7 Grundlagen der Evolution von Software-Architekturen

Holger Krahn, Bernhard Rumpe

In diesem Kapitel wird die *Evolution von Software-Architekturen im Kleinen* vorgestellt. Diese Form der Evolution nutzt sehr kleine, effiziente und schnell ausführbare, systematische Schritte zur Adaption der vorhandenen Architektur eines Systems. Die Architektur wird dabei so adaptiert, dass diese sich besser für die Erweiterung um neue Funktionalität eignet. Dabei bleibt das migrierte System jederzeit lauffähig und kann so sowohl weiter im Produktiveinsatz verbleiben, aber wichtiger noch permanenten Tests unterzogen werden.

Wesentliches Kernelement einer solchen Migration, wie sie später in Kapitel 9 für Architekturen im Großen diskutiert wird, ist eine erste Phase, in der ein Reengineering des Syystems stattfindet, das die Rekonstruktion wesentlicher Merkmale der Architektur in einer meist deutlich abstrahierten und grafisch aufbereiteten Form beinhaltet. Die bei einem Reengineering wiedergewonnene Architekturbeschreibung bildet die wesentliche Grundlage für die Planung der Migration. Auch eine »sanfte« Migration ist noch eine relativ große Aufgabe, da mit ihr zumindest komplette Teilsysteme ausgetauscht werden.

Dieses Kapitel ist wie folgt gegliedert:

- In Abschnitt 7.1 wird die grundlegende Vorgehensweise motiviert und die Vor- und Nachteile sowie die Anwendbarkeit werden diskutiert.
- Abschnitt 7.2 enthält einen Überblick über vorhandene und im Entstehen begriffene Werkzeuge und Techniken, die die Evolution im Kleinen unterstützen.
- Abschnitt 7.3 gibt eine Übersicht über Ansätze der Evolution, die oft unter dem Stichwort »Refactoring« vorgestellt werden.
- Ein komplexeres Beispiel für Refactoring-Techniken auf Basis eines Architekturstils für verteilte, asynchron kommunizierende Systeme wird in Abschnitt 7.4 vorgestellt.

- Die offenen Werkzeug- und Forschungsthemen sowie mögliche Wege zum schnellen Einsatz ausgesuchter Evolutionstechniken werden in Abschnitt 7.5 diskutiert.
  Eine mathematische Präzisierung der in Abschnitt 7.4 behandelten Refactoring-Techniken befindet sich in [KR05].

7.1 Grundlegende Motivation zur Evolution

Die Architektur ist über die Lebenszeit – vom Beginn der Entwicklung bis zum letztmaligen Start – eines Software-Systems typischerweise das stabilste Element. Dies gilt sowohl für die Hardware-Architektur als auch für die logische Architektur der Software-Komponenten. Die Software-Architektur wird frühzeitig geplant, um dann Funktionalitäten und Datenstrukturen des Systems in diese Architektur einzupassen. Entsprechend schwierig und teuer ist es bereits in der Entwicklungsphase, eine Architektur nachträglich zu ändern, insbesondere wenn diese Anpassung ad hoc und ohne geeignete konzeptuelle und werkzeugtechnische Grundlage vorgenommen wird. Oft werden zum Beispiel durch Testen erreichte Qualitätsstände bei einer Architekturanpassung leichtfertig verspielt und die Fehlerraten steigen zunächst deutlich an.

Gerade weil es so kostspielig ist, Software-Architekturen zu modifizieren, werden diese häufig nicht angepasst und stattdessen neue Funktionalität in nicht optimaler Form hinzugefügt. Dadurch entstehen Software-Systeme mit dem »Huckepack«-Syndrom:

- Viele Funktionalitäten wurden nachträglich, oft unter Umgehung der vorgesehenen, aber nicht ganz adäquaten Schnittstellen und durch Verletzung von Datenkapselungen, dem System hinzugefügt,
- Code wird teilweise nicht mehr genutzt oder ist in ähnlicher Form doppelt vorhanden,
- Code ist nicht in der Kompaktheit und für die Effizienz formuliert, wie es möglich wäre.

Die Wartbarkeit beziehungsweise Weiterentwickelbarkeit von Huckepack-Systemen nimmt mit einer bestimmten Komplexitätsstufe rapide ab und die Systeme müssen früher oder später durch neue Implementierungen abgelöst werden. Systeme können aber durch systematische Anpassung der Architektur für neue Funktionalität und im Gegenzug auch durch konsequente Entfernung unnützen Codes und eine daraus folgende Vereinfachung der Architektur dauerhaft wartbar bleiben und damit ihre Entwicklungskosten wesentlich besser amortisieren.

Neuere Untersuchungen [Gla04] zeigen, dass bereits über 40% der Gesamtkosten eines Software-Systems auf die Wartung entfallen. Dabei dient ein Großteil der als Wartung bezeichneten Tätigkeiten eigentlich der Software-Evolution. Es

ist zu erwarten, dass der Anteil der Evolution im Software-Lebenszyklus weiter steigt [Gla98].

Ein weiterer Vorteil einer verbesserten Evolution von Software-Architekturen besteht darin, dass bei der Planung der Architektur nicht alle eventuell auftretenden Erweiterungen bereits von Anfang an zu berücksichtigen sind. Gerade die Auslegung einer Architektur auf zukünftige potenzielle Erweiterungen führt zu einer wesentlichen Steigerung der Architekturkomplexität, die dann potenziert in die Komplexität der Realisierung von Funktionen eingeht, auch wenn diese Funktionen von den geplanten Erweiterungen unabhängig sind. Agile Vorgehensweisen [Bec03, Coc02] fordern deshalb einen Fokus auf möglichst einfache Architekturen, die dann durch so genannte »Refactoring«-Techniken erweitert und angepasst werden sollen. Refactoring [Fow00, Opd92] wird dort auf Modellebene diskutiert, aber letztendlich auf Ebene des Programmcodes durchgeführt. Dadurch steht keine bzw. nur eine geringe Abstraktion zur Verfügung und es ist oft schwer, die architekturelle Essenz auf dieser Detailebene zu erkennen. In diesem Kapitel wird deshalb über die reine Codeebene hinaus beschrieben, welche Möglichkeiten zur Evolution von Architekturmodellen existieren.

7.1.1 Methode

Der konsequente Einsatz von kleinen und systematischen Schritten zur Evolution einer Software-Architektur ist vor allem dann sinnvoll, wenn diese Technik in eine Entwicklungsmethode eingebettet ist. Die Evolution in kleinen Schritten kann bei entsprechender Werkzeugunterstützung bedeuten, dass die Architekturevolution binnen einer Stunde bis hin zu wenigen Tagen durchgeführt werden kann. In diesem Fall ist eine langwierige Planung der Evolution unnötig oder es kann sogar auf die Anwendung eines Reverse-Engineering-Ansatzes verzichtet werden. Sind die Entwickler in der Lage, Änderungen an einer Architektur relativ eigenverantwortlich, schnell und selbständig durchzuführen, dann ist es sinnvoll, dass sich die Aktivitäten Entwicklung und Evolution abwechseln und ein Reverse-Engineering ist dann weder sinnvoll noch notwendig.

Der erfolgreiche Einsatz von agilen Methoden [RS02] hat uns gezeigt, dass eine solche evolutionäre Vorgehensweise unter bestimmten Rahmenbedingungen hervorragend funktionieren kann. Die mit Extreme Programming populär gewordenen »Refactoring«-Techniken [Fow00, Opd92] bilden einen erfolgreichen Ansatz zur Evolution, der allerdings im Wesentlichen auf die Codeebene beschränkt ist. Tatsächlich können Transformationstechniken auf Modellebene noch wesentlich besser angewandt werden, weil Modelle generell abstrakter sowie konzeptuell und semantisch reichhaltiger sind. Um eine agile Vorgehensweise zu unterstützen, muss allerdings eine feste Integration der transformierbaren Modelle mit dem Code existieren, die zum Beispiel dadurch hergestellt werden kann, dass das Produktionssystem aus dem abstrakten Architekturmodell und darin integrierten Codefragmenten generiert werden kann und so das Modell quasi als abstrakte Programmiersprache

dient [Rum05]. Abschnitt 7.4 beschreibt ein Beispiel eines solchen Refactoring-Ansatzes auf Modellebene.

Grundsätzlich bleibt bei der Anwendung von transformationellen Evolutionstechniken sowohl auf Code- als auch auf Modellebene die Einbettung in eine agile Vorgehensweise ideal [Rum04]. Wesentliche Kernelemente der agilen Vorgehensweise sind dabei:

- System und automatisierte Tests werden zur Qualitätssicherung parallel entwickelt.
- Es kann agil und bedarfsorientiert zwischen den Aktivitäten Entwicklung und Architekturevolution gewechselt werden.
- Die Architekturplanungsphase kann sich zunächst auf eine Kernarchitektur beschränken, die in späteren Iterationsstufen erweiterbar ist.
- Die Planung und Evolution der Architektur erfolgt in Gruppen auf Basis des Architekturmodells, wodurch ein ähnlicher Effekt der Qualitätssicherung wie beim Pair Programming entsteht.
- Die Entwicklung des Systems und die Modifikation der Software-Architektur erfolgt nicht als »Big Bang«, sondern in kleinen, beherrschbaren Iterationen, die durch Feedback mit Hilfe automatisierter Tests und eines Kunden begleitet werden.
- Automatisierte Tests werden zeitnah mit dem System entwickelt und überdecken das Produktionssystem nach bestimmten Kriterien. Die meisten Tests können bei der Evolution der Software-Architektur ohne großen Zusatzaufwand wiederverwendet werden und amortisieren damit den initialen Aufwand zur Erstellung automatisierter Tests relativ schnell.

Insbesondere die Nutzung einer ausführlichen Sammlung automatisierter Tests ist ein in der Praxis nicht zu unterschätzendes Qualitätssicherungsmerkmal. Neben der Entdeckung einer hohen Anzahl von Fehlern bei der Entwicklung und Abnahme des Produktionssystems ist darüber hinaus vor allem auch die Möglichkeit zum Regressionstest essenziell. So kann ohne hohen Aufwand die Sicherstellung der Korrektheit der bereits realisierten Funktionalität innerhalb der modifizierten Software-Architektur erkannt werden. Dies ist besonders wichtig beim Einsatz kleiner, systematischer Transformationsschritte, bei denen idealerweise nach jedem Einzelschritt eine Qualitätssicherung durch automatisierte Tests stattfindet.

Aus dem Zusammenspiel vorhandener Testsammlungen, einer explizit modellierten Architektur, der Generierung bzw. Kompilation des lauffähigen Systems aus diesem Architekturmodell und der Transformationstechniken für Architekturen entsteht eine effektive Methode zur Evolution von Software-Architekturen. Natürlich sind dafür vorhandene Techniken und Werkzeuge Voraussetzung. Deshalb werden nachfolgend die wichtigsten Ansätze in Bezug auf ihre Eignung zur Architekturevolution diskutiert.

7.1.2 Anwendbarkeit der Evolution

Evolutionäre Techniken können nur bei eng umrissenen Rahmenbedingungen sinnvoll angewendet werden. Vorbereitende und begleitende methodische Maßnahmen ermöglichen unter Umständen den Einsatz auch in einem weniger geeigneten Umfeld.

- Die Architektur des Software-Systems muss möglichst explizit gegeben sein. Idealerweise erfolgt dies durch ein Architekturmodell, das eigenständig entwickelt und durch Generierungstechniken in das Produktionssystem überführt wurde sowie mit diesem synchronisiert ist. Um dies zu realisieren, wird entweder der aus einem Modell generierte Code als nicht mehr modifizierbare Zwischenstufe des Compilers betrachtet oder ein »Round-Trip Engineering«-Verfahren erlaubt die Synchronisation beider Sichten auf das System.
- Die Architektur sollte einen gewissen Abstraktionsgrad besitzen, so dass eine Übersichtlichkeit gegeben ist, die es dem Nutzer erlaubt, den Zugang zur Architektur wieder zu gewinnen.
- Das Projekt darf nicht zu groß sein. Zur Parallelisierung der Entwicklungsarbeiten benötigen große Projekte eine relativ detaillierte und stabile Beschreibung der Architektur, um damit Subsysteme und deren Schnittstellen festzulegen. Die Modifikation der Architektur ist daher in solchen Projekten immer ein zeitaufwendiger Konsensprozess mehrerer beteiligter Gruppen. In [JR01] wurde deshalb eine hierarchische Strukturierung von großen agilen Projekten mit einem expliziten Steuerungsteam vorgeschlagen. Dieses Team ist dann verantwortlich für die Planung und Umsetzung von Architekturänderungen, die projektübergreifend sind.
- Manche Modifikationen einer Software-Architektur betreffen den Austausch eines Frameworks, einer Middleware-Plattform oder eines Betriebssystems. Die Durchführung des tatsächlichen Austauschs ist ein großer Schritt, dem durchaus das »Big-Bang«-Syndrom anhaftet: Nach der Migration ist zunächst die Fehlerrate wieder deutlich angestiegen.
  Allerdings lässt sich ein solch großer Schritt des Austauschs einer zugrunde liegenden Technologie evolutionär vorbereiten, so dass eine geeignete Qualitätssicherung auch hier vorgenommen werden kann. Das Upgrade einer Framework- oder Betriebssystemversion ist dabei relativ harmlos, unterliegt aber denselben Prinzipien wie der Austausch einer Kommunikations-Middleware: Durch transformationelle Evolution wird zunächst eine Zwischenschicht zwischen Applikation und auszutauschendem Element eingezogen, die genau die neue Funktionalität anbietet und zunächst auf die alte übersetzt. Ist das neue Element reichhaltiger an Funktionen, so werden Vereinfachungen im Applikationskern möglich. In einem zweiten Schritt wird der Austausch vorgenommen. In der Praxis bleiben leider heute oft die Zwischenschichten erhalten, so dass bei mehrfacher Migration eine komplexe

Schichtung von Adaptern entsteht («Bubble-Effekt«). Aber natürlich ist es möglich, statt eines Umbaus der Applikation mit dem Ziel der Entfernung der Zwischenschicht auch eine Integration der Zwischenschicht in eine erweiterte Applikation vorzunehmen. Abbildung 7.1 skizziert diese Vorgehensweise.

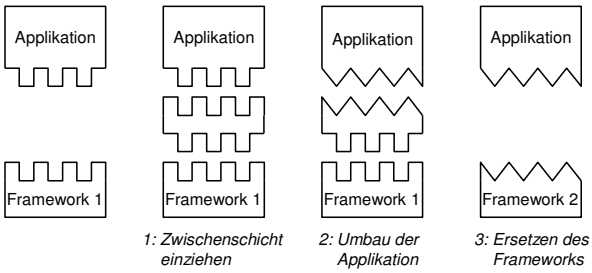


Abbildung 7.1: Migration auf ein aktualisiertes Framework

7.1.3 Domänenspezifische Sprachen

Model Driven Architecture (MDA) [MDA] (vgl. auch Kapitel 6) entwickelt sich derzeit zu einer Technologie, die es erlaubt, Software schneller, effizienter und qualitativ hochwertiger zu erstellen. MDA wird wie viele Themen der Informatik zunächst in seinen kurzfristig verfügbaren Möglichkeiten eventuell überschätzt, besitzt aber in seinem Kern wesentliche, langfristig effizienz- und qualitätssteigernde Konzepte, die bei der Darstellung einer Architektur zu berücksichtigen sind. Die Object Management Group (OMG), die bereits CORBA und UML zur industriellen Einsatzreife gebracht hat, wird schon alleine aufgrund ihrer Dominanz auch dem MDA-Ansatz industrielle Geltung verschaffen.

Tatsächlich gibt es eine Reihe von Transformationen jenseits von Plattformabhängigkeit und UML-nahen Modellen [MDA, Béz05], die teilweise auch bereits im Einsatz sind, zum Beispiel für die Anbindung einer grafischen Oberfläche, der Generierung von Datenstrukturen, der Integration eines Persistenzmechanismus in die Applikation oder die Abbildung von Teilmodellen auf Schnittstellen zur Kommunikation.

Zunächst hat sich die Entwicklung von MDA-Techniken vor allem auf die allgemein gültigen Konzepte zur Darstellung von Modellen und Transformationen konzentriert. Zurzeit entstehen immer mehr MDA-nahe Ansätze, die es erlauben, problemangepasste und damit meist auf eine Domäne spezialisierte Sprachen, so genannte »domänenspezifische Sprachen« (Domain Specific Languages, DSLs) einzusetzen. Beispielhaft sei der Ansatz von Microsoft [Mic, GSCK04] genannt, der es in Zukunft sehr einfach machen wird, DSLs zu definieren. Die domänenspezifischen

Konzepte einer so definierten DSL werden auf ein vorhandenes oder selbst erstelltes Framework abgebildet, das einzeln einsetzbare Wissenskomponenten enthält und durch geeignete vorhandene Schnittstellen entsprechend der vorgegebenen DSL komponiert und konfiguriert werden kann.

Durch diese Form der Spezialisierung verlieren Generierungstechniken an Breite in der Anwendbarkeit, gewinnen aber innerhalb der Anwendungsdomäne sehr viel Potenzial zur Effektivitätssteigerung, da so spezialisierte Frameworks ansprechbar sind.

Der Nutzen von DSLs liegt genau darin, für ein Aufgabengebiet spezialisierte, anwendungsnahe Abstraktionen zur Verfügung zu stellen. Der erfolgreiche Einsatz zeigt eine Effektivitätssteigerung um den Faktor 3-10 [TS01], wenn man von dem durchaus limitierten und nur einmaligen Aufwand, eine DSL zu erlernen, absieht und davon ausgeht, dass DSLs nicht für jedes Projekt neu erstellt werden müssen, sondern zumindest firmenweit wiederverwendbar sind.

Dem unbestreitbar großen Vorteil der Nutzung einer abstrakten DSL zur Spezifikation und gleichzeitigen High-Level-Programmierung steht zunächst aber noch das Problem gegenüber, dass die entstehende Vielfalt an DSL-Techniken einen Vergleich und eine Übersetzung ineinander notwendig macht.

Dennoch sind DSLs speziell zur Darstellung von Software-Architekturen sehr hilfreich, weil sie kompakt und übersichtlich sind. MDA-Transformationstechniken eignen sich sowohl zur Generierung des Produktionscodes aus der Architektur-DSL als auch zur Evolution. Speziell zur Evolution einer in DSL formulierten Software-Architektur können relativ einfache Transformationen eingesetzt werden, wobei durch begleitende Qualitätssicherungsmaßnahmen, wie in Abschnitt 7.1.1 beschrieben, die Qualität der resultierenden Architektur effizient sichergestellt werden kann. In Abschnitt 7.4 wird dies an einer speziell für asynchron kommunizierende Systeme entwickelten DSL demonstriert.

7.2 Ansätze und Konzepte zur Software-Evolution

Die verschiedenen Ansätze und Konzepte zur Software-Evolution haben gemeinsam, dass sie die folgenden drei Aspekte behandeln:

1. Welches Artefakt wird verändert?
2. Welcher Aspekt des Systems soll verändert werden?
3. Welcher Aspekt des Systems soll unverändert bleiben?

Wichtig für das Verständnis der verschiedenen Ansätze ist, dass keiner nur eine Veränderung behandelt, sondern gleichzeitig immer auch dargestellt wird, welche Aspekte unverändert bleiben. Unter anderem wurde im Workshop »Formal Foundations of Software Evolution« [MW01] die Kernthese herausgestellt, dass der Grundsatz »separation of concerns« auch auf Modellebene essenziell ist. Darunter versteht man die Trennung und Modularisierung von verschiedenen Aspekten ei-

nes Software-Systems mit dem Ziel, dass diese möglichst unabhängig voneinander modifizierbar sind. Während auf Quellcodeebene vor allem Aspekte wie Nebenläufigkeit, Verteilung und Persistenz von der Programmlogik zu trennen sind, ist bei der Architekturevolution vor allem darauf zu achten, Teile der Architektur zu trennen, die sich mit unterschiedlichen Geschwindigkeiten verändern. Zum Beispiel zeigen Untersuchungen, dass die Logik einzelner Funktionsblöcke sich schneller verändert als die unterliegende Architektur. Daher ist eine architektonische Trennung sinnvoll. [MW02]

Betrachtet man die Veränderungen eines Software-Systems auf Quellcodeebene, kann die von Weiser [Wei84] eingeführte Technik des »program slicing« eingesetzt werden, um Aussagen über die Auswirkungen einer bestimmten Umstrukturierung zu treffen, was als »impact analysis« bezeichnet wird. Zhao et al. [ZYXX02] übertragen diese Technik auf Modellebene, indem sie eine Variante auf Software-Systeme anwenden, die in der Architekturbeschreibungssprache Wright [All97] beschrieben sind.

Die Evolution von Software-Projekten auf Quellcodeebene wird werkzeugtechnisch dadurch unterstützt, dass Versionsverwaltungen wie CVS eingesetzt werden. Darauf aufbauend lassen sich auch Konfigurationsmanagementsysteme verwenden. Zusätzlich zur Versionsverwaltung existieren hier Varianten eines Artefakts, die je nach Kontext, wie zum Beispiel dem verwendeten Betriebssystem, gewählt werden können. Unter einer Konfiguration versteht man dabei eine Menge von Artefakten, die zusammen ein Gesamtsystem bilden. Roshandel et al. setzen in [RHMRM04] ein Konfigurationswerkzeug ein, um die Evolution einer Architekturbeschreibung zu erfassen. Sie sind dabei nicht auf eine bestimmte Architekturbeschreibungssprache festgelegt, die die Evolution meistens unzureichend unterstützt, sondern fügen den Evolutionsaspekt durch ein zusätzliches System hinzu.

7.3 Refactoring: Evolution im Kleinen

Die Arbeiten von Johnson und Opdyke [JO93, Opd92] und vor allem das Buch von Fowler [Fow00] haben einen großen Einfluss auf die Software-Entwicklung gehabt. Der folgende Satz fasst die Kernaspekte des Refactorings zusammen:

> »Refaktorisieren ist der Prozess, ein Software-System so zu verändern, dass das externe Verhalten nicht geändert wird, der Code aber eine bessere interne Struktur erhält.«
>
> Martin Fowler [Fow00]

Refactoring betrachtet den Quellcode eines Systems als das zu ändernde Artefakt. Dabei soll eine bessere interne Struktur erreicht werden, ohne das externe Verhalten eines Programms zu verändern. Fowler ist sich der Schwierigkeiten bewusst, die Programmstellen zu erkennen, die sich für ein Refactoring eignen, vertritt aber

den Standpunkt, dass »kein System von Metriken die informierte menschliche Intuition [erreicht]« [Fow00]. Daher verwendet er den Begriff »schlechter Geruch«, auch in Bezug auf das betrachtete Artefakt »Code-Smells« genannt. Er führt 22 Anzeichen an, die auf eine schlechte Programmstruktur hindeuten, d.h. Stellen, an denen sich eine Refaktorisierung wahrscheinlich als besonders nützlich erweist. Refactorings werden von Fowler systematisch beschrieben: mit einem Namen, einer kurzen Zusammenfassung, einer Motivation mit den verbundenen Vor- und Nachteilen, der genauen Vorgehensweise und Beispielen zur Anwendung.

Die manuelle Anwendung von Refactorings auf Quellcodeebene ist fehleranfällig und kann daher kontraproduktiv sein. Der »Smalltalk Refactoring Browser« [RBJ97] war das erste Werkzeug, das eine semiautomatische Ausführung erster Refactorings erlaubt. Neuerdings werden diese Techniken immer mehr in integrierten Entwicklungsumgebungen verwendet, die eine professionelle Software-Entwicklung erlauben (z.B. Eclipse, IntelliJ Idea, Code Guide[1]). [MDJ02]

Eine neuere Untersuchung von Tokuda und Batory [TB01] zeigt anhand von zwei Fallstudien, die eine existierende Programmmodifikation aus realen Software-Projekten mittels Refactorings nochmals durchführen, dass dieses Vorgehen die Evolution einer Software beschleunigen kann. Die Autoren schätzen eine Zeitersparnis von Faktor 8-10 und führen dieses vor allem auf einen geringeren Testaufwand und ein einfacheres Programmdesign zurück. Des Weiteren heben sie die Erleichterung beim Ausprobieren neuer Designs hervor, da sich Änderungen schnell und mit weniger Programmier- und Testaufwand realisieren lassen.

Die Idee des Refaktorisierens wurde von vielen Autoren aufgegriffen und weiterentwickelt. In [Ker05] werden Refactorings verwendet, um gezielt so genannte Entwurfsmuster zu erreichen [GHJV04] (vgl. auch Abschnitt 17.3.2). Durch Refaktorisierungstechniken kann eine gebräuchliche und oft angewandte Architektur eines Systems herbeigeführt werden, deren Vor- und Nachteile gut untersucht sind.

In [TDDN00] wurde ein Metamodell entwickelt, das es ermöglicht, Refactorings unabhängig von einer konkreten (objektorientierten) Programmiersprache zu definieren. Das Modell abstrahiert dabei von Unterschieden wie dynamische bzw. statische Bindung und beschreibt die Refactorings deklarativ mit Vor- und Nachbedingungen. In [GSMD03] wird eine ähnliche Beschreibung gewählt, nur dass hier die Unabhängigkeit von der Programmiersprache durch den Einsatz der UML erreicht wird. Die Vor- und Nachbedingungen werden in OCL formuliert und erlauben somit eine Integration in eine vorhandene UML-Werkzeuglandschaft.

Zu dem Ansatz, den Fowler vorschlägt, die Programmstellen, die refaktorisiert werden sollen, intuitiv durch »Code-Smells« zu erkennen, gibt es Alternativen: In [SSL01] werden Refaktorisierungsmöglichkeiten halbautomatisch erkannt und grafisch für den Benutzer dargestellt. In [EM02] wird eine vollautomatische Erkennung von »Code-Smells« in Java-Quellcode dem Nutzer grafisch dargestellt. Demeyer et al. [DDN00] zeigen, wie Metriken eingesetzt werden, um den Einsatz

[1] vgl. www.refactoring.com/tools.html

von Refactorings in Software-Projekten automatisch zu bestimmen und somit genauer untersuchen zu können. Tourwé und Mens [TM03] verwenden Logic-Meta-Programming-Techniken, um Programme auf nötige Umstrukturierungen zu untersuchen.

Duplizierter Code ist der »Code-Smell«, der sich am besten automatisch untersuchen lässt: Ducasse et al. [DRD99] erkennen duplizierten Code unabhängig von der eingesetzten Programmiersprache.

In [HKKI04] untersuchen die Autoren ein einzelnes freies Parsergeneratorprojekt und können etwas über 2% des Quellcodes durch semiautomatisches Entfernen von dupliziertem Code löschen. Der Erfolg der Refactorings wird dabei durch Testläufe demonstriert, die jeweils für eine Grammatik mittels des ursprünglichen und des refaktorisierten Projekts einen Parser generieren lassen und diese vergleichen. Dieser Ansatz verschiebt wie in [SS04] erklärt, den Fokus von der Unveränderthеit des Verhaltens des eigentlichen Projekts hin zu der Unveränderthеit des Verhaltens der erzeugten Parser.

Wie in Abschnitt 7.1.1 beschrieben, können diese Prinzipien der Restrukturierung von Artefakten in der Software-Entwicklung auch auf Modelle angewendet werden. Im Sinne der Software-Evolution auf Modellebene können Refactorings eingesetzt werden, um bestimmte Qualitätseigenschaften einer Software zu verbessern. Dabei stehen vor allem die folgenden Aspekte im Vordergrund [MT04]:

- Erweiterbarkeit
- Modularität
- Wiederverwendbarkeit
- Komplexität
- Wartbarkeit
- Effizienz

Correa und Werner [CW04] geben Refactorings für UML-Klassendiagramme an, die mit OCL-Invarianten versehen sind. Dabei verwenden die Autoren in Anlehnung an die »Code-Smells« von Fowler den Begriff »OCL-Smells«, die einem Entwickler die Identifizierung von OCL-Ausdrücken ermöglicht, die verändert werden sollten. In [TDDN01] werden nicht nur Refaktorisierungsregeln für UML-Klassendiagramme, sondern ebenfalls für Zustandsautomaten angegeben. Die Autoren benutzen dabei sieben Refactorings, die eine Umstrukturierung der Modelle erlauben. Der »Refactoring Browser for UML« [BSF02] ist ein Plug-in für ein UML-Modellierungswerkzeug, das die Refaktorisierung von Klassen-, Zustands- und Aktivitätsdiagrammen ermöglicht.

Die dargelegten Ansätze zur Evolution von Code- oder UML-Modellen zeigen, dass einerseits einiges an Potenzial für diese Technik vorhanden ist, andererseits aber eine integrierte, allgemeine Theorie und ihre technische Umsetzung noch fehlt. Der von der OMG ausgerufene Standardisierungsvorschlag [Gro02] ist ein weiterer

geeigneter Schritt in diese Richtung, bei dem vor allem die explizite Darstellung der Transformationen im Vordergrund steht.

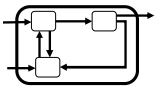
7.4 Software-Evolution am Beispiel eines Geschäftsprozessmodells

Für ein besseres Verständnis der evolutionären Vorgehensweise wird diese anhand eines einfachen Geschäftsprozessmodells veranschaulicht. Die dabei verwendete Notation hat die folgenden grafischen Elemente:

- Komponenten mit expliziten Schnittstellen (Ports)

- Gerichtete Nachrichtenkanäle

- Hierarchische Dekomposition



In der der Modellierungssprache zugrunde liegenden Domäne wird Kommunikation grundsätzlich als Daten- oder Nachrichtenfluss dargestellt. Komponenten kommunizieren nur explizit über diese Datenflüsse und besitzen keine gemeinsame Datenbasis. Dadurch sind Verhalten und Verhaltensänderungen explizit modellierbar. Ein Nachrichtenkanal überträgt unidirektional in der angegebenen Richtung und ist in der Lage, Nachrichten zu puffern; Sender und Empfänger sind deshalb wie bei einem E-Mailsystem relativ entkoppelt. Allerdings wird festgelegt, dass sich Nachrichten in einem Kanal nicht überholen können. Eine formalisierte Beschreibung dieses Ansatzes befindet sich in [KR05].

Eine Firma produziert wie in Abbildung 7.2 vereinfachend beschrieben aus Rohstoffen in der Produktion bestimmte Produkte. Diese Produkte werden durch eine Verkaufsabteilung vertrieben. Das Management der Firma lässt sich stets die aktuellen Produktions- und Verkaufszahlen von den anderen Abteilungen übermitteln, um gemäß dem Grundsatz von Angebot und Nachfrage einen Preis für die Produkte festzulegen und den Produktionsprozess zu steuern.

Da die Produktionszahlen und die Verkäufe dem Management direkt und ungefiltert zugetragen werden, sind für das Management die Planungsvorgänge sehr aufwendig. In Zukunft möchte das Management daher, dass eine tägliche Zusammenfassung für die Entscheidungen ausreicht. Die Entscheidungsgrundlage soll dabei unverändert bleiben und somit auch das für einen Kunden beobachtbare Verhalten in Form der Anzahl der produzierten Waren. Technisch ist diese Un-

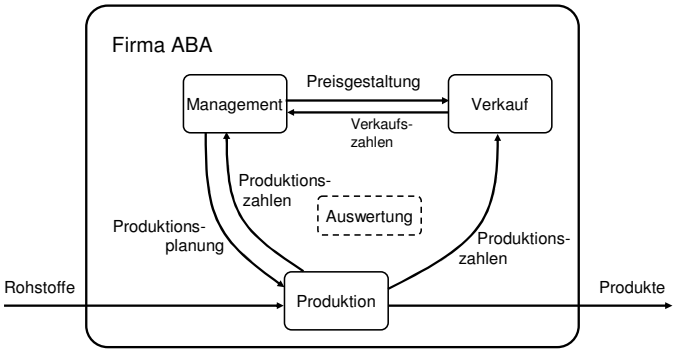


Abbildung 7.2: Die Firma ABA vor der Umstrukturierung

veränderbarkeit der externen Nachrichtenflüsse durch geeignete Invarianten oder später noch diskutierte Verhaltensanforderungen darstellbar.

Ziel ist es, die Verantwortung für die Zusammenfassung und Auswertung der Produktions- und Verkaufszahlen vom Management an eine neue Auswertungsabteilung zu delegieren. Bei der internen Umstrukturierung wird eine Systematik verwendet, die sicherstellt, dass möglichst viele der Anpassungen (Transformationen) derart strukturiert durchgeführt werden, dass sie per Konstruktion korrekt sind. Dies ist leider nicht für alle Transformationen möglich. Deshalb werden nachfolgend weitere Maßnahmen zur Sicherung korrekter Transformationen diskutiert.

Als erster Schritt wird die Komponente »Auswertung« wie aus Abbildung 7.3 ersichtlich ist, neu zur Firma hinzugefügt. Technisch bedeutet das Hinzufügen einer Komponente ohne Anbindung an vorhandene Komponenten durch Nachrichtenflüsse, dass diese Komponente keine Auswirkung auf das Verhalten der Firma hat und die Modifikation daher per Konstruktion richtig ist.

Damit die neue Komponente ihre Funktion erfüllen kann, benötigt sie die erforderlichen Eingabedaten. Sie erhält daher über neu instanzierte Kanäle dieselben Produktions- und Verkaufszahlen wie das Management (vgl. Abbildung 7.4). Da die Komponente keinen Ausgabekanal besitzt, kann weiterhin nichts Negatives passieren.

Durch die Eingangsdaten ist die Auswertungskomponente in der Lage, Berichte für das Management zu erstellen. Diese beinhalten eine Zusammenfassung der Produktions- und Verkaufzahlen und sind deutlich knapper als die bisherigen Informationen.

Das Management erhält nun dieselben Information doppelt: einmal detailliert, sobald ein Produkt produziert bzw. verkauft ist, und zum anderen in regelmäßigen Abständen als Bericht von der Auswertungskomponente. Das Management sollte nun in der Lage sein, seine Entscheidungen ausschließlich durch die Berichte zu treffen, da diese zumindest die für eine Entscheidung notwendigen Informationen

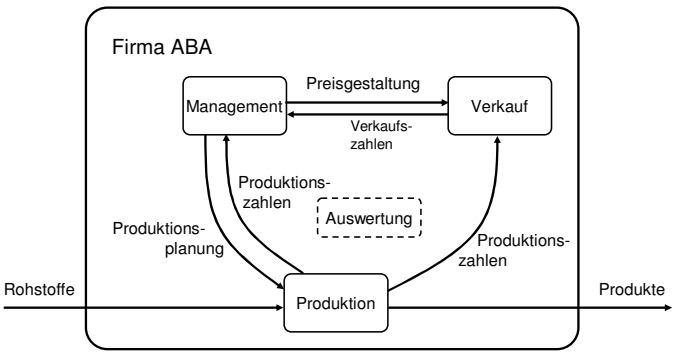


Abbildung 7.3: Komponente Auswertung wird hinzugefügt

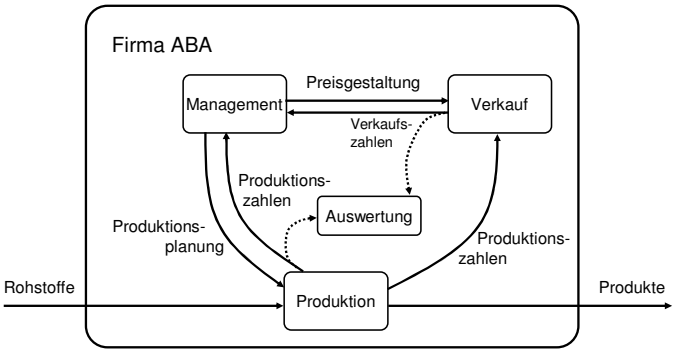


Abbildung 7.4: Auswertung und Management erhalten dieselben Daten

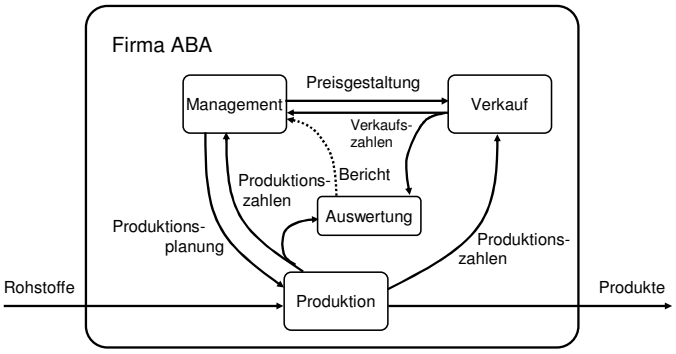


Abbildung 7.5: Auswertung erstellt nun regelmäßig Berichte

enthalten. Daher können die alten Informationsflüsse, wie in Abbildung 7.6 gezeigt, abgeschaltet werden.

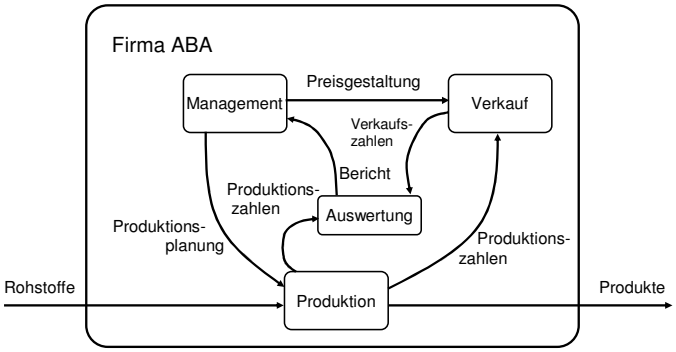


Abbildung 7.6: Das Geschäftprozessmodell nach der Umstrukurierung

Mit der gezeigten Vorgehensweise wurde die Transformation zunächst in mehrere Teilaktionen zerlegt. Der besonders kritische Schritt, die Eingabekanäle des Managements zu wechseln, wurde hier allerdings noch nicht diskutiert. Dazu ist es notwendig, Verhalten und Verhaltensbeschreibungen zu betrachten.

Um die einzelnen Evolutionsschritte besser nachvollziehen zu können, kann eine auf Focus [BS01, Rum96] basierende Formalisierung verwendet werden, die sich besonders gut dazu eignet, Systeme zu beschreiben, die über Nachrichten auf gerichteten Kanälen miteinander kommunizieren. Focus bietet einen Formalismus, der sowohl Verfeinerung von Verhalten und Struktur also auch eine Komposition besitzt. Das Verhalten des Kompositums kann aus den Verhaltensbeschreibungen der Komponenten und der Kompositionsstruktur präzise bestimmt werden und die Verfeinerung einer Komponente führt immer auch zu einer Verfeinerung des Kompositums. Diese für eine präzise Analyse und für die evolutionäre Weiterentwicklung einer Architektur notwendigen Eigenschaften werden allerdings mit dem Preis eines relativ komplexen mathematischen Mechanismus bezahlt. Eine kompakte Darstellung befindet sich in [KR05].

Die präzise Formulierung der Auswirkungen von Transformationen einschließlich der Verhaltensänderung erlaubt eine genauere Analyse der Korrektheitsbedingungen, als dies bei natürlichsprachlichen Beschreibungen möglich ist. So lässt sich beispielweise leicht erkennen, dass eine Validierung einiger Transformationen unnötig ist, da sie per Konstruktion korrekt sind. Die präzise und explizite Formulierung der Transformationen verbessert deren Wiederverwendung. Damit lässt sich ein mit [Fow00] vergleichbarer Katalog von Refactorings erstellen, der Transformationen auf Modellebene anbietet, die wichtige funktionale und architektonische Eigenschaften eines Systems erhalten, während sie ausgesuchte Teile und Komponenten des Systems weiterentwickeln.

7.4.1 Validierung von Refactorings durch Testen

Neben Transformationen, die durch ihre Konstruktion die gewünschten Eigenschaften des Systems erhalten, gibt es Transformationen, bei denen zusätzliche Maßnahmen zur Sicherung der Korrektheit notwendig sind. Neben der formalen Verifikation der Korrektheitsbedingungen von Transformationsanwendungen gibt es den wesentlich praktischeren Ansatz, der auf Tests basiert. Dabei ist natürlich zu beachten, dass Testen nicht die Abwesenheit von Fehlern beweisen kann, sondern nur deren Vorkommen [Dij70]. Dennoch kann eine Testmethode, die Randfälle erfasst und eine hohe Testfallüberdeckung aufweist, praktisch ausreichend viele Fehler ausschließen. Die wesentlichen Vorteile gegenüber formalen Methoden zur Verifikation sind die leichtere Erlernbarkeit der Technik, die deutlich einfachere Anwendbarkeit, die damit verbundene Kostenersparnis und die skalierende Fähigkeit, Tests auch für große Systeme zu erstellen.

Das verwendete Testsystem sollte die folgenden Eigenschaften erfüllen:

- Initialisierung, Ausführung und Testfallüberprüfung sind automatisiert und wiederholbar.
- Das Ergebnis ist deterministisch.
- Es treten keine Seiteneffekte auf.
- Eine gute Abdeckung von Randfällen wird erreicht.

Dieser Ansatz wird im Folgenden an der zentralen Invariante des verwendeten Beispiels demonstriert. Die Intention der Umstrukturierung ist, dass das Management nur noch die regelmäßigen Berichte beachten muss, aber dennoch dieselben Entscheidungen für die Produktion und den Verkauf trifft. Dabei muss das nach außen beobachtbare Verhalten, also die Managemententscheidungen, selbst unverändert bleiben. Das automatisierte Testframework soll also prüfen, ob die alte Komponente M und die neue M‴ dasselbe Verhalten zeigen. Daher werden, wie in Abbildung 7.7 gezeigt, beide Komponenten parallel in das System eingebaut und auf dasselbe Verhalten geprüft.

Die Komponente PreisTest erhält die Preisgestaltung der noch im Betrieb befindlichen alten Managementkomponente und der zu Testzwecken instanziierten neuen Managementkomponente als Eingabe. Diese beiden vergleicht sie und zeigt das Ergebnis eindeutig für den Entwicklern an, was hier durch eine Ampel signalisiert wird. Natürlich ist speziell dieses Beispiel in der Realität unwahrscheinlich, weil kaum zwei unabhängige Managements in einem Unternehmen existieren werden. Wie aber bereits eingangs gesagt, funktioniert diese Architekturbeschreibung genauso gut im Bereich von Hardware und Software, wo eine Replizierung von Komponenten meist wenig aufwändig ist.

Das dynamische Verhalten des Systems ist in Abbildung 7.8 durch ein UML-Sequenzdiagramm dargestellt. Die Produktion und der Verkauf fungieren als Testtreiber, das bedeutet, sie versorgen die Testobjekte mit Daten. Die originale Managementkomponente wird bei der Testauswertung als ein Orakel benutzt. Die

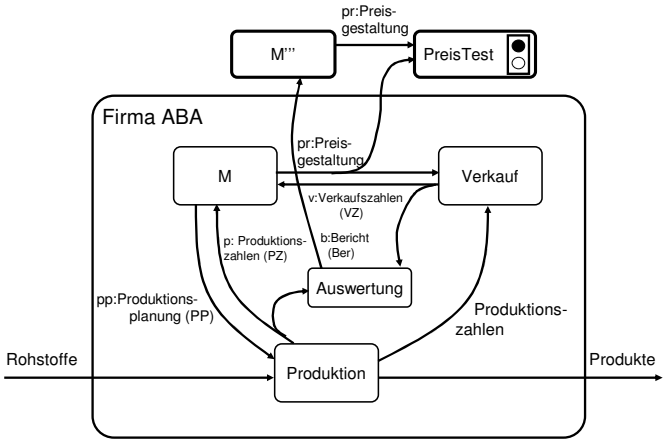


Abbildung 7.7: Erweiterung durch eine Testkomponente

Vergleichskomponente PreisTest vergleicht die Ergebnisse des Orakels mit den Ausgaben der Testobjekte. Im dargestellten Beispiel stimmen diese überein (»Der Preis beträgt 82 TSD Euro«). Die Testbeobachtung (»Täglicher Bericht«) ergibt sich aufgrund nicht dargestellter Werte des Vortags, die im Testtreiber festgelegt werden.

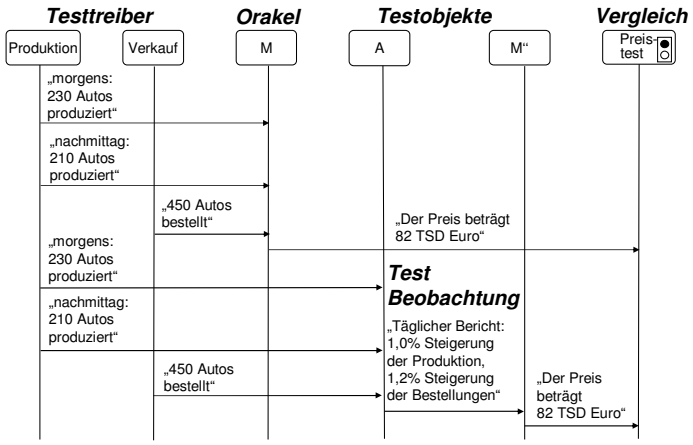


Abbildung 7.8: Der Testablauf als Sequenzdiagramm

7.5 Fazit

Der diesem Kapitel zugrunde liegende Kalkül erlaubt eine Unterspezifikation von Komponenten und lässt sich daher gut in einem agilen Entwicklungsprozess einsetzen. Die Programmierung eines Systems löst diese Unterspezifikation auf, da eine Kodierung eine genaue Spezifikation erfüllt. Somit gibt es viele Möglichkeiten, eine Spezifikation zu implementieren, ein konkretes System beinhaltet aber nur genau eine davon. Bei der Umstrukturierung kann das System so verändert werden, dass eine Implementierung in eine andere geändert wird, die beide die Spezifikation erfüllen. Diese Veränderung des Verhaltens erschwert das Testen, da direkte Vergleiche nicht unbedingt erfolgreich sind. Hierfür ist ein unscharfes Vergleichen und die automatische Erkennung von Äquivalenzklassen hilfreich.

Literaturverzeichnis

[All97] Allen, R.: *A Formal Approach to Software Architecture*. Dissertation, Carnegie Mellon School of Computer Science, Januar 1997. Auch als CMU Technical Report CMU-CS-97-144 erschienen.

[Bec03] Beck, K.: *Extreme Programming : die revolutionäre Methode für Softwareentwicklung in kleinen Teams*. Programmer's choice, Addison-Wesley, 2003, ISBN 3-8273-2139-5. Einheitssachtitel: Extreme programming explained »dt.«

[Béz05] Bézivin, J.: On the Unification Power of Models. In: *Software and Systems Modeling* 4 (2005), Nr. 2, S. 171–188

[BS01] Broy, M.; Stølen, K.: *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer-Verlag, 2001, ISBN 0-387-95073-7

[BSF02] Boger, M.; Sturm, T.; Fragemann, F.: Refactoring Browser for UML. In: Wells und Williams [WW02], S. 77–81

[Coc02] Cockburn, A.: *Agile Software-Entwicklung*. Addison-Wesley, 2002, ISBN 3-8266-1346-5. Einheitssachtitel: Agile Software Development »dt.«

[CW04] Correa, A. L.; Werner, C. M. L.: Applying Refactoring Techniques to UML/OCL Models. In: Baar, T.; Strohmeier, A.; Moreira, A. M. D.; Mellor, S. J., Hrsg., *UML*, Springer-Verlag, 2004, Nr. 3273 in Lecture Notes in Computer Science, ISBN 3-540-23307-5, S. 173–187

[DDN00] Demeyer, S.; Ducasse, S.; Nierstrasz, O.: Finding refactorings via change metrics. In: *OOPSLA*, 2000, S. 166–177

[Deu02] van Deursen, A., Hrsg.: *Proceedings / Ninth Working Conference on Reverse Engineering (WCRE02)*, IEEE Computer Society Press, 2002, ISBN 0-7695-1799-4

[Dij70] Dijkstra, E. W.: Notes on Structured Programming. April 1970, URL `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`. Private Korrespondenz

[DRD99] Ducasse, S.; Rieger, M.; Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In: *ICSM*, 1999, S. 109–118

[EM02] van Emden, E.; Moonen, L.: Java Quality Assurance by Detecting Code Smells. In: van Deursen [Deu02]

[Fow00] Fowler, M.: *Refactoring : Wie Sie das Design vorhandener Software verbessern*. Professionelle Softwareentwicklung, Addison-Wesley, 2000, ISBN 3-8273-1630-8. Einheitssachtitel: Refactoring : improving the design of existing code »dt.«

[GHJV04] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. Programmer's Choice, Addison-Wesley, Juli 2004, ISBN 3-8273-2199-9. Einheitssachtitel: Design patterns »dt.«

[Gla98] Glass, R.: Maintenance: Less Is Not More. In: *IEEE Software* 15 (1998), Nr. 4, S. 67–68

[Gla04] Glass, R. L.: Learning to Distinguish a Solution from a Problem. In: *IEEE Software* 21 (2004), Nr. 3, S. 111–112

[Gro02] Group, T. O. M.: Request for Proposals: MOF 2.0 Query / Views / Transformations. April 2002, URL `http://www.omg.org/docs/ad/02-04-10.pdf`

[GSCK04] Greenfield, J.; Short, K.; Cook, S.; Kent, S.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004, ISBN 0-471-20284-3

[GSMD03] van Gorp, P.; Stenten, H.; Mens, T.; Demeyer, S.: Towards automating source-consistent UML Refactorings. In: *Proceedings of UML 03*, 2003. TODO: Buchtitel? Verlag, Seiten fehlen

[HKKI04] Higo, Y.; Kamiya, T.; Kusumoto, S.; Inoue, K.: Refactoring Support Based on Code Clone Analysis. In: Bomarius, F.; Iida, H., Hrsg., *PROFES*, Springer-Verlag, 2004, Nr. 3009 in Lecture Notes in Computer Science, ISBN 3-540-21421-6, S. 220–233

[JO93] Johnson, R. E.; Opdyke, W. F.: Refactoring and Aggregation. In: Nishio, S.; Yonezawa, A., Hrsg., *ISOTAS*, Springer-Verlag, 1993, Nr. 742 in Lecture Notes in Computer Science, ISBN 3-540-57342-9, S. 264–278

[JR01] Jacobi, C.; Rumpe, B.: Hierarchical XP. Improving XP for Large-Scale Projects in Analogy to Reorganization Processes. In: Succi, G.; Marchesi, M., Hrsg., *Extreme Programming Examined*, Addison-Wesley, 2001, ISBN 0-201-71040-4, S. 83–102

[Ker05] Kerievsky, J.: *Refactoring to Patterns*. Addison-Wesley signature series, Addison-Wesley, 2005, ISBN 0-321-21335-1

[KR05] Krahn, H.; Rumpe, B.: *Evolution von Software-Architekturen*. Technischer Bericht Informatik-Bericht 2005-04, Technische Universität Braunschweig, Carl-Friedrich-Gauß Fakultät für Mathematik und Informatik, Mai 2005, URL

http://www.sse.cs.tu-bs.de/publications/EvolutionVonSoftwareArchitekturen.pdf

[MDA] Model Driven Architecture. URL http://www.omg.org/mda/. Siehe obmg04! - SaschaMueller

[MDJ02] Mens, T.; Demeyer, S.; Janssens, D.: Formalising Behaviour Preserving Program Transformations. In: *ICGT 2002*, 2002, Nr. 2505 in Lecture Notes in Computer Science. TODO

[Mic] Microsoft Coorperation: Domain Specific Language (DSL) Tools. URL http://lab.msdn.microsoft.com/vs2005/teamsystem/workshop/dsltools/

[MT04] Mens, T.; Tourwé, T.: A Survey of Software Refactoring. In: *IEEE Transactions on Software Engineering* 30 (2004), Nr. 2, S. 126–139

[MW01] Mens, T.; Wermelinger, M.: *Proceedings of the Workshop on Formal Foundations of Software Evolution*. Technical Report UNL-DI-1-2001, Departamento de Informatica Faculdade de Ciencias e Tecnologia Universidade Nova de Lisboa, 2001, URL ftp://progftp.vub.ac.be/tech_report/2001/vub-prog-tr-01-03.pdf

[MW02] Mens, T.; Wermelinger, M.: Separation of concerns for software evolution. In: *Journal of software maintenance and evolution : research and practice* 14 (2002), Nr. 5, S. 311–315

[Opd92] Opdyke, W.: *Refactoring Object-Oriented Frameworks*. Technischer Bericht UIUCDCS-R 92-1759, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992. Dissertation an der University of Illinois at Urbana-Champaign

[RBJ97] Roberts, D.; Brant, J.; Johnson, R.: A Refactoring Tool for Smalltalk. In: *Theory and Practice of Object Systems* 3 (1997), S. 253–263

[RHMRM04] Roshandel, R.; van der Hoek, A.; Mikic-Rakic, M.; Medvidovic, N.: Mae – A System Model and Environment for Managing Architectural Evolution. In: *ACM Transactions on Software Engineering and Methodology* 13 (2004), Nr. 2, S. 240–276

[RS02] Rumpe, B.; Schröder, A.: Quantitative Survey on Extreme Programming Projects. In: Wells und Williams [WW02], S. 43–46

[Rum96] Rumpe, B.: *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996

[Rum04] Rumpe, B.: *Modellierung mit UML : Sprache, Konzepte und Methodik*. Xpert.press, Springer-Verlag, 2004, ISBN 3-540-20904-2

[Rum05] Rumpe, B.: *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Xpert.press, Springer-Verlag, 2005, ISBN 3-540-20905-0

[SS04] Streckenbach, M.; Snelting, G.: Refactoring Class Hierarchies with KABA. In: *Proceedings of OOPSLA 04*, 2004. TODO: Seiten, Verlag fehlen

[SSL01] Simon, F.; Steinbückner, F.; Lewerentz, C.: Metrics Based Refactoring. In: *Proceedings of European Conference Software Maintenance and Reengineering*, 2001, S. 157–169

[TB01] Tokuda, L.; Batory, D.: Evolving Object-Oriented Designs with Refactorings. In: *Journal of Automated Software Engineering* 8 (2001), S. 89–120

[TDDN00] Tichelaar, S.; Ducasse, S.; Demeyer, S.; Nierstrasz, O.: A Meta-model for Language-Independent Refactoring. In: *Proceedings ISPSE 2000*, 2000, IEEE Computer Society Press

[TDDN01] Tichelaar, S.; Ducasse, S.; Demeyer, S.; Nierstrasz, O.: Refactoring UML models. In: *Proceedings of UML 01*, Springer-Verlag, 2001, Nr. 2185 in Lecture Notes in Computer Science, ISBN 3-540-42667-1

[TM03] Tourwé, T.; Mens, T.: Identifying Refactoring Opportunities Using Logic Meta Programming. In: *Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, 2003

[TS01] Tolvanen, J.-P.; S., K.: Domain-Specific Modeling: 10 times faster than UML. In: *Proceedings of Embedded Systems Conference, Stuttgart, Germany*, 2001

[Wei84] Weiser, M.: Program Sicing. In: *IEEE Transactions on Software Engineering* 10 (1984), Nr. 4, S. 352–357

[WW02] Wells, D.; Williams, L. A., Hrsg.: *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, Nr. 2418 in Lecture Notes in Computer Science, Springer-Verlag, 2002, ISBN 3-540-44024-0

[ZYXX02] Zhao, J.; Yang, H.; Xiang, L.; Xu, B.: Change impact analysis to support architectural evolution. In: *Journal of software maintenance and evolution : research and practice* 14 (2002), Nr. 5, S. 317–333