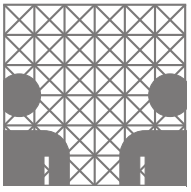
Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften



Im Rahmen des Seminars zu

Software Reengineering (SRE)

Analyse des Papers „Archimetrix - Improved Software Architecture Recovery in the Presence of Design Deficiencies“

Tim Krämer

E-Mail: 7kraemer@informatik.uni-hamburg.de
Matr.-Nr.: 5945287
Studiengang: M.Sc. Informatik

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung befasst sich mit dem Paper „Archimetrix - Improved Software Architecture Recovery in the Presence of Design Deficiencies“, das im Rahmen der 16. „European Conference on Software Maintenance and Reengineering“ im Jahr 2012 veröffentlicht wurde. [Platenius et al., 2012] Die vorliegende wissenschaftliche Arbeit ist in englischer Sprache verfasst und zum Teil bei der „Deutschen Forschungsgruppe“ (DFG) in Zusammenarbeit mit dem Sonderforschungsbereich „On-The-Fly Computing“ (CRC 901) entstanden. [SFB901, 2013]

Im Folgenden werden zunächst die Autoren vorgestellt und daraufhin der Aufbau und die Struktur der Arbeit dargelegt. Im dritten Kapitel soll auf die Hauptbeiträge der Arbeit eingegangen werden. Hierbei werden besondere Aspekte wie Motivation und Ziele hervorgehoben und die Ergebnisse präsentiert.

2 Übersicht: Aufbau, Struktur und Autoren

Auf insgesamt 10 Seiten, inklusive Abstract und Literaturverzeichnis, stellen die Autoren Marie C. Platenius, Markus von Detten und Steffen Becker das „Archimetrix“-Projekt vor. Alle Autoren sind Mitarbeiter der Fachgruppe Softwaretechnik am Heinz Nixdorf Institut der Universität Paderborn. Sie sind Mitarbeiter im SFB 901 „On-The-Fly Computing“. Platenius und von Detten erreichten ihren Master Abschluss im Bereich Informatik an der Universität Paderborn (vgl. [von Detten, 2013] u. [Platenius, 2013]), Steffen Becker ist Juniorprofessor in der Fachgruppe Softwaretechnik. [Becker, 2013]

Das Paper macht mit einem kurzen und prägnanten Abstract auf den weiteren Inhalt neugierig, indem die vorherige Situation im Bereich des „reverse engineering of component-based systems“ beschrieben und auf die fehlenden Möglichkeiten der Mängelbeseitigung (deficiency removal) hingewiesen wird. Weiter sei es mit dem „Archimetrix“-Ansatz möglich einige der Mängel zu erkennen und zu beseitigen, welches anhand des CoCoME (Common Component Modeling Example) getestet worden sei und zu vergleichbaren Verbesserungen an der so wiederhergestellten Softwarearchitektur geführt habe.

Das Paper öffnet nach dem Abstract mit einer Einleitung und ist in verschiedene Abschnitte strukturiert. Nach der Einleitung werden die Grundlagen der benutzten Reverseengineeringmethoden erklärt und der Ablauf des „Archimetrix“-Ansatzes anhand von Beispielen illustriert.

Abschnitt IV beschreibt detailliert die Erweiterungen des bisher benutzen Reeingineeringprozesses.

Die nächsten drei Abschnitte erklären die, in der Einleitung erwähnten, drei Schritte zur Verbesserung des bisherigen Verfahrens. So wird in Abschnitt V die Erkennnung der mängelempfindlichsten Kompententen (most eficiency-sensitive components) veranschaulicht. Abschnitt VI behandelt das Ranking von Designmängeln (design deficiencies). Und Abschnitt VII zeigt, wie die Vorschau der Korrekturauswirkungen dieser Mängel erstellt wird.

Die Erkenntnisse, die schließlich aus den Test mit dem CoCoME gewonnen wurden, werden in Abschnitt VIII präsentiert.

Abschnitt IX und X benutzen die Autoren um die Aussichten und Grenzen darzulegen und ihre Arbeit zu anderen wissenschaftlichen Publikationen einzuordnen.

3 Hauptbeiträge und Ergebnisse des Papers

An dieser Stelle soll der Inhalt des Papers zusammengefasst und anschließend die Ergebnisse evaluiert werden. Im Folgenden wird der Inhalt des Papers in den drei wesentlichen Bereichen Grundlagen und Analyse, Verfahren des „Archimetrix"-Ansatzes und Ergebnisse und Diskussion wiedergegeben. Es folgt eine kritische Betrachtung der Stärken und Schwächen dieser Arbeit unter Betrachtung der Einordnung zur wissenschaftlichen Community. Hier stehen Aspekte wie Zitationen, aufbauende Arbeiten und weitere Arbeiten der Autoren im Vordergrund.

3.1 Grundlagen und Analyse

In der Einleitung weisen die Autoren zunächst auf die Notwendigkeit von aktuellen und zutreffenden Modellen von komplexen Softwaresystemen hin. Sie erklären, dass viele der existierenden Ansätze diese Modelle aus Quellcode bestehender Software zu generieren, bestehende Mängel in der Software nicht so sicher erkennen können, wie Entwickler dazu in der Lage sind. Konsequenterweise wird direkt angemerkt, dass „clustering tools" diese Arbeit wesentlich schneller erledigen können. Auf Grund dieser beiden Aussagen schließen die Autoren, dass ein kombinierter Ansatz (combined reverse and reengineering) benötigt wird mit dem Designmängel automatisch erkannt werden. Diese sollen dem Entwickler sinnvoll vorgelegt und während des Prozesses mit geeigneten Beseitigungsstrategien behandelt werden und somit in einem besseren Architekturmodell resultieren.

Außerdem wird die, dem „Archimetrix"-Ansatz zu Grunde liegende, Reengineeringmethode vorgestellt, die im wesentlichen den bereits bestehenden Prozess aus „Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems" [von Detten and Becker, 2011] um drei Schritte erweitert. Die Schritte beinhalten das Identifizieren von lohnenswerten Komponenten, Ranking der Mängel in den identifizierten Komponenten, und Vorschau der Auswirkungen der Korrektur dieser Mängel. Um eine Vergleichbarkeit der resultierenden Architekturmodelle zu erreichen benutzen die Autoren die Referenz Implementation des „Common Component Modeling Example" [Rausch et al., 2008].

Im Abschnitt II gehen die Autoren zunächst auf kompententenbasierende Softwarearchitekturen ein und erklären „Clustering-Based Reverse Engineering" am Beispiel des Tools „SoMoX"[Krogmann, 2012]. Schritt 1 des „Archimetrix"-Prozesses benutzt „SoMoX", dass eine Kombination von Metriken benutzt um heuristisch ein Modell der Softwarekomponenten zu erstellen. An dieser Stelle wird sehr detailliert auf Verfahrensweise dieser Form des Reverseengineerings eingegangen und Beispiele für Metriken dargelegt. Eine weitere

Grundlage im Abschnitt II sind die Designmängel (Design Deficiencies), die es zu identifizieren und beheben gilt. Am Beispiel von „Interface Violations“ werden verschiedene Mängelbeseitigungsstrategien (removal strategies) vorgestellt.

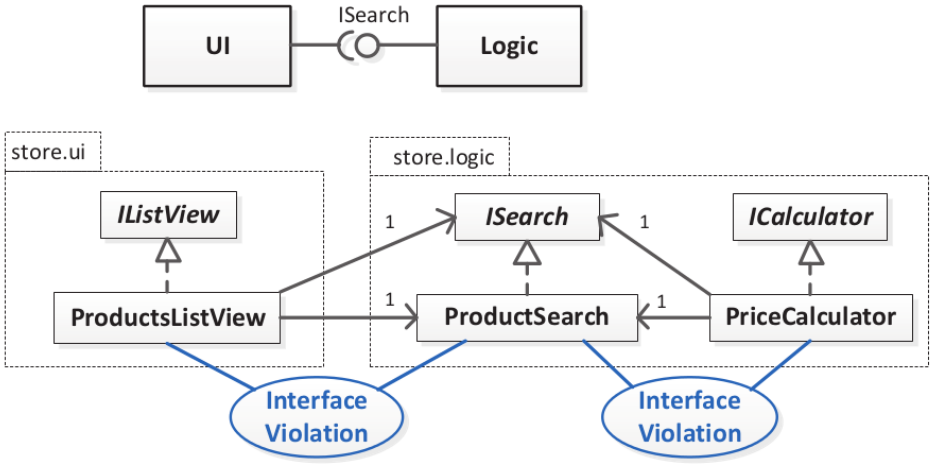


Abbildung 3.1: Einfaches Beispiel einer Architektur

Abbildung 3.1, entnommen aus Abschnitt III des Papers, dient als fortlaufendes Beispiel für die vorliegende Arbeit. Anhand der Abbildung werden zunächst am Beispiel eines Warenhauses (store) zwei vorliegende Interface Violations aufgezeigt (Die Klasse `ProductsListView` greift sowohl auf die Klasse `ProductSearch` sowie auf ihr Interface `ISearch` zu. Genauso wie die Klasse `PriceCalculator` auf `ISearch` und `ProductSearch` zugreift). Der Unterschied der beiden Interface Violations ist dem Menschen durch die Abbildung 3.1 schnell klar; Während die erste Violation paketübergreifend stattfindet und damit unbedingt behoben werden sollte, gehören die Klassen der zweiten Violation wahrscheinlicher zusammen, sodass die Violation hier toleriert werden könnte.

3.2 Verfahren des „Archimetrix“-Ansatzes

Abschnitt IV ist dem eigentlichen Ansatz des Papers gewidmet, so wird zunächst erklärt wie mit Hilfe von drei zusätzlichen Schritten die Nachteile eines automatischen Reverseengineerings mit der Clusteringmethode ausgeglichen werden können.

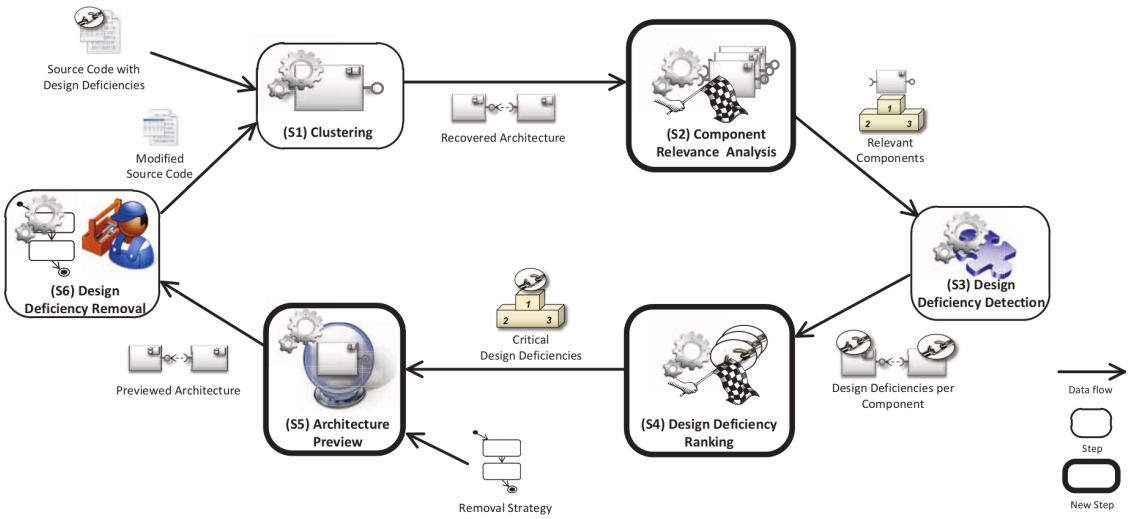


Abbildung 3.2: Erweiterter Reengineeringzyklus

Aufbauend auf der Masterarbeit der Autorin [von Detten and Becker, 2011] zeigt Abbildung 3.2 den resultierenden Reengineeringvorgang, nachdem die Schritte „component relevance analysis“, „design deficiency rating“ und „architecture preview“ hinzugefügt wurden. Mit diesem Verfahren ist es dem Entwickler möglich, die Architektur seminautomatisch aus dem vorliegendem Code zu generieren, indem arbeitsintensive Schritte – wie die Komponentenanalyse – mit Hilfe von automatischen Clusteringmethoden übernommen werden, wichtige Entscheidungen hingegen, die zur Verbesserung der resultierenden Architektur führen, werden dem Entwickler überlassen.

Im Abschnitt V beschreiben die Autoren zunächst den Schritt der sog. „Component Relevance Analysis“, die dem Entwickler erlaubt zu entscheiden, bei welchen Komponenten sich eine Mängelerkennung (deficiency detection) überhaupt lohnt. Relevante Komponenten werden durch eine Komplexitätsmetrik (Complexity Metric) oder eine Angrenzungsmetrik (Closeness to Threshold Metric) identifiziert. Die Komplexitätsmetrik identifiziert komplexe Komponenten. Komponenten die aus vielen Klassen, Attributen, Methoden und Interfaces bestehen sind sehr groß und dementsprechend schwer zu warten und anzupassen. Dadurch erhöht sich auch das Risiko von Interface Violations in diesen Komponenten. Deshalb seien komplexe Komponenten relevanter für die Mängelerkennung als weniger komplexe.

Während des „clustering steps“ (Schritt 1) werden einzelne identifizierte Komponenten anhand eines Grenzwertes (merging value) zusammengeführt. Dieser Wert wird durch unterschiedliche Metriken berechnet und entscheidet ob die Komponentenkandidaten zu einer größeren Komponente zusammengeführt werden sollten. Liegt der Wert unter einer bestimmten Grenze wird keine neue Komponente erschaffen, liegt er darüber werden die Komponenten zusammengeführt.

Besonders relevant für diese Metrik sind ggf. vorhandene Interface Violations, die dazu führen, dass die Metriken einen höheren Grenzwert berechnen und durch damit

verbundene Zusammenführungen in größeren Komponenten und einer schlechteren Architektur resultieren. Besonders relevant für die Designmängelbeseitigung sind also die Komponentenkandidaten bei denen der errechnete Wert nahe dem Grenzwert liegt, da hier eine Mängelbeseitigung im Endeffekt in einer besseren Architektur resultieren kann.

Die Autoren wählen in diesem Abschnitt den Ansatz, dass bei der Relevanzentscheidung beide Metriken gleich gewichtet werden. Abbildung 3.3 zeigt das Ergebnis einer Relevanzanalyse als Koordinatensystem.

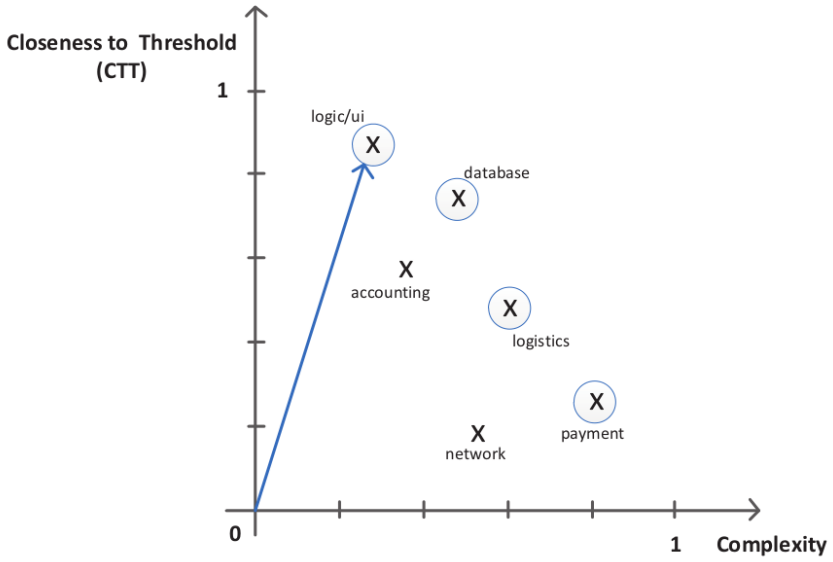


Abbildung 3.3: Ergebnis der Relevanzanalyse

Grundlage der Analyse ist weiterhin das Beispiel aus Abbildung 3.1. In diesem Fall wurde die `logic/ui` Komponente, nach Auswahl des Pareto-Optimum und auf Basis der geometrischen Distanz, als am meisten relevant identifiziert. Da sich die Identifikationsstrategien Pareto-Optimum und geometrische Distanz problemlos auf viele Dimensionen anwenden lassen, können mit diesem Verfahren beliebig viele Metriken bei der Relevanzentscheidung beachtet werden. Für die konkreten Formeln wird auf die Master-Arbeit der Autorin [Platenius, 2011] verwiesen.

Für das, in der Einleitung bereits erwähnte Ranking der relevanten Komponenten werden im Abschnitt VI des Papers verschiedene Metriken im Detail vorgestellt, die Autoren nutzen hier das Beispiel aus Abbildung 3.1 um die „Class Locations Metric“, „External Accesses Metric“ und „Higher Interface Adherence Metric“ zu veranschaulichen. Am Ende des Abschnitts wird auf die eigentliche Berechnung des Rankings eingegangen, die auch hier wieder mit dem Pareto-Optimum arbeitet und deshalb weiterhin einfach um weitere Metriken erweiterbar ist, wie bereits in Abschnitt V erklärt. Eine beispielhafte Auswertung der mit dieser Methode gefundenen Interface Violations kann in Abbildung 3.4 betrachtet werden.

	Roles				Ranking		Corresponding
	interface	accessedMethod	accessingClass	accessingMethod	Distance	Pareto Opt.	Components
#1	PersistenceContext	getEntityManager	EnterpriseQueryImpl	getQueryEnterpriseById	0.5562	yes	PC 46 & PC 90
#2	PersistenceContext	getEntityManager	EnterpriseQueryImpl	getMeanTimeToDelivery	0.5562	yes	PC 46 & PC 90
#3	PersistenceContext	getEntityManager	StoreQueryImpl	queryAllStockItems	0.4047	no	PC 90
#4	PersistenceContext	getEntityManager	StoreQueryImpl	queryLowStockItems	0.4047	no	PC 90
#5	PersistenceContext	getEntityManager	StoreQueryImpl	queryOrderById	0.4047	no	PC 90
#6	PersistenceContext	getEntityManager	StoreQueryImpl	queryProductById	0.4047	no	PC 90
#7	PersistenceContext	getEntityManager	StoreQueryImpl	getStockItems	0.4047	no	PC 90
#8	PersistenceContext	getEntityManager	StoreQueryImpl	queryStockItemById	0.4047	no	PC 90
#9	PersistenceContext	getEntityManager	StoreQueryImpl	queryStoreById	0.4047	no	PC 90
#10	PersistenceContext	getEntityManager	StoreQueryImpl	queryProducts	0.4047	no	PC 90
#11	PersistenceContext	getEntityManager	StoreQueryImpl	queryStockItem	0.4047	no	PC 90

Abbildung 3.4: In CoCoME erkannte Interface Violations, nach Distanz und Pareto-Optimum sortiert.

In Abbildung 3.4 ist leicht erkennbar, dass die beiden oberen Interface Violations tatsächlich zu einer relevanten Veränderung in der resultieren Architektur führen (`PC 46 & PC 90`).

Nachdem die Autoren nun ausführlich dargelegt haben mit welcher Strategie sie dem Entwickler die relevanten Komponenten zur näheren Untersuchung und Mängelbeseitigung darbieten, wird der semiautomatische Vorgang im Kapitel VII durch eine Vorschau der Korrekturkonsequenzen erweitert. An dieser Stelle wird argumentiert, dass das Ergebnis des Reeingineeringprozesses durch menschliche Entscheidungen verbessert wird. Die Vorschau hilft dem Reengineerer zu beurteilen, wie die ausgewählte Mängelbeseitigungsstrategie die resultierende Architektur beeinflusst und zu entscheiden, ob diese Änderung im Einklang mit seinen Anforderungen steht. Neben den vordefinierten Mängelbeseitigungsstrategien, können die Mängel auch manuell beseitigt werden, wofür „Archimetrix“ jedoch keine Vorschau erstellt.

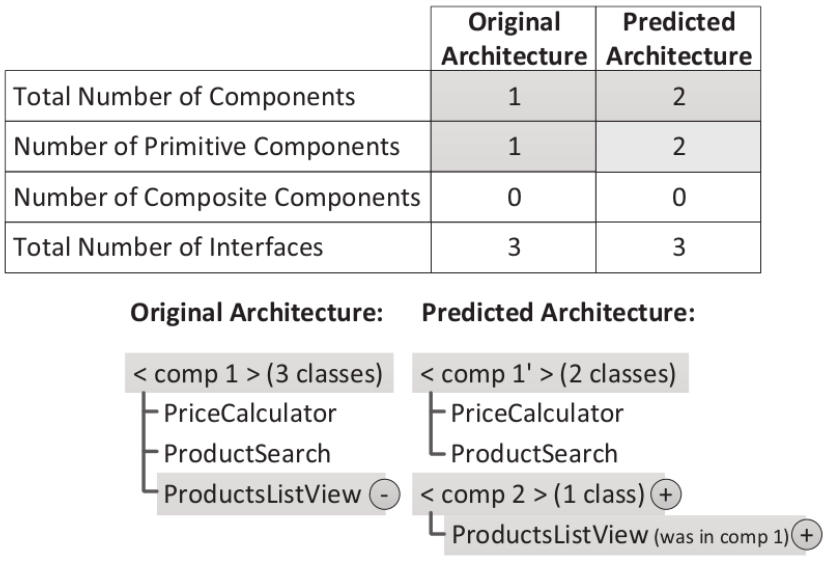


Abbildung 3.5: Beispiel der Architekturvorschau

Um die Unterschiede zwischen der ursprünglichen Architektur und der Vorschau für

den Benutzer zu vereinfachen, werden diese wie in Abbildung 3.5 hervorgehoben. Neben einer umfangreichen Erklärung der Abbildung, stellen die Autoren in diesem Abschnitt in Aussicht, dass die Vorschau der beiden Architekturen in Zukunft als „component diagrams“ visualisiert werden.

3.3 Ergebnisse und Diskussion

Die Autoren implementierten ihren Ansatz in dem Tool „Archimetrix“ und mit Hilfe dieses Tools in der Lage ihren Ansatz anhand der Referenzimplementation des CoCoMEs zu validieren. Die Referenzimplementation wurde manuell erstellt und enthält 127 Klassen mit über 5000 Zeilen Java Code und enthält mehrere Designmängel. Für den Vergleich existiert außerdem die dokumentierte angedachte Architektur mit der die wiederhergestellte Architektur verglichen werden kann. Konkret stellen die Autoren sich dafür drei Fragen:

1. Ist die berechnete Relevanz der im ersten Schritt ausgewählten Komponente ein guter Indikator für die Erkennung von lohnenswerten Designmängeln?
2. Führt das Beheben der Mängel, die im zweiten Schritt ein hohes Ranking erfahren haben, zu Veränderungen in der resultierenden Architektur? Und im Gegensatz, führen die niedrig gerankten zu keiner Veränderung?
3. Ist die resultierende Architektur, nach Behebung der relevanten Designmängel näher an der dokumentierten, angedachten Architektur?

Im Abschnitt VIII dokumentieren die Autoren genau, die Ergebnisse nach den einzelnen Schritten ihres Verfahrens und beantworten so nach- und nach die gestellten Fragen. Sie kommen mit einer ausführlichen Erklärung zu dem Schluss, dass das Ranking der Designmängel (design deficiency ranking) tatsächlich eine Unterstützung für den Reengineerer in dieser Situtation bietet. Abbildung 3.6 zeigt die, durch clustering generierte Architektur des CoCoMEs. Für weitere Details wird an dieser Stelle auf die Master-Arbeit der Autorin [Platenius, 2011] verwiesen.

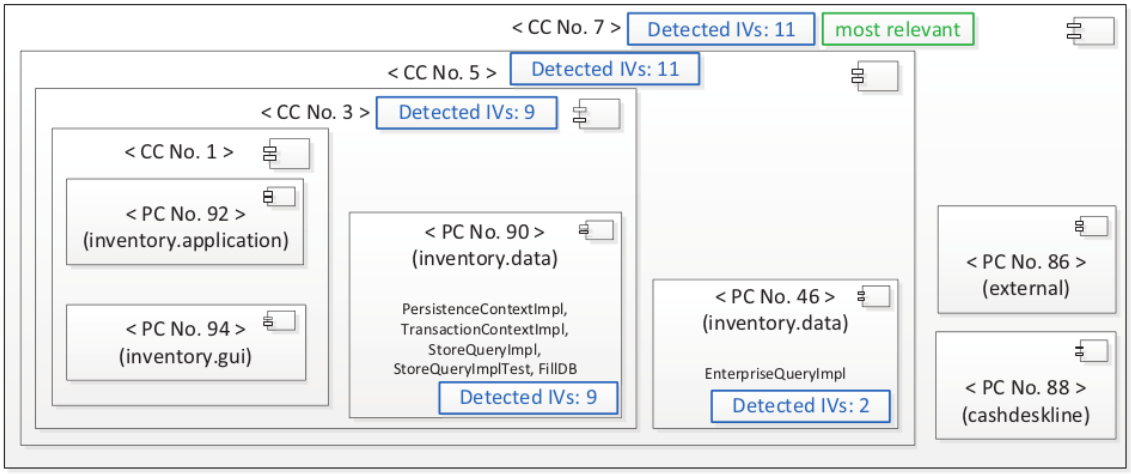


Abbildung 3.6: Konkrete resultierende Architektur nach einem Clusteringdurchlauf des CoCoME.

Im Folgenden beinhaltet das vorliegende Paper eine umfassende Diskussion und Verweise auf verwandte Veröffentlichungen. Die Autoren legen Wert darauf, dass ihr Paper als Grundlage für weitere Fragestellungen und Werkzeuge dient und weisen auf Einschränkungen und zukünftige Erweiterungsmöglichkeiten hin. Insgesamt denken sie, dass ihre Arbeit vielversprechende Ergebnisse geliefert hat und Möglichkeiten für zukünftige Erweiterungen bietet.

4 Kritische Betrachtung

Im folgenden Abschnitt sollen die Struktur und der Sprachgebrauch des vorliegenden Papers betrachtet werden. Die Struktur des Papers beinhaltet dabei u.a. Kriterien wie Gliederung, Referenzen und Vollständigkeit. Im Bereich Sprachgebrauch soll dann auf das Detail-Niveau und die Lesbarkeit des Papers eingegangen werden. Auch der Bezug zur wissenschaftlichen Community und verwandte Veröffentlichungen soll hier beachtet werden.

Abschließend wird die Aussage des Papers im Rahmen einer Zusammenfassung wiedergegeben.

4.1 Struktur und Sprachgebrauch

4.1.1 Abstract

Im Abstract erläutern die Autoren kurz den Sachverhalt und die Idee ihrer Arbeit. Schnell wird der Leser motiviert und die Relevanz des Themas dargelegt.

4.1.2 Formale Aspekte

Die Arbeit ist durchgängig verständlich geschrieben und überzeugt durch einen stets vorhandenen „roten Faden“. Der Text überzeugt mit wenigen Ausnahmen durch kurze und klare Sätze. Das Vokabular ist einer wissenschaftlichen Arbeit angemessen gewählt und vermittelt dem Leser schnell den benötigten Wortschaft dieses Fachgebiets.

4.1.3 Gliederung / Strukturierung

Die Einleitung gibt einen guten Überblick über die Gliederung des Papers. Die einzelnen Abschnitte sind treffend benannt und wurden mit Hilfe von Unterabschnitten und Abbildungen in gut lesbare Blöcke unterteilt. Positiv fällt die ausführliche Diskussion und die Einordnung zu anderen wissenschaftlichen Arbeiten auf. Abstract und Zusammenfassung sind angenehm kurz und vermitteln dem überfliegendem Leser die Ergebnisse in kurzer Zeit. Die sehr detaillierten Schritte zum Ergebnis hingegen wirken etwas zu detailliert und sind teilweise nur unter höchster Konzentration nachvollziehbar. Hilfreich dabei sind die zahlreichen Abbildungen, durch deren Visualisierung dem Leser einige Sachverhalte erst klar werden. Die Ergebnisse werden in angemessenem Umfang diskutiert und Limitierungen, sowie Erweiterungsmöglichkeiten aufgezeigt.

5 Zusammenfassung

Das vorliegende Paper stellt einen verbesserten Ansatz für das Reengineering von komponentenbasierter Software in Form des Tools „Archimetrix“ vor.

Während des Reengineeringvorgangs werden Kompententen mit wichtigen Designmängeln identifiziert und nach Einfluss auf die resultierende Architektur bewertet. Damit wird die Anwendung von vordefinierten Mängelbeseitigungsstrategien (pre-defined deficiency removal strategies) unterstützt und eine Vorschau für die resultierenden Änderungen zur Verfügung gestellt. Diese Schritte führen zu einer signifikanten Verbesserung des Reverseengineerings von Softwarearchitekturen, was die Autoren Anhang der „Common Component Modeling Example Application“ demonstrieren.

Der hier vorgestellte Ansatz unterstützt Reengineerer bei der Generierung von Softwarearchitekturen, indem es eine Methode zur Designmängelbeseitigung schon während der Suche nach vorhandenen Architekturen zu Verfügung stellt.

Zukünftig soll eine Verbesserung des Tools durch neue Metriken zur Erkennung von relevanten Designmängeln und eine verbesserte Visualisierung der Architekturvorschau erreicht werden.

Literaturverzeichnis

[Becker, 2013] Becker, S. (2013). Lebenslauf: Jun.-Prof. Dr.-Ing. Steffen Becker. http://www.hni.uni-paderborn.de/swt/mitarbeiter/130148509900101/. [Online; Zugriff am 28. April 2013].

[Krogmann, 2012] Krogmann, K. (2012). *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*, volume 4. KIT Scientific Publishing.

[Platenius, 2011] Platenius, M. C. (2011). Reengineering of design deficiencies in component-based software architectures. Master's thesis, University of Paderborn.

[Platenius, 2013] Platenius, M. C. (2013). Lebenslauf: Marie Christin Platenius. http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/personen/marie-christin-platenius/lebenslauf.html. [Online; Zugriff am 28. April 2013].

[Platenius et al., 2012] Platenius, M. C., von Detten, M., and Becker, S. (2012). Archimetrix: Improved software architecture recovery in the presence of design deficiencies. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on Software Maintenance and Reengineering*, pages 255–264. IEEE.

[Rausch et al., 2008] Rausch, A., Reussner, R. H., Mirandola, R., and Plasil, F. (2008). *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153. Springer.

[SFB901, 2013] SFB901 (2013). Website der SFB901 der Universität Paderborn. http://sfb901.uni-paderborn.de. [Online; Zugriff am 29. April 2013].

[von Detten, 2013] von Detten, M. (2013). Lebenslauf: Dr. Markus von Detten. http://www.hni.uni-paderborn.de/swt/mitarbeiter/ehemalige/130155581000101/. [Online; Zugriff am 28. April 2013].

[von Detten and Becker, 2011] von Detten, M. and Becker, S. (2011). Combining clustering and pattern detection for the reengineering of component-based software systems. In *Proceedings of the joint ACM SIGSOFT conference-QoSA and ACM SIGSOFT symposium-ISARCS on Quality of software architectures-QoSA and architecting critical systems-ISARCS, QoSA-ISARCS*, volume 11.

Abbildungsverzeichnis

Alle Abbildungen und Tabellen sind aus [Platenius et al., 2012] entnommen.