	65/2	4/2	65/2
UndercoverTransferObject	-	-	-	-
Zeit für die Mustersuche:	79s	219s	82s	627s

P: Positive, TP: True Positive

Tabelle 6.2: Vorkommen der modifizierten Muster in den Referenzimplementierung von CoCoME

Ergebnisse der Analyse der Data-Komponente Die Schwachstellensuche in der in Abschnitt 6.1 beschriebenen Komponente Data der rekonstruierten Architektur ergab insgesamt elf Vorkommen von Interface Violations, die auch durch manuelle Code-Inspektion nachgewiesen werden konnten. Neun der Interface Violations wurden in der Komponente StoreQueryImpl erkannt, die mehrfach die Schnittstelle von PersistenceContextImpl verletzt. Die anderen beiden Interface Violations konnten in EnterpriseQueryImpl erkannt werden, welche dieselbe Schnittstelle verletzt.

Die Vorkommen von *UnauthorizedCall* sind bis auf zwei Vorkommen, die auch als Interface-Violation erkannt wurden, als False Positives bewertet worden, weil die annotierten Klassen laut Dokumentation [HKW+08] als Datentypen in der Schnittstelle der untersuchten Komponente verwendet werden (vergl. Notizen in Abb. 6.1). Da Datentypen, die in einer Schnittstelle verwendet werden, selbst nicht Teil der Schnittstelle sind, wurde der Zugriff auf diese Datentypen als *UnauthorizedCall* detektiert. Dieser Zusammenhang wird von dem Schwachstellenmuster jedoch nicht berücksichtigt und muss daher in zukünftigen Arbeiten ergänzt werden.

Die in der Data-Komponente detektierten Mustervorkommen sind für beide Clusterings (1 und 2) identisch. Trotz der Unterschiede in der rekonstruierten Softwarearchitektur konnten somit keine Auswirkungen des Clusterings auf die Analyseergebnisse festgestellt werden.

Ergebnisse der Analyse des gesamten Systems Eine Schwachstellensuche, welche die gesamte Referenzimplementierung von CoCoME berücksichtigt, deckte zum einen weitere Schwachstellenvorkommen auf, zum anderen aber auch Defizite einiger Schwachstellenmuster.

Weder durch die Mustersuche, noch durch eine manuelle Code-Inspektion konnte eine Vererbung über Komponentengrenzen hinweg identifiziert werden, weshalb das nicht Detektieren von *InheritanceBetweenComponents* keine False Negatives darstellt und als korrekt angesehen wird.

Bei allen Vorkommen von *IllegalMethodAccess* handelt es sich jeweils um Interface Violations innerhalb einer konzeptionellen Komponente. Die Vorkommen wurden daher alle als False Positives bewertet. Die vielen False Positives überraschen jedoch nicht. Die Einhaltung von Schnittstellen wird in einem komponentenbasierten Softwaresystem lediglich zwischen Komponenten gefordert und nicht innerhalb einer Komponente. Deswegen sind Kopplung und Interface Violations zwei der wichtigsten Indizien für die Zuordnung von Klassen zu einer Komponente im Rahmen des Clusterings. In Zukunft sollte versucht werden, die Anzahl der False Positives für das Muster *IllegalMethodAccess* durch die Spezifikation weiterer Eigenschaften der Schwachstelle einzuschränken. Es könnte beispielsweise das Verhältnis von Zugriffen über Interfaces zu Zugriffen, die nicht über ein Interface stattfinden, berücksichtigt werden, da anzunehmen ist, dass bei konzeptionell verschiedenen Komponenten die meisten oder zumindest einige Zugriffe über Interfaces stattfinden.

Unter den Vorkommen von *IllegalMethodAccessBetweenComponents* sieht das Verhältnis von True und False Positives besser aus, da Interface Violations zwischen Komponenten nicht erlaubt sind. Bei den meisten davon handelt es sich um die bereits bei der Analyse der Data-Komponente detektierten Interface Violations zwischen PersistenceContextImpl, StoreQueryImpl und EnterpriseQueryImpl. Die beiden als False Positive bewerteten Vorkommen sind auf die in Clustering 1 und 2 rekonstruierte Softwarearchitektur zurückzuführen, welche mit der konzeptionellen Architektur nicht übereinstimmt.

Durch manuelle Code-Inspektion konnten bis auf die Zugriffe auf Enumeration-Typen keine Zugriffe auf Variablen zwischen Klassen identifiziert werden, die als Vorkommen von *IllegalVariableAcces* oder *IllegalVariabelAccessBetweenComponents* hätten annotiert werden müssen. Da die Zugriffe auf Enumerations innerhalb einer Komponente erlaubt sind, ist das nicht Detektieren von *IllegalVariableAccess* korrekt. Die Vorkommen von *IllegalVariabelAccessBetweenComponents* wurden alle als True Positive bewertet, obwohl es sich dabei nicht wirklich um einen Zugriff auf Variablen zwischen Komponenten handelt. Enumerations sind lediglich Datentypen und sollten daher vom Clustering ausgeschlossen werden. Durch ihre Verwendung wird jedoch Kopplung verursacht, wodurch das Clustering beeinflusst wird. Die Mustervorkommen dienen daher einem Reengineer als Hinweis, Enumerations von dem Clustering auszuschließen. Die durch das Clustering 1 rekonstruierten Komponenten sind weniger abstrakt als die durch das Clustering 2 rekonstruierten Komponenten. Weil dadurch mehr Zugriffe über Komponentengrenzen hinweg stattfinden, konnten in der von Clustering 1 rekonstruierten Architektur auch mehr Vorkommen von *IllegalVariableAccessBetweenComponents* detektiert werden.

An den Vorkommen des Schwachstellenmusters *NonToCommunication* ist der Einfluss des Clusterings auf die Analyse sichtbar. In der durch das Clustering 1 rekonstruierten Architektur konnten lediglich zwei Vorkommen des Musters detektiert werden. In der von Clustering 2 rekonstruierten Architektur wurden hingegen 23 Vorkommen detektiert. Bei allen Vorkommen handelt es sich nicht um Transferobjekte, jedoch teilweise um Klassen mit ähnlichen Eigenschaften, wie zum Beispiel Klassen zum Versenden von Ereignis-Benachrichtigungen.

Bei der Analyse des gesamten Systems wurden unabhängig vom Clusterings 65 Vorkommen von *UnauthorizedCall* detektiert. Bis auf die zwei Vorkommen, die auch bei der Analyse der Data-Komponente als True Positive bewertet wurden, wurden alle als False Positive bewertet. Wie bei den False Positives der Data-Komponente, handelt es sich um Zugriffe auf Datentypen, die zur Kommunikation über die Schnittstellen der betroffenen Komponenten verwendet werden.

Vorkommen von *UndercoverTransferObject* wurden im Rahmen der musterbasierten Schwachstellensuche nicht detektiert. In einer manuellen Code-Inspektion wurden jedoch Klassen identifiziert, wie zum Beispiel Product, die als Vorkommen von *UndercoverTransferObject* hätten annotiert werden müssen. In beiden Clusterings wurden diese jedoch korrekt als Datentyp erkannt und daher keiner Komponente zugeordnet, weshalb auch keine Vorkommen von *UndercoverTrans-*

ferObject erkannt wurden.

Architektur nach Beseitigung der Interface Violations Um zu validieren, ob sich die identifizierten Schwachstellen tatsächlich auf die rekonstruierte Architektur von CoCoME auswirken, wurden die identifizierten Interface Violations in CoCoME beseitigt. Anschließend wurde die Softwarearchitektur noch einmal mit den selben Metrikgewichten wie zuvor in Clustering 1 und 2 rekonstruiert.

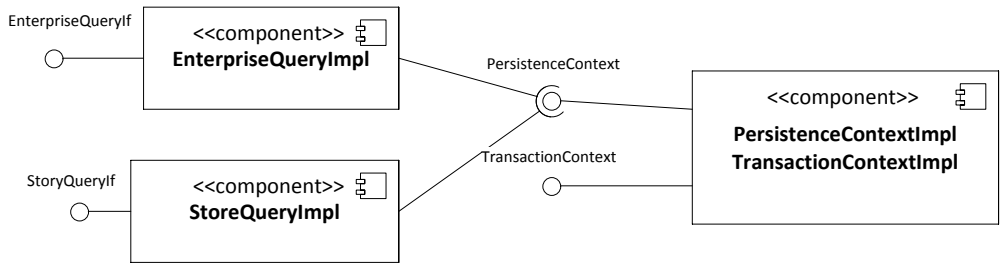

Abbildung 6.3: Ausschnitt der rekonstruierten Architektur der CoCoME Referenzimplementierung (Clustering 1) nach Beseitigung der Interface Violations

Die Abbildung 6.3 zeigt den bereits in Abschnitt 6.1 vorgestellten Ausschnitt der Architektur von CoCoMe, der in Clustering 1 nach Beseitigung der Interface Violations rekonstruiert wurde. Entgegen der Grundannahme dieser Arbeit, dass die Beseitigung von Schwachstellen in einem System die konkrete Architektur an die konzeptionelle Architektur annähert, hat sich die rekonstruierte Architektur von der konzeptionellen scheinbar entfernt. Obwohl PersistenceContextImpl und TransactionContextImpl nun, anders als zuvor, nur noch eine Komponente realisieren, stehen die drei rekonstruierten Komponenten in keiner Kompositionsbeziehung zueinander. Weil die gesamte in Clustering 1 rekonstruierte Architektur lediglich eine Kompositionsbeziehung und damit so gut wie keine Hierarchisierung aufweist, kann angenommen werden, dass die Wahl der Metrikgewichte in Clustering 1 nicht geschickt war und damit hätte angepasst werden müssen.

Der Ausschnitt des gleichen Teils der Architektur, der in Clustering 2 rekonstruiert wurde, wird im Anhang C in Abbildung C.2 dargestellt. Auch in Clustering 2 konnte eine Veränderung der rekonstruierten Architektur festgestellt werden. Die Komponente Data wurde dieses mal zusammen mit anderen Komponenten aus der *Controller*-Schicht der MVC-Architektur von CoCoME in einer zusammengesetzten Komponente rekonstruiert. Eine andere Wahl von Metrikgewichten hätte auch in diesem Fall vermutlich zu einer besseren rekonstruierten Architektur geführt.

Das Experiment zeigt, dass die identifizierten Schwachstellen eine Auswirkung auf die rekonstruierte Architektur haben und bestätigt damit eine der Grund-

annahmen dieser Arbeit. Die Annahme, dass sich die konkrete Architektur durch Beseitigung von Schwachstellen an die konzeptionelle Architektur annähert, konnte weder bestätigt, noch widerlegt werden. Um darüber eine Aussage treffen zu können, hätten die beim Clustering verwendeten Metrikgewichte noch einmal angepasst werden müssen, was jedoch im Rahmen dieser Arbeit nicht mehr möglich war.

6.4 Schlussfolgerung

Die Analyse der Referenzimplementierung von CoCoME zeigt, dass selbst in einem als gut erachteten, komponentenbasiert entwickelten Softwaresystem noch Schwachstellen enthalten sein können und sich diese auf die Architektur auswirken. Einige dieser Schwachstellen konnten mit dem vorgestellten kombinierten Reverse-Engineering-Ansatz trotz technischer Einschränkungen aufgedeckt werden.

Die Validierung an zwei unterschiedlichen Clusterings hat gezeigt, dass die Analyseergebnisse des kombinierten Ansatzes von der rekonstruierten Softwarearchitektur beeinflusst werden. Die rekonstruierte und die konzeptionelle Architektur stimmen jedoch nur selten überein [KSB+11]. Wenn diese sich stark von einander unterscheiden, kann dies dazu führen, dass Schwachstellen von der Mustersuche nicht detektiert werden. Eine Verbesserung des Clusterings könnte daher in Zukunft auch zu besseren Ergebnissen des vorgestellten Reverse-Engineering-Ansatzes führen, indem weniger Schwachstellen verborgen bleiben.

Die teilweise vielen False Positives zeigen, dass einige der Schwachstellenmuster noch weiter verfeinert werden müssen. In den verwendeten Musterspezifikationen wurden bisher keine optionalen Teile von Schwachstellen modelliert. Die Erweiterung der Schwachstellenmuster um optionale Teile könnte in Zukunft die Muster verfeinern und in Kombination mit einer automatisierten Bewertung [Tra06] die Auswertung von Mustervorkommen vereinfachen.

Bei der Auswertung von Mustervorkommen wäre es für einen Reengineer auch interessant zu erfahren, inwiefern sich ein Schwachstellenvorkommen auf das Clustering auswirkt. Um solche Auswirkungen nachvollziehen zu können, müssten die Entscheidungen während des Clusterings protokolliert werden und die Abwesenheit einer Schwachstelle simuliert werden, um einen Vergleich der Architekturen mit und ohne Schwachstelle zu ermöglichen.

Die Möglichkeit, einzelne Komponenten eines Softwaresystems getrennt vom Rest des Systems zu untersuchen, konnte dazu genutzt werden, sich ein genaues Bild über ausgewählte Komponenten zu verschaffen, ohne durch Mustervorkommen aus anderen Komponenten des Systems bei der Auswertung behindert zu werden. Außerdem konnte die Analyse dadurch deutlich beschleunigt werden.

Der sich noch in einem frühen Entwicklungsstadium befindende Story-Diagramm-Interpreter unterstützt momentan noch nicht das Matching von unidirektionalen Links entgegen ihrer Richtung. Diese Einschränkung konnte bei den verwendeten

Schwachstellenmustern durch ihre Anpassung umgangen werden. Andere Muster sind jedoch denkbar, bei denen dies nicht möglich ist, weshalb die noch fehlende Funktionalität des Interpreters auch den in dieser Arbeit vorgestellten Ansatz noch einschränkt.

7 Verwandte Arbeiten

Dieses Kapitel widmet sich der Abgrenzung dieser Arbeit zu verwandten Arbeiten. Im Folgenden werden zunächst einige Clustering-basierte und musterbasierte Reverse-Engineering-Ansätze diskutiert. Abschließend werden andere Ansätze, die eine Kombination von Clustering und Mustersuche realisieren, vorgestellt.

7.1 Clustering-basiertes Reverse Engineering

In der Literatur sind viele Clustering-basierte Reverse-Engineering-Ansätze zu finden. Ducasse und Pollet [DP09], sowie Koschke [Kos05] liefern einen Überblick über existierende Varianten des Clustering-basierten Reverse Engineerings. Die Arbeit von Ducasse und Pollet kategorisiert bestehende Ansätze (darunter auch Nicht-Clustering-basierte Ansätze) dazu nach Zielen, Prozessen, Eingaben, Techniken und Ausgaben. Koschke untersucht in seiner Arbeit die verschiedenen Sichten, die von Clustering-basierten Ansätzen zur Softwarearchitektur-Rückgewinnung auf ein Softwaresystem ermöglicht werden und identifiziert insbesondere Forschungsbedarf bei der Erkennung von Architekturmustern [SG96, BMR+96].

Müller et al. [MOTU93] präsentieren einen der ersten automatisierten Ansätze zur Softwarearchitektur-Rückgewinnung. Grundelemente eines Systems (z. B. Variablen, Prozeduren, Module, Subsysteme) werden dabei basierend auf der Berechnung von Metrikwerten für Kopplung und Kohäsion gruppiert. Das in dieser Arbeit verwendete Clustering-basierte Werkzeug SoMoX unterstützt darüber hinaus weitere Metriken und berücksichtigt auch mögliche Zusammenhänge von Werten verschiedener Metriken.

Koschke [Kos02] analysiert und vergleicht verschiedene Techniken zur Softwarearchitektur-Rückgewinnung und schlägt ein Framework vor, welches verschiedene Ansätze kombiniert, um bessere Ergebnisse zu erhalten. Die von ihm berücksichtigten Ansätze sind alle Clustering-basiert und die Art, wie sie miteinander kombiniert werden, ist vergleichbar mit dem Einsatz verschiedener Metriken in SoMoX. Koschke schlägt vor, detailliertere und präzisere Informationen als die von Clustering-basierten Ansätzen zu verwenden, um noch bessere Ergebnisse zu erhalten. Die in dieser Arbeit vorgestellte Kombination eines Clustering-basierten und eines musterbasierten Reverse-Engineering-Ansatzes ist eine Möglichkeit, dies zu tun.

Maqbool und Babri [MB04] untersuchen in ihrer Arbeit verschiedene, in Clustering-Algorithmen oft verwendete Metriken und kombinieren die vielversprechendsten davon in einem neuen Algorithmus, um bessere Ergebnisse zu erzielen. Der

Einfluss von Schwachstellen in der Implementierung eines Softwaresystems auf das Clustering wird von ihnen jedoch nicht berücksichtigt. Bei der von ihrem Algorithmus rekonstruierten Architektur handelt es sich lediglich um Gruppierungen ähnlicher Elemente aus der Implementierung eines Systems. Die Rekonstruktion einer Softwarearchitektur nach einer Komponentendefinition, wie die von Szyperski [Szy02], wird von ihnen nicht angestrebt.

7.2 Musterbasiertes Reverse Engineering

Neben den Clustering-basierten Ansätzen existieren auch viele musterbasierte Reverse-Engineerings-Ansätze. Einen Überblick über die gängigen, in der Literatur zu findenden, musterbasierten Ansätze liefern Dong et al. [DZP09]. Die Arbeit kategorisiert die Ansätze unter anderem nach Aspekt der Muster (Verhaltens- oder Strukturmuster), Repräsentation eines Softwaresystems, Matching-Verfahren, Visualisierung von Musterfunden und Grad der Automatisierung.

Keller et al. [KSRP99] betrachten in ihrem Reverse-Engineering-Ansatz Entwurfsmuster [GHJV95] als Entwurfskomponenten und versuchen, diese in der Implementierung bestehender Softwaresysteme teilweise automatisiert zu erkennen. Im Gegensatz zu dem von Keller et al. beschriebenen Ansatz handelt es sich bei den Komponenten, die von SoMoX rekonstruiert werden, um Komponenten nach der Definition von Szyperski [Szy02].

Sartipi [Sar03] versucht die Softwarearchitektur eines Systems mit Hilfe von Architekturmustern zu rekonstruieren. Die Architekturmuster sollen in seinem Ansatz von einem Experten der Domäne, aus der auch das analysierte Softwaresystem stammt, spezifiziert und damit auf das analysierte System angepasst werden. Dieser Schritt ist vergleichbar mit der Konfiguration einer Clustering-basierten Analyse mit SoMoX durch die Angabe von Metrikgewichten, erfordert jedoch domänenspezifisches Expertenwissen. Außerdem berücksichtigen die Architekturmuster von Sartipi lediglich simple Eigenschaften der Architektur, wie die Anzahl von Beziehungen zwischen bestimmten Komponenten. Der musterbasierte Ansatz von Reclipse unterstützt hingegen deutlich komplexere Struktureigenschaften.

Trifu et al. [TSG04] versuchen in ihrem Ansatz Schwachstellen in einem Softwaresystem zu erkennen, um sie anschließend zu beseitigen. Bei den Schwachstellen handelt es sich um Schwachstellen in Bezug auf ein zuvor selektiertes Qualitätsmerkmal von Software, wie zum Beispiel Performanz oder Wartbarkeit. Zur Erkennung von Schwachstellen wird in ihrem Ansatz das musterbasierte jGoose [jGo11] verwendet. Die Autoren stellen jedoch fest, dass die Mustersuche meist viele potentielle Schwachstellen (sowohl True als auch False Positives) aufdeckt, die alle von einem Reengineer manuell untersucht und bearbeitet werden müssen. Der im Rahmen dieser Arbeit vorgestellte kombinierte Reverse-Engineering-Ansatz nutzt das Clustering, um die Mustersuche anschließend auf einzelnen Komponenten anzuwenden und damit die Analyse zu beschleunigen, die Anzahl der detektierten Schwachstellen zu reduzieren und damit handhabbarer zu machen.

Munro [Mun05] und Salehie et al. [SLT06] versuchen, Schwachstellen anhand von Metriken zu erkennen. Dabei werden charakteristische Eigenschaften von Schwachstellen verwendet, um Strategien zur Erkennung von Schwachstellen zu formulieren. Die beiden Ansätze verwenden Metriken jedoch nicht, um die Softwarearchitektur eines Softwaresystems zu rekonstruieren und unterscheiden sich damit von dem in dieser Arbeit beschriebenen Ansatz.

Moha et al. [MGDLM10] stellen einen Ansatz vor, indem Schwachstellen textbasiert spezifiziert werden. In ihrem Ansatz werden anschließend Algorithmen zur Erkennung von Schwachstellen aus ihrer Spezifikation generiert und auf ein Softwaresystem angewendet. Die Auswirkungen von Schwachstellen auf die Softwarearchitektur eines Systems werden dabei nicht berücksichtigt.

Das Werkzeug SA4J [SA411] von IBM dient der Strukturanalyse von Java-basierten Softwaresystemen. Neben der Erkennung von Schwachstellen bietet das Werkzeug auch eine „What if“-Funktion, welche eine Vorschau auf die Struktur der analysierten Software in Folge einer Änderung an der Implementierung ermöglicht. Im Gegensatz zu SoMoX, das die Softwarearchitektur eines Systems in Form eines Komponentendiagramms rekonstruiert, repräsentiert SA4J die Struktur von Software als Graph, dessen Knoten Pakete, Klassen und Schittstellen sind.

7.3 Kombiniertes Reverse Engineering

In der Literatur sind bislang nur wenige Reverse-Engineering-Ansätze zu finden, die Clustering- und musterbasierte Verfahren kombinieren. Dieser Abschnitt stellt einige verwandte Verfahren vor, die eine Kombinationen von Clustering und Mustersuche realisieren und diskutiert deren Ansätze.

Bauer und Trifu [BT04] kombinieren in ihrem Ansatz Clustering und Mustersuche, um die Softwarearchitektur eines System zu rekonstruieren. Die Mustersuche wird dabei zur Erkennung von zum Beispiel Adaptern, Proxies oder Kompositionsbeziehungen eingesetzt, welche die Autoren als „Architectural Clues“ bezeichnen. Diese sind oft Teil eines Architekturmusters und werden in dem Ansatz zur Berechnung verschiedener Arten von Kopplung verwendet. Die Kopplung wird zur Erzeugung einer Multigraph-Repräsentation des analysierten Systems verwendet, auf dem das Clustering durchgeführt wird. Die Ergebnisse des Clusterings bauen in dem Ansatz vollständig auf denen der Mustersuche auf. Daher muss die Mustersuche auf das gesamte System angewendet werden. Dies kann für große Systeme sehr aufwändig sein. Der im Rahmen dieser Arbeit vorgestellte Ansatz verwendet das Clustering, um den Suchraum für die Mustersuche einzuschränken. Bauer und Trifu verwenden Entwurfsmuster lediglich als Hilfsmittel, um die Softwarearchitektur eines Systems zu rekonstruieren. Die Erkennung von Entwurfs- oder Schwachstellenmustern ist kein Ziel des Ansatzes. Der Einfluss von Schwachstellen auf das Clustering wird in ihrem Ansatz nicht berücksichtigt.

Sonargraph (ehemals SonarJ und Sotograph) [Son11] ist eine kommerzielle Reengineering-Werkzeugkette, die einem Softwarearchitekten ermöglicht, eine kon-

zeptionelle Softwarearchitektur eines Softwaresystems als Architekturmuster zu spezifizieren und diese mit der konkreten Architektur des Systems abzugleichen, um Verstöße gegen die konzeptionelle Architektur in der Implementierung zu erkennen. Sonargraph verwendet eine Vielzahl von Softwaremetriken bei der Analyse der konkreten Softwarearchitektur und bietet umfangreiche Refactoring-Maßnahmen zur Verbesserung der Softwarequalität an. In Sonargraph wird die konzeptionelle Softwarearchitektur als bekannt vorausgesetzt, die vor der Analyse von einem Softwarearchitekten spezifiziert werden muss, sodass fortan die konkrete Architektur während der Weiterentwicklung eines Softwaresystems überwacht werden kann. Im Gegensatz dazu wird im Rahmen dieser Arbeit versucht, die konkrete Architektur durch Beseitigen von möglichen Schwachstellen an eine dem Softwarearchitekten noch unbekannte konzeptionelle Softwarearchitektur anzunähern und ihm dadurch beim Verstehen der Software und deren konzeptioneller Architektur zu helfen.

Basit und Jarzabek [BJ05] nutzen eine Mustersuche, um identische bzw. sehr ähnliche Teile im Quellcode eines Softwaresystems, so genannte *Clone Patterns*, zu erkennen. Anschließend wird eine Clustering-basierte Analyse auf den Dateien, die den Quellcode enthalten, durchgeführt, um Dateien mit großen strukturellen Ähnlichkeiten zueinander zu gruppieren. Die Autoren behaupten, dass ihr Ansatz zur Erkennung von Mustern mit niedrigem Abstraktionslevel, wie zum Beispiel Idiome, bis hin zu Architekturmustern genutzt werden kann. Der Ansatz strebt jedoch keine Erkennung von Mustern an, die zuvor spezifiziert werden können, sondern leitet sich die Muster während der Analyse aus dem Quellcode ab. Entwurfs- und Schwachstellenmuster werden daher nicht adressiert. Eine Kategorisierung von Clone Patterns in *gut* und *schlecht*, wie es bei Entwurfs- und Schwachstellenmustern der Fall ist, wird nicht vorgenommen.

Arcelli Fontana und Zanoni [FZ11] kombinieren eine Clustering-basierte Softwarearchitektur-Rekonstruktion und eine musterbasierte Erkennung von Entwurfsmustern in einem Werkzeug. In ihrem Werkzeug werden die Analyseverfahren jedoch isoliert voneinander angewendet und berücksichtigen Ergebnisse des jeweils anderen Verfahrens nicht. Das Werkzeug kombiniert daher das Clustering und die Mustersuche ähnlich, wie in der Abbildung 1.1 in Kapitel 1 beschrieben, die den Ausgangspunkt dieser Arbeit dokumentiert. Die Erkennung von Schwachstellen wollen Arcelli Fontana und Zanoni in zukünftigen Arbeiten adressieren.

8 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und erläutert. Abschließend werden Ideen für Erweiterungen und Verbesserungen des vorgestellten Ansatzes in einem Ausblick kurz vorgestellt.

8.1 Zusammenfassung

Durch wiederkehrende Erweiterungen und Wartungsarbeiten können sich im Laufe der Zeit Schwachstellen in der Implementierung eines Softwaresystems ansammeln. Da Reverse-Engineering-Ansätze zur Rekonstruktion der Softwarearchitektur die Implementierung als Grundlage ihrer Analyse verwenden, können sie nur die konkrete Softwarearchitektur rekonstruieren. Weil diese aber, wie am Beispiel der Interface Violation in dieser Arbeit gezeigt wurde, durch Schwachstellen in der Implementierung beeinflusst wird, kann sie von der konzeptionellen Softwarearchitektur abweichen.

Um Schwachstellen, die einen solchen Einfluss auf die Softwarearchitektur ausüben, zu identifizieren, wurden im Rahmen dieser Arbeit Clustering- und musterbasierte Techniken in einem Reverse-Engineering-Prozess kombiniert. Der Clustering-basierte Ansatz rekonstruiert dabei die konkrete Softwarearchitektur eines Systems, die einen Überblick über das analysierte System liefert. Die Komponenten des Systems können anschließend mit Hilfe des musterbasierten Ansatzes nach potentiellen Schwachstellen durchsucht werden. Dabei aufgedeckte Schwachstellen können dann in der Implementierung des Systems von einem Reengineer beseitigt werden. Durch die Beseitigung der Schwachstellen wird auch ihr Einfluss auf die konkrete Softwarearchitektur entfernt. In Folge dessen kann eine auf Grund von Schwachstellen vorliegende Diskrepanz zwischen konkreter und konzeptioneller Architektur reduziert werden.

Die Anwendung des musterbasierten Reverse Engineerings auf einzelne Komponenten birgt zwei Vorteile gegenüber einer Analyse des gesamten Softwaresystems. Zum einen sind Schwachstellenvorkommen auf die analysierten Komponenten beschränkt, wodurch sie den Fokus des Reengineers berücksichtigen und der Aufwand, der bei der Auswertung der Analyseergebnisse entsteht, verringert wird. Zum anderen benötigt die Analyse einer Komponente deutlich weniger Zeit, als es für das gesamte System der Fall ist. In Folge dessen kann das musterbasierte Reverse Engineering auch Teile von Systemen analysieren, die sonst zu groß für den musterbasierten Ansatz wären bzw. zu viele Mustervorkommen als Ergebnis einer Analyse aufdecken würden.

Im Rahmen dieser Arbeit wurde der Zusammenhang von einigen Schwachstellen und ihrer Auswirkung auf die Softwarearchitektur identifiziert und Muster zur Erkennung der Schwachstellen spezifiziert. Alle in dieser Arbeit erarbeiteten Schwachstellen beeinflussen die Kopplung eines Systems und damit einen der wichtigsten Indikatoren für die Rekonstruktion einer Softwarearchitektur. Experimente am Beispiel des komponentenbasierten Softwaresystems CoCoME belegen, dass einige der in dieser Arbeit erarbeiteten Schwachstellen selbst in beispielhaft entwickelten Softwaresystemen vorkommen können und sich auf deren Softwarearchitektur auswirken. Die Annahme, dass die Beseitigung von Schwachstellen zu einer Annäherung der konkreten an die konzeptionelle Architektur führt, konnte hingegen nicht bestätigt werden. Um diese zu bestätigen oder zu widerlegen bedarf es weiterer Experimente, die im Rahmen dieser Arbeit nicht mehr durchgeführt werden konnten. Ein weiteres Ergebnis der Experimente ist, dass die rekonstruierte Architektur starken Einfluss auf die Schwachstellensuche nehmen kann und bei großer Abweichung von der konzeptionellen Architektur Schwachstellen übersehen werden können. Zum einen bedeutet es, dass eine Verbesserung des Clustering-basierten Ansatzes auch zu einer Verbesserung des in dieser Arbeit vorgestellten kombinierten Ansatzes führt. Zum anderen aber auch, dass Schwachstellenmuster in Zukunft toleranter gegenüber solchen Abweichungen formuliert werden müssen.

8.2 Ausblick

Bei der Spezifikation der Schwachstellenmuster wurden bisher keine optionalen Teile modelliert. Die Erweiterung der Schwachstellenmuster um optionale Teile könnte in Zukunft dazu verwendet werden, Abweichungen zwischen rekonstruierter und konzeptioneller Architektur durch solche Teile zu identifizieren. Eine automatisierte Bewertung eines Mustervorkommens [Wen07] könnte anschließend die Auswertung der True und False Positives vereinfachen.

Im Rahmen dieser Arbeit wurden lediglich einige wenige Schwachstellenmuster beschrieben und spezifiziert, um die Konzepte dieser Arbeit an einem beispielhaften Softwaresystem prüfen zu können. Die zu Grunde liegenden Schwachstellen üben alle Einfluss auf die Metriken *Coupling* oder *InterfaceViolation* aus. In zukünftigen Arbeiten wäre daher die Identifikation und Spezifikation weiterer Schwachstellenmuster, deren Schwachstellen auch andere Metriken betreffen, wünschenswert, um den Katalog von Schwachstellenmustern zu erweitern. Eine Evaluierung des Ansatzes an größeren Softwaresystemen als CoCoME wäre ebenfalls wünschenswert, um die Grenzen der Anwendbarkeit des Ansatzes zu prüfen.

Mit dem in dieser Arbeit vorgestellten Ansatz ist es möglich, die Mustersuche auf eine oder mehrere Komponenten eines Softwaresystems zu beschränken. Weil Mustervorkommen sich theoretisch auch über weite Teile des Systems erstrecken können, ist es möglich, dass sie im Rahmen einer Mustersuche verborgen bleiben. Insbesondere für die Analyse von großen Softwaresystemen, die nur in Teilen musterbasiert analysiert werden können, wäre es wichtig zu erfahren, ob solche

Muster existieren und wenn ja, inwiefern sie eine Einschränkung des Ansatzes dieser Arbeit sind. Daher sollte dieser Sachverhalt in zukünftigen Arbeiten untersucht werden.

Der in dieser Arbeit vorgestellte Ansatz kann zwar zur Erkennung von Schwachstellen in einem System verwendet werden, doch kann ein Reengineer den Einfluss einer Schwachstelle auf die Softwarearchitektur momentan nur nachvollziehen, indem die Schwachstelle in der Implementierung beseitigt und ein erneutes Clustering durchgeführt wird. Dieser Prozess kann sehr langwierig, aufwändig und fehleranfällig sein. Eine mögliche Erweiterung des in dieser Arbeit vorgestellten Ansatzes wäre die Unterstützung eines Reengineers durch Vorschläge zur Beseitigung einer Schwachstelle. Viele Schwachstellen, wie zum Beispiel die Interface Violation, können durch simple Refactoring-Maßnahmen beseitigt werden, andere bedürfen komplexerer Maßnahmen. Für diese Erweiterung wird eine Wissensdatenbank mit typischen Refactoring-Maßnahmen benötigt. Die Refactoring-Maßnahmen könnten durch Modelltransformationen beschrieben werden, sodass die Beseitigung einer Schwachstelle teil- oder vollautomatisiert durchgeführt werden kann.

Bei der Entscheidung eines Reengineers, welche der erkannten Schwachstellenvorkommen zuerst beseitigt werden sollten, fehlt es momentan an Entscheidungskriterien. Um effizient vorzugehen, wird ein Reengineer die Schwachstellen mit dem größten Einfluss auf die Softwarearchitektur zuerst beseitigen wollen. Dies ist insbesondere dann wichtig, wenn viele Schwachstellenvorkommen erkannt wurden. Momentan kann ein Reengineer diesbezüglich nur Vermutungen anstellen. In zukünftigen Arbeiten sollte daher eine Relevanzanalyse umgesetzt werden, welche die Metrikwerte, die während des Clusterings ermittelt werden, protokolliert und den Einfluss einer Schwachstelle auf das Clustering misst. Auf diese Weise könnten Schwachstellen nach ihrem Einfluss auf die Softwarearchitektur bewertet, sortiert und für Vorschläge zur Beseitigung von bestimmten Schwachstellen verwendet werden.

Im Rahmen dieser Arbeit wurde lediglich die Erkennung von Schwachstellenmustern in komponentenbasierten Softwaresystemen verfolgt, doch ist auch die Erkennung von Entwurfsmustern [GHJV95] in einem solchen System möglich. Folglich sollen in Zukunft auch Entwurfsmuster in komponentenbasierten Softwaresystemen mit dem vorgestellten Ansatz erkannt werden. Das musterbasierte Reverse Engineering wurde in dieser Arbeit derart erweitert, dass neben einem AST-Modell, das die Implementierung eines Systems repräsentiert, auch das Modell der rekonstruierten Softwarearchitektur musterbasiert analysiert wird. In zukünftigen Arbeiten könnte der kombinierte Ansatz daher auch dazu verwendet werden, die Softwarearchitektur eines Systems nach Vorkommen von Architekturmustern zu analysieren und damit dem von Koschke [Kos05] identifizierten Forschungsbedarf bei der Erkennung von Architekturmustern nachzukommen.

Anhang A

Modifizierte Schwachstellenmuster

In Kapitel 6 wurde dokumentiert, dass die Schwachstellenmuster, die in Abschnitt 4.3 vorgestellt wurden, auf Grund technischer Einschränkungen zu vielen False-Positives unter den von der Mustersuche detektierten Vorkommen führen. Von einigen Schwachstellenmustern konnten deswegen keine Vorkommen detektiert werden. Die Schwachstellenmuster wurden deswegen verändert bzw. an die technischen Einschränkungen angepasst. Die folgenden Abbildungen zeigen die Schwachstellenmuster und eines der Korrespondenzmuster, die angepasst wurden.

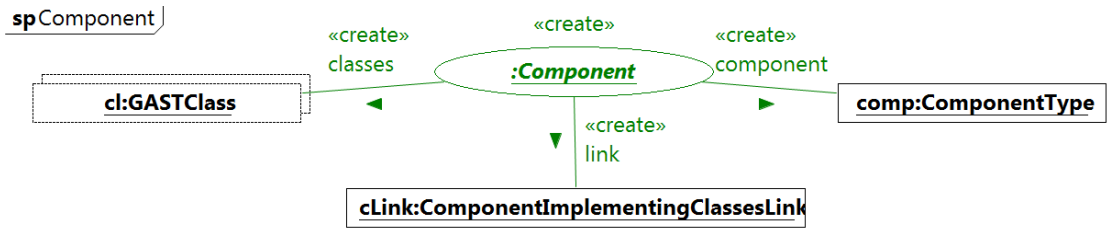

Abbildung A.1: Erweiterung des *Component*-Musters um die Rolle link

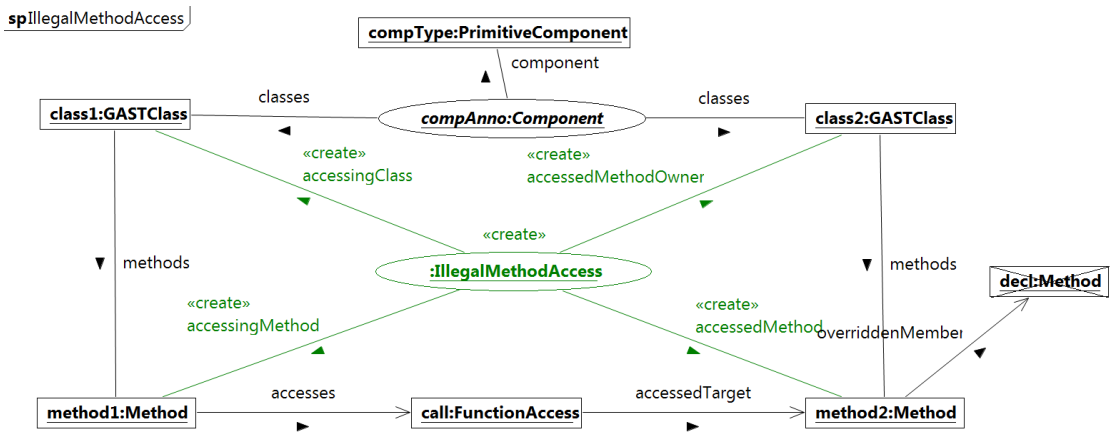

Abbildung A.2: Beschränkung des *IllegalMethodAccess*-Musters auf Interface Violations innerhalb einer Komponente und Filterung von Bibliotheksklassen

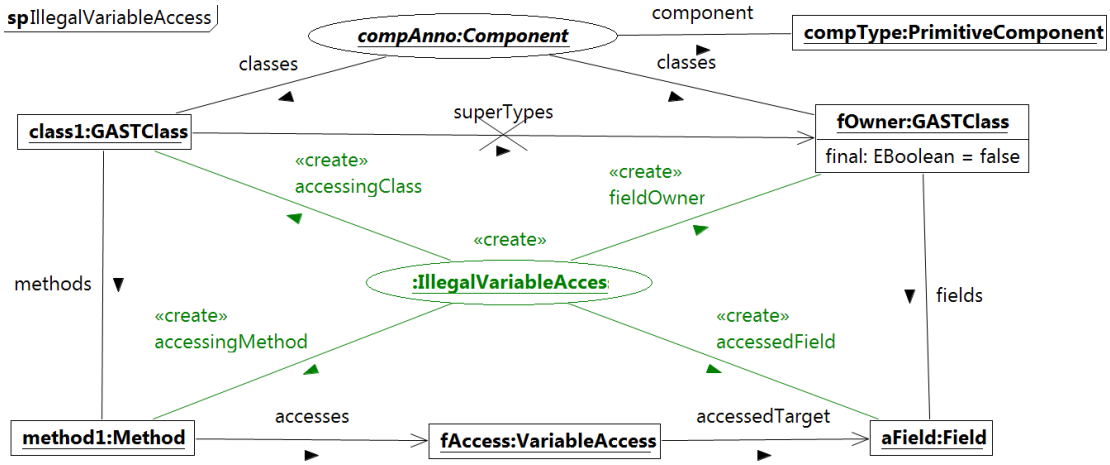

Abbildung A.3: Beschränkung des *IllegalVariableAccess*-Musters auf Interface Violations innerhalb einer Komponente und Filterung von Bibliotheksklassen und Enums

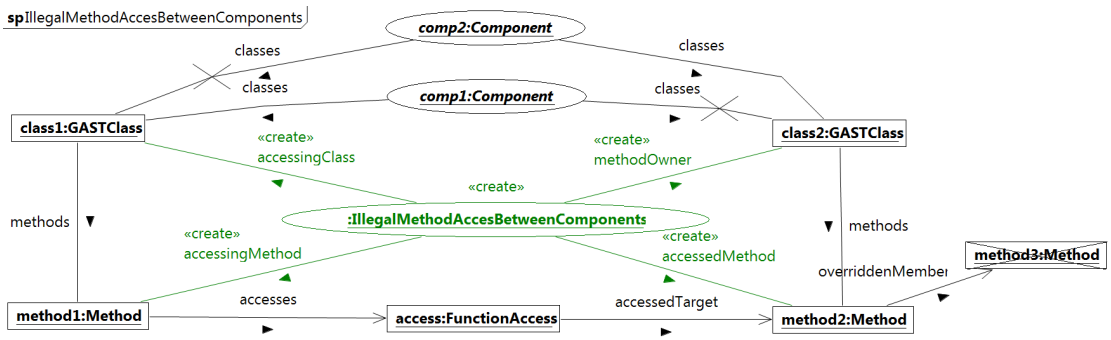

Abbildung A.4: Beschränkung des *IllegalMethodAccessBetweenComponents*-Musters auf Interface Violations zwischen zwei Komponenten. Filterung von Bibliotheksklassen ist hierbei unnötig, da dies bereits durch die Komponentenzuordnung geschehen ist.

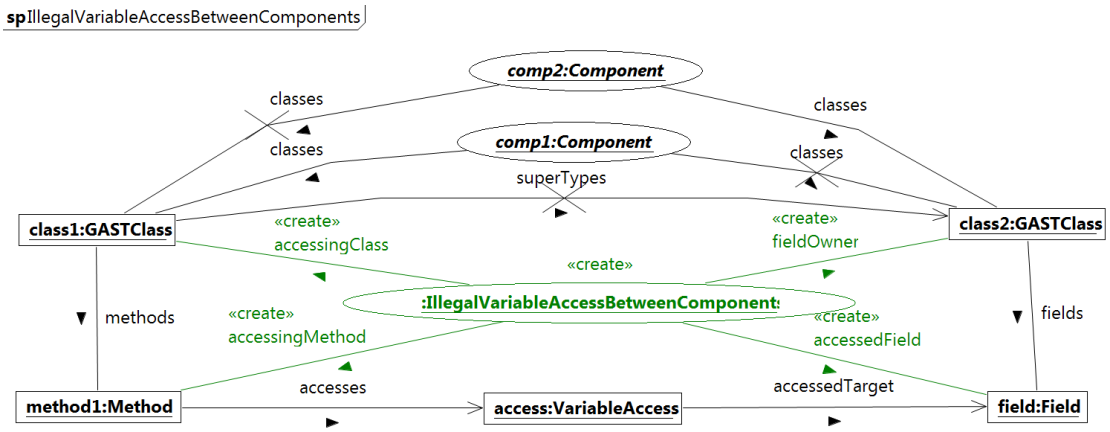

Abbildung A.5: Beschränkung des *IllegalVariableAccessBetweenComponents*-Musters auf Interface Violations zwischen zwei Komponenten. Filterung von Bibliotheksklassen ist hierbei unnötig, da dies bereits durch die Komponentenzuordnung geschehen ist.

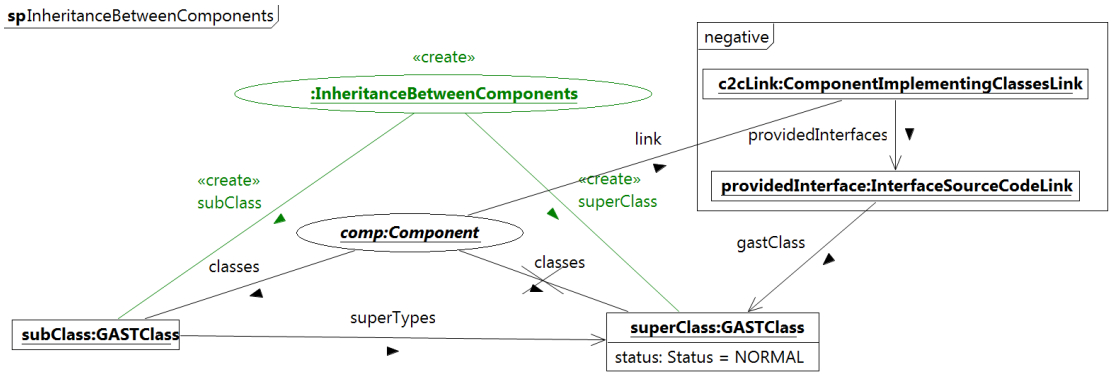

Abbildung A.6: Anpassung des *InheritanceBetweenComponents*-Musters zur Filterung von Bibliotheksklassen. Außerdem wurden unidirektionale Links vermieden, die vom Story-Diagramm-Interpreter nicht erkannt werden können.

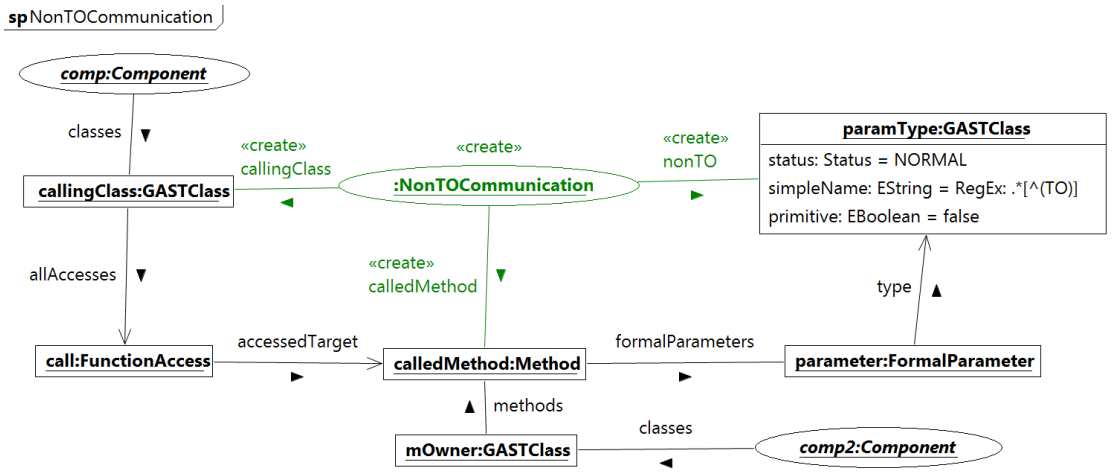

Abbildung A.7: Anpassung des *NonToCommunication*-Musters zur Filterung von Parametertypen, die durch Bibliothekssllassen, primitiven Datentypen oder durch Transferobjekte getypt sind. Transferobjekte werden anhand des Suffix TO im Namen gefiltert. Außerdem wurden unidirektionale Links vermieden, die vom Story-Diagramm-Interpreter nicht erkannt werden können.

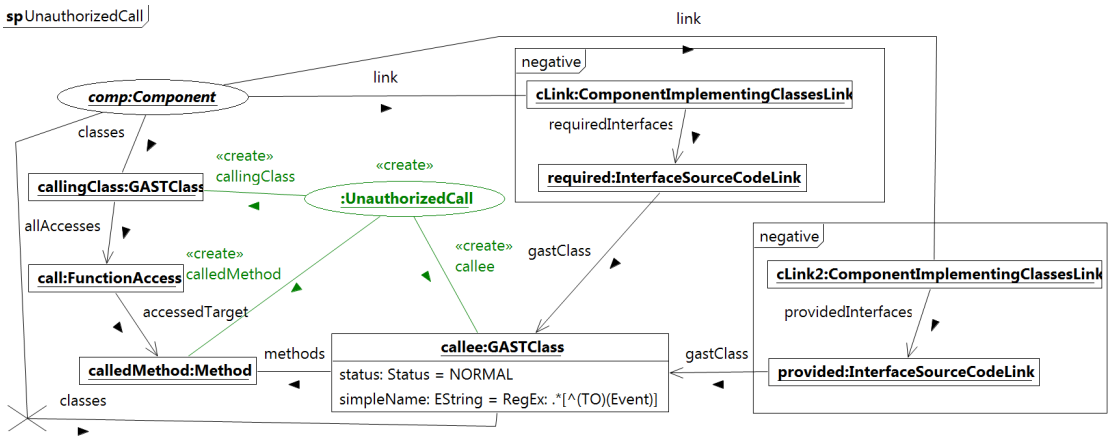

Abbildung A.8: Anpassung des *UnauthorizedCall*-Musters zur Filterung von Bibliotheks-, Transferobjekt- und Event-Klassen. Außerdem wurden unidirektionale Links vermieden, die vom Story-Diagramm-Interpreter nicht erkannt werden können.

Anhang B

Weitere Musterspezifikationen

In Kapitel 4.3 wurde das Muster *NonToCommunication* vorgestellt, welches Kommunikation zwischen Komponenten über Objekte, die keine Transferobjekte sind, aufdecken soll. Dort wurde erläutert, dass noch eine zweite Variante des Musters benötigt wird. Diese bezieht sich auf die Kommunikation über Objekte, die zu den Schnittstellen einer Komponente angehören und in Abbildung B.1 dargestellt wird.

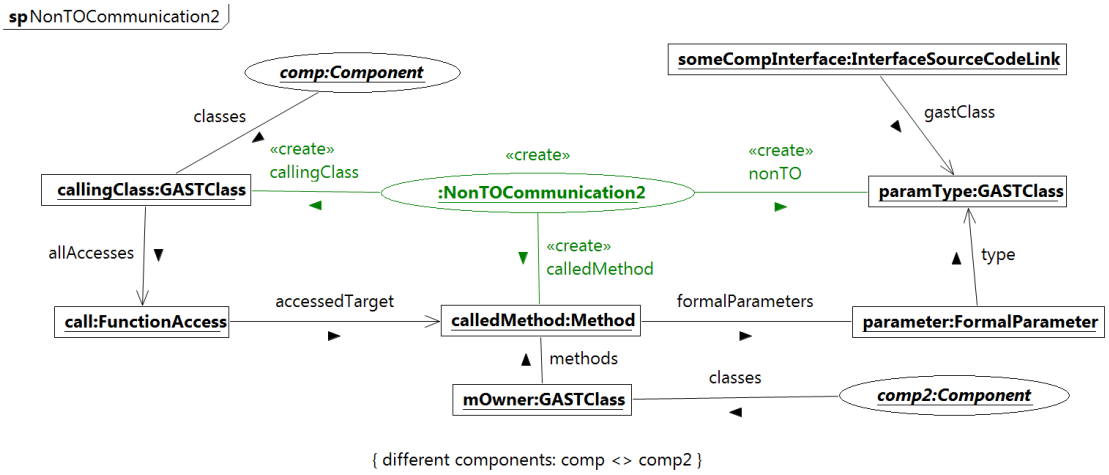

Abbildung B.1: Musterspezifikation zur Erkennung von Kommunikation über nicht-Transferobjekte - Variante über eine *required*- oder *provided*-Schnittstelle

Anhang C

Rekonstruktion der CoCoME-Architektur

Um die Softwarearchitektur der von CoCoME in Kapitel 6 zu rekonstruieren wurden in SoMoX die in der Tabelle C.1 dargestellten Metrikgewichte verwendet. Der in Abschnitt 6.1 vorgestellte Ausschnitt der rekonstruierten Architektur, der mit Hilfe der Metrikgewichte in der dritten Spalte der Tabelle rekonstruiert wurde, wird in Abbildung C.1 dargestellt.

	Clustering 1	Clustering 2
Metrik	Gewicht	Gewicht
Package Mapping	70	80
Directory Mapping	70	80
DMS	5	5
Low Coupling	0	0
High Coupling	15	15
Low Name Resemblance	5	5
Mid Name Resemblance	15	15
High Name Resemblance	30	30
Highest Name Resemblance	45	45
Low SLAQ	0	0
High SLAQ	15	15
Composition: Interface Adherence	40	70
Clustering Composition Threshold Max Value	100	65
Clustering Composition Threshold Min Value	25	3
Clustering Composition Threshold Decrement	10	15
Merge: Interface Violation	10	20
Clustering Merge Threshold Max Value	100	100
Clustering Merge Threshold Min Value	45	25
Clustering Merge Threshold Increment	10	10

Tabelle C.1: Gewählte Gewichtung von Metrikwerten im Rahmen des ersten und zweiten Clusterings von CoCoME

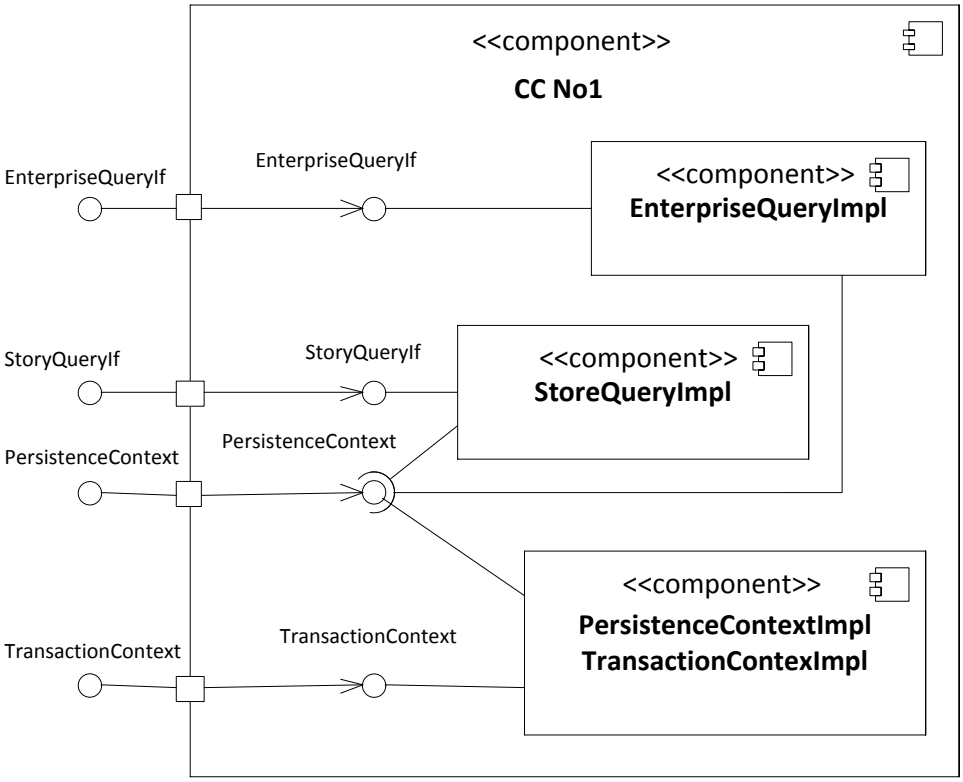

Abbildung C.1: Ausschnitt der rekonstruierten Architektur der CoCoME Referenzimplementierung (Clustering 2)

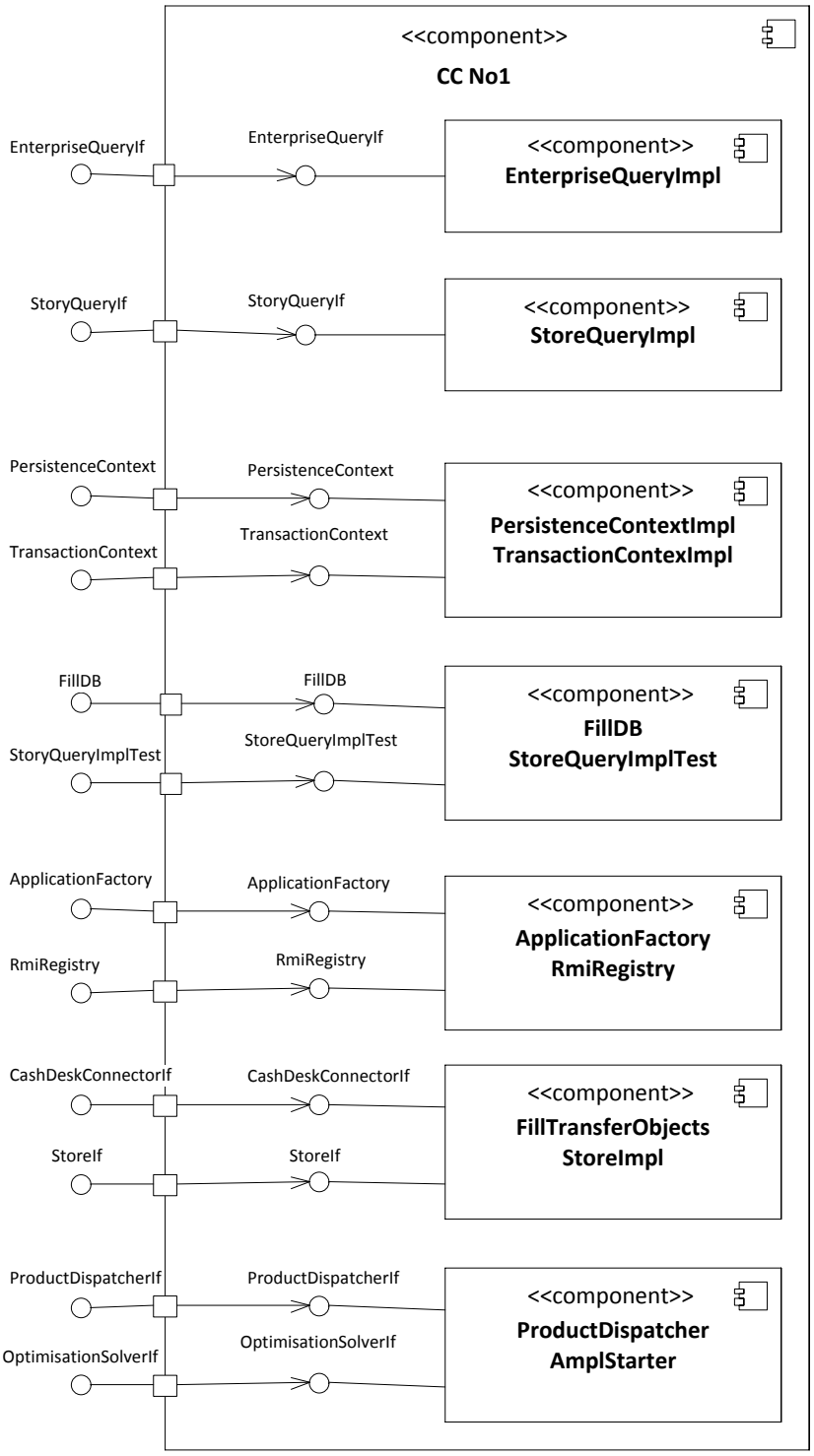

Abbildung C.2: Ausschnitt der rekonstruierten Architektur der CoCoME Referenzimplementierung (Clustering 2) nach Beseitigung der Interface Violations. Die Verwendung von Schnittstellen innerhalb der Komponente wurde aus Platzgründen nicht dargestellt.

Anhang D

Annotations- und Engine-Metamodell

In Kapitel 5 wurde die Anpassung der Generierung von Suchalgorithmen für Muster in Reclipse vorgestellt. In einer Mustersuche in Reclipse werden Mustervorkommen in Wirtsgraphen annotiert. Die Annotation eines bestimmten Mustervorkommens wird durch eine für das Muster generierte Unterklasse von ASGAnnotation repräsentiert. Solche Annotationen werden von generierten Suchalgorithmen erzeugt, die in Engine-Klassen gekapselt werden. Engine-Klassen werden für jede nicht abstrakte Musterspezifikation durch eine Unterklasse von AnnotationEngine realisiert. Die Abbildung D.1 zeigt das Metamodell, das für die Generierung von Annotations- und Engine-Klassen verwendet wird.

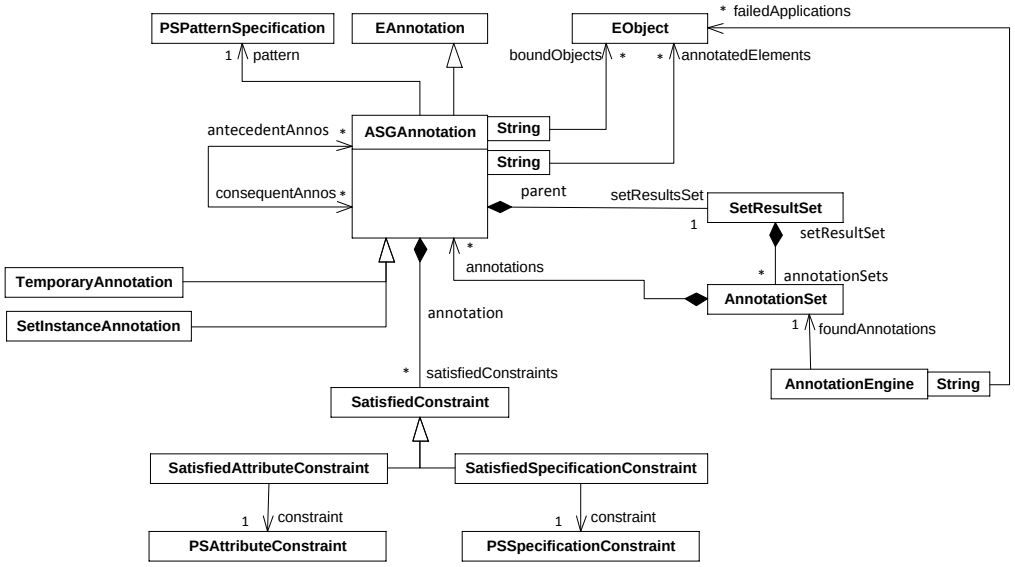

Abbildung D.1: Metamodell der Annotationen und Engines

Anhang E

Komponenten und Eclipse Plug-ins

Die im Rahmen dieser Arbeit erstellten Eclipse Plug-ins wurden auf der CD hinterlegt, welche der Ausarbeitung beiliegt. Zum einen enthält die CD eine Eclipse IDE, in der bereits alle nötigen Plug-ins installiert wurden. Zum anderen wurde der CD eine Eclipse Update Site beigefügt, welche die Installation der Plug-ins in eine bestehende Eclipse Installation ermöglicht. Ein Eclipse Workspace mit den Projekten, die den Quellcode der Plug-ins enthalten, wurden ebenfalls der CD hinzugefügt. Ein weiterer Workspace enthält ein Java-Projekt mit dem Quellcode von CoCoME und ein Projekt mit den im Rahmen dieser Arbeit entstandenen Musterkatalogen. Dieser kann zum Testen des kombinierten Reverse-Engineering-Ansatzes genutzt werden.

Im Folgenden werden die Abschnitt 5.1 beschriebenen Komponenten und ihre Eclipse Plug-ins aufgelistet. Die Komponente SISSy Parser (5) und das GAST-Metamodell (1) kann zusammen mit dem dafür benötigten Palladio Component Model (PCM) per Update Site installiert werden und wird hier daher nicht gelistet. Die dafür nötige Update Site ist `http://q-impress.ow2.org/release`.

Anschließend werden weitere Eclipse Plug-ins gelistet, zu denen eine Implementierungsabhängigkeit besteht und die daher zur Ausführung der Plug-ins benötigt werden.

Repository	svn://svn.forge.objectweb.org/svnroot/
Plug-in	eu.qimpress.samm

Tabelle E.1: Service Architecture Meta Model (2)

Repository	http://svn.codespot.com/a/eclipselabs.org/sdm-commons/
Plug-in	org.storydriven.modeling

Tabelle E.2: Story-Diagramm Metamodell (3)

Repository	https://dsd-serv.uni-paderborn.de/svn/reclipse-emf/
Plug-in	org.reclipse.structure.specification

Tabelle E.3: Musterspezifikations Metamodell (4)

Repository	svn://svn.forge.objectweb.org/svnroot/q-impress/
Plug-in	eu.qimpress.sourcecodedecorator

Tabelle E.4: Source Code Decorator Metamodell (6)

Repository	https://dsd-serv.uni-paderborn.de/svn/reclipse-emf/
Plug-in	org.reclipse.generator
Plug-in	org.reclipse.generator.ui
Plug-in	org.reclipse.structure.generator
Plug-in	org.reclipse.structure.generator.ui

Tabelle E.5: Reclipse Suchalgorithmen Generator (7)

Repository	https://dsd-serv.uni-paderborn.de/svn/reclipse-emf/
Plug-in	org.reclipse.math
Plug-in	org.reclipse.gast
Plug-in	org.reclipse.structure.specification.edit
Plug-in	org.reclipse.structure.specification.editor
Plug-in	org.reclipse.structure.specification.ui
Plug-in	org.reclipse.metrics
Plug-in	org.reclipse.behavior.specification

Tabelle E.6: Reclipse Musterspezifikations-Editor (8)

Repository	svn://svn.forge.objectweb.org/svnroot/q-impress/
Plug-in	eu.qimpress.dtmc
Plug-in	org.jgrapht
Plug-in	org.somox.ProvidedRequiredIds.edit
Plug-in	org.somox.ProvidedRequiredIds.editor
Plug-in	org.somox.metrics.dsl
Plug-in	org.somox.metrics.dsl.generator
Plug-in	org.somox.provreqid
Plug-in	org.somox.analyzer.sissymodelanalyzer
Plug-in	org.somox.analyzer.sissymodelanalyzer.ui
Plug-in	org.somox.core
Plug-in	org.somox.filter
Plug-in	uk.ac.shef.dcs.simmetrics
Plug-in	eu.qimpress.dtmc.edit
Plug-in	eu.qimpress.dtmc.editor
Plug-in	eu.qimpress.gast
Plug-in	eu.qimpress.gast.edit
Plug-in	eu.qimpress.gast.editor
Plug-in	eu.qimpress.identifier
Plug-in	eu.qimpress.identifier.edit
Plug-in	eu.qimpress.reverseengineering.gast2seff
Plug-in	eu.qimpress.qualityannotationdecorator
Plug-in	eu.qimpress.qualityannotationdecorator.edit
Plug-in	eu.qimpress.qualityannotationdecorator.editor
Plug-in	eu.qimpress.samm.edit
Plug-in	eu.qimpress.samm.editor
Plug-in	eu.qimpress.seff
Plug-in	eu.qimpress.seff.edit
Plug-in	eu.qimpress.seff.editor
Plug-in	eu.qimpress.sourcecodedecorator
Plug-in	eu.qimpress.sourcecodedecorator.edit
Plug-in	eu.qimpress.sourcecodedecorator.editor
Plug-in	org.somox.resources.defaultmodels
Plug-in	org.somox.ui

Tabelle E.7: SoMoX Clustering (9)

Repository	https://www.hpi.uni-potsdam.de/giese/gforge/svn/storyeditor/
Plug-in	de.mdelab.sdm.interpreter.sde.sdeInterpreterComponent
Plug-in	de.hpi.sam.storyDiagramEcore
Plug-in	de.mdelab.sdm
Plug-in	de.mdelab.sdm.interpreter.common
Plug-in	de.mdelab.sdm.interpreter.common.eclipse
Plug-in	de.mdelab.sdm.interpreter.ocl
Plug-in	org.storydriven.modeling.interpreter.adapter

Tabelle E.8: Story-Diagramm-Interpreter (10)

Repository	https://dsd-serv.uni-paderborn.de/svn/reclipse-emf/
Plug-in	org.reclipse.structure.inference
Plug-in	org.reclipse.structure.inference.annotations
Plug-in	org.reclipse.structure.inference.ui
Plug-in	org.reclipse.structure.inference.ui.matching
Plug-in	org.reclipse.interpreter.adapter.extension

Tabelle E.9: Reclipse Inferenz (11)

Repository	https://dsd-serv.uni-paderborn.de/svn/ma-travkin/
Plug-in	org.reclipse.models
Plug-in	org.reclipse.structure.inference.extended
Plug-in	org.reclipse.structure.inference.extended.ui

Tabelle E.10: Reclipse Inferenz-Erweiterung (12)

Repository	https://dsd-serv.uni-paderborn.de/svn/fujaba4eclipse/
Plug-in	AppIndependent4Eclipse

Tabelle E.11: Zusätzliche Plug-ins, die zur Ausführung benötigt werden

Repository	https://dsd-serv.uni-paderborn.de/svn/reclipse-emf/
Plug-in	org.fujaba.commons
Plug-in	org.fujaba.commons.notation
Plug-in	org.fujaba.sdm.ui

Tabelle E.12: Zusätzliche Plug-ins, die zur Ausführung benötigt werden

Literatur

[ABM+06] Anderer, Jochen; Bloch, Rainer; Mohaupt, Thomas; Neumann, Rainer; Schumacher, Alexa; Seng, Olaf; Simon, Frank; Trifu, Adrian; Trifu, Mircea: Methoden und Werkzeuge zur Sicherung der inneren Qualität bei der Evolution objektorientierter Softwaresysteme / FZI Forschungszentrum Informatik. 2006 (1-6-6/06). – Forschungsbericht. – ISSN 0944–3037

[BHT+10] Becker, Steffen; Hauck, Michael; Trifu, Mircea; Krogmann, Klaus; Kofron, Jan: Reverse Engineering Component Models for Quality Predictions. In: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track*, IEEE, 2010, S. 199–202

[BJ05] Basit, Hamid A.; Jarzabek, Stan: Detecting higher-level similarity patterns in programs. In: *SIGSOFT Software Engineering Notes* 30 (2005), September, Nr. 5, S. 156–165. – ISSN 0163–5948

[BMMM98] Brown, William J.; Malveau, Raphael C.; McCormick, Hays W.; Mowbray, Thomas J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1. Wiley, 1998. – ISBN 0471197130

[BMR+96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996. – ISBN 978–0–471–95869–7

[BT04] Bauer, Markus; Trifu, Mircea: Architecture-Aware Adaptive Clustering of OO Systems. In: *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, 2004. – ISSN 1534–5351, S. 3 – 14

[CKK01] Cho, Eun S.; Kim, Min S.; Kim, Soo D.: Component metrics to measure component quality. In: *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, 2001, S. 419 – 426

[CKK08] Chouambe, Landry; Klatt, Benjamin; Krogmann, Klaus: Reverse Engineering Software-Models of Component-Based Systems. In: Kontogiannis, Kostas (Hrsg.); Tjortjis, Christos (Hrsg.);

Winter, Andreas (Hrsg.): *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. Athens, Greece : IEEE Computer Society, April 1–4 2008. – ISBN 978–1–4244–2157–2, S. 93–102

[coc10] *CoCoME - Common Component Modelling Example*. November 2010. – `http://agrausch.informatik.uni-kl.de/CoCoME`

[DB11] von Detten, Markus; Becker, Steffen: Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems. In: *Proceedings of the 7th International Conference on the Quality of Software Architectures*, 2011 (QoSA 2011)

[DDN08] Demeyer, Serge; Ducasse, Stéphane; Nierstrasz, Oscar: *Object-Oriented Reengineering Patterns*. Switzerland : Square Bracket Associates, Juni 2008. – ISBN 978–3–9523341–2–6

[DHL+11] von Detten, Markus; Heinzemann, Christian; Lauder, Marius; Rieke, Jan; Travkin, Dietrich: A new Meta-Model for Story Diagrams. 2011. – Forschungsbericht

[DMT10] von Detten, M.; Meyer, M.; Travkin, D.: Reverse Engineering with the Reclipse Tool Suite. In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa*, 2010

[DP09] Ducasse, Stéphane; Pollet, Damien: Software Architecture Reconstruction: A Process-Oriented Taxonomy. In: *IEEE Transactions on Software Engineering* 35 (2009), Nr. 4, S. 573–591

[DZP09] Dong, Jing; Zhao, Yajing; Peng, Tu: A Review of Design Pattern Mining Techniques. In: *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* 19 (2009), September, Nr. 6, S. 823–855

[EMF10] *Eclipse Modelling Framework (EMF)*. November 2010. – `http://www.eclipse.org/emf`

[Epp99] Eppstein, David: Subgraph Isomorphism in Planar Graphs and Related Problems. In: *CoRR* cs.DS/9911003 (1999)

[FNTZ00] Fischer, Thorsten; Niere, Jörg; Torunski, Lars; Zündorf, Albert: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, Hartmut (Hrsg.); Engels, Gregor (Hrsg.); Kreowski, Hans-Jörg (Hrsg.); Rozenberg, Grzegorz (Hrsg.): *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT '98), November 16-20, 1998, Paderborn, Germany* Bd.

1764. Springer Berlin / Heidelberg, 2000. – ISBN 3–540–67203–6, S. 296 – 309

[Foc10] Fockel, Markus: *Interpretation von Graphtransformationsregeln zur statischen Erkennung von Software-Mustern*, University of Paderborn, Master's Thesis, Oktober 2010. – (In German)

[Fow99] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA : Addison-Wesley, 1999. – ISBN 0–201–48567–2

[Fuj10] *Fujaba*. November 2010. – `http://www.fujaba.de`

[FZ11] Fontana, Francesca A.; Zanoni, Marco: A tool for design pattern detection and software architecture reconstruction. In: *Information Sciences* 181 (2011), Nr. 7, S. 1306 – 1324. – ISSN 0020–0255

[GHJV95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns*. Addison Wesley, 1995

[GHS09] Giese, Holger; Hildebrandt, Stephan; Seibel, Andreas: Improved Flexibility and Scalability by Interpreting Story Diagrams. In: Magaria, Tiziana (Hrsg.); Padberg, Julia (Hrsg.); Taentzer, Gabriele (Hrsg.): *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)* Bd. 18, Electronic Communications of the EASST, 0 2009

[GSR05] Geiger, Leif; Schneider, Christian; Record, C.: Template- and modelbased code generation for MDA-Tools. In: Giese, Holger (Hrsg.); Zündorf, Albert (Hrsg.): *Proc. of the third International Fujaba Days 2005, Paderborn, Germany* Bd. tr-ri-06-275, University of Paderborn, September 2005, S. 1–6

[HKW+08] Herold, Sebastian; Klus, Holger; Welsch, Yannick; Deiters, Constanze; Rausch, Andreas; Reussner, Ralf; Krogmann, Klaus; Koziolek, Heiko; Mirandola, Raffaela; Hummel, Benjamin; Meisinger, Michael; Pfaller, Christian: CoCoME - The Common Component Modeling Example. In: Rausch, Andreas (Hrsg.); Reussner, Ralf (Hrsg.); Mirandola, Raffaela (Hrsg.); Plášil, František (Hrsg.): *The Common Component Modeling Example* Bd. 5153. Springer Berlin / Heidelberg, 2008. – 10.1007/978-3-540-85289-6_3. – ISBN 978–3–540–85288–9, S. 16–53

[jGo11] *jGoose*. Mai 2011. – `http://sourceforge.net/projects/jgoose/`

[KDW+10] Koziolek, Heiko; Doppelhamer, Jens; Weiss, Roland; Bilich, Carlos; Schlich, Bastian; Skuliber, Ivan; Zemljic, Marijan; Desic, Sasa; Huljenic, Darko; Beltran,Juan Carlos Flores: Project Delivarable D7.3 - Demonstrator Results Documentation. 2010. – Forschungsbericht. http://www.q-impress.eu/wordpress/wp-content/uploads/2010/10/D7.3-Demonstrator_Results_Documentation.pdf

[Kla09] Klatt, Benjamin: Software Model eXtractor (SoMoX). study thesis. In: *Universität Karlsruhe (TH), Software Design and Quality Group*, 2009

[Kos02] Koschke, Rainer: Atomic architectural component recovery for program understanding and evolution. In: *Proceedings of the International Conference on Software Maintenance*, IEEE, 2002. – ISSN 1063–6773, S. 478 – 481

[Kos05] Koschke, Rainer: Rekonstruktion von Software-Architekturen. In: *Informatik - Forschung und Entwicklung* 19 (2005), Nr. 3, S. 127–140. – 10.1007/s00450-005-0180-1. – ISSN 0178–3564

[Kro10] Krogmann, Klaus: *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Dissertation, 2010

[KSB+11] Koziolek, Heiko; Schlich, Bastian; Bilich, Carlos; Weiss, Roland; Becker, Steffen; Krogmann, Klaus; Trifu, Mircea; Mirandola, Raffaela; Martens, Anne: An Industrial Case Study on Quality Impact Prediction for Evolving Service-Oriented Software. In: *Proc. 33rd ACM/IEEE Int. Conf. on Software Engineering (ICSE'11) Software Engineering in Practice Track*, ACM, May 2011

[KSRP99] Keller, Rudolf K.; Schauer, Reinhard; Robitaille, Sébastien; Pagé, Patrick: Pattern-Based Reverse-Engineering of Design Components. In: *Proc. of the 21st International Conference on Software Engineering*, IEEE Computer Society Press, Mai 1999, S. 226–235

[Mar94] Martin, Robert. *OO Design Quality Metrics - An Analysis of Dependencies*. 1994

[MB04] Maqbool, O.; Babri, H.A.: The weighted combined algorithm: a linkage algorithm for software clustering. In: *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, 2004. – ISSN 1534–5351, S. 15 – 24

[MGDLM10] Moha, Naouel; Guéhéneuc, Yann-Gaël; Duchien, Laurence; Le Meur, Anne-Françoise: DECOR: A Method for the Specification and Detection of Code and Design Smells. In: *IEEE Trans. Softw. Eng.* 36 (2010), January, S. 20–36. – ISSN 0098–5589

[MOTU93] Müller, Hausi A.; Orgun, Mehmet A.; Tilley, Scott R.; Uhl, James S.: A reverse-engineering approach to subsystem structure identification. In: *Journal of Software Maintenance: Research and Practice* 5 (1993), Nr. 4, S. 181–204. – ISSN 1096–908X

[Mun05] Munro, Matthew J.: Product Metrics for Automatic Identification of "Bad Smell"Design Problems in Java Source-Code. In: *Proceedings of the 11th IEEE International Symposium on Software Metrics*, IEEE Computer Society, September 2005. – ISBN 0–7695–2371–4, S. 9–17

[Nie04] Niere, Jörg: *Inkrementelle Entwurfsmustererkennung*, University of Paderborn, Paderborn, Germany, Dissertation, 2004. – In German

[NSW+02] Niere, Jörg; Schäfer, Wilhelm; Wadsack, Jörg P.; Wendehals, Lothar; Welsh, Jim: Towards Pattern-Based Design Recovery. In: *Proceedings of the 24th International Conference on Software Engineering*, ACM Press, Mai 2002, S. 338–348

[OMG06] OMG: *OCL 2.0*, 2006. – OMG doc. ptc/06-05-01

[Par94] Parnas, David L.: Software Aging. In: *Proceedings of the 16th International Conference on Software Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1994. – ISBN 0–8186–5855–X, S. 279–287

[QBe10] *QBench*. November 2010. – `http://www.qbench.de`

[qim10] *Q-Impress*. November 2010. – `http://www.q-impress.eu/wordpress/`

[Ree79] Reenskaug, Trygve. *Model-View-Controller - Origins*. 1979

[SA411] *SA4J*. Mai 2011. – `http://www.alphaworks.ibm.com/tech/sa4j`

[SAM11] *Project Deliverable 2.1 - Service Architecture Meta-Model (SAMM)*. April 2011. – `http://www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service_architecture_meta-model.pdf`

[Sar03] Sartipi, Kamran: Software Architecture Recovery based on Pattern Matching. In: *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, IEEE Computer Society, 2003. – ISBN 0–7695–1905–9, S. 293–296

[SG96] Shaw, Mary; Garlan, David: *Software architecture - Perspectives on an emerging discipline*. Prentice Hall, 1996. – I–XXI, 1–262 S. – ISBN 978–0–13–182957–2

[SLT06] Salehie, M.; Li, S.; Tahvildari, Ladan: A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In: *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*, IEEE, 2006, S. 159 –168

[Som92] Sommerville, Ian: *Software Engineering*. 4^{th}. Addison Wesley, May 1992. – ISBN 0321210263

[Son11] *Sonargraph Product Family*. Mai 2011. – `https://www.hello2morrow.com/products/sonargraph`

[SSM06] Seng, Olaf; Simon, Frank; Mohaupt, Thomas: *Code Quality Management*. dpunkt Verlag, Heidelberg, 2006. – ISBN 978–3898643887

[Szy02] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming - Second Edition*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0201745720

[TDB11] Travkin, Oleg; von Detten, Markus; Becker, Steffen: Towards the Combination of Clustering-based and Pattern-based Reverse Engineering Approaches. In: *Proceedings of the 3rd Workshop of the GI Working Group L2S2 - Design for Future 2011*, 2011

[Tra06] Travkin, Dietrich: *Bewertung automatisch erkannter Instanzen von Software-Mustern*. Germany, University of Paderborn, Diploma Thesis, 2006. – In German

[TS04] Trifu, Mircea; Szulman, Peter. *Language Independent Abstract Metamodel for Quality Analysis and Improvement of OO Systems*. 2004

[TSG04] Trifu, Adrian; Seng, Olaf; Genssler, Thomas: Automated Design Flaw Correction in Object-Oriented Systems. In: *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR 2004)*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2107–X, S. 174–183

[Wen07] Wendehals, L.: *Struktur- und verhaltensbasierte Entwurfsmustererkennung*, University of Paderborn, Paderborn, Germany, Dissertation, 2007

[Zün01] ZÜNDORF, A. *Rigorous Object Oriented Software Development*. Habilitation Thesis, University of Paderborn. 2001