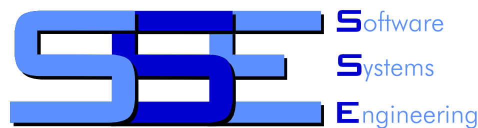
Technische Universität Braunschweig
Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik
Institut für Software Systems Engineering

Evolution von Software-Architekturen

Informatik-Bericht 2005-04

Holger Krahn
Bernhard Rumpe



Braunschweig, den 27. Mai 2005

[KR05b] H. Krahn, B. Rumpe.
Evolution von Software-Architekturen.
Informatik-Bericht 2005-04
Technische Universität Braunschweig, Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik, 2005
www.se-rwth.de/publications

Kurzfassung

In diesem technischen Bericht wird die Evolution von Software-Architekturen anhand von kleinen, effizient und schnell ausführbaren, systematischen Schritten dargestellt. Die Architektur wird dabei so adaptiert, dass diese sich besser für die Erweiterung um neue Funktionalität eignet. Die Umgestaltungsschritte werden durch einen dargestellten Kalkül formal beschrieben, wodurch die Implikationen präzise und bis hin zu notwendigen Beweisverpflichtungen festgelegt sind. Dabei bleibt das migrierte System in jedem Zwischenschritt lauffähig, kann so weiter im Produktiveinsatz verbleiben und permanenten Tests unterzogen werden.

Inhaltsverzeichnis

1 Einleitung **1**

2 Motivation **3**

2.1 Methodik 4

2.2 Anwendbarkeit der Evolution 6

2.3 Model Driven Architecture 7

2.4 Domänen-spezifische Sprachen (DSL) 9

3 Ansätze und Konzepte zur Software-Evolution **11**

4 Refactoring: Evolution im Kleinen **13**

5 Software-Evolution am Beispiel eines Geschäftsprozessmodells **17**

5.1 Grundlagen für einen Evolutions-Kalkül 20

5.2 Ein Evolutions-Kalkül 23

5.3 Eine Anwendung des beschriebenen Kalküls 26

5.4 Validierung von Refactorings durch Testen 29

6 Weitere Fragen **33**

Literaturverzeichnis **35**

1 Einleitung

In diesem technischen Bericht 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.

Der Bericht ist wie folgt gegliedert:

- In Kapitel 2 wird die grundlegende Vorgehensweise motiviert, die Vor- und Nachteile sowie die Anwendbarkeit diskutiert.
- Kapitel 3 enthält einen Überblick über vorhandene und im Entstehen begriffene Werkzeuge und Techniken, die die Evolution im Kleinen unterstützen.
- Kapitel 4 gibt eine Übersicht über Ansätze der Evolution, die oft unter dem Stichwort „Refactoring" vorgestellt werden.
- Ein komplexeres Beispiel auf Basis eines Architekturstils für verteilte, asynchron kommunizierende Systeme wird in Kapitel 5 vorgestellt.
- Die offenen Werkzeug- und Forschungsthemen werden in Kapitel 6 diskutiert und mögliche Wege zum schnellen Einsatz ausgesuchter Evolutionstechniken diskutiert.

2 Motivation

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 Hardwarearchitektur 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 mit der Effizienz oder Kompaktheit 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 und dass ein Großteil der als Wartung bezeichneten Tätigkeiten eigentlich der Software-Evolution dienen. 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 potentielle 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.

2.1 Methodik

Der konsequente Einsatz von kleinen und systematischen Schritten zur Evolution einer Software-Architektur ist vor allem dann sinnvoll, wenn diese Technik in eine geeignete Entwicklungsmethodik eingebettet ist. Die Evolution in kleinen Schritten kann bei geeigneter Werkzeugunterstützung bedeuten, dass die Architektur-Evolution 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 und 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]. Kapitel 5 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 essentiell. 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. Compilation des lauffähigen Systems aus diesem

Architekturmodell und der Transformationstechniken für Architekturen entsteht eine effektive Methodik 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.

2.2 Anwendbarkeit der Evolution

Evolutionäre Techniken können nur unter eng umrissenen Umständen sinnvoll angewendet werden. Unter Umständen lassen sich jedoch durch vorbereitende und begleitende methodische Maßnahmen evolutionäre Techniken auch in einem weniger geeigneten Umfeld einsetzen.

- 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 notwendigerweise 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 zugrundeliegenden Technologie evolutionär vorbereiten, so dass eine geeignete Qualitätssicherung auch hier vorgenommen werden kann. Das Upgrade einer Framework- oder Betriebssystem-Version 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 2.1 skizziert diese Vorgehensweise.

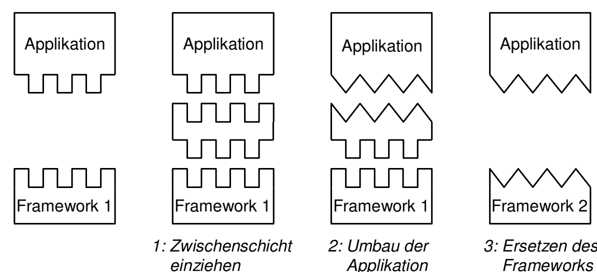


Abbildung 2.1: Schema zur evolutionären Migration auf ein aktualisiertes Framework

2.3 Model Driven Architecture

Model Driven Architecture (MDA) [MDA] entwickelt sich derzeit zu einer Technologie, die es erlaubt, Software noch 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.

Der ursprüngliche Kern von MDA besteht aus der separaten Definition von plattform-unabhängigen (PIM) und plattform-spezifischen (PSM) Teilen der Software, um beide unabhängig voneinander wieder verwenden zu können und damit einen ähnlichen Effekt zu erreichen wie er auch in [Sie04] beschrieben ist. Der Applikationskern wird dabei völlig plattform-unabhängig realisiert. In diesem Applikationskern liegt teuer erworbenes Wissen über Geschäftsabläufe oder Gerätesteuerungen verborgen. Die explizite Kapselung des Applikationskerns erlaubt dann dessen leichtere Wiederverwendung zum Beispiel bei einer Migration auf eine andere Plattform. Der Applikationskern wird dabei idealerweise nicht in Form von Code sondern von abstrakteren Modellen entwickelt, die durch generative Transformationen in Code umgesetzt werden.

Der plattform-spezifische Teil kann demgegenüber meist nicht als eigenständiges Modell definiert werden, sondern wird durch ein Transformationssystem definiert, das zur vorhandenen Applikation zum Beispiel technik-spezifische Funktionalität hinzufügt und es so mit einem Framework oder Betriebssystem integriert. Solche Transformationen sollten ähnlich zu Compilern unabhängig von den übersetzten Applikationen sein und daher für viele Applikationen einsetzbar sein. Abbildung 2.2 illustriert dieses Prinzip.

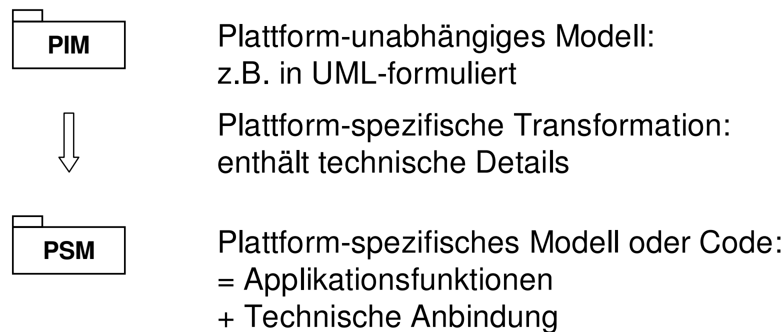


Abbildung 2.2: Grundprinzip der Model Driven Architecture (MDA)

Tatsächlich sind eine Reihe von Transformationen jenseits von Plattformabhängigkeit und UML-nahen Modellen [MDA, Béz05] und teilweise auch bereits im Einsatz, zum Beispiel für die Anbindung einer graphischen Oberfläche, der Generierung von Datenstrukturen, der Integration eines Persistenzmechanismus in die Applikation oder die Abbildung von Teilmodellen auf Schnittstellen zur Kommunikation. Da diese Transformationen jeweils eigenständige Einheiten darstellen, nur auf Teile des Ausgangsmodells angewendet werden bzw. nur Teile des Ergebnisses erzeugen, entsteht in der Praxis eine projektspezifische Landschaft von parallelen und sequentiell angewandten Transformationen, wie beispielhaft in Abbildung 2.3 dargestellt ist. Durch die Zerlegung der Transformationen in kleine, spezialisierte Einheiten lassen sich einzelne, relativ allgemein einsetzbare Transformationen entwickeln und applikationsunabhängig wieder verwenden.

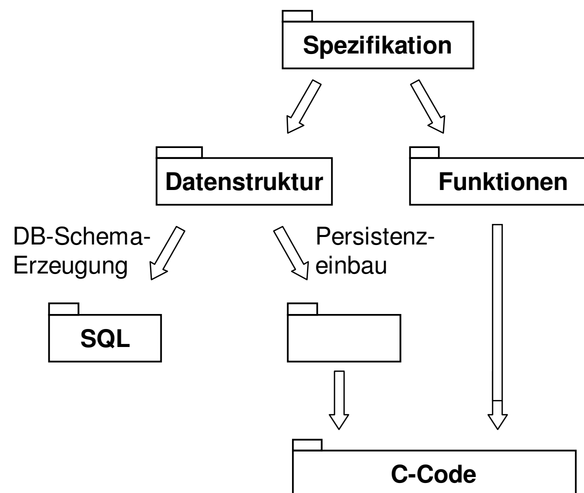


Abbildung 2.3: Anwendungslandschaft von MDA: Sequentiell und parallel angewandte Transformationen

Eine Reihe der als MDA-Anwendungen verstandenen Transformationen sind jedoch nicht vollständig automatisierbar. Im Beispiel ist etwa die Extraktion der Datenstruktur und der Funktionsbeschreibungen normalerweise nicht aus einer (informellen) Spezifikation ableitbar, während die Definition der SQL-Statements zur Erzeugung und Manipulation einer Datenbank zwar weitgehend automatisiert werden kann, aber für die Optimierung mit dem Anwender oft eine interaktive Transformation sinnvoll ist. Teilweise ist es aber auch vorstellbar, dass eine Automatisierung der Übersetzung ggf. unter Einstellung geeigneter Parameter machbar erscheint, aber von den aktuell verfügbaren Werkzeugen noch nicht adäquat unterstützt wird.

Die mangelnde Werkzeugunterstützung hat vor allem zwei Gründe:
MDA war ursprünglich vor allem als Methodik zum Einsatz von UML-Modellen für die PIM gedacht. Den Anspruch zur allgemeingültigen Modellierungssprache der UML, der zu einer relativ schwachen Semantikdefinition geführt hat und die Komplexität der Sprache limitieren jedoch die Nützlichkeit von Transformationen. Es hat sich daher relativ schnell gezeigt, dass MDA-Techniken über die UML hinaus vor allem auch für domänen-spezifische Aufgabenstellungen geeignet ist. Darin werden Domänen-spezifische Sprachen (DSL) und Modelle genutzt, um spezialisierte Konzepte in kompakter Form darzustellen. Die darin inhärente Spezifität kann bei einer Generierung dann genutzt werden, ganze Komponenten wieder zu verwenden.

2.4 Domänen-spezifische Sprachen (DSL)

Zunächst hat sich die Entwicklung von MDA-Techniken vor allem auf die allgemeingültigen Konzepte zur Darstellung von Modellen und Transformationen konzentriert.

Mittlerweile entstehen immer mehr MDA-nahe Ansätze, die es erlauben, problemangepasste und damit meist auf eine Domäne spezialisierte Sprachen, so genannte „Domänen-spezifische Sprachen“ (DSL) einzusetzen. Beispielhaft sei der Ansatz von Microsoft [Mic, GSCK04] genannt, der es in Zukunft sehr einfach machen wird, DSL's zu definieren. Die domänen-spezifischen Konzepte einer so definierten DSL werden auf ein vorhandenes oder selbst erstelltes Framework abgebildet, das einzeln einsetzbare Wissenskomponenten enthält und durch geeignet 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 Potential zur Effektivitätssteigerung, da so spezialisierte Frameworks ansprechbar sind.

Der Nutzen von Domain Specific Languages (DSL) liegt genau darin, für ein Aufgabengebiet spezialisierte, anwendungsnahe Abstraktionen zur Verfügung zu stellen. Der erfolgreiche Einsatz von DSL's 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 DSL's nicht für jedes Projekt neu erstellt werden müssen, sondern zumindest firmenweit wiederverwendbar sind.

Dem unbestreitbar immensen 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 2.1 beschrieben, die Qualität der resultierenden Architektur effizient sichergestellt werden kann. In Kapitel 5 wird dies an einer speziell für asynchron kommunizierende Systeme entwickelten DSL demonstriert.

3 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 im Workshop „Formal Foundations of Software Evolution“ [MW01] wurde die Kernthese herausgestellt, dass der Grundsatz „Separation of concerns“ auch auf Modellebene essentiell ist. Darunter versteht man die Trennung und Modularisierung von verschiedenen Aspekten eines 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 u. a. [ZYXX02] übertragen diese Technik auf Modellebene, indem sie eine Variante auf Software-Systeme anwenden, die in 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 das verwendete Betriebssystem gewählt werden können. Unter einer Konfiguration versteht man dabei eine Menge von Artefakten, die zusammen ein Gesamtsystem bilden. Roshandel u. a. setzen in [RvdHMRM04] ein Konfigurationswerkzeug ein, um die Evolution einer Architekturbeschreibung zu erfassen. Sie sind dabei nicht auf eine bestimmte Architekturbeschreibungssprache festgelegt, die die Evolution meistens unzureichend unterstützen, sondern fügen den Evolutionsaspekt durch ein zusätzliches System hinzu.

4 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 ein, 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, dass eine semi-automatische 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

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

einer Software beschleunigen kann. Die Autoren schätzen eine Zeitersparnis vom 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]. Somit kann durch Refaktorisierungstechniken eine gebräuchliche und oft angewandte Architektur eines Systems herbeigeführt werden, dessen Vor- und Nachteile gut untersucht sind.

In [TDDN00] wurde ein Meta-Modell entwickelt, das es ermöglicht, Refactorings unabhängig von einer konkreten (objekt-orientierten) Programmiersprache zu definieren. Das Modell abstrahiert dabei von Unterschieden wie dynamische bzw. statische Bindung und beschreiben die Refactorings deklarativ mit Vor- und Nachbedingungen. In [vGSMD03] 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 graphisch für den Benutzer dargestellt. In [vEM02] wird eine vollautomatische Erkennung von „Code-Smells" in Javaquellcode dem Nutzer grafisch dargestellt. Demeyer u. a. [DDN00] zeigen, wie Metriken eingesetzt werden, um den Einsatz 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 u. a. [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 semi-automatisches Entfernen von duplizierten Code entfernen. 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 Unverändertheit des Verhaltens des eigentlichen Projekts hin zu der Unverändertheit des Verhaltens der erzeugten Parser.

Wie in Abschnitt 2.1 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 ermöglichen, die verändert werden sollten. In [TDDN01] werden nicht nur Refaktorisierungsregeln für UML-Klassendiagramme, sondern auch Zustandsautomaten angeben. Die Autoren verwenden dabei sieben Refactorings, die eine Umstrukturierung der Modelle erlauben. Der „Refactoring Browser for UML“ [BSF02] ist ein Plugin 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 Potential für diese Technik vorhanden ist, andererseits aber eine integrierte, allgemeine Theorie und ihre technische Umsetzung noch fehlt. Das von der OMG ausgerufene Standardisierungsproposal [Gro02] ist ein weiterer geeigneter Schritt in diese Richtung, bei dem vor allem die explizite Darstellung der Transformationen im Vordergrund steht.

5 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 graphischen Elemente:

- Komponenten mit expliziten Schnittstellen (Ports)

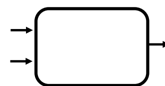


Abbildung 5.1: Ports

- Gerichtete Nachrichtenkanäle

Abbildung 5.2: Kanäle

- Hierarchische Dekomposition

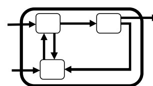


Abbildung 5.3: Dekomposition

In der der Modellierungssprache zugrundeliegenden 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

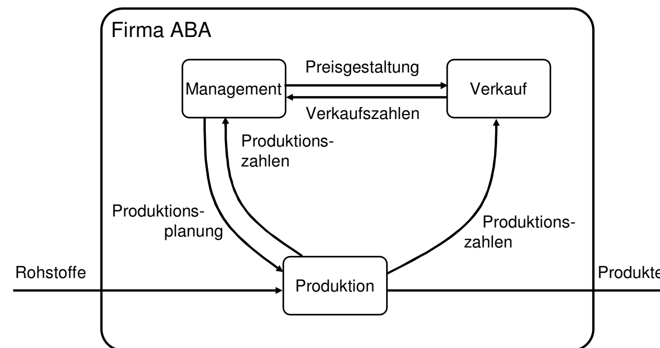


Abbildung 5.4: Die Firma ABA vor der Umstrukturierung

Lage Nachrichten zu puffern, Sender und Empfänger sind deshalb wie bei einem Emailsystem relativ entkoppelt. Allerdings wird festgelegt, dass sich Nachrichten in einem Kanal nicht überholen können.

Eine Firma produziert wie in Abbildung 5.4 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 das für einen Kunden beobachtbare Verhalten in Form der Anzahl der produzierten Waren. Technisch ist diese Unveränderbarkeit der externen Nachrichtenflüsse durch geeignete Invarianten, bzw. 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) so 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 5.5 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.

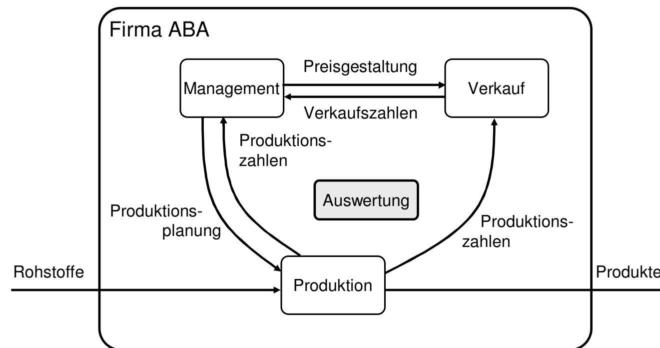


Abbildung 5.5: Komponente Auswertung wird hinzugefügt

Damit die neue Komponente ihre Funktion erfüllen kann, muss sie die notwendigen Eingabedaten erhalten. Sie erhält daher über neu instanzierte Kanäle dieselben Produktions- und Verkaufszahlen wie das Management (vgl. Abbildung 5.6). Da die Komponente keinen Ausgabekanal hat, kann weiterhin nichts Negatives passieren.

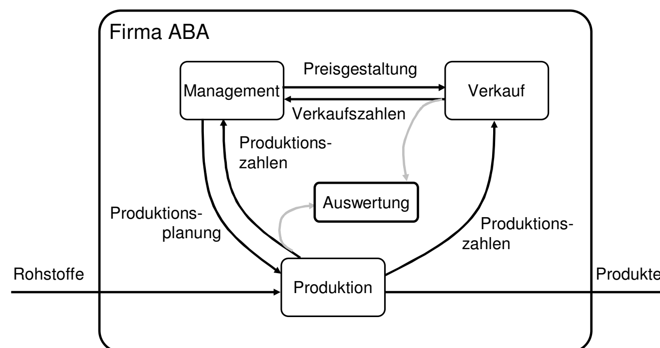


Abbildung 5.6: Komponente Auswertung erhält dieselben Daten wie das Management

Durch die Eingangsdaten ist die Auswertungs-Komponente 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 detailiert, sobald ein Produkt produziert bzw. verkauft ist und zum anderen in regelmäßigen Abständen als Bericht von der Auswertungs-Komponente. Das Management sollte nun in der Lage sein, seine Entscheidungen ausschließlich durch die Berichte zu treffen, da diese zumindestens die für eine Entscheidung notwendigen Informationen enthalten. Daher können die alten Informationsflüsse, wie in Abbildung 5.8 gezeigt, abgeschaltet werden.

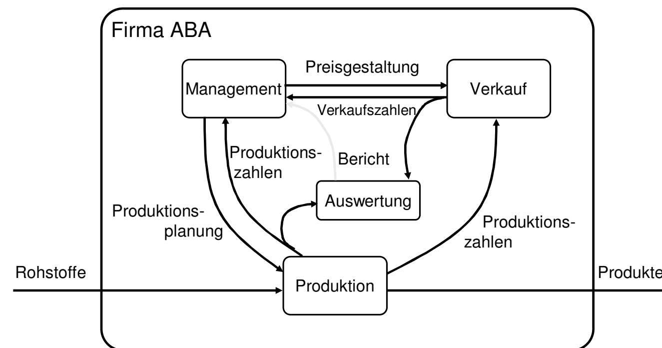


Abbildung 5.7: Komponente Auswertung erstellt nun regelmäßig Berichte

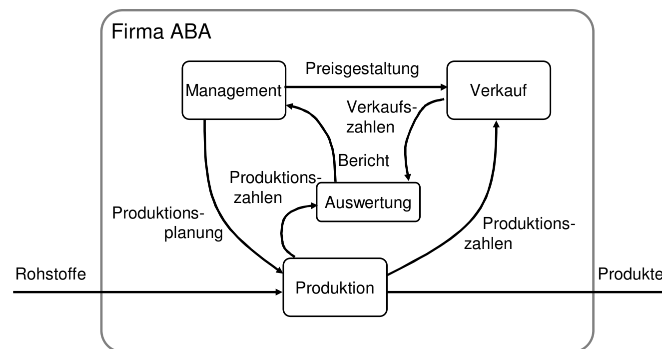


Abbildung 5.8: Das Geschäftprozessmodell nach der Umstrukurierung

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.

5.1 Grundlagen für einen Evolutions-Kalkül

Um die einzelnen Evolutionsschritte besser nachvollziehen zu können, wird im Folgenden eine auf Focus [BS01, Rum96] basierende Formalisierung verwendet, 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, der nachfolgend in kompakter Form vorgestellt wird. Am Ende dieses Abschnitts wird dieser Mechanismus relativ gut gekapselt.

Ströme Die Kommunikation einzelner Komponenten untereinander wird als zeitliche Beobachtung auf unidirektionalen Kommunikationskanälen beschrieben. Mathematisch wird eine solche Beobachtung durch eine endliche oder unendliche Sequenz („Strom“) von Daten dargestellt. Ströme ($M^\omega = M^* \cup M^\infty$) sind geordnete Tupel von endlicher (M^*) oder unendlicher Länge (M^∞) über einer Grundmenge M, die alle möglichen Nachrichten enthält, die auftreten können. Ein Strom ist somit eine zeitlich geordnete Sequenz von Nachrichten auf einem beobachteten Kanal. Zeitliche Abstände werden dabei nicht festgelegt. Für Ströme lässt sich eine Präfixordnung $s \sqsubseteq t$ definieren, die beschreibt, wenn ein Strom s der Anfang eines anderen Stroms t ist. Es gilt also $s \sqsubseteq t$, wenn t eine längere Beobachtung darstellt als s. Der leere Strom ist bezüglich dieser Ordnung das kleinste Element und wird mit $[]$ bezeichnet.

Kanal Ein Kanal beschreibt die Verbindung zwischen zwei Komponenten, auf der ein Strom von Nachrichten fließt. Ein Kanal hat einen Namen $k.\mathrm{name}$ und einen Nachrichtentyp $k.\mathrm{mess}$, der die Menge der darauf potentiell beobachtbaren Nachrichten beschreibt. Der Typ eines Kanals (Kanaltyp) wird durch die Ströme $\overrightarrow{k} = k.\mathrm{mess}^\omega$ der darauf beobachtbaren Nachrichten beschrieben.

In einem konkreten Systemablauf wird einem Kanal zusätzlich eine Belegung in Form eines Stroms $k.\mathrm{stream} \in \overrightarrow{k}$ zugeordnet.

Das Konzept der Kanäle wird auf Gruppen ausgeweitet: $K.\mathrm{name}$ ist demnach eine Menge von Namen. Die möglichen Beobachtungen auf diesen Kanälen werden durch ein Kreuzprodukt $\overrightarrow{K} = \otimes_{k \in K} \overrightarrow{k}$ über die einzelnen Kanäle beschrieben. Analog ist mit $K.\mathrm{stream}$ ein benanntes Tupel mit jeweils einer konkreten Beobachtung pro Kanal beschrieben und für $Q \subseteq K$ kann für eine Kanalbelegung $l \in \overrightarrow{K}$ ein Teiltupel $l|_Q \in \overrightarrow{Q}$ selektiert werden.

Komponente Ein Tupel $c = (c.\mathrm{name}, c.\mathrm{in}, c.\mathrm{out}, c.\mathrm{behav})$ stellt eine Komponente c dar. Dabei ist $c.\mathrm{name}$ der Name der Komponente. $c.\mathrm{in}$ und $c.\mathrm{out}$ sind Mengen von Eingabe- und Ausgabekanälen und bilden die Schnittstellen der Komponente. $c.\mathrm{behav}$ beschreibt das Verhalten der Komponente.

Verhalten einer Komponente Eine Komponente ist durch ihr Verhalten auf den Schnittstellen gekennzeichnet. Dabei werden die Ausgaben auf Basis der Eingaben festgelegt. Bei einer deterministischen Komponente kann dies als mathematische

Abbildung v von den Eingabekanälen auf Ausgabekanäle beschrieben werden. Die Funktion $v : \overrightarrow{c.\text{in}} \rightarrow \overrightarrow{c.\text{out}}$, die die Ströme auf den Eingabekanälen auf die Ströme auf den Ausgabekanälen abbildet, ist also geeignet das Verhalten einer Komponente zu beschreiben. Eine Komponente kann eine einmal erfolgte Ausgabe nicht zurücknehmen. Formal lässt sich diese Tatsache für die Funktion v als Monotonie bezüglich der Präfixordnung $\sqsubseteq$ definieren:

$$\forall s, t \in \overrightarrow{c.\text{in}} : s \sqsubseteq t \ \Rightarrow \ v(s) \sqsubseteq v(t)$$

Die Spezifikation einer Komponente muss nicht notwendigerweise eine einzelne Funktion v sein, denn sowohl Unterspezifikation als auch Nichtdeterminismus der Komponente können durch eine Menge von Funktionen als potentielle Verhaltensbeschreibungen dargestellt werden. Während einer Entwicklung werden Komponenten oft zunächst unvollständig spezifiziert und erst zu einem späteren Zeitpunkt präzisiert. Daher wird als Verhaltensbeschreibung für eine Komponente c eine Menge von Verhaltensfunktionen eingesetzt:

$$c.\text{verh} \subseteq \overrightarrow{c.\text{in}} \rightarrow \overrightarrow{c.\text{out}}$$

Dieser etwas komplexe Mechanismus wird wie eingangs bereits beschrieben, genutzt, weil man damit in der Lage ist, Komposition und Verfeinerung nachfolgend elegant zu formulieren und beides zueinander kompatibel ist [BS01, Rum96].

System Ein System besitzt zwei Sichten: In der externen Sicht sind Name, Schnittstellen und Gesamtverhalten beschrieben. Weil ein System in der externen Sicht behandelt werden kann wie eine Komponente, besteht so die Möglichkeit zur hierarchischen Komposition. In der internen Sicht besitzt ein System zusätzlich eine architekturelle Dekomposition in Komponenten.

Ein System S ist durch ein Tupel $S = (S.\text{name}, S.\text{comp}, S.\text{in}, S.\text{out}, S.\text{behav})$, beschrieben, das gegenüber einer einfachen Komponente zusätzlich die Zerlegung in eine Menge von Komponenten $S.\text{comp}$ beinhaltet. Das Verhalten des Systems $S.\text{behav}$ ist festgelegt durch die Komposition des Verhaltens seiner Komponenten. Die Komposition der Komponenten-Verhalten wird durch den nachfolgend präzisierten Operator $\oplus$ beschrieben:

$$V = \oplus_{c \in S.\text{comp}} \; c.\text{behav}$$

Die Signatur des komponierten Verhaltens beinhaltet aber immer noch interne Kanäle. Nach Reduktion der Signatur von V auf die extern sichtbaren Schnittstellen muss V dann zu $S.\text{behav}$ identisch sein. Seien die intern verfügbaren Kanäle beschrieben durch

$$K = \bigcup S.\text{comp.out} \ \cup \ \bigcup S.\text{comp.in}$$

Eine Funktion v ist genau dann Teil des Gesamtverhaltens S.behav wenn jede Eingabebelegung $i \in \overrightarrow{S.\text{in}}$ so zu einer Beobachtung $l \in \overrightarrow{K}$ aller Kanäle des Systems ausgeweitet werden kann ($l|_{S.\text{in}} = i$), dass diese Beobachtung auch für jede Komponente $c \in S$.comp passt ($l|_{c.\text{out}} \in c.\text{verh}(l|_{c.\text{in}})$). Mathematisch formuliert:

$$\forall i \in \overrightarrow{S.\text{in}} : \ \exists l \in \overrightarrow{K} : \ l|_{S.in} = i \ \wedge \ \forall c \in S.\text{comp} : l|_{c.\text{out}} \in c.\text{behav}(l|_{c.\text{in}})$$

Komposition Komponenten können sequentiell, parallel oder in zahlreichen anderen Formen miteinander verbunden werden. Vereinfachend wird in diesem Ansatz davon ausgegangen, dass die Verbindung über die Gleichheit der Namen von Ein- und Ausgabekanälen festgelegt ist. Mit dieser Annahme kann der gerade definierte Kompositionsoperator $\oplus$ beliebige Kommunikationsstrukturen bilden und mehrere Ziel-Komponenten mit gleichnamigen Eingangskanälen von derselben Quelle durch selektiven Broadcast bedienen. Allerdings dürfen die Komponenten eines Systems jeden Namen nur einmal als Ausgabekanal verwenden. Die benutze Komposition basiert mathematisch auf der Funktionskomposition.

Verfeinerung des Verhaltens Eine Verfeinerung einer Komponente oder eines Systems S bedeutet, dass dessen interne Struktur weiterentwickelt wird, ohne dass sich das nach außen beobachtbare Verhalten des Systems ändert.

$$S \rightsquigarrow S' \Leftrightarrow_{\text{def}} S'.\text{verh} \subseteq S.\text{verh}$$

Wenn sich bei einer Verfeinerung eines Sytems S die Ein- bzw. Ausgabekanäle nicht ändern, also $S.\text{in} = S'.\text{in}$ und $S.\text{out} = S'.\text{out}$ gilt, sind Verfeinerungen transitiv:

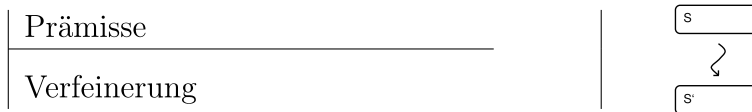
$$S \rightsquigarrow S' \wedge S' \rightsquigarrow S'' \Rightarrow S \rightsquigarrow S''$$

Durch diese Eigenschaft ist es möglich, Umstrukturierungen eines Systems schrittweise vorzunehmen. Einzelne Regeln können nacheinander ausgeführt und zu komplexeren Taktiken zusammengestellt werden, wie dieses auch bei Refactorings möglich ist. Im Folgenden wird eine Adaption des Kalküls aus [PR97, PR99] verwendet, um die Umstrukturierungen eines Systems zu beschreiben.

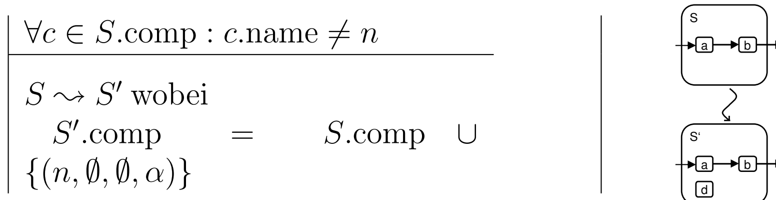
5.2 Ein Evolutions-Kalkül

Auf Basis der Grundlagen wird in diesem Abschnitt eine Sammlung von Regeln entwickelt, die die Modifikation der Interna eines Systems zulassen, ohne das spezifizierte externe Verhalten unzulässig zu verändern. Das heißt, interne Modifikationen dürfen extern nur als Verfeinerungen sichtbar werden.

Die Verfeinerungsregeln, die eine Komponente in eine andere transformieren, verwenden die folgende Syntax, wobei die Prämisse erfüllt sein muss, um die Verfeinerung anwenden zu können. Ein oder mehrere Abbildungen rechts illustrieren die Regel.



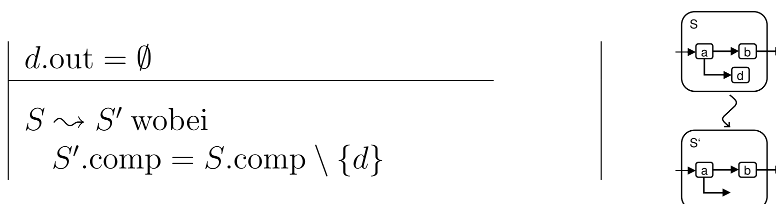
Hinzufügen von Komponenten. Eine neue Komponente d mit Namen n kann in ein System S unter der Einschränkung neu eingefügt werden, dass sie zunächst nicht über Ein- und Ausgabekanäle mit der Umgebung verbunden ist. Somit kann diese neue Komponente zu keiner Verhaltensänderung des Gesamtsystems führen. Später können dann durch andere Regeln sukzessive Ein- und Ausgabekanäle hinzugefügt werden. Die eingefügte Komponente zeigt anfangs aufgrund der fehlenden Aus- und Eingabekanäle ein leeres beobachtbares Verhalten, das hier mit α bezeichnet wird.



$\forall c \in S.\text{comp} : c.\text{name} \neq n$

$S \rightsquigarrow S'$ wobei $S'.\text{comp} = S.\text{comp} \cup \{(n, \emptyset, \emptyset, \alpha)\}$

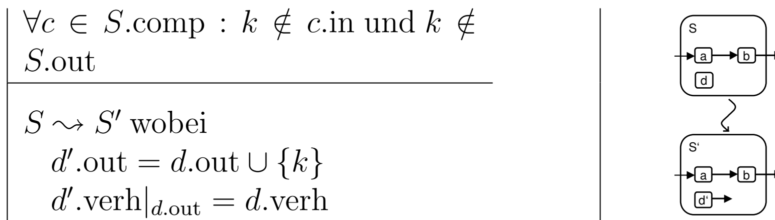
Entfernen von Komponenten. Auf ähnliche Weise wie Komponenten ohne Kanäle hinzugefügt werden, kann eine Komponente d entfernt werden, die nicht mit dem Rest des Systems oder der Umgebung verbunden ist. In Verallgemeinerung zur vorhergehenden Regel darf Komponente d auch entfernt werden, wenn sie noch Eingabekanäle hat:



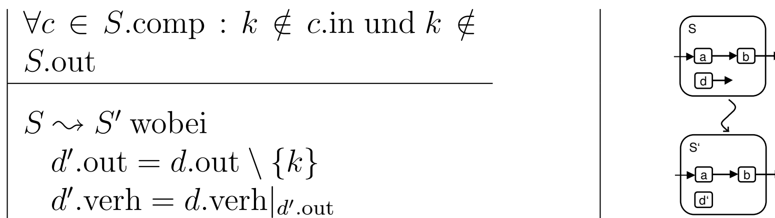
$d.\text{out} = \emptyset$

$S \rightsquigarrow S'$ wobei $S'.\text{comp} = S.\text{comp} \setminus \{d\}$

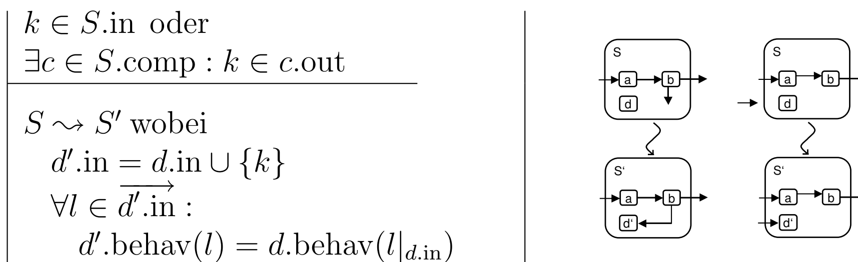
Hinzufügen von Ausgabekanälen. Ein Ausgabekanal k kann zu einem System S an eine Komponente d eingefügt werden, wenn er nicht Eingabekanal einer Komponente oder ein Ausgabekanal des komponierten Systems ist. Über den Strom dieses Kanals wird nur die Aussage getroffen, dass das Hinzufügen keine Änderungen an den anderen Ausgabekanälen hervorruft.



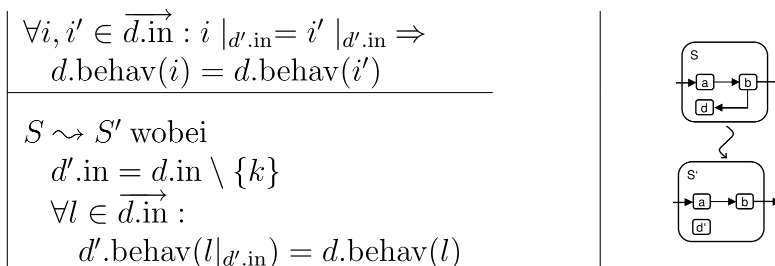
Entfernen von Ausgabekanälen. Ein Ausgabekanal k kann aus einem System S bzw. von einer Komponente d entfernt werden, falls er nirgendwo genutzt wird: also er weder Eingabekanal einer Komponente noch ein Ausgabekanal des Systems ist.



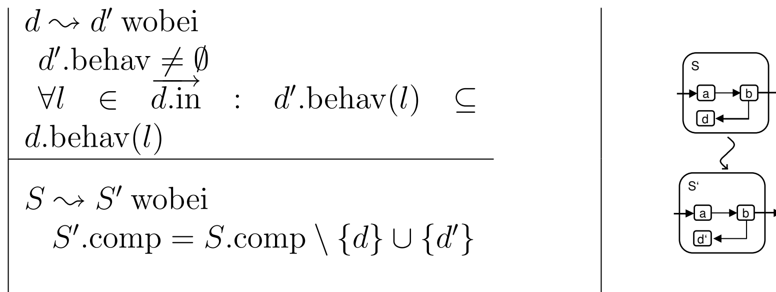
Hinzufügen von Eingabekanälen. Ein existenter Eingabekanal k kann zu einer Komponente d hinzugefügt werden, um so der Komponente mehr Informationen zur Verfügung zu stellen.



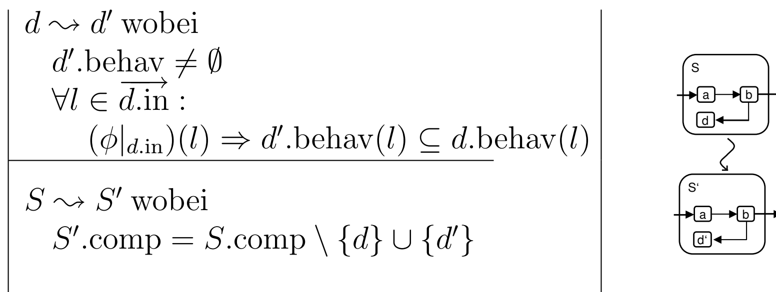
Entfernen von Eingabekanälen. Etwas komplexer ist das Entfernen eines Eingabekanals: Ein Eingabekanal kann von einer Komponente nur entfernt werden, falls sich dadurch Ihr Verhalten nicht ändert. Dafür ist es nötig, dass sie dieselben Informationen über andere Kanäle erhält oder den Eingabekanal gar nicht nutzt. Diese Eigenschaft lässt sich formal fassen, indem die Ausgaben der Komponente für Eingaben, die sich nur im zu entfernenden Kanal unterscheiden, nicht unterschiedlich sein dürfen.



Verfeinerung des Verhaltens. Nach Definition der Komposition und Verfeinerung bedeutet die Verfeinerung einer Komponente, dass automatisch auch das Gesamtverhalten des Systems verfeinert wird [PR97, PR99].



Diese relativ einfache Form der Verhaltensanpassung ist gelegentlich zu scharf, da sie eine Verhaltensverfeinerung der Komponente auf allen Eingaben darstellt. Aufgrund der Einbettung der Komponente in den Kontext der Systemstruktur kann aus der Einbettung abgeleitet werden, welche Eingaben für die Komponente überhaupt auftreten können. Solche Eingaben lassen sich in Form einer Invariante $\phi : \overrightarrow{K} \rightarrow \text{Bool}$ über die Beobachtungen beschreiben und die Verfeinerungsbeziehung dann auf die in der Invariante enthaltenen Beobachtungen einschränken. Diese Invariante kann systeminterne Eigenschaften darstellen, aber auch Kontextwissen (Annahmen) über den Einsatz des Systems beinhalten.

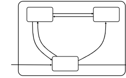
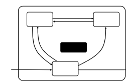
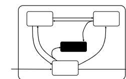
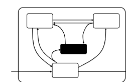
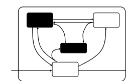
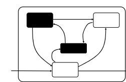


Der Kalkül zur Evolution von Architektur kann um weitere Regeln für die Expansion und Faltung von Subzuständen erweitert werden, auf deren Darstellung hier aus Platzgründen verzichtet wird. Der geneigte Leser sei dazu unter anderem auf [PR97, PR99] verwiesen.

5.3 Eine Anwendung des beschriebenen Kalküls

Der im Beispiel eingeführte Geschäftsprozess wurde bereits durch eine relativ einfache Umstruktierung auf neue Anforderungen angepasst. Diese Evolution vollzieht sich nicht in einem großen, sondern in mehreren automatisch angewandten kleinen Schritten für die die nun mit präziser Grundlage versehenen Refactoring-Regeln verwendet werden.

Zunächst noch einmal eine Zusammenfassung der einzelnen Schritte zur Umstrukturierung:

0. Ausgangssituation der Firma 
1. Einführung der Auswertungskomponente 
2. Hinzufügen von Eingabedaten 
3. Auslieferung der Berichte 
4. Umstellen des Managements 
5. Entfernen der alten Informationsflüsse 

Für eine genauere Betrachtung des Beispiels sind in Abbildung 5.9 die Ströme der einzelnen Kanäle mit Namen bezeichnet.

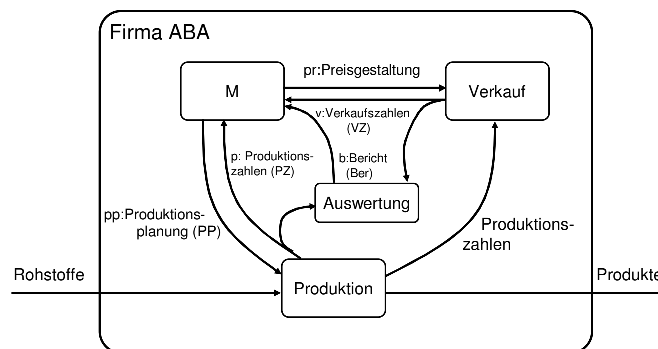


Abbildung 5.9: Bezeichnung der Kanäle

Bei der Anwendung der Refactoring-Regeln sind natürlich die jeweiligen Prämissen zu erfüllen. Für die Schritte 1 und 2 ist dieses offensichtlich. In Schritt 3 wird die Schnittstelle der Managementkomponente M verfeinert, d. h. es wird ein zusätzlicher Eingabekanal für den Bericht hinzugefügt und die Komponente somit nach M′ weiterentwickelt. Eine Gegenüberstellung der beiden Signaturen zeigt die zusätzlichen Kanäle:

$$\begin{aligned} \mathrm{M}: &\ \mathrm{ProdZ}^\omega \times \mathrm{VerkZ}^\omega && \rightarrow \mathrm{Preisgestaltung}^\omega \times \mathrm{ProdPlan}^\omega \\ \mathrm{M}': &\ \mathrm{ProdZ}^\omega \times \mathrm{VerkZ}^\omega \times \mathrm{Ber}^\omega && \rightarrow \mathrm{Preisgestaltung}^\omega \times \mathrm{ProdPlan}^\omega \end{aligned}$$

Wenn das Management zunächst die neu verfügbaren Berichte ignoriert, ist die für das Hinzufügen des Eingabekanals notwendige Prämisse automatisch erfüllt.

Die zentrale Idee des Schritts 4 dieser Umgestaltung ist, dass die neue Management-Komponente M'' die Entscheidungen allein auf der Grundlage der neu verfügbaren Berichte treffen kann. Wichtig ist, dass sie dabei dieselben Entscheidungen wie die alte Komponente M' trifft, sofern sie einen verlässlichen Bericht erhält. Es wird damit eine interne Verhaltensveränderung vorgenommen, die aber ein nach außen unverändertes Verhalten zur Folge hat. Ob der neue Bericht verlässlich ist, wird durch eine geeignete Invariante ϕ über die alten und den neuen Eingabekanal formuliert. Konkret lässt sich die Aussage, dass die Information nun doppelt vorhanden ist, durch die Invariante ϕ beschreiben, die angibt dass aus Verkaufs- und Produktionszahlen deterministisch ein Bericht erstellt wird. Damit wird die Invariante über den Kanälen p, v und b unter Nutzung einer geeigneten Spezifikation für die Auswertung definiert.

$$\phi(p, v, b) \equiv \quad b = \mathrm{Auswertung.behav}(p, v)$$

Für die Anwendung der Verhaltensverfeinerung unter Nutzung der Invariante ist noch die Umsetzung der Prämisse sicherzustellen, die durch folgende Eigenschaft gesichert werden kann:

$$\begin{aligned} &\forall p \in \mathrm{ProdZ}^\omega, \forall v \in \mathrm{VerkZ}^\omega, \forall b \in \mathrm{Ber}^\omega : \\ &\quad \phi(p, v, b) \;\Rightarrow\; \mathrm{M}''(p, v, b) \;=\; \mathrm{M}'(p, v, b) \end{aligned}$$

In Schritt 5 werden die alten Eingabekanäle entfernt, d. h. die Managementkomponente M'' zu M''' weiterentwickelt. Dazu müssen die Management-Entscheidungen nun allein auf der Grundlage der Reports getroffen werden können und es somit gleichgültig sein, welche Produktions- und Verkaufszahlen im Detail gemeldet werden, sofern diese zum selben Bericht führen:

$$\begin{aligned} &\forall p, q \in \mathrm{ProdZ}^\omega, u, v \in \mathrm{VerkZ}^\omega, b \in \mathrm{Ber}^\omega : \\ &\quad \phi(p, u, b) \wedge \phi(q, v, b) \Rightarrow \mathrm{M}''(p, u, b) = \mathrm{M}''(q, v, b) \end{aligned}$$

Aufgrund dieser Eigenschaft, lässt sich nun M''' mit der folgenden Formel definierend festlegen und damit zeigen, dass letztendlich das Verhalten von M''' mit dem von M identisch ist:

$$\begin{aligned} &\forall p \in \mathrm{ProdZ}^\omega, \forall v \in \mathrm{VerkZ}^\omega, \forall b \in \mathrm{Ber}^\omega : \\ &\quad \phi(p, v, b) \Rightarrow \mathrm{M}'''(b) = \mathrm{M}''(p, v, b) = \mathrm{M}'(p, v, b) = \mathrm{M}(p, v) \end{aligned}$$

Die präzise Formulierung der Auswirkungen von Transformationen einschließlich der Verhaltensänderung erlaubt eine genauere Analyse der Korrektheitsbedingungen als dies bei einer natürlichsprachlichen Beschreibung möglich ist. So lässt sich beispielweise leicht erkennen, dass eine Validierung einiger Transformationen unnötig ist, das 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 archtektonische Eigenschaften eines Systems erhalten während sie ausgesuchte Teile und Komponenten des Systems weiterentwickeln.

5.4 Validierung von Refactorings durch Testen

Neben bereits beschriebenen 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 Testmethodik, 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 determistisch.
- Es treten keine Seiteneffekte auf.
- Eine gute Abdeckung von Randfällen wird erreicht.

Dieser Ansatz wird im Folgenden an der zentralen Invariante des verwendeteten 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 Management-Entscheidungen 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 5.10 gezeigt, beide Komponenten parallel in das System eingebaut und auf dasselbe Verhalten geprüft.

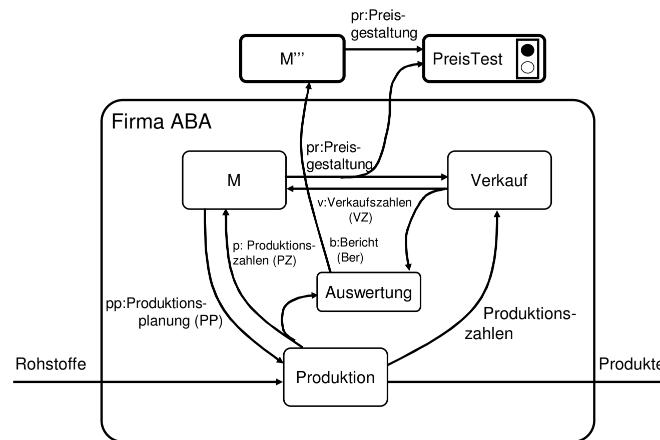


Abbildung 5.10: Erweiterung des Geschäftsprozessmodells durch eine Testkomponente

Die Komponente PreisTest erhält die Preisgestaltung der noch im Betrieb befindlichen alten Mangement-Komponente und die zu Testzwecken instanzierten neuen Management-Komponente 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 aufwendig ist.

Das dynamische Verhalten des System ist in Abbildung 5.11 durch ein UML-Sequenzdiagramm dargestellt. Die Produktion und der Verkauf fungieren als Testtreiber, das bedeutet, sie versorgen die Testobjekte mit Daten. Die orginale Management-Komponente wird bei der Testauswertung als ein Orakel benutzt. Die Vergleichskomponente Preisvergleich vergleicht die Ergebnisse des Orakels mit den Ausgaben der Testobjekte.

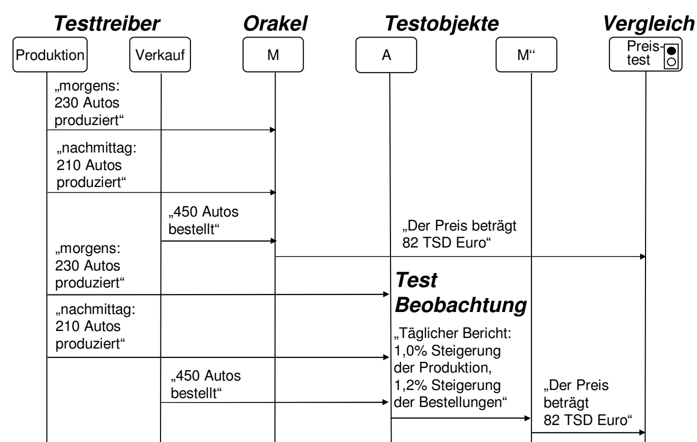


Abbildung 5.11: Der Testablauf als Sequenzdiagramm

6 Weitere Fragen

Der für dieses Bericht verwendete 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 Codierung 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] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon School of Computer Science, January 1997. Ebenfalls CMU Technical Report CMU-CS-97-144.

[Bec03] K. Beck. *Extreme Programming : die revolutionäre Methode für Softwareentwicklung in kleinen Teams*. Programmer's choice. Addison-Wesley, 2003. Einheitssachtitel: Extreme programming explained „dt.".

[Béz05] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 2005. erscheint 2005.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer-Verlag, 2001.

[BSF02] M. Boger, T. Sturm, and F. Fragemann. Refactoring Browser for UML. In Wells and Williams [WW02], pages 77–81.

[Coc02] A. Cockburn. *Agile Software-Entwicklung*. Addison-Wesley, 2002. Einheitssachtitel: Agile Software Development „dt.".

[CW04] A. L. Correa and C. M. L. Werner. Applying refactoring techniques to uml/ocl models. In T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, editors, *UML*, number 3273 in Lecture Notes in Computer Science, pages 173–187. Springer-Verlag, 2004.

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

[Dij70] E. W. Dijkstra. Notes on structured programming. Private Korrespondenz, April 1970.

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.

[Fow00] Martin Fowler. *Refactoring : Wie Sie das Design vorhandener Software verbessern*. Professionelle Softwareentwicklung. Addison-Wesley, 2000. Einheitssachtitel: Refactoring : improving the design of existing code „dt.“.

[GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software*. Programmer's Choice. Addison-Wesley, July 2004. Einheitssachtitel: Design patterns „dt.“.

[Gla98] Robert L. Glass. Maintenance: Less is not more. *IEEE Software*, 15(4):67–68, 1998.

[Gla04] Robert L. Glass. Learning to distinguish a solution from a problem. *IEEE Software*, 21(3):111–112, 2004.

[Gro02] The Object Management Group. Request for proposals: Mof 2.0 query / views / transformations, April 2002.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[HKKI04] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In F. Bomarius and H. Iida, editors, *PROFES*, number 3009 in Lecture Notes in Computer Science, pages 220–233. Springer-Verlag, 2004.

[JO93] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In Shojiro Nishio and Akinori Yonezawa, editors, *ISOTAS*, number 742 in Lecture Notes in Computer Science, pages 264–278. Springer-Verlag, 1993.

[JR01] C. Jacobi and B. Rumpe. Hierarchical XP. Improving XP for Large-Scale Projects in Analogy to Reorganization Processes. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, pages 83–102. Addison-Wesley, 2001.

[Ker05] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley signature series. Addison-Wesley, 2005.

[MDA] Model driven architecture. http://www.omg.org/mda/.

[MDJ02] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *ICGT 2002*, volume 2505 of *Lecture Notes in Computer Science*, 2002. TODO.

[Mic] Microsoft Coorperation. Domain specific language (dsl) tools.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[MW01] Tom Mens and Michel Wermelinger. Proceedings of theworkshop 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. TODO.

[MW02] Tom Mens and Michel Wermelinger. Separation of concerns for software evolution. *Journal of software maintenance and evolution : research and practice*, 14(5):311–315, 2002.

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

[PR97] J. Philipps and B. Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97*. IEEE CS Press, 1997.

[PR99] J. Philipps and B. Rumpe. Refinement of Pipe And Filter Architectures. In *FM'99, LNCS 1708*, pages 96–115, 1999.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3:253–263, 1997.

[RS02] B. Rumpe and A. Schröder. Quantitative Survey on Extreme Programming Projects. In Wells and Williams [WW02], pages 43–46.

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

[Rum04] Bernhard Rumpe. *Modellierung mit UML : Sprache, Konzepte und Methodik*. Xpert.press. Springer-Verlag, 2004.

[Rum05] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Xpert.press. Springer-Verlag, 2005.

[RvdHMRM04] Roshanak Roshandel, André van der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae – a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276, 2004.

[Sie04] Johannes Siedersleben. *Moderne Software-Architektur : Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, 2004.

[SS04] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with kaba. In *Proceedings of OOPSLA 04*, 2004.

[SSL01] Frank Simon, Frank Steinbückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of European Conference Software Maintenance and Reengineering*, pages 157–169, 2001.

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

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

[TDDN01] Sander Tichelaar, Stéphanie Ducasse, Serge Demeyer, and Oscar Nierstrasz. Refactoring uml models. In *Proceedings of UML 01*, volume 2185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, 2003.

[TS01] J.-P. Tolvanen and Kelly S. Domain-specific modeling: 10 times faster than uml. In *Proceedings of Embedded Systems Conference, Stuttgart, Germany,*, 2001.

[vEM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In Arie van Deursen, editor, *Proceedings of WCRE02*. IEEE Computer Society Press, 2002.

[vGSMD03] Pieter van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent uml refactorings. In *Proceedings of UML 03*, 2003.

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

[WW02] D. Wells and L. A. Williams, editors. *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*, volume 2418 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[ZYXX02] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. Change impact analysis to support architectural evolution. *Journal of software maintenance and evolution : research and practice*, 14(5):317–333, 2002.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2000-09

2000-09	T. Firley	Regular languages as states for an abstract automaton
2001-01	K. Diethers	Tool-Based Analysis of Timed Sequence Diagrams
2002-01	R. van Glabbeek, U. Goltz	Well-behaved Flow Event Structures for Parallel Composition and Action Refinement
2002-02	J. Weimar	Translations of Cellular Automata for Efficient Simulation
2002-03	H. G. Matthies, M. Meyer	Nonlinear Galerkin Methods for the Model Reduction of Nonlinear Dynamical Systems
2002-04	H. G. Matthies, J. Steindorf	Partitioned Strong Coupling Algorithms for Fluid-Structure-Interaction
2002-05	H. G. Matthies, J. Steindorf	Partitioned but Strongly Coupled Iteration Schemes for Nonlinear Fluid-Structure Interaction
2002-06	H. G. Matthies, J. Steindorf	Strong Coupling Methods
2002-07	H. Firley, U. Goltz	Property Preserving Abstraction for Software Verification
2003-01	M. Meyer, H. G. Matthies	Efficient Model Reduction in Non-linear Dynamics Using the Karhunen-Loève Expansion and Dual-Weighted-Residual Methods
2003-02	C. Täubner	Modellierung des Ethylen-Pathways mit UML-Statecharts
2003-03	T.-P. Fries, H. G. Matthies	Classification and Overview of Meshfree Methods
2003-04	A. Keese, H. G. Matthies	Fragen der numerischen Integration bei stochastischen finiten Elementen für nichtlineare Probleme
2003-05	A. Keese, H. G. Matthies	Numerical Methods and Smolyak Quadrature for Nonlinear Stochastic Partial Differential Equations
2003-06	A. Keese	A Review of Recent Developments in the Numerical Solution of Stochastic Partial Differential Equations (Stochastic Finite Elements)
2003-07	M. Meyer, H. G. Matthies	State-Space Representation of Instationary Two-Dimensional Airfoil Aerodynamics
2003-08	H. G. Matthies, A. Keese	Galerkin Methods for Linear and Nonlinear Elliptic Stochastic Partial Differential Equations
2003-09	A. Keese, H. G. Matthies	Parallel Computation of Stochastic Groundwater Flow
2003-10	M. Mutz, M. Huhn	Automated Statechart Analysis for User-defined Design Rules
2004-01	T.-P. Fries, H. G. Matthies	A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods
2004-02	B. Mathiak, S. Eckstein	Automatische Lernverfahren zur Analyse von biomedizinischer Literatur
2005-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme
2005-02	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization
2005-03	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling
2005-04	H. Krahn, B. Rumpe	Evolution von Software-Architekturen