Fachhochschul-Diplomstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Refactoring-Muster der WAM-Modellarchitektur

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Ingenieur (Fachhochschule)

Eingereicht von

Stefan Michael Hofer

Betreuer: Dipl.-Inform. Joachim Sauer, C1 WPS GmbH, Hamburg
Begutachter: Mag. Dr. Berthold Kerschbaumer

Juni 2006

Inhaltsverzeichnis

Kurzfassung **iii**

Abstract **iv**

Danksagung **vi**

Erklärung **vii**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Fragestellung und Zielsetzung . . . 2
1.3 Übersicht . . . 2

2 Grundlagen **3**
2.1 Der WAM-Ansatz . . . 3
2.1.1 Überblick über den WAM-Ansatz . . . 3
2.1.2 Konzeptionsmuster . . . 4
2.1.3 Entwurfsmuster . . . 5
2.1.4 Wichtige WAM-Regeln . . . 7
2.1.5 Das JWAM-Rahmenwerk . . . 8
2.2 Refactoring . . . 8
2.3 WAM-Refactorings . . . 9
2.4 Muster . . . 10

3 Vom Muster zum Musterkatalog **11**
3.1 Fokus und Gültigkeit der Refactorings . . . 11
3.2 Identifikation möglicher Refactorings . . . 12
3.3 Struktur der Muster . . . 12
3.4 Unit-Tests und Refactoring . . . 13
3.5 Gliederung des Katalogs . . . 14

4 Katalog von WAM-Refactorings **17**
4.1 Modellierung korrigieren . . . 17
4.1.1 Ersetze Wert durch Fachwert . . . 17
4.1.2 Ersetze allgemeinen Fachwert durch speziellen Fachwert . . . 20
4.1.3 Ersetze Material durch Fachwert . . . 24
4.2 Wiederverwendbarkeit erhöhen . . . 28
4.2.1 Baue fachliche Materialhierarchie auf . . . 28
4.2.2 Lasse Werkzeug mit Aspekt arbeiten . . . 31
4.2.3 Verschiebe Fachlogik in Service . . . 34
4.3 Komplexität beherrschen . . . 38
4.3.1 Extrahiere Automat . . . 38
4.3.2 Zerlege Werkzeug in kleinere Werkzeuge . . . 43

5 Schlusswort **49**
5.1 Zusammenfassung 49
5.2 Bewertung 49
5.2.1 Nutzen des Musterkatalogs 49
5.2.2 Grenzen zwischen Refactoring und Erweiterung 50
5.2.3 Technische WAM-Refactorings 50
5.3 Ausblick 50
5.3.1 Weitere Kategorien und Muster 50
5.3.2 WAM-spezifische Smells 51
5.3.3 Automatisierung von WAM-Refactorings 51
5.3.4 Refactorings in Modellarchitekturen 51

Literaturverzeichnis **52**

Kurzfassung

In dieser Arbeit werden Refactorings untersucht, die spezifisch für den Werkzeug-und-Material-Ansatz (WAM-Ansatz) sind. Durch die Beschreibung dieser Refactorings soll deren Anwendung bei der Entwicklung von Software nach dem WAM-Ansatz erleichtert werden. Ein „Katalog” genanntes Verzeichnis beinhaltet die als Muster beschriebenen und kategorisierten Refactorings. Mit Hilfe dieses Katalogs werden folgende Fragen beantwortet:

1. Welche Refactorings sind spezifisch für den WAM-Ansatz?
2. Wie sieht eine für Entwickler nützliche Beschreibung dieser Refactorings aus?
3. Wie lassen sich WAM-spezifische Refactorings kategorisieren?
4. Schränkt der WAM-Ansatz die Durchführbarkeit bestimmter Refactorings durch die Berücksichtigung fachlicher Strukturen ein?

Die vorgestellten Refactorings sind Verallgemeinerungen von Refactorings, die in einem Software-Projekt umgesetzt wurden. Eine für Software-Entwickler nützliche Beschreibung muss sowohl die Motivation der Refactorings und eine Anleitung zur Umsetzung enthalten, als auch allgemein anwendbar sein. Mit Mustern lassen sich diese Anforderungen umsetzen. Daher wurde eine Musterbeschreibung entwickelt, die an bewährte Musterkataloge angelehnt ist. Ein Teil der Musterbeschreibung behandelt die Wechselwirkung zwischen WAM-spezifischen und unspezifischen Refactorings. Die Arbeit mit Refactoring-Mustern kann durch eine Gliederung des Katalogs vereinfacht werden. Verschiedene Gliederungskriterien werden zu diesem Zweck untersucht und bewertet. Die Gliederung nach dem Zweck der Refactorings ist wegen ihrer Problemorientierung die beste der diskutierten Möglichkeiten.

Der Katalog gliedert sich in drei Teile mit insgesamt acht Refactorings:

- Modellierung korrigieren
  - Ersetze Wert durch Fachwert
  - Ersetze allgemeinen Fachwert durch speziellen Fachwert
  - Ersetze Material durch Fachwert
- Wiederverwendbarkeit erhöhen
  - Baue fachliche Materialhierarchie auf
  - Lasse Werkzeug mit Aspekt arbeiten
  - Verschiebe Fachlogik in Service
- Komplexität beherrschen
  - Extrahiere Automat
  - Zerlege Werkzeug in kleinere Werkzeuge

Abstract

This thesis characterizes and categorizes refactorings that are specific to the Tools & Materials Approach (T&M Approach) by describing them as patterns. The resulting catalog of refactoring patterns answers the following questions:

1. Which refactorings are specific to the T&M Approach?
2. How can they be described so that software developers can make use of that description?
3. How can those refactorings be categorized?
4. Which effect does the T&M Approach have on the applicability of refactorings?

With help of the catalog, T&M software developers are able to apply refactorings more easily and safely.

Refactoring, the improvement of code quality without altering its behavior, is technically motivated and has become a well-established technique in software engineering in recent years. In contrary to refactoring, the T&M Approach is domain driven. Domain concepts are represented by metaphors, including "tool", "material" (thus the name Tools & Materials Approach), "automaton", and "service". The use of such metaphors makes it possible for developers and domain experts to talk in the same language and to reflect the application domain by the way the inner structure of software is organized. This is called "structural similarity". The approach also formulates design patterns and rules for the implementation of the metaphors. They are combined in the T&M model architecture. The model architecture is common to all software developed with the approach and makes it possible to identify distinct refactorings. These are aware of the significance of the code's structure for sustaining structural similarity between the software architecture and the application domain. Such refactorings were named "T&M refactorings" and feature three characteristics: Firstly, they are refactorings, which means that they must not alter the code's functionality. Secondly, they may change how the domain is modeled. Any change has to conform to the rules of the T&M model architecture. The last characteristic is that their application may be triggered by a change of the domain's model.

The refactoring patterns collected were derived from refactorings applied in a real-world project. Describing refactorings as patterns offers several benefits. Patterns present the core of a solution to a recurring problem in such a way that the solution can be used over and over without ever doing it the same way twice. This means that the solution is abstracted away from project specifics. Also, patterns have proofed successful in many areas like presenting solutions for common object-oriented design issues.

The pattern description can either focus on the abstract concepts of the T&M model architecture or on their platform-specific implementation. The latter was chosen, as with the JWAM framework a reference implementation is available for the Java platform. Describing patterns at a platform specific layer increases their applicability to that platform. It omits variations of the description caused by different technological means of how to implement the concepts of the model architecture. Nevertheless, the refactorings are applicable to all T&M software because they are tied to the model architecture. However, the description assumes that the JWAM framework is used.

To detail the patterns, their structure is of significant importance. Several approved pattern catalogs served as examples for useful pattern structures. The pattern description starts with a meaningful, characterizing name and a short description of the problem and the solution. Then, the motivation for this refactoring is explained and possible contraindications are discussed. An instruction consisting of single steps

explains how the refactoring is applied and is followed by an example of such an application. Concluding, consequences such as interrelation of non-specific refactorings are presented.

To work with the refactoring patterns efficiently, the catalog has to be organized so that refactorings can be found more quickly. Furthermore, this makes it easier to compare refactorings that address similar problems. Several criteria for grouping were evaluated, including which element of the model architecture is affected or what kind of structural change the refactoring causes. Since problem orientation was considered the most important characteristic for the organization of the refactorings, they are grouped by their main purpose. Again, this accords to other pattern catalogs. The T&M refactorings are grouped in the categories c*orrecting modeling*, *increasing reusability*, and *handling complexity*. The refactorings in the first category change how an artifact of the application domain is represented in the software. As a result, the structural similarity between the domain and the software model is improved. Increased usability tackles one of the main causes of erroneous software: duplicated logic. The complexity caused by bulky classes can be reduced by splitting them up into smaller ones and letting them work together according to the T&M model architecture.

In total, eight T&M refactorings were found. The ones that correct modeling are:

- Replace value by domain value: A domain artifact is not represented appropriately by a standard value type. Replace it by a custom-made value type.
- Replace domain value by specialized domain value: The domain value that currently represents a domain artifact is too generic. Replace it by a specialized version of that domain value.
- Replace material by domain value: A material behaves like a value. Replace the material by a domain value.

The following three refactorings increase reusability.

- Introduce domain driven hierarchy of materials: Several materials show similarities. Extract that similarities by introducing a hierarchy of materials that is both good object-oriented design and domain driven.
- Let a tool work with an aspect rather than a material: A tool needs to perform the same operations on several similar materials. Let the tool work with an aspect that is provided by all these materials.
- Move domain logic to service: The same domain logic was implemented several times. Move it to a service and replace the implementations by service calls.

Two of the refactorings found deal with handling complexity.

- Extract automaton: Tools or services perform connected steps of a business process. Extract those steps and move them to an automaton.
- Split tool into smaller tools: A tool has many responsibilities and is thus very complex. Find self-contained subtasks and move them into their own tool.

In conclusion, this paper shows that there are refactorings that are specific to the T&M Approach because they affect the elements of its model architecture. By describing them as patterns and categorizing them, they become applicable more easily. Thus, the catalog is a useful collection of solutions to recurring refactoring needs in software that uses the JWAM framework. The pattern description takes into account that non-specific refactorings might interfere with the T&M model architecture by accidentally changing the code's structure. Because most of the presented T&M refactorings are either specializations of other refactorings or consist of other refactorings, it is explained why T&M refactorings are better suited for T&M software than non-specific refactorings.

Danksagung

Ich möchte mich bei allen Menschen bedanken, die an der Entstehung dieser Arbeit mitgewirkt haben.

Bei meinem Betreuer Dipl.-Inform. Joachim Sauer kann ich mich gar nicht genug bedanken. Seine Anregungen und Korrekturen haben wesentlich zum Gelingen dieser Arbeit beigetragen. Auch bei meinem zweiten Betreuer, Mag. Dr. Berthold Kerschbaumer, bedanke ich mich für die Unterstützung. Wertvolle Rückmeldung erhielt ich durch die Präsentation meiner Diplomarbeit im Seminar *Ausgewählte Themen der Softwaretechnik* der Universität Hamburg. Ich danke allen Teilnehmern, insbesondere Prof. Dr.-Ing. Heinz Züllighoven, der mir einen Termin des Seminars zur Verfügung gestellt hat.

Für ihre Korrekturen und Ideen danke ich meinen Studienkollegen Lukas Burgstaller und Christoph Stichlberger.

Durch meine Entscheidung, diese Arbeit in Hamburg zu schreiben, konnte ich in den letzten Monaten nur selten bei meiner Familie und meinen Freunden sein. Ich danke euch für euer Verständnis und dass ihr euch bei meinen seltenen Besuchen für mich Zeit genommen habt.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hamburg, am 21. Juni 2006

Stefan Hofer

Kapitel 1

Einleitung

„Here's a challenge for you. Write a short description that tells someone how to tie bows in their shoelaces. Go on, try it! [...] Some things are better done than described."
Andrew Hunt und David Thomas in *The Pragmatic Programmer: From Journeyman to Master* [Hunt99, S. 218].

1.1 Motivation

Eine *Refactoring* genannte Entwicklungstechnik hat in den vergangenen Jahren einen hohen Stellenwert in der Software-Entwicklung eingenommen ([Kerievsky05], [Hunt99]). Diese Technik gehört „mittlerweile zum elementaren Handwerkszeug eines jeden Entwicklers" ([Lippert05, S. 52]). Auch in der Entwicklung von Software-Systemen nach dem Werkzeug-und-Material-Ansatz (WAM-Ansatz) findet Refactoring als etablierte Software-Entwicklungsmethode Anwendung. Der am Arbeitsbereich Softwaretechnik der Universität Hamburg entwickelte WAM-Ansatz legt durch sein Streben nach Software mit hoher Benutzungsqualität einen zyklisch-evolutionären Entwicklungsprozess nahe ([Züllighoven05]). Refactoring ist eine der grundlegenden Techniken solcher, *agil* genannten, Prozesse: Anforderungen werden erst nach und nach formuliert und können sich häufig ändern. Um keine unnötige Arbeit zu verrichten, soll die zu entwickelnde Software immer die einfachste Lösung zur Erfüllung der gerade relevanten Anforderungen sein ([Beck04]). Wird eine solche, einfache Lösung erweitert oder geändert, muss die Implementierung konsolidiert werden, um weiterhin Qualitätskriterien wie Verständlichkeit, Wartbarkeit und Erweiterbarkeit zu erfüllen. Dieser Vorgang wird als *Refactoring* bezeichnet. Er ist fix in der Philosophie agiler Software-Entwicklung verankert, und zwar als Schritt zwischen der Implementierung einer Funktionalität und dem darauf folgenden weiteren Ausbau einer Software. Die Existenz prägnanter Sprüche wie „First make it run, then make it right." im Umfeld agiler Entwicklung unterstreicht die Akzeptanz dieser Vorgehensweise bei den Entwicklern.

Viele Refactorings gleichen sich, was Problemstellung, Umsetzung und Lösung betrifft. Martin Fowler hat in seinem Buch *Refactoring: Improving The Design of Existing Code* ([Fowler00]) eine Sammlung („Katalog") von Refactorings veröffentlicht, in der wiederkehrende Refactorings als Muster beschrieben werden, um sie allgemein einsetzbar zu machen. Durch die in der Musterbeschreibung enthaltene Anleitung verfügen Entwickler über ein Werkzeug zur Umsetzung von Refactorings in objektorientierter Software.

Nach dem WAM-Ansatz gebaute Software-Systeme haben neben ihrer agilen Entwicklung auch einen gemeinsamen Architekturstil. Als Entwickler eines solchen Systems fiel mir auf, dass einige darin vorgenommene Refactorings entweder gar nicht in Fowlers Refactoring-Sammlung enthalten waren oder als spezielle Ausprägung eines dort aufgeführten Refactorings aufgefasst werden konnten. In jedem dieser Fälle gab es einen Zusammenhang zwischen dem Refactoring und dem WAM-typischen Architekturstil. Eine mit Fowlers Sammlung vergleichbare Aufarbeitung solcher Refactorings könnte deren Anwendung einfacher, sicherer und effizienter machen.

1.2 Fragestellung und Zielsetzung

Diese Arbeit soll folgende Fragen beantworten:

1. Welche Refactorings sind spezifisch für den WAM-Ansatz?
2. Wie sieht eine für Entwickler nützliche Beschreibung dieser Refactorings aus?
3. Wie lassen sich WAM-spezifische Refactorings kategorisieren?
4. Schränkt der WAM-Ansatz die Durchführbarkeit bestimmter Refactorings durch die Berücksichtigung fachlicher Strukturen ein?

Die erste Frage soll durch die Verallgemeinerung von Refactorings beantwortet werden, die in einem kommerziellen Entwicklungsprojekt umgesetzt wurden. Dazu wurde zunächst überprüft, ob vorgenommene Änderungen am Quelltext als Refactorings aufgefasst werden können. Solche Änderungen wurden dann auf ihre Auswirkung auf die Elemente der Architektur überprüft. Danach erfolgte die Abstrahierung des Refactorings von den projektspezifischen Architekturelementen hin zu den in der WAM-Modellarchitektur (siehe 2.1.1) definierten Elementen. Gesammelt werden die Beschreibungen dieser Refactorings in einem Katalog WAM-spezifischer Refactorings („WAM-Refactorings"). Das führt zur zweiten Frage, bei der die Art der Beschreibung der gefundenen Refactorings zur Diskussion steht. Diese soll es Entwicklern erlauben, die gefundenen Refactorings ähnlich denen in Fowlers Refactoring-Katalog einzusetzen. Dazu werden die Refactorings in Form von Mustern beschrieben, die sich an bewährten Mustern orientiert. Eine Sammlung von Refactoring-Mustern muss strukturiert werden, um damit effizient arbeiten zu können. Das wirft die dritte Fragestellung auf. Beantwortet wird sie durch die Vorstellung und Bewertung möglicher Kriterien zur Gliederung des Katalogs. Auf die letzte Frage wird in der Beschreibung der einzelnen Refactorings eingegangen. Dort werden gegebenenfalls Wechselwirkungen zwischen WAM-spezifischen und unspezifischen Refactorings erläutert.

Nicht Teil der Zielsetzung sind umfangreiche Erläuterungen zu WAM und Refactoring. Die Grundlagen des benötigten Wissens werden am Beginn der Arbeit dargelegt, um Lesern ohne speziellem Vorwissen das Verständnis dieser Arbeit zu ermöglichen. Für detaillierte Informationen zu den behandelten Themen wird jedoch auf die zitierte Literatur verwiesen.

Lesern, die keine Software nach dem WAM-Ansatz entwickeln, soll diese Arbeit als Beispiel für Musterbeschreibungen und den Ablauf von Refactorings in der Praxis dienen. Wer den WAM-Ansatz zur Entwicklung von Software einsetzt, kann nicht nur das Problembewusstsein für die Anwendung von Refactorings im Umfeld von WAM schärfen, sondern die Arbeit auch als Anleitung zur Identifikation und Umsetzung von Refactorings im eigenen Projekt heranziehen.

1.3 Übersicht

In Kapitel 2 werden die Grundlagen dieser Arbeit erläutert und benötigte Begriffe eingeführt. Neben etabliertem Wissen über den WAM-Ansatz, Refactorings und Muster gehören dazu Zusammenhänge zwischen diesen Gebieten sowie selbst definierte Begriffe.

Das nachfolgende Kapitel 3 spannt den Bogen von der Problemstellung der Arbeit hin zur Lösung. Dazu wird der Identifikationsprozess von Refactoring-Potentialen beleuchtet und danach ein Beschreibungsschema für WAM-Refactorings entwickelt. Anschließend werden mögliche Gliederungskriterien der WAM-Refactorings untersucht, um aus einer Ansammlung von Refactorings einen Refactoring-Katalog erstellen zu können.

Den Hauptteil der Arbeit bildet Kapitel 4, der Katalog von WAM-Refactorings. Der Katalog ist dreigeteilt und enthält insgesamt acht Refactorings.

Abgeschlossen wird die Arbeit durch eine Zusammenfassung und Bewertung der Ergebnisse sowie durch einen Ausblick auf weitere Fragestellungen im Umfeld von Refactoring und WAM-Ansatz.

Kapitel 2

Grundlagen

2.1 Der WAM-Ansatz

Dieser Abschnitt gibt einen Überblick über den WAM-Ansatz und die WAM-Modellarchitektur. Die für diese Arbeit relevanten Teile werden schrittweise näher erläutert, von den Konzepten bis hin zu Konstruktionsrichtlinien. Um die Umsetzung der Konzepte geht es auch in der Vorstellung des JWAM-Rahmenwerks. Detaillierte Informationen über WAM bietet das Hauptwerk zum WAM-Ansatz ([Züllighoven05]).

2.1.1 Überblick über den WAM-Ansatz

Der Werkzeug-und-Material-Ansatz (WAM-Ansatz) ist ein Ansatz für die Entwicklung anwendungsorientierter Software. Unter Anwendungsorientierung versteht man das Bestreben, die Arbeit der zukünftigen Benutzer eines Software-Systems optimal zu unterstützen. Die Struktur anwendungsorientierter Software orientiert sich an den Konzepten und Beziehungen der Anwendungsdomäne. In [Züllighoven05, S. 102] werden drei Charakteristika anwendungsorientierter Software aufgeführt:

- „The functionality of a software system is oriented to the tasks involved in the application domain."
- „The manipulation of a software system is user-friendly."
- „The processes and interactions defined within a software system can be easily adapted to the actual requirements of a working context."

Mit Hilfe von Metaphern aus der menschlichen Arbeitsorganisation werden fachliche Strukturen in Software abgebildet. Ausgehend von Leitmetaphern wie „Expertenarbeitsplatz" wird mit Entwurfsmetaphern (z.B. Werkzeug, Material) eine Vision der zu erstellenden Software formuliert, wodurch Entwickler und zukünftige Benutzer mit gleichem Vokabular über die Software sprechen können. Diese Strukturähnlichkeit zwischen realer Arbeitsorganisation in den Fachabteilungen und der Organisation von Quelltext ist eine der zentralen Ideen des WAM-Ansatzes. Es genügt nicht, dass die Anwender ihre Arbeitsorganisation in der Software widergespiegelt sehen; auch auf der Architekturebene der Software müssen diese Strukturen erkennbar sein. Nur so können beide Seiten von der Strukturähnlichkeit profitieren: Die Benutzer können mit der neuen Software ihre Aufgaben in einer ihnen vertrauten Art erledigen, und die Entwickler können die fachlichen Strukturen als Orientierungshilfe für den (Aus-)Bau der Software verwenden ([Züllighoven05]).

Eine anwendungsorientierte Software-Architektur kann nicht unabhängig von der Anwendungsdomäne entworfen werden. Daher werden die Strukturierungsprinzipien des WAM-Ansatzes in einer so genannten *Modellarchitektur* festgehalten. In [Züllighoven05] wird der Begriff wie folgt definiert:

> „A model architecture abstracts from the characteristic features of a set of similar software architectures. It defines the kind of elements used, their connection, and the rules of their combination. In addition, a model architecture includes criteria for the composition of elements

> into modeling and construction units, and guidelines for the desing and quality of a specific architecture." [Züllighoven05, S. 282]

Die Elemente der WAM-Modellarchitektur sind nach den Entwurfsmetaphern benannt, daher auch das Akronym *WAM* als Abkürzung für *Werkzeug*, *Automat* und *Material*. Dies sind neben *Service* die wichtigsten Entwurfsmetaphern. Viele Regeln der WAM-Modellarchitektur betreffen verbotene oder notwendige Beziehungen zwischen Elementen. Einige der wichtigsten werden in Abschnitt 2.1.4 aufgeführt. Die Einhaltung bzw. Verletzung der Regeln ist eine der in der Definition angesprochenen Qualitätsrichtlinien. Als Entwurfsrichtlinie finden sich zwei Arten von Mustern, die auf den Entwurfsmetaphern basieren:

- Konzeptionsmuster dienen zur Modellierung des Anwendungsbereichs durch die Entwickler.
- Entwurfsmuster sind eine Konstruktionsanleitung für die Umsetzung der modellierten Elemente.

Die für diese Arbeit relevanten Muster sind in den Abschnitten 2.1.2 und 2.1.3 zusammengefasst.

Zur Unterstützung bei der Implementierung von Software nach dem WAM-Ansatz gibt es das in Java geschriebene JWAM-Rahmenwerk. Es ist nicht domänenspezifisch und bietet unter anderem Funktionalität für den Umgang mit Elementen der Modellarchitektur und deren Darstellung (siehe 2.1.5).

Der WAM-Ansatz befasst sich nicht nur mit Software-Architektur, sondern auch mit Projekt- und Produktentwicklung, für die Vorgehensmodelle und Dokumententypen vorgeschlagen werden. In der Projektentwicklung nimmt die Aufteilung in Kernsystem und Ausbaustufen eine zentrale Stelle ein, in der Produktentwicklung das iterativ-zyklische Vorgehensmodell. Für ein solches Modell sieht der WAM-Ansatz in jedem Zyklus ein Review des entstehenden Software-Systems durch die Fachabteilung vor („Autor-Kritiker-Zyklus"). Zahlreiche Projekte haben gezeigt, dass sich so genannte agile Vorgehensmodelle wie *eXtreme Programming* ([Beck04]) gut für die Entwicklung nach dem WAM-Ansatz eignen. Zum einen sind sie iterativ-zyklisch, zum anderen sind in ihnen neben Autor-Kritiker-Zyklen auch noch andere Maßnahmen (unter anderem in den Bereichen Änderungsmanagement und Kundenbeziehung) zur anwendungsorientierten Entwicklung vorgesehen ([Beck01]). Das Vorgehen in kleinen Schritten, in denen vorhandener Quelltext adaptiert und erweitert wird, legt besondere Implementierungstechniken nahe. Dazu gehört, vorhandenen Quelltext bei Bedarf vor einer Änderung erst in eine einfachere Form zu bringen (*Refactoring*, siehe 2.2), um ihn besser anpassen zu können. Damit beim häufigen Ändern von bestehendem Quelltext keine Fehler eingebaut werden, können automatisierte Tests *(Unit Tests*, siehe 3.4) zur Kontrolle eingesetzt werden. Dies ist nur einer von vielen Anlässen, automatisierte Tests zu verwenden.

Abschließend werden einige Begriffe mit Bezug zu WAM eingeführt, die zur sprachlichen Vereinfachung dieser Arbeit gedacht sind.

- Ein ***WAM-Element*** ist ein Element der WAM-Modellarchitektur, beispielsweise ein Werkzeug oder Material.
- Als ***WAM-Modell*** wird die Abbildung fachlicher Strukturen in WAM-Elemente bezeichnet. Als Mittel dazu dienen die Konzeptionsmuster.
- Unter ***WAM-System*** ist ein nach dem WAM-Ansatz gebautes Software-System zu verstehen. Die Architektur des Software-Systems kann als Exemplar der WAM-Modellarchitektur gesehen werden, d.h. sie gehorcht den Regeln der WAM-Modellarchitektur.

2.1.2 Konzeptionsmuster

Konzeptionsmuster setzen Entwurfsmetaphern um. Die wichtigsten Konzepte für die in dieser Arbeit relevanten Metaphern sind im Folgendenzusammengefasst.

Materialien sind die Arbeitsgegenstände oder Abstraktionen, die Benutzer im Rahmen ihrer Tätigkeiten mit Werkzeugen betrachten, ändern und erzeugen. An ihnen sollen nur fachlich sinnvolle Operationen ausführbar sein. Materialien werden nie selbst aktiv. Konten, Ordner und Dokumente werden in der Regel als Materialien abgebildet.

Werkzeuge bieten Sichten auf Materialien an und erlauben Benutzern, Materialzustände zu sondieren und zu verändern. Alle diese Funktionalitäten sind fachlich motiviert, d.h. sie unterstützen die Benutzer bei der Bewältigung ihrer Aufgaben. Ein Beispiel für ein Werkzeug ist ein Bearbeiter von Personendaten.

Automaten sind den Werkzeugen ähnlich, arbeiten aber ohne Benutzerinteraktion auf Materialien. Sie übernehmen immer gleichförmige Routinetätigkeiten und können bei Bedarf vor dem Start der Tätigkeit konfiguriert werden. Typischerweise arbeiten Automaten nach dem Start längere Zeit selbstständig als Hintergrundprozess. Datenarchivierung oder Dokumenterstellung lässt sich gut mit Automaten unterstützen.

Services (auch: *Dienstleister*) fassen Fachlogik zusammen, die unabhängig von konkreten Werkzeugen oder Automaten ist. Sie arbeiten ebenfalls mit Materialien, stellen diese aber im Unterschied zu Werkzeugen nicht dar. Die Bereitstellung von Materialien aus externen Systemen (Datenbanken, Netzwerke etc.) lässt sich etwa in einem Service implementieren.

2.1.3 Entwurfsmuster

Wie bei den Konzeptionsmustern werden für die Entwurfsmuster die wichtigsten Zusammenhänge für die in dieser Arbeit relevanten Metaphern angeführt. Für die Umsetzung der Entwurfsmuster gibt es Konstruktionsrichtlinien, von denen ebenfalls ein Ausschnitt präsentiert wird.

Fachwerte sind fachlich motivierte Erweiterungen der primitiven Datentypen objektorientierter Programmiersprachen. Solche Werttypen zeichnen sich durch ihre Unveränderbarkeit aus. Beispielsweise bleibt eine *5* immer eine *5* - durch die Addition von *1* entsteht ein neuer Wert *6*. Der Wert *5* hat sich nicht verändert. In objektorientierten Sprachen werden neue Werttypen als Klassen definiert. Werte werden also durch Objekte dargestellt. Um die Wertsemantik dieser Objekte sicherzustellen, dürfen in den Klassen keine ändernden Methoden definiert sein. Zumindest konzeptionell existiert für jeden Wert eines Werttyps nur ein Objekt. Weil dies in Implementierungen nicht zwingend so sein muss, sollten Objektvergleiche anhand des Werts erfolgen, nicht anhand der Identität. Der Wertebereich von Fachwerten wird entweder durch andere Fachwerte definiert (*zusammengesetzte Fachwerte)* oder durch primitive Datentypen der jeweiligen Programmiersprache (*elementare Fachwerte*). Zur internen Repräsentation wird oft ein zu großer Wertebereich verwendet, daher muss bei der Erzeugung von Fachwerten aus externen Repräsentationen (z.B. eine Zeichenkette „1234") sichergestellt werden, dass nur Objekte mit erlaubten Werten erzeugt werden können. Dazu gibt es eigene Methoden zur Überprüfung der Gültigkeit der externen Repräsentation (z.B. Kann aus „1234" eine Kontonummer erzeugt werden?) .

Materialien bestehen aus anderen Materialien oder Fachwerten. Ihre fachlich motivierte Schnittstelle wird als *Aspekt* bezeichnet. Verschiedene Teilmengen der Schnittstelle können in unterschiedlichen Aspekten zusammengefasst werden. Somit kann ein Material über mehrere Aspekte verfügen. WAM-Elemente, die Materialien verwenden, kennen immer nur Aspekte eines Materials[1], wodurch sie mit allen Materialien umgehen können, die über gleiche Aspekte verfügen. Die Trennung von fachlicher Schnittstelle und Material bringt weitere Vorteile, wenn ein Material von unterschiedlichen WAM-Elementen bearbeitet wird. Ändert sich ein Aspekt eines Materials, muss sich das nicht zwangsläufig auf die anderen Aspekte auswirken. Die Verwendung von Aspekten stellt sicher, dass ein Material immer unter einer möglichst schmalen Schnittstelle bekannt ist.

Werkzeuge präsentieren, erzeugen und bearbeiten Materialien. Dazu können sie bei Bedarf andere Werkzeuge, Automaten und Services verwenden. Wenn Werkzeuge nicht alle Eigenschaften eines Materials bearbeiten oder anzeigen, brauchen sie nur einen Teil der Schnittstelle des Materials zu kennen. In diesem Fall, oder wenn an mehreren Materialien die gleichen Arbeiten verrichtet werden sollen, arbeiten Werkzeuge mit Aspekten von Materialien. Anhand ihrer Konstruktionsweise werden zwei Arten von Werkzeugen unterschieden, nämlich *monolithische* und *komplexe Werkzeuge*. Erstere sind einfacher und nur in Oberfläche und Logik gegliedert, während zweitere in Oberfläche, Benutzerinteraktion und Fachlogik aufgeteilt sind. Monolithische Werkzeuge bestehen aus einer Klasse für die GUI und einer Klasse, die sowohl fachliche Operationen anbietet als auch für die Anbindung der GUI sorgt (siehe Abbildung 2.1). In komplexen Werkzeuge sorgt eine *Interaktionskomponente* (*IAK*) dafür, dass die *Funktionskomponente* (*FK*) vollständig von der GUI abstrahiert und sich auf die Fachlogik beschränken kann. GUI, IAK und FK werden von einer *Werkzeugklasse* umschlossen (siehe Abbildung 2.2). Abhängigkeiten zwischen den Kom-

[1] Ein Aspekt kann aber identisch mit der gesamten Schnittstelle eines Materials sein.

ponenten (siehe 2.1.4) werden vermieden, wenn eine der zwei Benutzt-Beziehungen durch lose Kopplung (siehe z.B. *Beobachter* [GoF95]) ersetzt wird. In den Abbildungen 2.1 und 2.2 sind Benutzt-Beziehungen als Pfeile mit durchgezogener Linie und lose Kopplungen als Pfeile mit gestrichelter Linie dargestellt.

Abbildung 2.1: Monolithisches Werkzeug mit Benutzt-Beziehungen

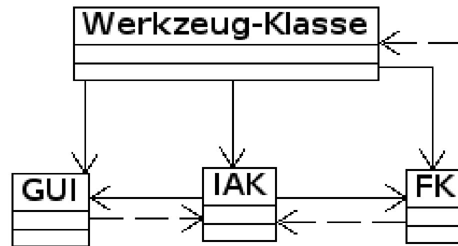

Abbildung 2.2: Komplexes Werkzeug mit Benutzt-Beziehungen

Ein weiteres Merkmal von Werkzeugen ist die Möglichkeit, Werkzeuge ineinander einzubetten. Umfangreiche Aufgaben können zergliedert und auf mehrere Werkzeuge (*Subwerkzeuge*) verteilt werden, die dann von einem weiteren Werkzeug (*Kontextwerkzeug*) aktiviert werden. Dabei können die Oberflächen der Subwerkzeuge entweder in die des Kontextwerkzeugs integriert werden oder sie sind eigenständig (etwa als Dialoge) realisiert. Beispielsweise könnte die Aufgabe „Verwaltung von Patienten in einem Krankenhaus" in zwei Teilaufgaben „Stammdatenverwaltung" und „Verwaltung medizinischer Behandlungen" mit eigenen Werkzeugen aufgeteilt werden. Über ein Kontextwerkzeug „Patientenverwaltung" ließen sich dann die beiden Subwerkzeuge auswählen und anzeigen. Um Abhängigkeiten zwischen den Werkzeugen zu vermeiden (siehe 2.1.4) wird die Kommunikation zwischen den Werkzeugen mit loser Kopplung, etwa den Mustern *Beobachter* oder *Zuständigkeitskette* ([GoF95]) umgesetzt. Abbildung 2.3 zeigt die Beziehungen zwischen Kontext- und Subwerkzeugen. Lose Kopplungen sind als Pfeile mit gestrichelter Linie dargestellt.

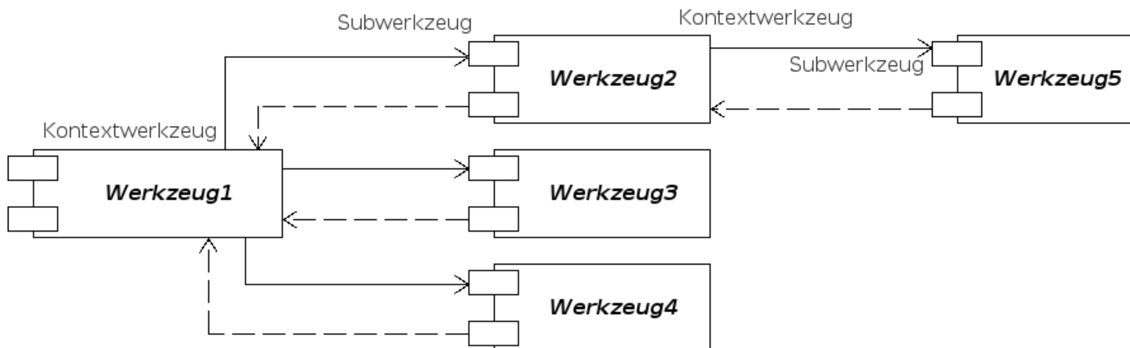

Abbildung 2.3: Kontext- und Subwerkzeuge

Automaten manipulieren Materialien, zeigen diese im Gegensatz zu Werkzeugen aber nicht an. Sie können von Werkzeugen, Services oder anderen Automaten gestartet werden. Wenn ein Automat vorher konfiguriert werden muss, kann der Start über ein eigenes Konfigurationswerkzeug erfolgen. Der Zustand eines Automaten kann mit sondierenden Operationen abgefragt werden, zum Beispiel ob er schon konfiguriert wurde. Mit seinem Aufrufer kann ein Automat über synchrone oder asynchrone Kommunikationswege verbunden sein. Wenn ein gestarteter Automat eine unbestimmte Zeit arbeitet, sollen eventuell benötigte

Ergebnisse mittels asynchroner Kommunikation an den Aufrufer weitergeleitet werden. Synchron gekoppelte Automaten blockieren ihre Aufrufer, wenn diese auf Ergebnisse warten müssen.

Services bieten eine Technologie-unabhängige, fachlich motivierte Schnittstelle. Die Semantik der Schnittstelle bleibt gleich, auch wenn sich die darunter liegende Technologie ändert. Services erwarten und liefern an ihrer Schnittstelle Fachwerte oder Materialien, machen aber keine Annahmen über deren Verwendung. In der Regel sind Services mehrbenutzerfähig und verteilbar. Sie erfordern daher entweder eine entsprechende Zustandsverwaltung oder sind als zustandslose Services implementiert, deren Operationen als atomar betrachtet werden.

2.1.4 Wichtige WAM-Regeln

Die WAM-Modellarchitektur definiert für ihre Elemente Regeln. Einige wichtige Gebots- und Verbots-Regeln werden hier für die vorgestellten WAM-Elemente angeführt. Eine Sammlung dieser Regeln beschreibt [Karstens05].

Fachwerte dürfen von allen WAM-Elementen verwendet werden, selbst aber nur andere Fachwerte kennen. Sie müssen sich wie Werte verhalten und sich extern repräsentieren lassen (z.B. als Zeichenketten).

Materialien dürfen außer anderen Materialien nur Fachwerte kennen. Realisiert ein Material einen Aspekt, so muss es die im Aspekt formulierten Anforderungen erfüllen.

Werkzeuge arbeiten mit Materialien und Fachwerten. Sie können dazu andere Werkzeuge, Services und Automaten benutzen. Für beide Werkzeug-Varianten (komplex und monolithisch) gilt, dass sich ihre Bestandteile nicht gegenseitig kennen dürfen, weil dadurch zyklische Abhängigkeiten entstünden. Zyklen zwischen Teilen[2] von Software verhindern, dass man diese Teile unabhängig voneinander testen und wiederverwenden kann. Auch bei Beziehungen zwischen Werkzeugen sind zyklische Abhängigkeiten nicht erwünscht. Subwerkzeuge dürfen daher weder einander noch ihr Kontextwerkzeug kennen. Nicht zyklische Beziehungen zwischen Subwerkzeugen sind nur erlaubt, wenn sie in einem Kontextwerkzeug-Subwerkzeug-Verhältnis stehen (siehe Abbildung 2.3).

Services und *Automaten* arbeiten mit Materialien und Fachwerten. Sie können dazu andere Services und Automaten benutzen und dürfen keine Werkzeuge kennen.

Abbildung 2.4 fasst die vorgestellten WAM-Elemente und ihre erlaubten Beziehungen zusammen. Reflexive Beziehungen werden in der Abbildung nicht dargestellt.

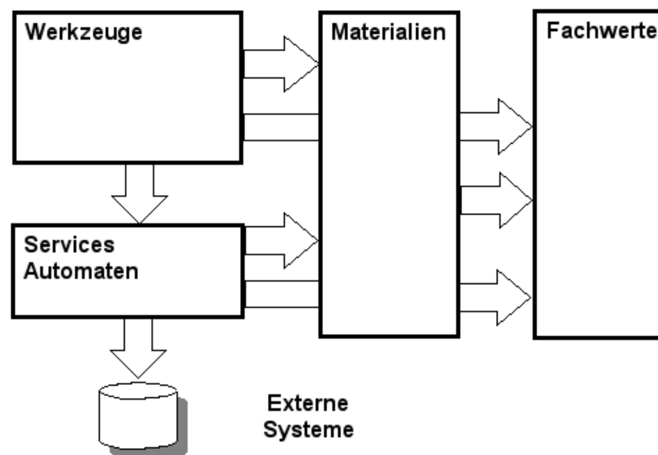

Abbildung 2.4: WAM-Elemente und erlaubte Benutzungsbeziehungen

[2]Zyklen kann es innerhalb verschiedener Ebenen geben, etwa zwischen Klassen, Modulen und Schichten.

2.1.5 Das JWAM-Rahmenwerk

Das JWAM-Rahmenwerk[3] ([JWAM05]) ist ein freies, quelloffenes und nicht domänenspezifisches Java-Rahmenwerk, das die Implementierung von WAM-Systemen in Java unterstützt. Es stellt Interfaces und Klassen für die WAM-Elemente bereit und implementiert die Entwurfsmuster des WAM-Ansatzes. In den Beispielen, die mit dem Rahmenwerk ausgeliefert werden, finden sich einige häufig benötigte Fachwerte. Des Weiteren beinhaltet JWAM eine so genannte Arbeitsumgebung zur Verwaltung von Werkzeugen und Materialien. Dazu gehört unter anderem eine Registratur für Services, Automaten und Werkzeuge, wodurch die Erzeugung von Objekten dieser Typen vereinfacht wird. Häufig benötigte Abläufe, wie das Ausstatten eines Werkzeug mit einer GUI, sind in Klassen des Rahmenwerks festgelegt und können in abgeleiteten Klassen mittels Einschub-Methoden (siehe [GoF95]) angepasst werden. Ein Einbettungsmechanismus erleichtert den Bau von Kontext- und Subwerkzeugen, indem Anfragen von Werkzeugen über Zuständigkeitsketten weitergeleitet werden.

Die in WAM enthaltenen Vorgehensweisen, Dokumenttypen etc. (siehe Abschnitt 2.1.1) sind nicht Teil des JWAM-Rahmenwerks.

Ein Rahmenwerk wie JWAM setzt die Entwurfsmuster mit zur Implementierungssprache passenden Konstruktionsweisen um. Derart konkretisiert wird das Erkennen von fachlichen Strukturen im Quelltext vereinfacht, weil die WAM-Elemente anhand von Klassenhierarchien identifiziert werden können und viele Beziehungen (beispielsweise durch Einschub-Methoden) bereits geknüpft sind. Für Entwickler ergibt sich ein schärferes Bild auf das System, als dies ohne Rahmenwerk der Fall wäre.

2.2 Refactoring

In der Fachliteratur findet sich keine einheitliche Definition für den Begriff „Refactoring”. Definitionen finden sich unter anderem in [Roock04], [Westphal06] und [Kerievsky05]). Häufig zitiert wird die Definition von Martin Fowler:

> „Refactoring [is] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.” [Fowler00, S. 53]

In [Schulz04] wird der Begriff „Refactoring” genau durchleuchtet und auf die Schwachstellen der Definition nach Fowler hingewiesen: Refactorings haben nicht nur einfacheres Verständnis und bessere Erweiterbarkeit zum Ziel. Ein weiterer Zweck kann beispielsweise die Behebung von Architekturverstößen sein. Ein weiterer Schwachpunkt ist die Formulierung „observable behavior”, weil „beobachtbares Verhalten” sich nach Art der Anwendung stark unterscheidet, etwa bei einem Echtzeitsystem im Vergleich zu einer Arbeitsplatz-Software.

Manchmal findet sich in Definitionen der Hinweis auf gleich bleibende Semantik von Quelltext (so in [Roock04]). Eine solche Definition ist für diese Arbeit ungeeignet, weil sich nach dem WAM-Ansatz in der Semantik des Quelltextes die fachlichen Strukturen widerspiegeln. Genau diese Semantik ist Änderungen, etwa durch Erkennen falscher Modellierung oder durch Wachstum einer Software, unterworfen, die keinen Einfluss auf die Funktionalität haben.

Um trotz des Fehlens einer einheitlichen Definition und der Unzulänglichkeiten verbreiteter Definitionen mit klaren Begriffen arbeiten zu können, wird für diese Arbeit eine eigene Definition verwendet:

Definition *Refactoring*: Ein Refactoring ist eine Umstrukturierung von Software, die

- der Qualitätsverbesserung dient,
- die Funktionalität nicht verändert,
- sich in einzelnen Schritten umsetzen lässt und
- nach Ausführung jedes Schritts einen korrekten, lauffähigen Zustand hinterlässt.

[3] Diese Arbeit bezieht sich immer auf JWAM 2.

Die ersten beiden Punkte grenzen ein Refactoring von einer beliebigen Änderung des Quelltextes ab. Refactorings sind zielgerichtet, d.h. mindestens ein Qualitätsmerkmal von Software soll verbessert werden, ohne dabei die Funktionalität zu beeinflussen. Wie in der Definition von Fowler ist es ohne Berücksichtigung des Anwendungsgebiets eines Refactorings nicht möglich, den Begriff „Funktionalität" näher einzugrenzen. Mit dem dritten Punkt wird sichergestellt, dass ein Refactoring als Anleitung beschrieben werden kann. Der letzte Punkt verfeinert den dritten, weil die gewählte Schrittfolge in der Praxis umsetzbar sein muss. Refactorings sind nicht immer atomar und mitunter so umfangreich, dass noch vor der vollständigen Umsetzung veränderter Quelltext in die Produktivversion integriert wird. Dadurch können beispielsweise parallel laufende Entwicklungsarbeiten von den umgesetzten Schritten profitieren.

Anmerkung: In [Schulz04] werden drei Bedeutungen des Begriffs „Refactoring" im Deutschen identifiziert:

1. Als „Schema, welches auf Software angewendet werden kann",
2. als „konkrete Anwendung eines solchen Schemas" und
3. als „allgemeine Disziplin, solche Schemata zu kennen und mit ihnen umzugehen".

In dieser Arbeit kommt „Refactoring" in allen drei Bedeutungen vor. Falls die Bedeutung nicht aus dem Kontext klar wird oder eine Unterstreichung der Bedeutung beabsichtigt ist, wird der Begriff wie folgt umschrieben: „Anwendung" oder „Ausführung eines Refactoring" (zweite Bedeutung) und „Technik des Refactoring" (dritte Bedeutung).

2.3 WAM-Refactorings

Refactoring ist eine wesentliche Technik in agilen Entwicklungsmethoden, die bei der Entwicklung von WAM-Systemen zum Einsatz kommen (zum Zusammenhang zwischen WAM und agiler Entwicklung siehe Abschnitt 2.1.1). Der Begriff ***WAM-Refactoring*** bezeichnet Refactorings, die nur in WAM-Systemen auftreten. Refactorings lassen sich also in WAM-spezifische und unspezifische einteilen. Orthogonal dazu lässt sich unterscheiden, ob Refactorings zu Änderungen in einer fachlich motivierten Architektur führen, beispielsweise der WAM-Modellarchitektur. In solchen Architekturen ist der Quelltext nach fachlichen Einheiten strukturiert und ein Refactoring kann diese Strukturen verändern. Für das Beispiel der WAM-Modellarchitektur heißt das, dass ein Refactoring des Quelltext eine Änderung im WAM-Modell bewirkt. Das Refactoring ändert die Semantik des Quelltextes so, dass im zugrunde liegenden WAM-Modell des Systems ein WAM-Element durch mindestens ein anderes (mitunter vom gleichen Typ) ersetzt wird. Solche Refactorings werden in dieser Arbeit *fachliche Refactorings* (wenn sie die fachlich motivierte Architektur ändern) bzw. *technische Refactorings* (wenn sie diese nicht ändern) genannt. Beispiele für technische Refactorings sind die in [Fowler00] beschriebenen Refactorings, fachliche Refactorings sind etwa die in Kapitel 4 enthaltenen. Tabelle 2.1 stellt die beiden Kategorien von Refactorings gegenüber.

	mit Änderung der fachl. Architektur	**ohne Änderung der fachl. Architektur**
WAM-spezifisch	fachliches WAM-Refactoring	technisches WAM-Refactoring
unspezifisch	fachliches Refactoring	technisches Refactoring

Tabelle 2.1: Kategorien von Refactorings

Anmerkung: Im Folgenden wird *Refactoring* immer als Oberbegriff verwendet. Ist ein technisches, fachliches oder WAM-Refactoring gemeint, wird dies explizit durch die in Tabelle 2.1 angeführten Begriffe gemacht.

Folgende Beispiele sollen die Unterscheidung zwischen technischem und fachlichen WAM-Refactoring erleichtern:

- Der Quelltext eines umfangreiches Werkzeugs, das zur Eingabe und Berechnung von Daten dient, soll übersichtlicher und verständlicher gemacht werden. Dazu fasst man die unterstützen Tätigkeiten fachlich sinnvoll zusammen (etwa in „Eingabe” und „Berechnung”) und teilt sie in zwei separate Werkzeuge auf. Diese können als Subwerkzeuge so in ein Kontextwerkzeug eingebettet werden, dass an der Benutzeroberfläche kein Unterschied zur vorherigen Lösung auszumachen ist. Als Muster ist dieses Refactoring in Abschnitt 4.3.2 beschrieben. Im WAM-Modell des Systems treten dabei an Stelle eines Werkzeugs zwei Werkzeuge, es handelt sich also um ein fachliches WAM-Refactoring.
- Wenn das beschriebene Werkzeug als monolithisches Werkzeug implementiert ist, bietet sich eine andere Möglichkeit zur Beherrschung der Komplexität an. Es kann zu einem komplexen Werkzeug mit separaten Klassen für GUI, Interaktion und fachlicher Funktion umgebaut werden. Ein solches Refactoring verändert nur die Umsetzung des WAM-Elements Werkzeug im Quelltext, nicht aber dessen Rolle im WAM-Modell. Daher ist dies ein Beispiel für ein technisches WAM-Refactoring.

2.4 Muster

Im Bereich der Software-Entwicklung wird unter *Muster* (engl.: *pattern*) oft die Definition des Architekten Christopher Alexander verstanden (siehe [Züllighoven05]):

> „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” Zitiert nach [GoF95, S. 2]

Die drei wichtigsten Punkte der Definition sind:

- Beschreibung des Problems
- Beschreibung der Lösung
- Allgemeine Anwendbarkeit

Sowohl Problem als auch Lösung müssen so beschrieben werden, dass das Muster in seiner Anwendungsdomäne (z.B. objektorientierter Entwurf) allgemein anwendbar ist. Dieses Konzept zur Lösung wiederkehrender Probleme einer Anwendungsdomäne wurde unter anderem in [GoF95] und in [Fowler00] umgesetzt.

Anmerkung: In dieser Arbeit werden Wortbildungen aus „Muster” und anderen Wörtern immer zusammen geschrieben (beispielsweise „Musterkatalog” statt „Muster-Katalog”). „Muster” hat dabei immer die in diesem Abschnitt vorgestellte Bedeutung. Weitere Wortbedeutungen, wie etwa „beispielhaft”, werden durch andere Wortwahl ausgedrückt.

Kapitel 3

Vom Muster zum Musterkatalog

Dieses Kapitel spannt den Bogen von der Problemstellung der Arbeit hin zu deren Lösung: Ausgangspunkt ist die Entscheidung, auf welchem Abstraktionsniveau die WAM-Refactorings beschrieben werden sollen. Nach dem Erkennen eines Refactoring-Potentials müssen zu dessen Umsetzung ähnliche, wiederkehrende Entwurfsentscheidungen getroffen werden. Dieser Vorgang lässt sich mit einer passenden Beschreibung der Refactorings als Muster vereinfachen. Abschließend wird gezeigt, wie sich die gefundenen Muster in Form eines Katalogs sinnvoll gliedern lassen.

3.1 Fokus und Gültigkeit der Refactorings

Die Beschreibung eines Refactorings kann auf verschiedenen Ebenen, etwa auf Quelltextebene oder auf Architekturebene erfolgen (siehe Abbildung 3.1). Eine projektspezifische Beschreibung ist nicht wiederverwendbar und daher nicht geeignet. Je abstrakter ein Refactoring beschrieben ist, desto schwieriger lässt es sich umsetzen, weil der Entwickler die Schritte erst für jeden Anwendungsfall konkretisieren muss. Im Gegensatz zum WAM-Ansatz erfolgt daher die Beschreibung der Refactorings in einer bestimmten Programmiersprache. Die Beschreibung setzt die Eigenschaften und Möglichkeiten der Sprache Java voraus. Weiter spezialisiert werden die Refactorings durch die Einbeziehung des JWAM-Rahmenwerks. Durch die Festlegung auf Rahmenwerk und Plattform verringert sich der Interpretationsspielraum beim Anwenden der Refactorings und es müssen weniger Variationen beschrieben werden. Obwohl die Beschreibungen der Muster auf JWAM fokusiert sind, bleiben die Refactorings selbst auch ohne den Einsatz von JWAM gültig. Sowohl die Probleme als auch die Lösungen der Muster beziehen sich auf die WAM-Modellarchitektur. Ohne Rahmenwerk bliebe die Struktur der Refactorings gleich, ihre Umsetzung müsste jedoch verallgemeinert werden. In der Beschreibung der Muster wird darauf eingegangen (siehe 3.3).

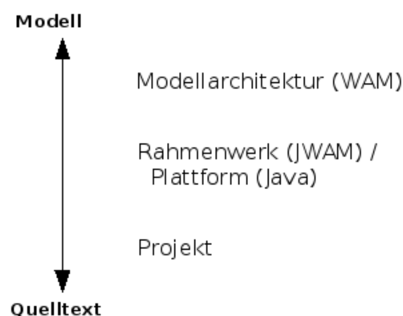

Abbildung 3.1: Mögliche Beschreibungsebenen von WAM-Refactorings

3.2 Identifikation möglicher Refactorings

Die im Katalog beschriebenen WAM-Refactorings sind Verallgemeinerungen von Refactorings, die in einem realen Software-Projekt durchgeführt wurden. Aus diesen stammt auch der Quelltext der Beispiele. Um die Entscheidungen, die zu diesen Refactorings geführt haben, nachvollziehen und selbst WAM-Refactorings einsetzen zu können, wird hier der Identifikationsprozess eines Refactoring-Potentials vorgestellt.

Zumindest in agilen Projekten findet Refactoring als Teil eines iterativen Entwicklungszyklus statt. In [Roock04] wird ein Programmierzyklus beschrieben, der zu Beginn und zum Ende der Implementierung einer neuen Funktionalität ein Refactoring stellt. So wird etwa vor Implementierung eines neuen Werkzeugs zuerst eine „Bestandsaufnahme” des vorhandenen Quelltexts gemacht, die in der Extraktion von wiederverwendbaren Teilen von Werkzeugen münden kann. Nachdem das neue Werkzeug fertig gestellt ist, beseitigt ein abschließendes Refactoring mögliche Problemstellen des neuen Quelltextes, wie etwa schlecht gewählte Bezeichnungen oder duplizierte Logik.

Häufig wird die Notwendigkeit eines Refactoring auch in einem Quelltext-Review festgestellt. In solchen Reviews wird oft im Quelltext nach *code smells* (siehe [Fowler00]) oder *architecture smells* (siehe [Roock04]) gesucht. Unter *smells* versteht man Indikatoren für mögliche Probleme in einem Quelltext. Diese Indikatoren sind, im Gegensatz zu Architekturverstößen, subjektiv und können je nach Projekt und Entwickler unterschiedlich wahrgenommen werden. Ob eine Klasse „zu groß” ist, um noch einfach verständlich zu sein, kann nicht objektiv gesagt werden. Werkzeuge zur statischen Quelltext- und Architekturanalyse wie der Sotograph [Sotograph06] können die Durchführung von Reviews unterstützen (siehe [Becker-Pechau06]). Solche Werkzeuge überprüfen objektive Kriterien eines Systems. Dazu gehören etwa Architekturverstöße, die durch den Vergleich von Soll- und Ist-Architektur entdeckt werden können. Metrikbasierte Analysen können durch das Aggregieren von Kennzahlen beim Erkennen von Smells helfen. In WAM-Systemen kontrolliert man bei statischen Analysen, ob die Regeln der WAM-Architektur (siehe 2.1.4) eingehalten werden. Dazu kann der Sotograph um spezielle Abfragen erweitert werden, die in [Karstens05] beschrieben sind. Für diese Arbeit wurde sämtliche Software aber ohne die Erweiterung analysiert. Daher waren die gefundenen Schwachstellen meist allgemeinerer Natur und nicht WAM-spezifisch (Ausnahmen sind beispielsweise verbotene Subwerkzeug-Beziehungen). Dennoch ließen sich die Ergebnisse der Sotograph-Analysen als Entscheidungshilfe für oder wider die Durchführung von WAM-Refactorings heranziehen. Dabei zeigte sich, dass viele der Refactorings um „Kristallisationspunkte” angeordnet sind. Das heißt, dass einige wenige WAM-Elemente und die Beziehungen zwischen diesen den Ausgangspunkt für viele Refactorings bilden. Dazu gehören Materialien, Fachlogik in Werkzeugen und die Wechselbeziehung zwischen Materialien und Werkzeugen.

3.3 Struktur der Muster

Mit Hilfe von Mustern lassen sich Refactorings so beschreiben, dass sie einerseits unabhängig von konkretem Quelltext betrachtet werden können, andererseits genau genug nachvollziehbar sind, um vielfach angewendet zu werden. Nur wenn diese beiden Bedingungen erfüllt sind, macht eine Beschreibung von Refactorings auch jenseits eines bestimmten Software-Systems Sinn. Die gewählte Struktur der Muster lehnt sich an bewährte Musterbeschreibungen ([GoF95] und [Kerievsky05]) an. Sie soll problemorientiert, praxisnah und verständlich sein.

- Name: Ein klarer, den Zweck des Musters verdeutlichender Name. Einprägsame Namen erleichtern auch das Wiedererkennen und die Diskussion von Mustern.

- Übersicht: Einleitend werden die Problemstellung und Lösung in wenigen Sätzen beschrieben. Eine Skizze (UML-Klassendiagramm) zeigt Ist- und Soll-Zustand anhand eines Beispiels. Bereits anhand der Übersicht soll der Leser erkennen können, ob ein bestimmtes Muster für die Lösung seines Problems in Frage kommt. Die Skizze erhöht die Verständlichkeit, weil sie ein einfaches, nachvollziehbares Beispiel zeigt und nicht sofort alle Details des Musters

- Motivation: Zuerst wird anhand des skizzierten Beispiels ein typischer Anwendungsfall vorgestellt. Dann folgt eine Verallgemeinerung, unter welchen Voraussetzungen dieses Muster (nicht) angewendet werden soll. Die in der Übersicht präsentierte Skizze des Ist- und Soll-Zustands wird ebenfalls verallgemeinert.
- Schritte: Anleitung zur Anwendung des Refactorings.
- Beispiel: Das Beispiel wird wieder aufgegriffen und schrittweise umstrukturiert. Die Schritte werden dadurch nachvollziehbarer.
- Konsequenzen: Mögliche Auswirkungen und Nebeneffekte der Anwendung des Refactorings, unter anderem auf WAM-Modell, technische Refactorings und verschiedene Qualitätsmerkmale von Software. Das Muster wird dadurch in einen Kontext gestellt, der die Auswirkungen des Musters besser erkennbar macht.
- Zusammenhänge: Erklärt Zusammenhänge und Unterschiede zu anderen Mustern des Katalogs. Das erleichtert das Abwägen von Alternativen.
- Verallgemeinerung: Inwieweit lässt sich dieses Muster auf die WAM-Modellarchitektur (also unter Ausblendung der JWAM-Spezifika, siehe dazu auch Abschnitt 3.1) verallgemeinern? Die Beantwortung dieser Frage ist vor allem für Leser interessant, die nicht das JWAM-Rahmenwerk verwenden. Da die Motivation bereits allgemein gehalten ist, geht der Punkt *Verallgemeinerung* stärker auf die Schritte ein.

Anmerkungen zu den Quelltextauszügen: Die Quelltextbeispiele sind in Java 5 implementiert. Die Quelltextkonventionen sehen deutschsprachige Bezeichner vor, Quelltext aus dem JWAM-Rahmenwerk verwendet englischsprachige. Die Namenskonventionen des Rahmenwerks sehen vor, dass Fachwertklassen auf „DV" (für *domain value*) enden. Exemplarvariablen beginnen immer mit einem Unterstrich und werden am Ende einer Klasse deklariert. Um die Übersichtlichkeit und Verständlichkeit zu erhöhen, wurde der Quelltext gegenüber dem Original an einigen Stellen vereinfacht. Auslassungen werden durch eine Ellipse (...) gekennzeichnet, sofern sie nicht offensichtlich sind. Die schrittweisen Veränderungen im Quelltext werden typografisch gekennzeichnet: Gelöschte Stellen werden ~~durchgestrichen~~, neu hinzugekommener Quelltext ist **fett** gedruckt.

Anmerkungen zu den UML-Diagrammen: Die abgebildeten Klassendiagramme zeigen meist nur Ausschnitte der Klassen und verzichten auf Details wie Methodenparameter, Sichtbarkeitsmodifikatoren und Multiplizitäten. Detailliertere Diagramme werden nur verwendet, wenn die zusätzlichen Informationen zum besseren Verständnis beitragen.

3.4 Unit-Tests und Refactoring

Bei der Beschreibung der Schritte wird vom Vorhandensein automatisierter Tests, so genannter *Unit-Tests*, für die betroffenen Klassen ausgegangen. Für neue Klassen und Methoden wird angenommen, dass diese ebenfalls durch Tests abgedeckt werden. Anhand des (Miss-)Erfolgs der Ausführung der Tests lassen sich die Schritte abgrenzen und nachvollziehen (so wird auch in [Kerievsky05] und [Westphal06] verfahren). Bei der Anwendung von Refactorings lässt sich durch Unit-Tests die Beibehaltung des Verhaltens sicherstellen[1]. Das reduziert die Wahrscheinlichkeit, durch Refactorings Fehler in den Quelltext zu bringen und gibt dem Entwickler „[...] das Vertrauen, dass die produzierte Software tatsächlich wie gewünscht funktioniert" ([Westphal06]).

Auf Grund dieser Vorteile werden Tests in den Musterbeschreibungen behandelt. Schon die Refactoring-Schritte alleine können in unterschiedlichen Reihenfolgen angeordnet werden, durch Hinzunehmen der Tests vervielfachen sich die Möglichkeiten, die Schritte anzuordnen. Die Beschreibungen gehen nicht auf Variationen in der Schrittfolge ein, sondern stellen nur einen sinnvollen Refactoring-Pfad vor.

[1] Absolute Sicherheit lässt sich nur für triviale Fälle mit vertretbarem Aufwand erreichen. Erfahrungsgemäß erleichtern sinnvolle Tests und hohe Testabdeckung die Entwicklung aber spürbar.

3.5 Gliederung des Katalogs

Der Hauptteil dieser Arbeit ist eine „Katalog” genannte Sammlung von acht fachlichen WAM-Refactorings:

- Ersetze Wert durch Fachwert (4.1.1)
- Ersetze allgemeinen Fachwert durch speziellen Fachwert (4.1.2)
- Ersetze Material durch Fachwert (4.1.3)
- Baue fachliche Materialhierarchie auf (4.2.1)
- Lasse Werkzeug mit Aspekt arbeiten (4.2.2)
- Verschiebe Fachlogik in Service (4.2.3)
- Extrahiere Automat (4.3.1)
- Zerlege Werkzeug in kleinere Werkzeuge (4.3.2)

Außer diesen gibt es noch weitere fachliche sowie technische WAM-Refactorings. In dem Software-Projekt, aus dem die Mustern abgeleitet wurden, ließen sich im Wesentlichen diese acht Muster identifizieren. Trotz der kleinen Anzahl an Mustern kann eine Gliederung des Katalogs Vorteile bringen: Das Auffinden von Mustern wird erleichtert und Zusammenhänge lassen sich besser erkennen, wenn ähnliche Refactorings gruppiert werden. Welche Muster sich ähneln, hängt davon ab, nach welchen Kriterien man sie vergleicht. Ist die Gliederung zu fein oder zu grob, geht der Blick auf die Zusammenhänge verloren und auch die Suche wird nicht einfacher.

Im Folgenden werden einige Kriterien vorgestellt und bewertet, anhand deren Ausprägungen sich die in dieser Arbeit beschriebenen Muster gruppieren lassen.

- Zweck: Eine bewährte Kategorie ist der Zweck oder die Problemstellung eines Refactorings. In [Fowler00] und [Kerievsky05] finden sich Kapitel wie *Organizing Data*, *Simplification* und *Generalization*, in denen Refactorings mit ähnlichen Zielen gruppiert sind. Die vorliegenden WAM-Refactorings lassen sich grob in *Modellierung korrigieren*, *Wiederverwendbarkeit erhöhen* und *Komplexität beherrschen* gliedern (siehe Tabelle 3.1).

Modellierung korrigieren	**Wiederverwendbarkeit erhöhen**	**Komplexität beherrschen**
Fachwert statt Wert (4.1.1)	Materialhierarchie (4.2.1)	Extrahiere Automat (4.3.1)
Spezieller Fachwert (4.1.2)	Werkzeuge arbeiten mit Aspekten (4.2.2)	Kleinere Werkzeuge (4.3.2)
Fachwert statt Material (4.1.3)	Verschiebe Logik in Service (4.2.3)	

Tabelle 3.1: Gliederung der WAM-Refactorings nach ihrem Zweck

- Struktur: Wenn Refactoring eine Änderung der Struktur von Quelltext darstellt, können verschiedene Arten von Strukturänderungen als Gliederungskriterium in Frage kommen. Der Katalog enthält zwei Arten von Strukturänderungen (siehe Tabelle 3.2):
  - Zerlegende Strukturänderungen: Ein Teil der Funktionalität eines WAM-Elements wird in ein anderes WAM-Element verschoben. Nach Abschluss des Refactorings sind daher mehrere WAM-Elemente zur Erfüllung der Aufgabe des ursprünglichen WAM-Elements nötig.
  - Transformierende Strukturänderungen: Ein WAM-Element ersetzt ein anderes oder es wird ein WAM-Element eingeführt, um eine fachliche Gegebenheit zu modellieren.

zerlegend	**transformierend**
Verschiebe Logik in Service (4.2.3)	Spezieller Fachwert (4.1.2)
Extrahiere Automat (4.3.1)	Fachwert statt Material (4.1.3)
Kleinere Werkzeuge (4.3.2)	Materialhierarchie (4.2.1)
	Werkzeuge arbeiten mit Aspekten (4.2.2)
	Fachwert statt Wert (4.1.1)

Tabelle 3.2: Gliederung der WAM-Refactorings nach Struktur

- Betroffene WAM-Elemente: Eine Möglichkeit ist, die Refactorings nach den betroffenen WAM-Elementen einzuteilen. Dieses Kriterium wird auch zur Gliederung der Sammlung von WAM-Regeln in [Karstens05] verwendet. Eine eindeutige Zuordnung von Refactoring-Mustern zu WAM-Elementen ist aber nur unter zwei Bedingungen möglich:

  1. Wie im Punkt „Struktur" angesprochen wurde, werden im Zuge mancher Refactorings Elemente durch andere ersetzt. In diesen Fällen muss festgelegt werden, ob sich die Zuordnung auf die Ausgangssituation oder auf das abgeschlossene Refactoring bezieht.
  2. In dem gewählten Zustand darf nur ein Typ von WAM-Elementen am Refactoring beteiligt sein. Das heißt, bei Wahl des Ausgangszustandes darf ein WAM-Element in ein oder mehrere andere übergeführt oder zerlegt werden. Im umgekehrten Fall darf am Zielzustand des Refactorings nur ein WAM-Element beteiligt sein.

  Nicht nur der Typ eines WAM-Elements kann als Kriterium herangezogen werden, sondern auch, ob sich die Typen von WAM-Elementen ändern. Es lassen sich also Refactorings mit und ohne Änderung von Elementtypen unterscheiden, wie Tabelle 3.3 zeigt.

Mit Typ-Änderung	**Ohne Typ-Änderung**
Fachwert statt Wert (4.1.1)	Spezieller Fachwert (4.1.2)
Fachwert statt Material (4.1.3)	Materialhierarchie (4.2.1)
Verschiebe Logik in Service (4.2.3)	Werkzeuge arbeiten mit Aspekten (4.2.2)
Extrahiere Automat (4.3.1)	Kleinere Werkzeuge (4.3.2)

Tabelle 3.3: Gliederung der WAM-Refactorings nach Änderung von Elementtypen

Für die Gliederung des Katalogs von WAM-Refactorings wurde der Zweck als Kriterium herangezogen, weil diese Einteilung der Anwendung der Refactorings am meisten entgegenkommt. Da nämlich am Anfang eines Refactorings immer ein Problem steht (etwa ein Smell, siehe 3.2), das es zu lösen gilt, ist der Zweck das primäre Suchkriterium. Alternative Lösungsmöglichkeiten für ein Problem finden sich bei der Gliederung nach dem Zweck in der gleichen Gruppe und können so miteinander verglichen werden

Die Arten von Strukturänderungen wurden nicht als Kriterium gewählt, weil Strukturänderungen nur ein Mittel zum Erreichen eines Zwecks sind. Dies entspricht nicht der oben beschriebenen problemorientierten Herangehensweise beim Refactoring. Auch die Gliederung nach betroffenen WAM-Elementen kommt nicht in Frage. Deren Bedingungen erfüllen nicht alle Muster des Katalogs. Das alternative Kriterium der Änderung von WAM-Typen ist mit nur zwei Ausprägungen (ob eine Änderung auftritt oder nicht) sehr grob und daher ungeeignet.

Schwierigkeiten bei der Einteilung nach Zweck gibt es, wenn Refactorings mehrere Zwecke erfüllen. *Extrahiere Automat* beispielsweise kann die Modellierung korrigieren und hilft, Komplexität zu beherrschen. In solchen Fällen wurde nach dem Hauptzweck gruppiert und in der Beschreibung der Refactorings die weiteren Zwecke als Konsequenz (siehe 3.3) mit aufgenommen. Für *Extrahiere Automat* heißt das, dass es im Abschnitt „Komplexität beherrschen" zu finden ist.

Die gewählte Gliederung lässt sich leicht erweitern, etwa um Refactorings, die Verstöße gegen die Regeln der WAM-Modellarchitektur korrigieren. Ein Abschnitt „Regelverstöße korrigieren" ist jedoch nicht in dieser Arbeit enthalten (siehe dazu Abschnitt 5.3.1).

Um die Einordnung der Refactorings in die WAM-Modellarchitektur zu verdeutlichen, zeigt Tabelle 3.4 eine zweidimensionale Gliederung aus den nach dem Zweck gruppierten Refactorings und den beteiligten WAM-Elementen.

WAM-Refactoring	**Werkzeug**	**Automat**	**Service**	**Material**	**Fachwert**
Fachwert statt Wert (4.1.1)					X
Spezieller Fachwert (4.1.2)					X
Fachwert statt Material (4.1.3)				X	X
Materialhierarchie (4.2.1)				X	
Werkzeuge arbeiten mit Aspekten (4.2.2)	X			X	
Verschiebe Logik in Service (4.2.3)	X		X		
Extrahiere Automat (4.3.1)	X	X	X		
Kleinere Werkzeuge (4.3.2)	X				

Tabelle 3.4: WAM-Refactorings und beteiligte WAM-Elemente

Kapitel 4

Katalog von WAM-Refactorings

„Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
Martin Fowler in *Refactoring: Improving The Design of Existing Code* [Fowler00].

4.1 Modellierung korrigieren

Die in diesem Abschnitt beschriebenen Refactorings verbessern die Strukturähnlichkeit zwischen WAM-Modell und Anwendungsdomäne. Dazu werden fachliche Gegenstände entweder durch ein anderen Typ von WAM-Element im Quelltext abgebildet (etwa durch einen Fachwert statt ein Material), oder ein Typ wird spezialisiert.

4.1.1 Ersetze Wert durch Fachwert

Übersicht

Ein fachlicher Gegenstand kann nicht adäquat mit einem Standard-Datentyp abgebildet werden.
Führe einen neuen Fachwert mit den benötigten Eigenschaften für den Gegenstand ein.

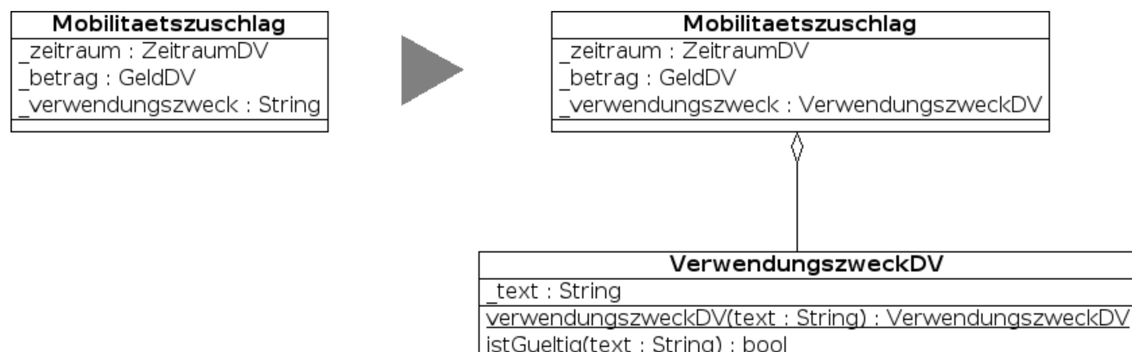

Abbildung 4.1: Der Verwendungszweck eines Mobilitätszuschlags wird als Fachwert statt als Java-Typ `String` modelliert.

Motivation

Bei der Entwicklung des Materials `Mobilitaetszuschlag` wurde der Verwendungszweck (erscheint am Beleg einer Banküberweisung) einer solchen Zahlung zunächst als primitiver Java-Datentyp `String`

abgespeichert. Um von externen Programmen weiterverarbeitet werden zu können, darf ein Verwendungszweck aber nicht mehr als 40 Zeichen haben. Aus diesem Grund wurde der Datentyp des Attributs vom Wert `String` auf den Fachwert `VerwendungszweckDV` geändert, der aus maximal 40 Zeichen besteht.

Anmerkung: Der Java-Datentyp `String` ist zwar eine Klasse, aber Objekte dieses Typs verhalten sich wie Werte. Daher wird der Typ im hier verwendeten Beispiel als Werttyp bezeichnet.

Mit den Standard-Werttypen einer Programmiersprache gibt es eine einfache Möglichkeit, Daten mit Wertsemantik zu implementieren. Verlangt der zu modellierende Wert aber mehr Semantik, als ein Standard-Typ anbietet, soll dieser Wert gemäß dem WAM-Ansatz als Fachwert modelliert werden. Wie im beschriebenen Beispiel kann diese Entscheidung nicht immer sofort getroffen werden. Um einen als Standard-Typ implementierten Wert in einen Fachwert überzuführen, kann *Ersetze Wert durch Fachwert* angewendet werden. In der neuen Fachwertklasse wird dann die zusätzliche Semantik eingefügt.

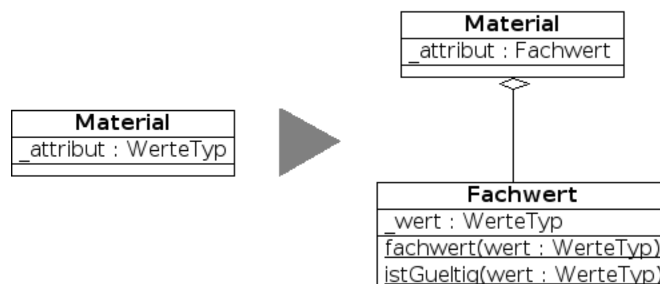

Abbildung 4.2: Der Verwendungszweck eines Mobilitätszuschlags wird als Fachwert statt als String modelliert.

Schritte

1. Lege eine neue Klasse mit allen benötigten Methoden (Objekterzeugung, Gültigkeitsprüfung etc.) für den Fachwert an, die das Interface `DomainValue` implementiert. Als externe Repräsentation des Fachwerts eignet sich der zu ersetzende Werttyp. Die Tests der Klasse überprüfen, ob sie das gleiche Verhalten wie der zu ersetzende Werttyp hat. Die Tests schlagen fehl.

2. Implementiere die Methoden des Fachwerts so, dass sie die Tests erfüllen. Dazu wird der Typ der externen Repräsentation auch zu internen Repräsentation genutzt, das heißt eine Exemplarvariable vom zu ersetzenden Werttyp wird angelegt. Fachwert und Werttyp besitzen nun gleiches Verhalten, die Tests des Fachwerts laufen daher durch.

3. Suche alle Stellen (auch in den Tests), an denen der Werttyp verwendet wird und ersetze diesen an den Stellen, wo sein Verhalten die fachlichen Gegebenheiten nicht abdeckt, durch den Fachwert. Da die Unterscheidung, wann welcher der beiden Typen nötig ist, von der Anwendungsdomäne abhängt, können dafür keine allgemein gültigen Regeln aufgestellt werden. Weil das Verhalten der beiden Typen noch immer gleich ist, laufen alle Tests durch.

4. Passe alle Tests, in denen der Fachwert verwendet wird, an das gewünschte Verhalten an. Alle Tests schlagen fehl. Sobald das gewünschte Verhalten im Fachwert implementiert ist, laufen wieder alle Tests durch.

Einen Sonderfall bildet die Transformation in einen Aufzählungstyp. Mit *Java 5* gibt es die Möglichkeit, solche Typen selbst zu definieren und als Fachwerte einzusetzen. Da auf Objekte eines Aufzählungstyps wie auf Klassenkonstanten zugegriffen wird, braucht man keine Methoden zur Gültigkeitsprüfung zu implementieren. Die Schritte für diese Variante des Refactorings sehen so aus:

1. Erzeuge einen Aufzählungstyp für den Fachwert und definiere einen parameterlosen, privaten Konstruktor und eventuell benötigte Methoden, etwa um auf Exemplarvariablen zuzugreifen oder zur Darstellung als String.

2. Erweitere den Fachwert um eine Exemplarvariable vom Typ des zu ersetzenden Werts und erweitere den Konstruktor um einen Parameter diesen Typs, so dass die Exemplarvariable initialisiert wird.

3. Definiere alle möglichen Werte des Fachwerts als Klassenkonstante. Der Parameter für den Konstruktoraufruf ist der entsprechende Wert der allgemeinen Fachwertklasse. Für die noch funktionslose neue Klasse wird ein Test geschrieben, der das erwartete Verhalten abprüft. Dieser Test schlägt fehl.

4. Verändere den neuen Fachwert so, dass er das gewünschte Verhalten zeigt. Der Test für den Fachwert läuft durch. Damit ist der neue Fachwert vollständig implementiert.

5. Suche alle Stellen (auch in den Tests), an denen der Werttyp verwendet wird und ersetze diesen an den Stellen, wo sein Verhalten die fachlichen Gegebenheiten nicht abdeckt, durch den Fachwert. Da die Unterscheidung, wann welcher der beiden Typen nötig ist, von der Anwendungsdomäne abhängt, können dafür keine allgemein gültigen Regeln aufgestellt werden.

Beispiel

1. Die Klasse `VerwendungszweckDV` wird angelegt und implementiert das Interface `DomainValue`. Abbildung 4.3 zeigt einige der Methoden der neuen Klasse. Die Tests der Methoden überprüfen, ob sie sich verhalten, wie es dem Java-Datentyp `String` entspricht (beispielsweise sind leere Zeichenketten gültig, nicht jedoch `null)`.

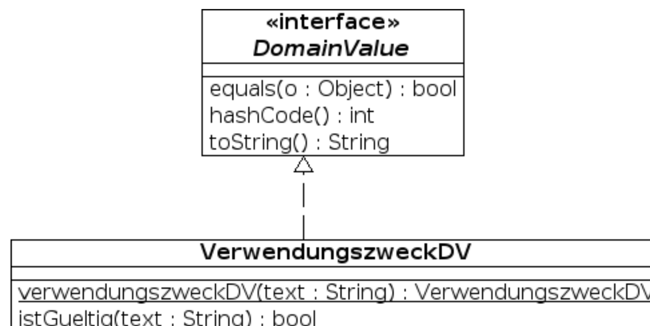

Abbildung 4.3: Eine Klasse für den neuen Fachwert `VerwendungszweckDV` wird angelegt.

```
public final class VerwendungszweckDV implements DomainValue {
  public static VerwendungszweckDV verwendungszweckDV(String text) {
    return null;
  }
  public static boolean istGueltig(String text) {
    return false;
  }
  private VerwendungszweckDV(String text) {
  }
  ...
}
```

2. Nun wird der Fachwert so erweitert, dass er sich wie ein String verhält und die Tests besteht. Der folgende Ausschnitt zeigt einige der veränderten Methoden.

```
public static VerwendungszweckDV verwendungszweckDV(String text) {
  assert istGueltig(text);
  return VerwendungszweckDV(text);
}
public static boolean istGueltig(String text) {
  return text != null;
}
private VerwendungszweckDV(String text) {
  _text = text;
}
private String _text;
```

3. Der neue Fachwert kommt unter anderem im Material `Mobilitaetszuschlag` zum Einsatz.

```
public class Mobilitaetszuschlag  {
  public void setzeVerwendungszweck(
    String VerwendungszweckDV verwendungszweck) {
      _verwendungszweck = verwendungszweck;
  }
  ...
  private String VerwendungszweckDV _verwendungszweck;
}
```

4. Nach dem das Refactoring mit dem dritten Schritt abgeschlossen wurde, kann der Fachwert so angepasst werden, dass er maximal 40 Zeichen langen Text zulässt.

```
public static boolean istGueltig(String text) {
  return text != null && text.length() < 41;
}
```

Konsequenzen

Durch das Ersetzen von Werten mit Fachwerten können ähnliche Klassen entstehen. Beispielsweise kann ein Bankname genauso wie der Verwendungszweck als String mit begrenzter Länge implementiert werden. Ob solche Klassen durch technische Refactorings zusammengelegt werden oder ob die separaten Fachwertklassen zu Gunsten der Strukturähnlichkeit bestehen bleiben sollen, erläutern die Konsequenzen des Refactorings *Ersetze allgemeinen Fachwert durch speziellen Fachwert* (siehe Abschnitt 4.1.2).

Zusammenhänge

Nach der Spezialisierung eines Typs vom Standard-Wert zum Fachwert kann eine weitere Spezialisierung von einem zu allgemeinen Fachwert hin zu einem spezielleren vollzogen werden. Dieser Schritt lässt sich mit dem Refactoring *Ersetze allgemeinen Fachwert durch speziellen Fachwert* (siehe 4.1.2) umsetzen.

Verallgemeinerung

Wird zur Implementierung eine vollständig objektorientierte Programmiersprache gewählt, besteht wegen des Fehlens von Werttypen kein Bedarf an *Ersetze Wert durch Fachwert*. Objekte mit Wertsemantik müssen ohnedies als Fachwerte implementiert werden. Manche Programmiersprachen wie C# lassen sich auch um benutzerdefinierte Werttypen erweitern, was die Verwendung von Fachwerten ersetzen kann.

4.1.2 Ersetze allgemeinen Fachwert durch speziellen Fachwert

Übersicht

Ein Fachwert bildet mehrere fachliche Gegenstände ab. Für einen dieser Gegenstände ist der Fachwert zu allgemein. Seine Eigenschaften lassen sich nur mit einem spezielleren Fachwert vollständig abbilden.

Führe einen neuen Fachwert mit den benötigten Eigenschaften für den Gegenstand ein.

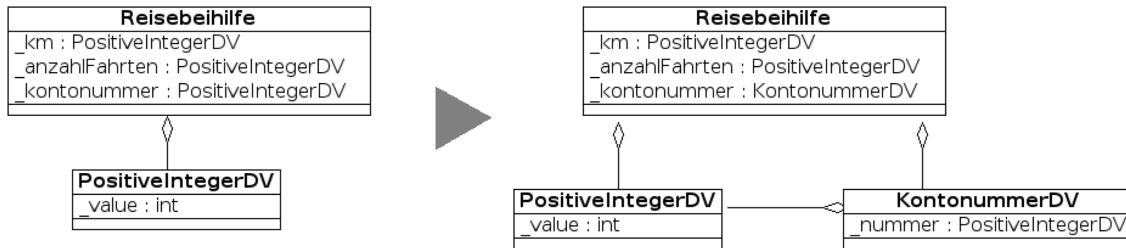

Abbildung 4.4: Eine Kontonummer wird als Spezialisierung des Typs PositiveIntegerDV modelliert

Motivation

Die ursprünglichen Modellierung des Materials „Reisebeihilfe" (eine Vergütung für Zivildienstleistende[1]) verwendete eine positive Ganzzahl als Kontonummer. Im Gegensatz zu Ganzzahlen sind Kontonummern in ihrer Stellenanzahl beschränkt, haben aber sonst gleiche Eigenschaften. Um die ungenaue Modellierung zu korrigieren, wurde ein neuer, auf `PositiveIntegerDV` basierender Fachwert geschaffen. Mittels eines Refactorings wurde zuerst eine neue, vom Verhalten her zu `PositiveIntegerDV` gleiche Klasse angelegt und dann so angepasst, dass nur noch eine bestimmte Anzahl von Stellen möglich ist.

Im Lauf der Entwicklung kann sich zeigen, dass ein bereits vorhandener Fachwert nicht ausreichend zur Abbildung eines fachlichen Gegenstands ist. Der Gegenstand hat Einschränkungen oder zusätzliche Eigenschaften im Vergleich zum verwendeten Fachwert. Falls zusätzliche Eigenschaften nötig sind, ist die Einführung eines neuen Fachwerts die logische Konsequenz. *Ersetze allgemeinen Fachwert durch speziellen Fachwert* ist ein guter Startpunkt für die Implementierung des neuen Fachwerts, weil die dabei entstehende Fachwertklasse benötigt wird, um die neue Eigenschaft dort zu verankern.

Wenn die Spezialisierung eines Fachwerts nur in eine Einschränkungen der Wertemenge besteht, kann eine parametrisierbarer, allgemeiner Fachwert eine Alternative zur Schaffung neuer Fachwerte sein. Vor- und Nachteile dieser Lösung diskutiert der Abschnitt *Konsequenzen*.

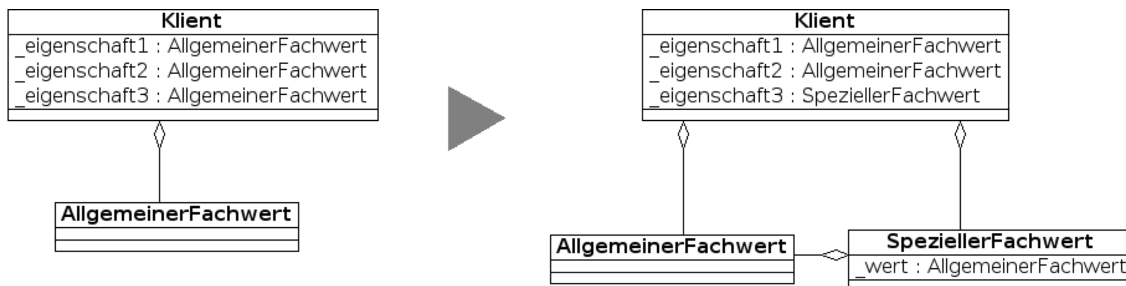

Abbildung 4.5: Ersetze allgemeinen Fachwert durch speziellen Fachwert

Schritte

1. Erzeuge eine Klasse für den Fachwert und definiere Methoden zur Objekterzeugung und Gültigkeitsprüfung. Diese Methoden weisen die gleiche Schnittstelle auf wie ihre Entsprechungen in der Klasse mit dem zu allgemeinen Fachwert. Für die noch funktionslose neue Klasse werden Tests geschrieben, die das erwartete Verhalten abprüfen. Diese Tests schlagen fehl.

[1] *Zivildienstleistender* wird im Folgenden oft mit *ZDL* abgekürzt.

2. Erweitere den speziellen Fachwert um eine Exemplarvariable vom Typ des allgemeinen Fachwerts und delegiere alle Methodenaufrufe an diesen. Statische Methoden werden an die Klasse des allgemeinen Fachwerts delegiert. Der spezielle Fachwert hat jetzt exakt die gleiche Funktionalität wie der allgemeine, daher schlagen die Tests für die neue Klasse noch immer fehl.

3. Suche alle Stellen (auch in den Tests), an denen der allgemeine Fachwert verwendet wird und ersetze bei Bedarf die Methodenaufrufe und Objekte durch die des neuen Typs. Da die Unterscheidung, wann welcher der beiden Fachwerte nötig ist, von der Anwendungsdomäne abhängt, können dafür keine allgemein gültigen Regeln aufgestellt werden. Nach Ausführung dieses Schritts schlagen die Tests für den neuen Fachwert noch immer fehl, die anderen Tests jedoch nicht. Sie testen zwar die Verwendung der neuen Fachwerte in Materialien, Werkzeugen etc., jedoch erwarten sie das gleiche Verhalten wie beim allgemeineren Fachwert.

4. Nachdem die Tests an allen Stellen, an denen der neue Fachwert verwendet wird, an das speziellere Verhalten angepasst wurde, schlagen diese fehl. Nun verändert man den neuen Fachwert so, dass er das gewünschte Verhalten zeigt. Alle Test laufen durch.

Wie in den Schritten von *Ersetze Wert durch Fachwert* (siehe Abschnitt 4.1.1) dargelegt wurde, bietet *Java 5* die Möglichkeit, Aufzählungstypen zu definieren und als Fachwerte einzusetzen. Die Schritte für diese Variante des Refactorings sehen so aus:

1. Erzeuge einen Aufzählungstyp für den Fachwert und definiere einen parameterlosen, privaten Konstruktor und eventuell benötigte Methoden, etwa um auf Exemplarvariablen zuzugreifen oder zur Darstellung als String.

2. Erweitere den speziellen Fachwert um eine Exemplarvariable vom Typ des allgemeinen Fachwerts und erweitere den Konstruktor um einen Parameter diesen Typs, so dass die Exemplarvariable initialisiert wird.

3. Definiere alle möglichen Werte des Fachwerts als Klassenkonstante. Der Parameter für den Konstruktoraufruf ist der entsprechende Wert der allgemeinen Fachwertklasse. Für die noch funktionslose neue Klasse wird ein Test geschrieben, der das erwartete Verhalten abprüft. Dieser Test schlägt fehl.

4. Verändere den neuen Fachwert so, dass er das gewünschte Verhalten zeigt. Der Test für den Fachwert läuft durch. Damit ist der neue Fachwert vollständig implementiert.

5. Suche alle Stellen (auch in den Tests), an denen der allgemeine Fachwert verwendet wird und ersetze bei Bedarf die Methodenaufrufe und Objekte durch die des neuen Typs. Da die Unterscheidung, wann welcher der beiden Fachwerte nötig ist, von der Anwendungsdomäne abhängt, können dafür keine allgemein gültigen Regeln aufgestellt werden.

Falls außer der Beschränkung der Wertemenge noch weitere Eigenschaften gegenüber dem allgemeinen Fachwert geändert werden müssen, kann dies nun in der Klasse des neuen Fachwerts implementiert werden.

Beispiel

Der Fachwert `PositiveIntegerDV` wird eingesetzt, um positive Ganzzahlen zu modellieren. In einem Projekt fand er unter anderem als Fachwert für verschiedene Anzahlen Verwendung, aber auch als Kontonummer. Das stellte ein Problem dar, weil beliebig lange Kontonummern erzeugt werden konnten. Mittels *Ersetze allgemeinen Fachwert durch speziellen Fachwert* wurde ein eigener Fachwert für Kontonummern, `KontonummerDV`, geschaffen, der eine positive Ganzzahl mit begrenzter Stellenanzahl modelliert. Hier zunächst ein Ausschnitt aus der Klasse `PositiveIntegerDV`:

```
public class PositiveIntegerDV implements DomainValue, Comparable {
  public static PositiveIntegerDV positiveIntegerDV(String zahl) {...}
  public static PositiveIntegerDV positiveIntegerDV(long zahl) {...}
```

```
  public static boolean istGueltig(String zahl) {...}
  ...
  private PositiveIntegerDV(long zahl) {
    _zahl = zahl;
  }
  ...
  private long _zahl;
}
```

1. Die neu angelegte Klasse `KontonummerDV` definiert bereits alle benötigten Methoden (hier ein Ausschnitt):

   ```
   public class KontonummerDV implements DomainValue{
     public static KontonummerDV kontonummerDV(String ktn) {
       return null;
     }
     public static boolean istGueltig(String ktn) {
       return false;
     }
     private KontonummerDV(long ktn) {
     }
     ...
   }
   ```

2. Zur Datenhaltung wird eine Variable vom Typ `PositiveIntegerDV` eingefügt. Nach diesem Schritt verhalten sich `PositiveIntegerDV` und `KontonummerDV` gleich.

   ```
   public static KontonummerDV kontonummerDV(String ktn) {
     return KontonummerDV(ktn);
   }
   public static boolean istGueltig(String ktn) {
     return PositiveIntegerDV.isValid(ktn);
   }
   private KontonummerDV(long ktn) {
     _nummer = PositiveIntegerDV.positiveIntegerDV(ktn);
   }
   private PositiveIntegerDV _nummer;
   ```

3. Der neue Fachwert kann nun eingesetzt werden, ohne das bisherige Verhalten des Systems zu verändern. An der GUI kann etwa die Definition eines Eingabefelds für eine Kontonummer von
   `private DomainValueTextFeld<PositiveIntegerDV> kontonummer;`
   auf
   `private DomainValueTextFeld<`**`KontonummerDV`**`> kontonummer;`
   geändert werden.

4. Nun wird KontonummerDV so verändert, dass nur noch maximal zehnstellige positive Ganzzahlen zulässig sind.

   ```
   public static boolean isValid(String ktn) {
     assert ktn != null;
     return PositiveIntegerDV.isValid(ktn) && ktn.length() <= 10;
   }
   ```

Konsequenzen

Fachwerte, die sich ähnlich sind, können unter Umständen doppelten Quelltext enthalten. Zwar lässt sich durch Objektkomposition viel duplizierter Quelltext vermeiden, trotzdem müssen für den neuen Fachwert Methoden zur Objekterzeugung, Gültigkeitsprüfung etc. geschrieben werden. Darauf könnte verzichtet

werden, wenn der zu allgemeine Fachwert so erweitert wird, dass er bei Bedarf auch das speziellere Verhalten zeigt. Für das Beispiel der Kontonummer ist es möglich, den Fachwert `PositiveIntegerDV` mit der gewünschten Stellenanzahl zu parametrisieren - schon können damit Objekte erzeugt werden, die sich wie solche des Fachwerts `KontonummerDV` verhalten. Das hat jedoch den Nachteil, dass sich Objekte des selben Fachwerts unterschiedlich verhalten. Ein Entwickler müsste beispielsweise für jedes Objekt vom Typ `PositiveIntegerDV` wissen, ob es gerade als Kontonummer verwendet wird. Bei der spezialisierten Fachwertklasse tritt an Stelle der impliziten Unterscheidung zwischen eingeschränktem und uneingeschränktem Fachwert eine explizite Unterscheidung anhand des Typs. Das heißt, das an Stelle des Entwicklers nun der Compiler überprüfen kann, welcher Typ verwendet werden muss. Auf Grund dieses Vorteils sind eigene Fachwerte für unterschiedliche fachliche Gegenstände meist die bessere Lösung. Daher sind in der Regel technische Refactorings zu vermieden, die zwecks Quelltextersparnis Fachwertklassen zusammenlegen. Würde durch die Modellierung spezieller Fachwertklassen aber eine Vielzahl von neuen, wenig benutzten Klassen entstehen, können parametrisierbare Fachwerte eine gute Alternative sein.

Zusammenhänge

Ersetze allgemeinen Fachwert durch speziellen Fachwert kann als Fortführung von *Ersetze Wert durch Fachwert* (Abschnitt 4.1.1) verstanden werden. Vor allem bei agiler Entwicklung wird man zuerst *Ersetze Wert durch Fachwert* und dann *Ersetze allgemeinen Fachwert durch speziellen Fachwert* anwenden, um die Modellierung schrittweise zu verbessern.

Verallgemeinerung

Spezielle Fachwerte können nicht nur durch Objektkomposition aus allgemeinen Fachwerten erzeugt werden, sondern auch durch Vererbung. In beiden Fällen macht man sich die Eigenschaften des allgemeineren Fachwerts zu Nutze, um den speziellen Fachwert mit weniger Aufwand implementieren zu können. In JWAM wird zur Konstruktion von Fachwerten keine Vererbung eingesetzt, weil Fachwerte per Konvention einen privaten Konstruktor und statische Methoden zur Objekterzeugung und Gültigkeitsprüfung haben. Ein abgeleiteter Fachwert würde die statischen Methoden seiner Basisklasse anbieten und somit nicht nur Objekte seines Typs erzeugen können. Die Verwendung des Fachwerts wäre dadurch mehrdeutig und fehleranfällig.

4.1.3 Ersetze Material durch Fachwert

Übersicht

Ein Material verhält sich wie ein Fachwert, wird also nie verändert.
Ersetze das Material durch einen Fachwert.

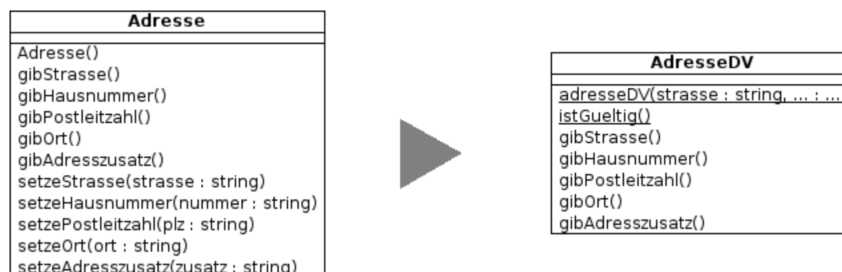

Abbildung 4.6: Aus dem Material Adresse wird ein Fachwert

Motivation

Eine Adresse wurde zunächst als Material modelliert, vermutlich weil sich Adressen laut Sprachgebrauch ändern können („Meine Adresse hat sich geändert"). Mit Ausnahme der Tests wurden sämtliche ändernden

Methoden (`setzeOrt(...)`) nur zum Initialisieren von Objekten verwendet. Bei einem Review wurde überprüft, ob ein Fachwert eine adäquatere Abbildung einer Adresse darstellt. Tatsächlich bekommen Personen oder Gebäude neue Adressen. Die Adressen selbst ändern sich nicht.

Wenn Objekte eines Material-Typs nie verändert werden, d.h. ihre verändernden Methoden nie oder nur zum Initialisieren aufgerufen werden, kann das auf einen Fehler in der Modellierung hindeuten. Vielleicht wurde der Gegenstand schon immer unpassend abgebildet oder die falsche Abbildung entstand als Folge einer Aufspaltung eines Materials in andere Materialien oder Fachwerte. *Ersetze Material durch Fachwer*t beseitigt überflüssige Methoden und sorgt dafür, dass die Strukturen des WAM-Modells und der Domäne ähnlicher werden.

Die Umkehr des Refactorings, also vom Fachwert zum Material, ist schwer vorstellbar. Fehlende Methoden zur Zustandsänderung eines Objekts würden (vor allem bei testgetriebener Entwicklung) im Gegensatz zu nicht verwendeten Methoden rasch auffallen.

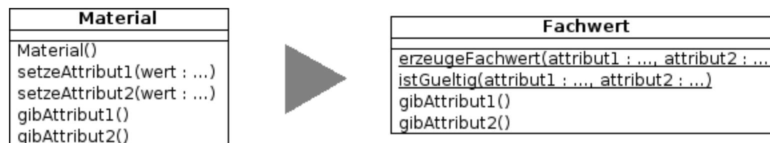

Abbildung 4.7: Aus einem Material wird ein Fachwert

Schritte

1. Erstelle eine neue Fachwertklasse, die alle lesenden Methoden des Materials sowie alle für Fachwerte übliche Methoden (Fabrikmethode, Gültigkeitsprüfung etc.) aufweist. Die Implementierung und die Tests der lesenden Methoden können dazu aus dem Material kopiert werden. Die Signatur dieser Methoden muss im Fachwert und im Material gleich sein. Der Test für den neuen Fachwert läuft durch.

2. Der Konstruktor des Materials wird um Parameter für alle Attribute erweitert, so dass der Konstruktor die gleichen Parameter wie die Fabrikmethode des Fachwerts aufweist. An Stelle von Methoden zum Setzen der Attribute werden diese im Konstruktor initialisiert. Sobald alle Konstruktoraufrufe angepasst wurden, können die ändernden Methoden gelöscht werden. Alle Tests laufen durch.

3. An Stelle der einzelnen Attribute wird im Material ein Objekt des neuen Fachwerts verwendet und alle Methoden auf dieses Objekte bzw. die Fachwertklasse umgeleitet. Der Konstruktor ruft also die Fabrikmethode der Fachwertklasse auf und leitet seine Parameter an sie weiter. Da nach wie vor alle Tests fehlerfrei sind, haben Fachwert und Material das gleiche Verhalten.

4. Ersetze in allen Deklarationen von Objekten des Materials den Typ mit der neuen Fachwertklasse. Nun ist das Programm so lange nicht übersetzbar, bis alle Konstruktoraufrufe mit Aufrufen der Fabrikmethode ersetzt sind. Da im zweiten Schritt aber die Parameter an die der Fabrikmethode angeglichen wurden, ist das Problem leicht zu beseitigen.

5. Wenn alle Tests laufen und das Material nur noch in seiner eigenen Testklasse benutzt wird, kann es (inklusive seiner Testklasse) gelöscht werden.

Beispiel

1. Die neue Fachwertklasse `AdresseDV` wird vollständig implementiert und getestet. Ihre fachlich motivierten Methoden sowie einige Klassenmethoden (Fabrikmethode etc.) sind im Klassendiagramm der Abbildung 4.6 angeführt. Alle Attribute einer Adresse werden der Fabrikmethode als Parameter übergeben.

```
public static AdresseDV adresseDV(String strasse, String nummer,
  String adresszusatz, String plz, String ort) {
```

```
  ...
}
```

2. Der parameterlosen Konstruktor in `Adresse` wird an die Fabrikmethode von `AdresseDV` angeglichen.

```
public Adresse(String strasse, String nummer,
  String adresszusatz, String plz, String ort) {
  ...
}
```

An statt Objekte mit einzelnen Methoden zu initialisieren, geschieht dies nun über den Konstruktor:

```
Adresse adresse = new Adresse("Hohlweg", "1a", "20259", "Hamburg");
adresse.setzeStrasse("Hohlweg");
adresse.setzeHausnummer("1a");
```

Die ändernden Methoden in `Adresse` werden jetzt nicht mehr benutzt und können gelöscht werden. Abbildung 4.8 zeigt die Klasse `Adresse` nach Abschluss dieses Schritts.

Adresse

Adresse(strasse : string, ... : ...)
gibStrasse()
gibHausnummer()
gibPostleitzahl()
gibOrt()
gibAdresszusatz()

Abbildung 4.8: Zwischenschritt von Material zu Fachwert

3. Statt die Straße, Hausnummer etc. in eigenen Variablen zu speichern, wird dazu ein Objekt des Fachwerts `AdresseDV` verwendet.

```
public Adresse(String strasse, String nummer,
  String adresszusatz, String plz, String ort) {
  _strasse = strasse;
  ...
  _adresse = AdresseDV.adresseDV(strasse, nummer, adresszusatz, plz, ort);
}

public String gibStrasse() {
  return _strasse;
  return _adresse.gibStrasse();
}
...
private String _strasse;
private AdresseDV _adresse;
```

4. Alle Variablen vom Typ `Adresse` müssen nun auf den Typ `AdresseDV` geändert werden. Bis auf die Aufrufe des Konstruktors, die gegen Aufrufe der statischen Fabrikmethode getauscht werden müssen, bleibt der Quelltext unverändert. Schließlich haben die fachlichen Methoden in beiden Klassen die gleiche Signatur.

```
Adresse adresse = new Adresse("Hohlweg", "1a", "20259",
"Hamburg");
AdresseDV adresse = AdresseDV.adresse("Hohlweg", "1a", "20259", "Hamburg");
```

5. Das Material `Adresse` wurde vollständig durch den Fachwert `AdresseDV` ersetzt und kann gelöscht werden.

Verallgemeinerung

Abgesehen von der Implementierung der Fachwertklasse hängt die Beschreibung von *Ersetze Material durch Fachwert* nicht von JWAM ab und kann daher auch in Systemen ohne JWAM eingesetzt werden.

4.2 Wiederverwendbarkeit erhöhen

Duplizierte Logik ist einer der häufigsten und fehlerträchtigsten Code Smells (siehe [Wake04]). Die Refactorings in diesem Abschnitt vermindern Duplikationen, in dem die Wiederverwendbarkeit von Fachlogik oder WAM-Elementen gesteigert wird. Einen ähnlichen Zweck verfolgen die Refactorings im Kapitel *Dealing with Generalization* in [Fowler00].

4.2.1 Baue fachliche Materialhierarchie auf

Übersicht

Verschiedene Materialien sind sich während der Entwicklung ähnlich geworden und weisen gleiche Eigenschaften und Zustände auf.
Baue eine Hierarchie von Materialien und Aspekten auf, die sowohl fachlich sinnvoll ist als auch gutes, objektorientiertes Design darstellt.

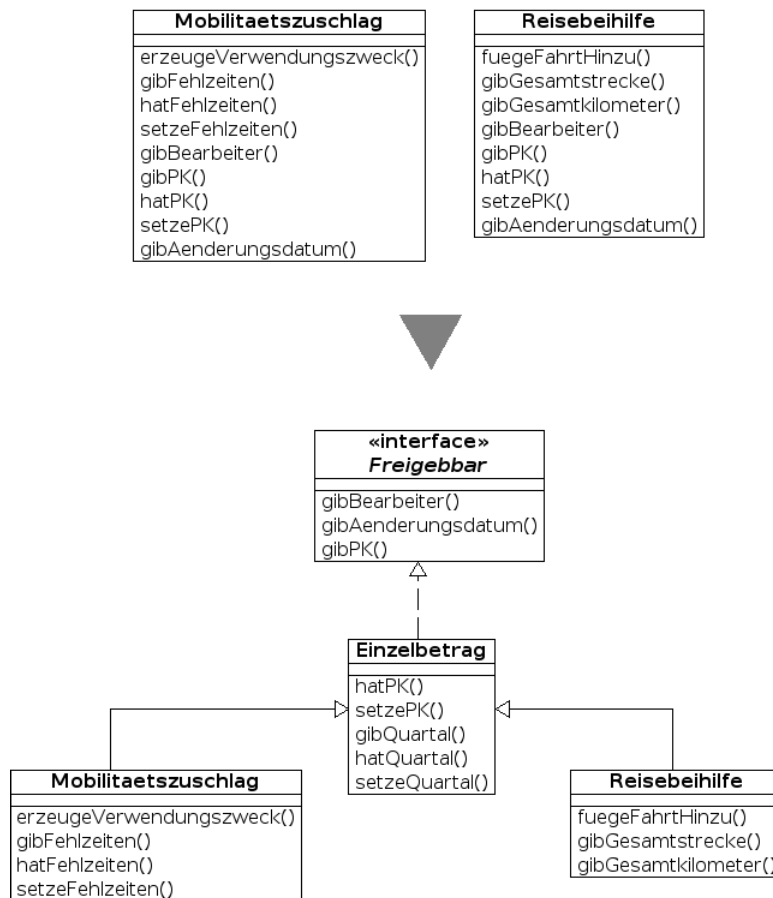

Abbildung 4.9: Mobilitaetszuschlag und Reisebeihilfe als Bestandteil einer Materialhierarchie

Motivation

Mobilitätszuschläge und Reisebeihilfen müssen nach Bearbeitung des Antrags durch einen Sachbearbeiter von einem zweiten Sachbearbeiter kontrolliert und zur Auszahlung freigegeben werden. Um sich einen Überblick über alle freizugebenden Anträge zu verschaffen, muss der freigebende Sachbearbeiter das

Personenkennzeichen (`PkDV`) des Zivildienstleistenden, den eingebenden Sachbearbeiter und das letzte Änderungsdatum kennen. Alle Zahlungen, die dem Freigabeprozess unterliegen, müssen diese Überblicksinformation liefern können. Dazu wurde der Aspekt `Freigebbar` eingeführt, den alle freigebbaren Materialien anbieten. Weil darüber hinaus noch weitere Gemeinsamkeiten zwischen Mobilitätszuschlägen und Reisebeihilfen bestehen, wurden die doppelt vorhandenen Methoden und Attribute der Klassen `Mobilitaetszuschlag` und `Reisebeihilfe` in eine abstrakte Basisklasse herausgezogen. Als Name für diese Klasse wurde statt einer technischen Bezeichnung (etwa *AbstraktesFreigebbaresMaterial*) eine fachlich motivierte verwendet. Mit *Einzelbetrag* wurde ein Begriff herangezogen, der in einem bestehenden Buchungssystem Verwendung fand. Das System rechnete Beträge beider Art als einzelne Buchungen ab.

Vererbung und Polymorphie sind zwei Eckpfeiler der objektorientierten Programmierung, die auch bei der Konstruktion von Materialien und Werkzeugen gewinnbringend eingesetzt werden können. Mitunter sollen polymorphe Ausprägungen eines Materials bearbeitet werden, wozu Vererbungshierarchien von Materialien benötigt werden. Der Aufbau solcher Hierarchien steht aber nicht nur unter einem objektorientierten Gesichtspunkt. Da Materialien fachliche Gegenstände abbilden, sollten Abstraktionen von Materialien ebenfalls einen fachlichen Bezug haben. Eine rein technisch motivierte Abstraktion beeinträchtigt die Strukturähnlichkeit. Da solche Hierarchien nicht immer explizit in der Domäne verankert sind, kann es nützlich sein, als Klassennamen Begriffe aus Altsystemen und Organisationsstrukturen zu verwenden oder gemeinsam mit den zukünftigen Benutzern neue Begriffe zu entwickeln.

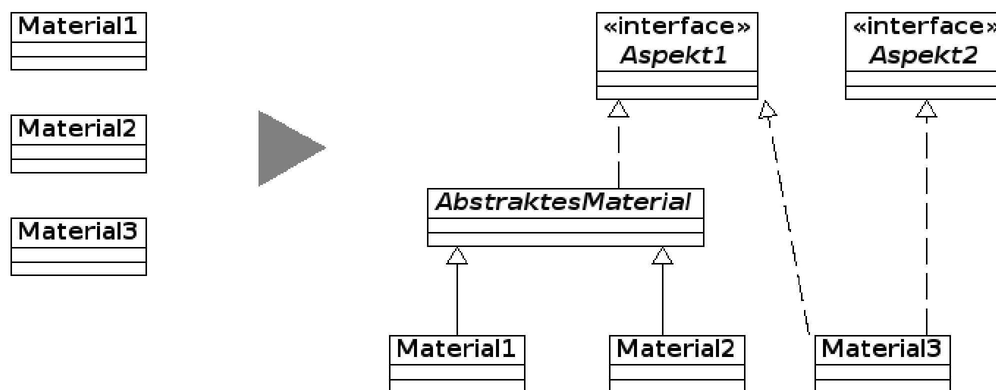

Abbildung 4.10: Materialien werden in einer Hierarchie angeordnet

Schritte

1. Identifiziere fachliche Kontexte, in denen verschiedene Materialien auf gleiche Art verwendet werden.

2. Führe für jeden dieser fachlichen Kontexte einen Aspekt in Form eines Interfaces ein, das von den Materialien implementiert wird (vgl. *Extract Interface* [Fowler00]). Der Aspekt bietet fachliche Methoden an und hat einen zum Kontext passenden Namen. Alle Tests laufen.

3. Ziehe gleich implementierte Methoden aus den Materialien in abstrakte Klassen heraus (vgl. *Extract Class* [Fowler00]) und lasse die Materialien von diesen erben. Für diese Abstraktionen sollen fachliche Namen verwendet werden. Alle Tests laufen.

Beispiel

1. Die Materialien `Mobiliaetszuschlag` und `Reisebeihilfe` durchlaufen einen Freigabeprozess und müssen dazu Informationen über den letzten Bearbeiter des Materials, das letzte Änderungsdatum und den antragstellenden Zivildienstleistenden bereitstellen.

2. Der Aspekt *Freigebbar* (siehe Abbildung 4.9) fasst die im ersten Schritt identifizierten fachlichen Anforderungen für die Freigabe von Mobilitätszuschlägen und Reisebeihilfen zusammen. Im Quelltext ist der Aspekt als Interface implementiert und erweitert das Interface `Material` aus dem JWAM-Rahmenwerk. `Material` dient nur zur Kennzeichnung von Materialien und definiert keine Methoden.

```
public interface Freigebbar extends Material {
  public PkDV gibPk();
  public DateDV gibAenderungsDatum();
  public BearbeiterKennzeichenDV gibBearbeiter();
}
```

   Dieses Interface wird von `Mobiliaetszuschlag` und `Reisebeihilfe` implementiert.

3. Außer der Freigabe weisen Mobilitätszuschläge und Reisebeihilfen noch weitere Gemeinsamkeiten auf. Dazu gehören unter anderem die Speicherung von Zivildienstplätzen und Personenkennzeichen der Antragsteller. Um duplizierten Quelltext zu beseitigen, wurden die Gemeinsamkeiten in eine abstrakte Basisklasse herausgezogen. Als Name der Klasse wurde aus den in *Motivation* beschriebenen Gründen `Einzelbetrag` gewählt. `Mobilitaetszuschlag` und `Reisebeihilfe` erben von `Einzelbetrag`. Da alle Einzelbeträge freigegeben werden müssen, wird das Interface `Freigebbar` bereits von der Oberklasse implementiert (siehe Abbildung 4.9). Nicht alle doppelt vorhandenen Methoden sind gleich implementiert, weshalb bei einigen lediglich die Methodensignatur in der Oberklasse definiert ist.

```
public abstract class Einzelbetrag implements Freigebbar {
  public abstract BearbeiterKennzeichenDV gibBearbeiter();

  public abstract DateDV gibAenderungsDatum();

  public boolean hatPk() {
    return _pk != null;
  }
  public PkDV gibPk() {
    return _pk;
  }
  public void setzePk(PkDV pkDV) {
    _pk = pkDV;
  }
  ...
  protected PkDV _pk;
}
```

   Die abstrakten Methoden der Klasse `Einzelbetrag` waren in der abgeleiteten Klasse `Reisebeihilfe` bereits implementiert. Die anderen in der Oberklasse vorhandenen Methoden brauchen in `Reisebeihilfe` nicht nochmals implementiert zu werden.

```
public class Reisebeihilfe extends Einzelbetrag implements
Freigebbar {
  public BearbeiterKennzeichenDV gibBearbeiter() {...};
  public abstract DateDV gibAenderungsDatum() {...};
  public boolean hatPk() {...}
  public PkDV gibPk() {...}
  public void setzePk(PkDV pkDV) {...}
  ...
  private PkDV _pkDV;
}
```

Analog zu `Reisebeihilfe` wird auch die Klasse `Mobilitaetszuschlag` angepasst.

```
public class Mobilitaetszuschlag extends Einzelbetrag implements
Freigebbar {...}
```

Konsequenzen

Mitunter enthalten mehrere Aspekte die gleichen Methoden, weil auch die mit den Aspekten abgebildeten fachlichen Anforderungen Überschneidungen haben. Diese Schnittmengen durch technische Refactorings in fachlich nicht nachvollziehbare Interfaces herauszuziehen würde dazu führen, dass die fachlichen Strukturen schwieriger im Quelltext wiedererkennbar wären. In den Materialklassen kommt es durch das Implementieren mehrerer Interfaces zu keinen Duplikationen, sodass diese WAM-spezifische Richtlinie auch aus Sicht des objektorientierten Entwurfs in Ordnung ist.

Zusammenhänge

Baue fachliche Materialhierarchie auf strukturiert Materialien und eliminiert mehrfach implementierte Methoden. Der Hauptgrund des Refactorings ist aber, dass Materialhierarchien die Voraussetzung für die Wiederverwendbarkeit von Werkzeugen, Services und Automaten sind. Daher kann nach dem Aufbau einer Hierarchie oft die Wiederverwendbarkeit von Werkzeugen verbessert werden, indem *Lasse Werkzeug mit Aspekt arbeiten (*siehe Abschnitt 4.2.2) angewendet wird.

4.2.2 Lasse Werkzeug mit Aspekt arbeiten

Übersicht

An verschiedenen Materialien einer Vererbungshierarchie sollen die gleichen Operationen ausgeführt werden. Für ein Material gibt es bereits ein Werkzeug zu diesem Zweck.
Lasse das Werkzeug mit einem Aspekt arbeiten, sodass es die Operationen an allen Materialien ausführen kann, die den Aspekt implementieren.

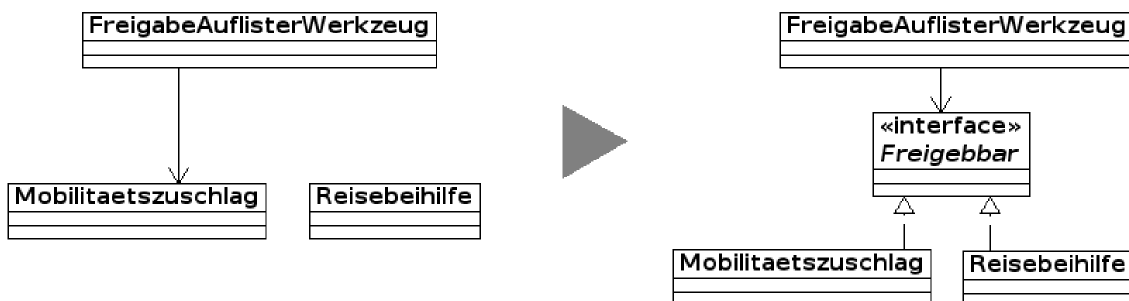

Abbildung 4.11: Das `FreigabeAuflisterWerkzeug` bearbeitet nicht mehr nur Mobilitätszuschläge.

Motivation

Das Werkzeug `FreigabeAuflisterWerkzeug` dient zum Anzeigen von Informationen über freizugebende Mobilitätszuschläge. Für Reisebeihilfen, die ebenfalls einem Freigabeprozess unterliegen, wurde ein aus fachlicher Sicht gleiches Werkzeug benötigt. Statt ein zweites zu implementieren sollte das Werkzeug so verändert werden, dass es mit allen freigebbaren Materialien umgehen kann. Auf der Grundlage der in *Baue fachliche Materialhierarchie auf* (siehe 4.2.1) eingeführten Materialhierarchie wurde zuerst ein weiteres Refactoring vollzogen, um danach das Werkzeug an die neuen Anforderungen anpassen zu können: Statt mit dem Material `Mobilitaetszuschlag` arbeitet die Klasse `FreigabeAuflister-Werkzeug` nun mit dem Aspekt `Freigebbar`. Danach wurde die Funktionalität des Werkzeugs erweitert, indem beim Erzeugen je nach Bearbeitungskontext verschiedene freigebbare Materialien mitgegeben wurden.

Ein bestehendes Werkzeug soll nicht länger auf einem Material arbeiten, sondern mit mehreren ähnlichen Materialien zurecht kommen. Dazu müssen zuerst die Gemeinsamkeiten der Materialien in Form einer Materialhierarchie herausgearbeitet werden, was mit mit *Baue fachliche Materialhierarchie auf* (siehe 4.2.1) erreicht werden kann. Die Wiederverwendbarkeit eines Werkzeugs wird durch das Refactoring erhöht, weil es nur noch einen Aspekt seiner Materialien kennt. Das Werkzeug kann mit allen Materialien arbeiten, die diesen Aspekt anbieten.

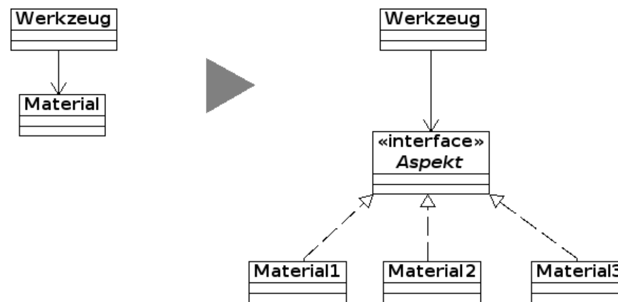

Abbildung 4.12: Ein Werkzeug arbeitet mit einem Aspekt statt direkt mit einem Material.

Schritte

1. Ersetze im Werkzeug den Typ des bisher bearbeiteten Materials mit dem Aspekt. Zu den anzupassenden Stellen gehören typischerweise Exemplarvariablen und überschriebene Einschubmethoden des JWAM-Rahmenwerks (`doUseMaterial()`, `doUpdateMaterial()` und `getAspect()`). Alle Tests laufen.

   Bearbeitet das Werkzeug das Material mit Methoden, die im Aspekt nicht vorgesehen sind, muss der Aspekt auf Unvollständigkeit überprüft werden. Fehlen dem Aspekt Methoden um seinen fachlichen Aufgabenbereich zu erfüllen, müssen diese ergänzt werden (eventuell mit *Baue fachliche Materialhierarchie auf*, siehe 4.2.1). Betreffen die fraglichen Methoden einen anderen fachlichen Kontext als der Aspekt vorsieht, muss ein Teil der Funktionalität des Werkzeugs in ein anderes Werkzeug ausgelagert werden (*Zerlege Werkzeug in kleinere Werkzeuge*, siehe 4.3.2).

2. Ersetze den Typ des Materials an den Stellen, an denen das Werkzeug erzeugt und ausgestattet wird. Alle Tests laufen.

3. Überprüfe die Namen des Werkzeugs sowie der Methoden und Attribute auf Verweise auf den alten Typ des Materials. Ändere gegebenenfalls die Namen so um, dass darin keine Bezeichnungen konkreter Materialien mehr vorkommen. Alle Tests laufen.

Nun kann die Funktionalität des Werkzeugs erweitert werden, indem beim Ausstatten andere Materialien mit gleichem Aspekt übergeben werden. Wenn das Werkzeug die bearbeiteten Materialien anderen Werkzeugen zur Verfügung stellt, müssen diese damit umgehen können.

Beispiel

1. In der Klasse `FreigabeAuflisterWerkzeug` wird der Typ des bearbeiteten Materials von `Mobilitaetszuschlag` auf den Aspekt `Freigebbar` geändert. Allerdings arbeitet das Werkzeug nicht nur mit einem Materialobjekt, sondern mit einer Liste solcher Objekte. Diese Liste (`DomainContainer`) ist selbst ein Material.

```
public class FreigabeAuflisterWerkzeug extends AbstractToolMono{
  public Class getAspect() {
        return DomainContainer.class;
  }
```

```
    protected void doUseMaterial(Thing material) {
      _materialien = (List<Mobilitaetszuschlag Freigebbar>) material;
      _gui.setzeMaterialListe(_materialien);
    }
    ...
    private List<Mobilitaetszuschlag Freigebbar> _materialien;
  }
```

2. Im Kontextwerkzeug (`HauptWerkzeug`) des Freigabeauflisters muss das Werkzeug nun mit Objekten des Typs `Freigebbar` ausgestattet werden. Die Objekte selbst werden über einen Serviceaufruf bezogen. Mit der Methode `tryToCreateSubtool(...)` wird der Freigabeauflister als Subwerkzeug erzeugt und mit Material ausgestattet.

```
public class HauptWerkzeug extends AbstractToolMono {
  ...
  private void erzeugeFreigabeAuflisterSubwerkzeug() {
    DomainContainer freigaben = new DomainContainer(
        _zdlService.gibFreizugebendeMobilitaetszuschlaege());
    _freigabeAuflister = tryToCreateSubtool(
      FreigabeAuflisterWerkzeug.class, "FreigabeAuflister", freigaben);
  }
  ...
  private Tool _freigabeAuflisterWerkzeug;
  private ZDLService _zdlService;
}
```

3. Mit `FreigabeAuflisterWerkzeug` hat die Klasse bereits einen passenden Namen. Auch die Namen der Methoden und Attribute geben keinen Hinweis darauf, dass das Werkzeug ursprünglich nur mit Mobilitätszuschlägen umgehen konnte.

Im Anschluss an das Refactoring wurde die Funktionalität erweitert, indem der Freigabeauflister je nach Einsatzkontext mit Mobilitätszuschlägen oder Reisebeihilfen ausgestattet wird.

```
public class HauptWerkzeug extends AbstractToolMono {
  ...
  private void erzeugeFreigabeAuflisterSubwerkzeug(
    Class<? extends Freigebbar> materialTyp) {

    DomainContainer freigaben = new DomainContainer(
        _zdlService.gibFreizugebendeMobilitaetszuschlaege());

    if (Mobilitaetszuschlag.class.isAssignableFrom(materialTyp)) {
      materialContainer.addAll(
        _zdlService.gibFreizugebendeMobilitaetszuschlaege());
    } else if (Reisebeihilfe.class.isAssignableFrom(materialTyp)) {
      materialContainer.addAll(
        _zdlService.gibFreizugebendeReisebeihilfen());
    }
    _freigabeAuflister = tryToCreateSubtool(
      FreigabeAuflisterWerkzeug.class, "FreigabeAuflister", freigaben);
  }
  ...
  private Tool _freigabeAuflisterWerkzeug;
  private ZDLService _zdlService;
}
```

Zusammenhänge

Eine Vorbedingung für die Durchführung dieses Refactorings ist eine Materialhierarchie, wie sie mit *Baue fachliche Materialhierarchie auf* (siehe 4.2.1) angelegt werden kann.

4.2.3 Verschiebe Fachlogik in Service

Übersicht

Mehrere Werkzeuge verfügen über teilweise gleiche Fachlogik.
Verschiebe diese Fachlogik in einen fachlichen Service.

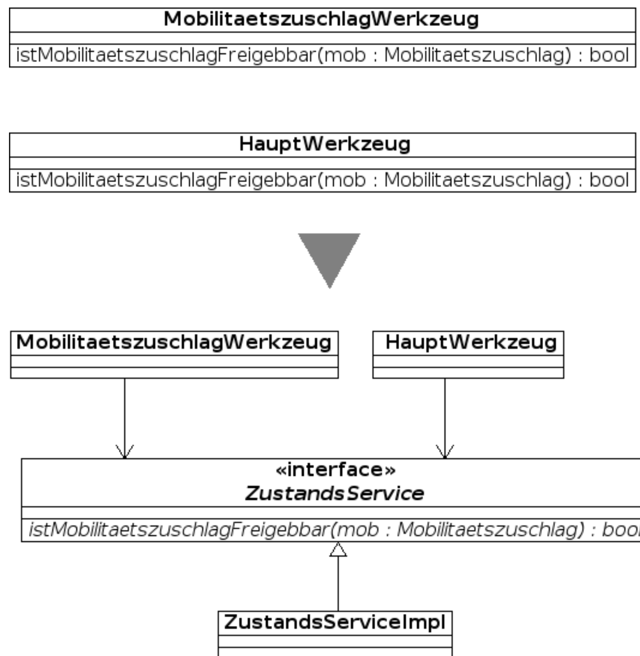

Abbildung 4.13: Eine in zwei Werkzeugen vorhandene Methode wird in einen Service verschoben

Motivation

In der Klasse `HauptWerkzeug` verschob ein Refactoring (*Extract Method* [Fowler00]) die Logik zur Zustandsüberprüfung eines Mobilitätszuschlags in eine eigene Methode. Dabei wurde bemerkt, dass eine Methode mit gleichem Zweck bereits in der Klasse `MobilitaetszuschlagWerkzeug` vorhanden war. Wenn sich die Bedingungen des Zustands ändern, muss die Überprüfung an zwei Stellen angepasst werden.

Duplizierte Logik ist einer der Hauptgründe für die Fehleranfälligkeit und schlechte Wartbarkeit von Software ([Wake04]). Werkzeug- bzw. Automaten-übergreifende Logik wird in fachliche Services ausgelagert, um mehrfachen Implementierungs- und Wartungsaufwand zu vermeiden. Im WAM-Ansatz gelten für fachliche Services aber Einschränkungen (siehe Abschnitt 2.1.3), die sie etwa von Bibliotheken abgrenzen. *Verschiebe Fachlogik in Service* soll daher tatsächlich nur auf fachlich zusammengehörige Logik angewendet werden.

Verschiebe Fachlogik in Service ist eine Adaption von *Move Method* ([Fowler00]). So wie im angeführten Beispiel kann es nötig sein, den Quelltext erst in einer Methode zusammenzufassen. Eventuell müssen die Schnittstellen der Methoden, die durch eine Service-Methode ersetzt werden sollen, erst vereinheitlicht werden. Solche vorbereitenden Maßnahmen werden in den Schritten nicht berücksichtigt.

Schritte

1. Definiere ein Service-Interface mit der zu verschiebenden Methode. Möglicherweise gibt es einen Service mit Aufgaben, die eine fachliche Einheit mit der zu verschiebenden Methode bilden und wo

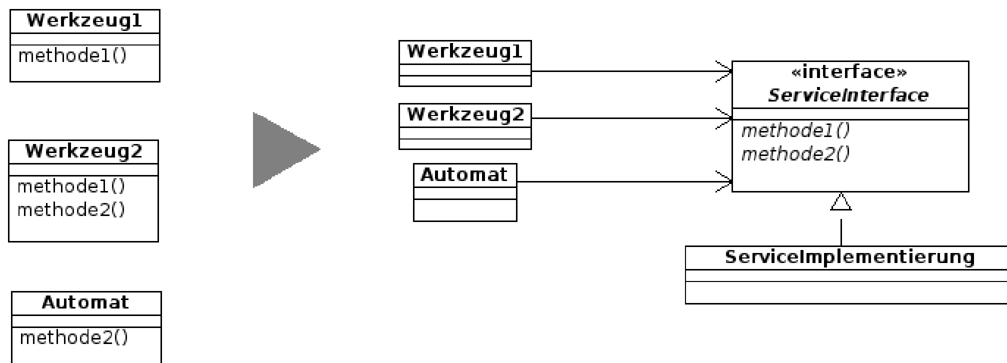

Abbildung 4.14: Verschiebe Fachlogik in Service

die Methode eingefügt werden kann[2]. In einer Klasse, die das Service-Interface implementiert, wird der Methoden-Rumpf erstellt. Der Test für die neue Service-Methoden schlägt noch fehl. Eventuell kann der Test aus der ursprünglichen Klasse kopiert werden.

2. Kopiere die Implementierung der Service-Methode aus der ursprünglichen Klasse in den Service. Die Tests für den Service laufen.
3. Registriere den Service in der JWAM Service-Registratur. Der Service ist jetzt verwendbar.
4. Ersetze in den ursprünglichen Klassen die Implementierung der verschobenen Methode durch Service-Aufrufe. Alle Tests laufen.
5. Statt in den ursprünglichen Klassen die lokalen Methoden mit dem Service-Aufruf zu verwenden, kann immer direkt auf den Service zugegriffen werden. Die Umleitungen lassen sich mit *Inline Method* ([Fowler00]) entfernen.

Beispiel

Zivildienstleistende haben Anspruch auf einen Fahrtzuschuss, genannt Mobilitätszuschlag. Vor der Auszahlung eines Zuschlags muss der Antrag in ein Software-System eingegeben und überprüft werden. Dabei ändert sich der Status des als Material modellierten Mobilitätszuschlags. Um zur Auszahlung freigegeben zu werden, muss er zuvor von zwei Sachbearbeitern überprüft worden sein. Die beiden Bearbeiter erfüllen ihre Aufgaben mit unterschiedlichen Werkzeugen. Daher muss an mehreren Stellen anhand des Zustands eines Mobilitätszuschlags unterschieden werden, welches Werkzeug aktiviert werden soll bzw. ob ein Werkzeug mit dem Material umgehen kann.

1. In mehreren Werkzeugen findet sich je eine Methode `istFreigebbar(Mobilitaetszuschlag mob)` mit gleicher Logik. Da es noch keinen Service gibt, in den sich Methoden zur Zustandsprüfung fachlich sinnvoll eingliedern ließen, wird eine Serviceschnittstelle `ZustandsService` mit einer Methode `istFreigebbar(Mobilitaetszuschlag mob)` definiert. `ZustandsServiceImpl` implementiert dieses Interface. Da der Service ausschließlich lokal verwendet wird, braucht die Methode keine Exceptions an der Schnittstelle zu definieren.

```
public interface ZustandsService extends Service {
  public boolean istFreigebbar(Mobilitaetszuschlag mob);
}

public class ZustandsServiceImpl implements ZustandsService {
  public synchronized boolean istFreigebbar(Mobilitaetszuschlag mob) {
    return false;
```

[2] Im Folgenden wird von einem neu zu erstellenden Service ausgegangen.

```
    }
}
```

2. Aus einem der Werkzeuge wird die Implementation der Methode in `ZustandsServiceImpl` kopiert. Damit erfüllt der Service seinen Test und ist fertig.

```
public synchronized boolean istFreigebbar(Mobilitaetszuschlag mob) {
  if (mob.hatZustand()) {
    ...
  }
}
```

3. Die Service-Implementierung wird in den Startklassen der Anwendung und der Tests in der Registratur angemeldet. Alle Werkzeuge sowie die Tests können den Service nun verwenden.

4. In den Werkzeugen mit der zu verschiebenden Methode muss ein Service-Objekt aus der Registratur bezogen werden, an dem `istFreigebbar()` aufgerufen werden kann. Dieser Aufruf ersetzt die Implementierung der lokalen `istFreigebbar()` Methoden.

```
public boolean istFreigebbar(Mobilitaetszuschlag mob) {
  if (mob.hatZustand()) {
    ...
  }
  return _service.istFreigebbar(mob);
}

private ZustandsService _service = (ZustandsService) Environment.
  instance().getServiceRegistry().getService(ZustandsService.class);
```

5. Da die Tests der Werkzeuge laufen, kann an Stelle der Aufrufe der lokalen Methode nun direkt der Service benutzt werden. Die nicht mehr benötigten lokalen Methoden werden gelöscht.

```
public boolean istFreigebbar(Mobilitaetszuschlag mob) {
  return _service.istFreigebbar(mob);
}

private void erzeugeWerkzeug(Mobilitaetszuschlag mob) {
  ...
  if (istFreigebbar(mob)) {
  if (_umgebungsService.istFreigebbar(mob)) {
    erzeugeMobilitaetszuschlagFreigabeSubwerkzeug();
  }
  ...
}
```

Konsequenzen

Der Einsatz von Services ist einer der Gründe für die überwiegend monolithische Konstruktion von Werkzeugen. Services nehmen die Rolle der Funktionskomponente ein, sind aber dabei nicht an ein bestimmtes Werkzeug gebunden - können und werden also von verschiedenen Werkzeugen verwendet.

Im ersten Schritt des Refactorings wird angeregt, neue Service-Methoden in einen bestehenden Service einzufügen, sofern die neue Methode in den Kontext des Services passt. Ist das nicht der Fall, stellt sich die Frage, wie viele Methoden nötig sind um eine eigene Service-Klasse zu rechtfertigen. Ob fachliche Einheit oder eine kleinere Anzahl von Interfaces und Klassen besser sind, muss von Fall zu Fall entschieden werden. Für das Beispiel der Zustandsprüfung fiel die Entscheidung für einen eigenen Service, weil noch drei weitere, ähnliche Methoden als Duplikate identifiziert und verschoben wurden.

Zusammenhänge

Als Nebeneffekt des Refactorings werden mindestens zwei Klassen kleiner. Weniger Logik und kleinere Klassen können die Komplexität innerhalb der Klassen reduzieren, allerdings entstehen dadurch mehr Klassen. Das Verhältnis zwischen reduziertem Umfang und steigender Anzahl der Klassen wird besser, je häufiger die ausgelagerten Methoden benutzt werden.

4.3 Komplexität beherrschen

Die Refactorings in diesem Abschnitt helfen, komplexe Klassen durch Zerlegung in kleinere Klassen einfacher und verständlicher zu machen, vergleichbar mit *Extract Class* [Fowler00]. Eine Folge der vereinfachten inneren Struktur der Klassen sind komplexere Beziehungen zwischen den Klassen.

4.3.1 Extrahiere Automat

Übersicht

Werkzeuge oder Services führen zusammenhängende Arbeitsschritte selbstständig aus.
Baue für diese Arbeitsschritte einen Automaten, inklusive der eventuell dazugehörenden Konfiguration.

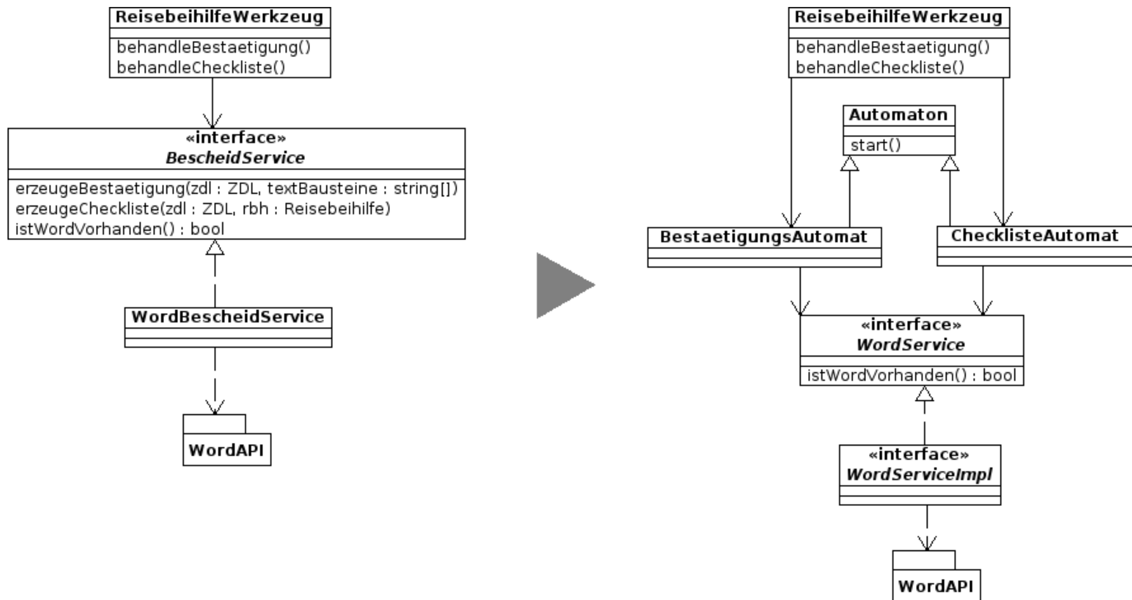

Abbildung 4.15: Automaten bündeln Logik zur Bescheiderstellung

Motivation

Zivildienstleistende bekommen Fahrtkosten, die durch Familienheimfahrt oder Dienstreisen entstehen, zum Teil durch Beantragung von Reisebeihilfe abgegolten. Nach Abarbeitung der Anträge müssen den Antragstellern verschiedene Bescheide zugestellt werden, deren Anfertigung für die Sachbearbeiter Routinearbeit ist. Der in Abbildung 4.15 links dargestellte Ausschnitt einer Software unterstützt die Sachbearbeiter dabei, in dem je nach Bescheidart die benötigten personen- und antragsbezogenen Daten in eine Bescheidvorlage kopiert werden. Im Reisebeihilfe-Werkzeug gibt es dazu Methoden (`behandleBestaetigung()`, `behandleCheckliste()`), die auf Knopfdruck des Benutzers Konfigurationsdialoge zeigen, in denen aus vorgefertigten Textbausteinen Teile des Bescheids zusammengestellt werden können. Die gewählten Textbausteine sowie die Daten des Zivildienstleistenden werden an einen Service weitergereicht, der die Erstellung der Bescheide übernimmt. Dazu kapselt und verwendet er eine Java-API zum Erzeugen von Microsoft Word Dokumenten. Eine Analyse mit dem Sotographen (siehe 3.2) zeigte, dass sowohl `ReisebeihilfeWerkzeug` als auch `WordBescheidService` sehr große Klassen (über 1000 Zeilen) mit einer Vielzahl an Methoden sind. Zudem ist die Logik zur Bescheiderstellung auf beide Klassen verteilt und `BescheidService` bietet zwei nicht zusammengehörige Funktionalitäten: Zum einen die fachlich motivierten Methoden zur Bescheiderstellung, zum anderen technisch motivierte Methoden zur Kapselung der verwendeten Bibliothek zur Ansteuerung von Word. Dass die Erstellung von Bescheiden eine automatisierbare Routinetätigkeit ist, wird durch die Struktur des Quelltext nur bei genauem Hinsehen

klar. Die entsprechenden Arbeitsschritte werden immer in einer fixen Reihenfolge ausgeführt, der Benutzer ist lediglich Auslöser des Prozesses. Zudem läuft die Erstellung der Word-Datei vom Benutzer unbemerkt im Hintergrund ab und er kann während dessen bereits andere Arbeiten erledigen.

Aus diesen Gründen stellt ein Automat ein geeignetes Mittel dar, um das Werkzeug zu vereinfachen und den Service von seiner Doppelrolle zu entlasten. Darüber hinaus entspricht die Semantik der Metapher Automat besser den Gegebenheiten der automatisierten Bescheiderstellung als die bisherige Lösung.

Wenn Werkzeugen und Services fachlich motivierte Methoden enthalten, die nacheinander und ohne Eingreifen des Benutzers abgearbeitet werden, können diese automatisierten Arbeitsschritte in einen Automaten extrahiert werden. Eine eventuell im Werkzeug durchgeführte Konfiguration des automatisierten Prozesses kann ebenfalls in den Automaten verlagert werden. Nur für umfangreiche Konfigurationsmaßnahmen bietet sich die Aufteilung in Automat und Konfigurationswerkzeug an. Wie in dem Beispiel des Bescheiddrucks können die zu verschiebenden Funktionen auf mehrere Klassen verteilt sein, was deren Identifikation erschwert. In der Skizze dieses Refactorings (Abbildung 4.16) wird dieser Fall berücksichtigt, bei dem *Extrahiere Automat* einen zusätzlichen Nutzen bietet: Zusammenhängende Logik wird an einer Stelle gebündelt.

Der Ausgangs- und Zielzustand dieses Refactorings kann je nach Anwendung stark variieren. Im in Abbildung 4.16 skizzierten Fall bleibt der Service erhalten, etwa weil `prozessSchritt3()` auf ein externes System zugreift (wie im Beispiel „Bescheiddruck”). Unter anderen Gegebenheiten könnte der Service unnötig und daher gelöscht werden, oder es ist überhaupt nur ein Service *oder* ein Werkzeug am Ausgangszustand beteiligt.

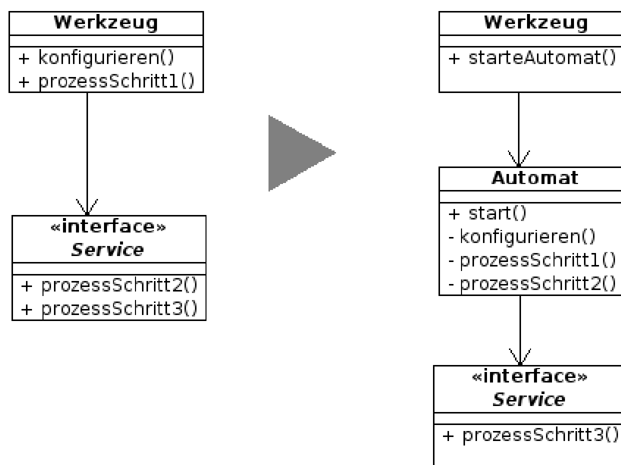

Abbildung 4.16: Ein Automat bündelt einen automatisierten, auf mehrere Klassen verteilten Prozess

Schritte

1. Erstelle einen von `Automaton` abgeleiteten Automat, dessen Methoden noch keine Logik aufweisen. Der Test für die Methode `start()` kann eventuell aus einem vorhandenen Test eines Werkzeugs bzw. Service kopiert werden. Dieser Test schlägt fehl.

2. Kopiere die einzelnen Methoden, die den zu automatisierenden Prozess ausmachen (in Abbildung 4.16: `prozessSchritt1` und `prozessSchritt2`), in den Automaten. Aufgerufen werden sie in der Methode `start()` des Automaten . Methoden, die in die Zuständigkeit eines Service fallen (z.B. Kapselung eines externen Systems), verbleiben dort (in Abbildung 4.16: `prozessSchritt3`). Damit die Klasse wieder übersetzbar wird, braucht der Automaten noch Informationen:

   (a) Die von den Methoden benötigten Materialien und Fachwerte müssen vom Aufrufer des Automaten mitgegeben werden. Das wichtigste Material des Automaten kann dabei mit `setMaterial()` (aus der Oberklasse `Automaton`) gesetzt werden. Für weitere benötigte Ob-

jekte müssen entsprechende Methoden ergänzt werden. Vor dem Start des Automaten muss sichergestellt sein, dass dieser alle nötigen Daten zur Verfügung hat.

(b) Wurde bisher dem Benutzer im Werkzeug eine Möglichkeit geboten, den Ablauf zu konfigurieren, so muss diese Konfiguration ebenfalls in den Automaten verschoben werden. Aufgerufen wird sie noch vor allen anderen Methoden in `start()`. Umfangreiche Konfiguration kann in ein eigenes Werkzeug verschoben werden. Das bestehende Werkzeug und das neue Konfigurationswerkzeug können mit *Zerlege Werkzeug in kleinere Werkzeuge* (siehe 4.3.2) getrennt werden.

Der Automat ist jetzt funktionsfähig, sein Test läuft durch.

3. Nun wird der Automat wie ein Service in die JWAM Registratur eingetragen. Dieser Schritt ist bei Automaten nicht unbedingt notwendig (JWAM sieht daher auch keine eigene Registratur für Automaten vor), erleichtert aber das Austauschen von Implementierungen von Automaten. Beispielsweise kann ein lange laufender Automat für Tests durch eine Attrappe (engl.: *mock*, siehe [Westphal06]) ersetzt werden.

4. Ein Werkzeug oder Service verwendet den Automat. Alle Tests laufen.

5. Lösche im Werkzeug bzw. Service alle nicht mehr benötigten Methoden sowie deren Tests. Alle Tests laufen.

Beispiel

Wie in Abbildung 4.15 zu sehen ist, wurde *Extrahiere Automat* mehrmals angewendet, um die Klassen `WordBescheidService` und `ReisebeihilfeWerkzeug` zu vereinfachen. Hier wird nur eine solche Extraktion beschrieben, bei deren Ausführung eine Klasse `ChecklisteAutomat` erzeugt wird. Um eine Checkliste zu erstellen, benötigt der Automat Informationen aus einem ZDL- und einem Reisebeihilfe-Objekt.

1. Die neue Klasse `ChecklisteAutomat` wird angelegt. Abbildung 4.17 zeigt die Schnittstelle des Automaten. Bis auf die Methode `kannGestartetWerden()` überschreiben alle die entsprechenden Methoden der Basisklasse `Automaton`. Mittels `kannGestartetWerden()` kann vor dem Start des Automaten überprüft werden, ob dieser alle benötigten Informationen zur Verfügung hat.

ChecklisteAutomat
hasMaterial() getMaterial() setMaterial() start() canHandleMaterial() kannGestartetWerden()

Abbildung 4.17: Öffentliche Methoden des neuen Automaten

2. Zunächst werden die Prozessschritte in den Automaten kopiert:

```
public class ChecklistenAutomat extends Automaton {
  public void start() {
    assert kannGestartetWerden() : "Automat ausreichend ausgestattet";
    erzeugeCheckliste(...);
  }
  ...
  private void erzeugeCheckliste(ZDL zdl, Reisebeihilfe rbh,
    String[] textBausteine)  {
    _wordService.erzeugeDokumentAusVorlage(gibWordSchabloneCheckliste());
    befuelleCheckliste(zdl, rbh);
```

```
    ...
  }
  private void befuelleCheckliste(ZDL zdl, Reisebeihilfe rbh,
    String[] textBausteine) {
    _wordService.schreibeTextInTextmarke("ZDL", zdl.gibName().toString());
    ...
  }
  ...
  BescheidService _wordService = ((WordService) Environment.instance().
    getServiceRegistry().getService(BescheidService.class));
}
```

Im Service verbleiben alle Methoden, welche die Word-API kapseln. Daher benötigt der Automat ein Service-Objekt (`_wordService`). Die Klasse lässt sich trotzdem noch nicht übersetzen, weil die Parameter der Methoden `erzeugeCheckliste()` und `befuelleCheckliste()` fehlen.

(a) Die `...material()` Methoden des Automaten werden überschrieben, damit sie mit Reisebeihilfe-Objekten umgehen könne. Um dem Automaten ein ZDL-Objekt übergeben zu können, werden zusätzliche Methoden implementiert.

```
public void setMaterial(Thing thing) {
  assert canHandle(thing) : "Vorbedingung verletzt:
    Material ist Reisebeihilfe";
  _rbh = (Reisebeihilfe) thing;
}
public void setzeZDL(ZDL zdl) {
  _zdl = zdl;
}
...
Reisebeihilfe _rbh;
ZDL _zdl;
```

(b) Nun fehlt noch die Konfiguration der Textbausteine, um den Quelltext übersetzbar zu machen. Checklisten werden vom Sachbearbeiter mit einer Liste von Texten konfiguriert und dann mit den Daten aus des Antragstellers und des Antrags personalisiert. Ein Dialog zur Konfiguration befindet sich im `ReisebeihilfeWerkzeug`, Methoden zum Erzeugen und Befüllen der Checkliste im `BescheidService`. Da die Konfiguration des Automaten nicht aufwändig ist, wird der Dialog nicht als eigenes Werkzeug realisiert.

```
private String[] konfigurieren() {
  String[] dialogErgebnis = null;
  JDialog dialog = new CheckboxListeDialog(...);
  ...
  return dialogErgebnis;
}
```

Die Methode `start()` kann nun fertig gestellt werden und der Test des Automaten läuft.

```
public void start() {
    assert kannGestartetWerden() : "Automat ausreichend ausgestattet";
    String[] textBausteine = konfigurieren();
    erzeugeCheckliste(_zdl, _rbh, textBausteine);
  }
```

3. Der Checklisten-Automat wird beim Programmstart in die JWAM Service-Registratur eingetragen.

```
ConfigurationFacade.registerService(ChecklistenAutomat.class, null);
```

4. In der Klasse `ReisebeihilfeWerkzeug` wird der Automat verwendet. Wenn der Benutzer auf den Knopf zur Bescheiderstellung drückt, wird `behandleCheckliste()` aufgerufen.

```
private void behandleCheckliste {
  String[] textBausteine = konfigurieren();
  erzeugeCheckliste(_zdl, _rbh, textBausteine);
  ChecklistenAutomat automat = (ChecklistenAutomat )Environment.
    instance().getServiceRegistry().getService(ChecklistenAutomat.class);
  automat.setMaterial(_rbh);
  automat.setzeZDL(_zdl);
  if (automat.kannGestartetWerden()) {
    automat.start();
  }
}
```

5. Die im Werkzeug nicht mehr benötigten Methoden `konfigurieren()`, `erzeugeCheckliste()` und `befuelleCheckliste()` werden gelöscht. Die Service-Implementierung `WordBescheid-Service` hat nur noch Methoden, die die Word-API kapseln. Interface und Implementierung bekommen daher einen passenderen Namen: `WordService` und `WordServiceImpl`. Abbildung 4.18 vergleicht Ausschnitte aus dem Service vor und nach dem Refactoring.

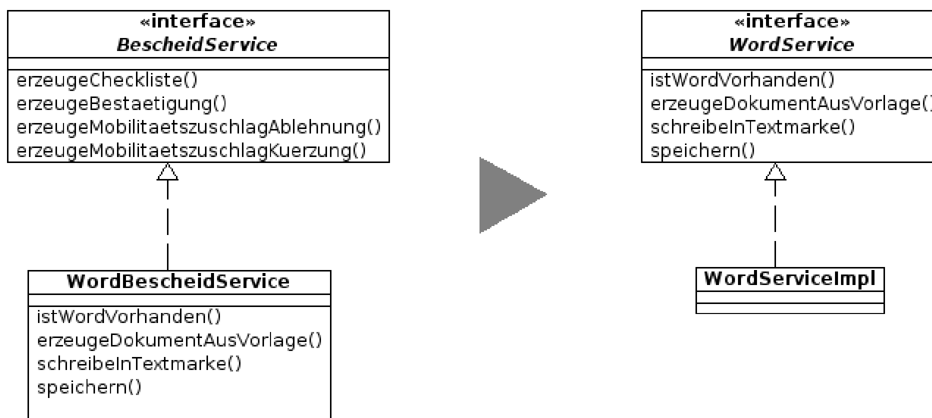

Abbildung 4.18: `BescheidService` bzw. `WordService` vor und nach dem Refactoring

Konsequenzen

Der Bedarf an *Extrahiere Automat* ist ein Zeichen einer unpassenden Modellierung und könnte daher auch unter *Modellierung korrigieren* eingeordnet werden. Weil aber verstreute Fachlogik und komplexe Klassen die einfacher zu identifizierenden Smells sind, findet sich dieses Refactoring unter *Komplexität beherrschen*.

Werden Methoden aus einem Werkzeug oder Service in verschiedene Automaten aufgeteilt (so wie im Beispiel), müssen vorher gemeinsam verwendete Hilfsmethoden in alle Automaten kopiert werden. Diese Vervielfachung von Quelltext ist zwar ein Argument gegen die Durchführung des Refactorings, kann aber nach Fertigstellung der Automaten mittels *Extract Class* [Fowler00] behoben werden: Gemeinsamkeiten der extrahierten Automaten werden in eine abstrakte Oberklasse verschoben. Abbildung 4.19 zeigt, dass dieses Problem im Beispiel aufgetreten ist und mit einem abstrakten Automaten gelöst wurde.

Zusammenhänge

Extrahiere Automat ähnelt *Zerlege Werkzeug in kleinere Werkzeuge* (siehe 4.3.2) was Zweck (Vereinfachen von Klassen mit umfangreicher Fachlogik) und Mittel (Auftrennung entlang fachlicher Grenzen) betrifft. Der Unterschied liegt in der Wahl des WAM-Elements, in das ein Teil der Fachlogik verschoben werden soll. Handelt es sich um zwar zusammengehörige, aber vom Benutzer einzeln gesteuerte Funktionen, entspricht das der Metapher Werkzeug. Wenn die Funktionen automatisch, ohne Benutzerinteraktion ablaufen, eignet sich ein Automat als Zielelement für die Fachlogik.

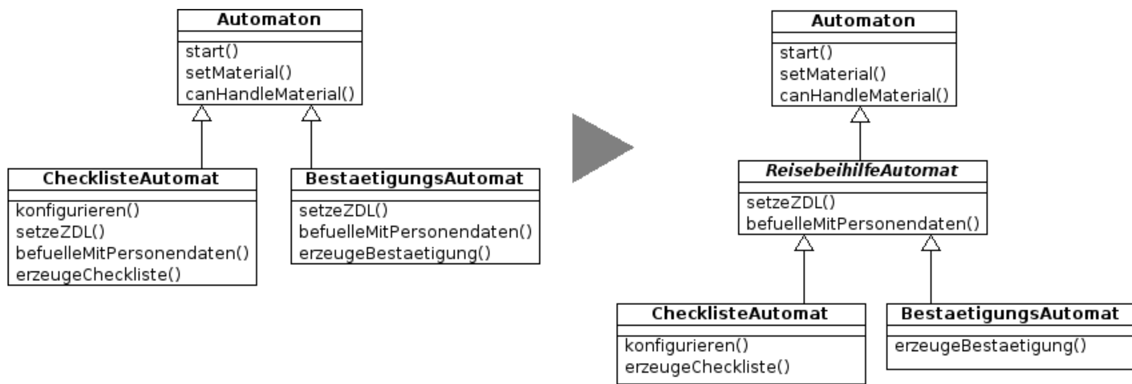

Abbildung 4.19: Einführen einer Oberklasse vermeidet Redundanzen

Verallgemeinerung

Extrahiere Automat hängt stärker von der Metapher Automat ab als von deren Implementierung in JWAM. Bei der Entscheidung für dieses Refactoring ist es wesentlich, ob die zu verschiebenden Arbeitsschritte einen zusammenhängenden, automatisierten Prozess ausmachen, also dem Charakteristikum eines Automaten. Diese Entscheidung ist unabhängig von Plattform und Rahmenwerk. Außerdem macht die Klasse `Automaton` in JWAM wenig Vorgaben über die Implementierung von Automaten: Sie sollen auf einem Material arbeiten und sich starten lassen. Das in den Schritten beschriebene Registrieren eines Automaten ist optional. Aus diesen Gründen lässt sich die Beschreibung von *Extrahiere Automat* auch ohne JWAM anwenden.

4.3.2 Zerlege Werkzeug in kleinere Werkzeuge

Übersicht

Der Aufgabenbereich eines Werkzeugs ist sehr groß, wodurch der Quelltext des Werkzeugs unübersichtlich wird.
Finde abgeschlossene Teilaufgaben in der fachlichen Logik und verteile diese Logik auf eigene Werkzeuge.

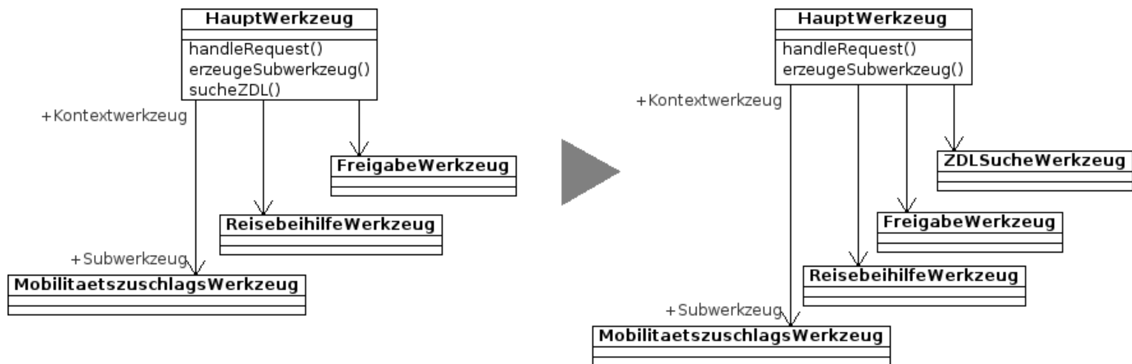

Abbildung 4.20: Die Funktionalität zum Suchen von Zivildienstleistenden wird in ein eigenes Werkzeug ausgelagert.

Motivation

Die Klasse `HauptWerkzeug` ist das oberste Kontextwerkzeug einer Software zur Verwaltung von Zahlungen an Zivildienstleistende, das heißt es verwaltet alle anderen Werkzeuge. In die GUI des Hauptwerk-

zeugs werden dynamisch die Benutzeroberflächen der Subwerkzeuge eingebettet. Auf Grund der wachsenden Anzahl an zu verwaltenden Werkzeugen wuchs die Klasse immer mehr. Zu den fachlichen Aufgaben der Klasse gehörte eine Funktion zum Suchen von Zivildienstleistenden nach deren Personenkennzahl. Um die Klasse zu vereinfachen, wurde die Suchfunktion in eine eigene Klasse `ZDLSucheWerkzeug` verschoben und dieses Werkzeug in das Hauptwerkzeug eingebettet.

Ein Mittel zur Vereinfachung umfangreicher Werkzeuge ist die Ausgliederung fachlicher Funktionalität in andere Werkzeuge. Aber auch schon bevor ein Werkzeug unübersichtlich groß wird kann das Refactoring eingesetzt werden. Kleinere Werkzeuge lassen sich meist besser erweitern als große, daher kann es die Erweiterung einer Teilaufgabe eines Werkzeugs unterstützen, in dem die betroffene Funktionalität erst ausgelagert wird. Im Gegensatz zu Vererbungshierarchien oder Objektkomposition, die auch den Quelltext von Werkzeugen reduzieren können, werden bei *Zerlege Werkzeug in kleinere Werkzeuge* die Werkzeuge entlang fachlicher Grenzen zerlegt (siehe Abbildung 4.21).

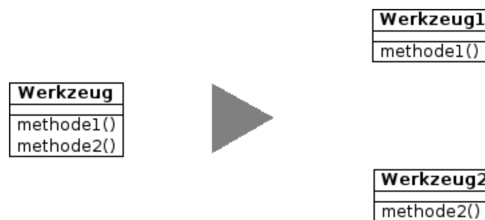

Abbildung 4.21: Ein Teil der Funktionalität eines Werkzeugs wird in ein eigenes Werkzeug ausgelagert.

Das Refactoring bewirkt kleinere, das heißt übersichtlichere und leichter verständliche Werkzeuge. Eventuell kann ihre Funktionalität nur als Ganzes eingesetzt werden. Dem Benutzer wird dann die gesamte Funktionalität durch Einbettung in ein Kontextwerkzeug zur Verfügung gestellt. Durch den Einbettungsmechanismus in JWAM kann eine solche Änderung ohne merkbare Änderungen an der Benutzeroberfläche durchgeführt werden. Es kann aber einfacher oder sogar gewünscht sein, die neu entstandenen Werkzeuge auch an der Oberfläche zu trennen. Auch eine solche Änderung erfüllt wegen der gleich bleibenden Funktionalität die Kriterien eines Refactorings. Die bei dem Refactoring entstehenden Werkzeuge sind kleiner als das ursprüngliche, bringen aber zusätzlichen Verwaltungsaufwand: Die Werkzeuge müssen ausgestattet und erzeugt werden. Vermutlich arbeiten sie auf den gleichen Materialien und müssen einander mittels loser Kopplung über Zustandsänderungen informieren. Dieser Aufwand bringt nur Vorteile, wenn die verschobene Fachlogik umfangreich genug ist oder das ursprüngliche Werkzeug nicht zusammenhängende Aufgaben erledigt hat. Im letzteren Fall kann der Gewinn an Strukturähnlichkeit die Nachteile des zusätzlichen Aufwands übertreffen.

Eingebettet können die neuen Werkzeuge auf zwei verschiedene Arten werden. Entweder, eines dient als Kontextwerkzeug des anderen (Abbildung 4.22) oder sie werden beide von einem anderen Kontextwerkzeug verwaltet (Abbildung 4.23). Die erste Variante (sie wurde auch im in Abbildung 4.20 gezeigten Beispiel verwendet) bedeutet für den Benutzer, dass `Werkzeug2` eine Teilaufgabe von `Werkzeug1` realisiert. Bei der zweiten Variante sind die Werkzeuge in ihrer fachlichen Funktion unabhängiger.

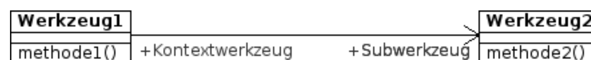

Abbildung 4.22: Die zerlegten Werkzeuge in einer Kontextwerkzeug-Subwerkzeug-Beziehung.

Die folgenden Beschreibung der Schritte setzt die Implementierung aller beteiligten Werkzeuge als monolithische Werkzeuge voraus. Zwar lässt sich das Refactoring auch auf komplexe Werkzeuge anwenden (siehe dazu den Abschnitt „Verallgemeinerung"), doch wird darauf aus zwei Gründen nicht näher eingegangen: Komplexe Werkzeuge sind in der Praxis weniger relevant als monolithische und auf Grund der Aufteilung in Interaktions- und Funktionskomponente schon von sich aus in kleineren Einheiten strukturiert.

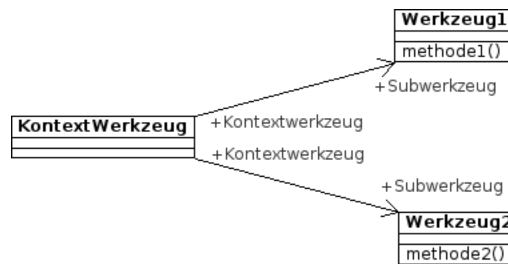

Abbildung 4.23: Die zerlegten Werkzeuge als Subwerkzeuge eines bestehenden Kontextwerkzeugs

Schritte

1. Erstelle eine von `AbstractToolMono` abgeleitete, monolithische Werkzeug-Klasse und passe sie an das zu bearbeitende Material an. Die Tests laufen durch.

2. Kopiere die GUI der auszulagernden Funktionalität in die neu angelegte GUI-Klasse des neuen Werkzeugs und binde sie an die Werkzeug-Klasse an. Da das Werkzeug noch nirgends mit GUI gestartet wird, laufen die Tests trotz fehlender Fachlogik durch.

3. Kopiere die Fachlogik in das neue Werkzeug. Die Tests der Logik können aus denen des alten Werkzeugs kopiert werden laufen durch. Eine Ausnahme bilden Oberflächentests (siehe Schritt 5). Das neue Werkzeug ist jetzt funktionstüchtig.

4. Binde das neue Werkzeug an ein Kontextwerkzeug an.

   (a) Eventuell muss das neues Werkzeug Ergebnis seiner Arbeit anderen Werkzeugen zugänglich machen oder braucht von diesen Daten. Solche Informationen müssen über das Kontextwerkzeug an die entsprechenden Subwerkzeuge weitergeleitet werden. Geeignete Mechanismen dazu sind die Entwurfsmuster *Zuständigkeitskette* und *Beobachter* [GoF95].

   (b) Soll die GUI des Subwerkzeugs in die des Kontextwerkzeugs eingebettet werden, muss in der GUI des Kontextwerkzeugs ein passender `ToolViewSlot` (z.B. eine von der Java Swing-Klasse `JPanel` abgeleitete Klasse) zur Verfügung stehen. Beim Erzeugen des Subwerkzeugs ruft das Kontextwerkzeug die Methode `tryToCreateSubtool()` auf und übergibt ihr das zu erzeugende Werkzeug und den Namen des Slot. Ohne einen gültigen Namen wird das Subwerkzeug nicht eingebettet. Ein Blick auf die Deklaration der Methode `tryToCreateSubtool()` aus der Klasse AbstractTool verdeutlicht dieser zwei Möglichkeiten:

   ```
   protected Tool tryToCreateSubtool (Class toolClass,
     String optionalEmbeddingName, Class optionalMaterialType)
   ```

   (a) Lösche die ins neue Werkzeug kopierten Methoden sowie GUI-Elemente und obsolete Tests des alten Werkzeugs.

   Da alle Tests der verschobenen Funktionalität ebenfalls verschoben wurden, müssen nun die Tests angepasst werden, die die Werkzeug-übergreifende Logik testen.

5. Oberflächentests zu ändern verursacht meist größeren Aufwand als das bei Tests ohne Einbeziehung der GUI ist. Daher werden diese erst nach Durchführung aller Änderungen umgestellt.

Beispiel

1. Die Klasse `ZDLSucheWerkzeug` wird von `AbstractToolMono` abgeleitet und so angepasst, dass sie mit ZDL-Objekten als Material arbeiten kann. Abbildung 4.24 zeigt die Klasse im Überblick und ein Ausschnitt aus dem Quelltext einige der Anpassungen.

ZDLSucheWerkzeug
+ getAspect() + canHandleMaterial() # doEquip() # doUseMaterial() # doUpdateMaterial()

Abbildung 4.24: Das neue, abgespaltene Werkzeug.

```
public Class getAspect() {
  return ZDL.class;
}
public boolean canHandleMaterial(Thing material) {
  return material instanceof ZDL;
}
```

2. Bisher bestand die Benutzeroberfläche der Suchfunktion aus zusammengehörigen Textfeldern. Aus zwei Gründen wird die GUI des neuen Werkzeugs als Dialog gestaltet und nicht in die GUI des Kontextwerkzeugs eingebettet: Zum einen ist der Aufwand dafür höher als für die Implementierung als Dialog, zum anderen gewinnt die Suchfunktion an Klarheit. Schließlich verhindert der modale Dialog, dass unvollständige Suchkriterien eingegeben werden und dabei das andere Werkzeug noch verwendet werden kann. Zur Verdeutlichung der Veränderung zeigt Abbildung 4.25 die GUI vor und nach Ausführung des zweiten Schritts. Das Anbinden der GUI erfolgt in der Schablonen-Methode `doEquip()` des Werkzeugs.

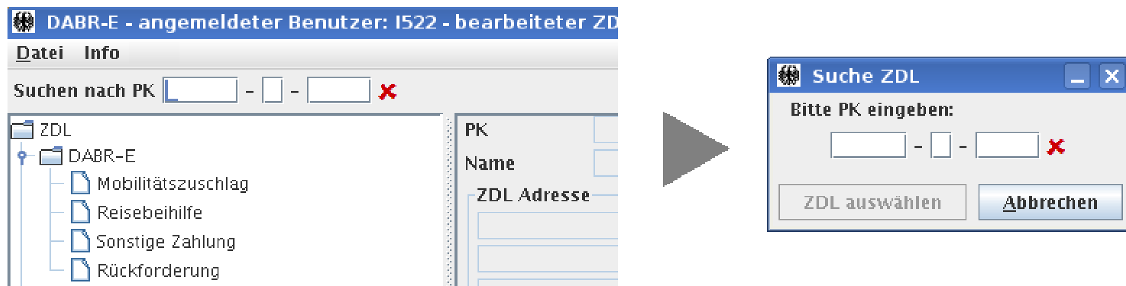

Abbildung 4.25: Die GUI der Suchfunktion vor und nach dem Refactoring.

```
protected void doEquip() {
  _gui = new ZDLSucheWerkzeugGUI();
  _gui.gibAuswaehlenButton().addActionListener();
  _gui.gibAbbrechenButton().addActionListener();
}
private ZDLSucheWerkzeugGUI _gui;
```

3. Nach der Oberfläche wird nun die Fachlogik der Suche in das neue Werkzeug kopiert. Da die eigentliche Suche eine Datenbankabfrage ist, wird sie, wie alle Datenbankzugriffe, von einem Service gekapselt. Der umfangreichere Teil der ins Werkzeug kopierten Logik beschäftigt sich mit der Validierung und Darstellung der eingegebenen Daten.

```
protected void doEquip() {
  ...
  _gui.gibEingabePanel().addKeyListener(new KeyAdapter() {
    public void keyReleased(KeyEvent e) {
      if (_gui.gibEingabePanel().isValidValue()) {
        sucheNachPk(_gui.gibEingabePanel().gibPkDV());
```

```
        }
      }
    }
  }

  private void sucheNachPk(PkDV pk) {
    if (_zdlService.hatZDL(pkDV)) {
      ...
    }
  }
```

4. Das Such-Werkzeug wird an ein bereits bestehendes Kontextwerkzeug, die Klasse `HauptWerkzeug`, angebunden (siehe Abbildung 4.20).

   (a) Mittels einer Zuständigkeitskette, die durch JWAM unterstützt wird, übergibt das Such-Werkzeug das gefundene ZDL-Objekt dem Kontextwerkzeug. Dieses aktualisiert alle anderen aktiven Werkzeuge mit den Daten des Zivildienstleistenden. `ZDLSucheWerkzeug` löst einen *Request* vom Typ `VerwendeZDLRequest` aus, sobald es einen ZDL gefunden hat und der Benutzer auf den „OK"-Knopf klickt. Das ZDL-Objekt wird dem Request als Parameter mitgegeben.

```
protected void doEquip() {
  ...
  _gui.gibAuswaehlenButton().addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      handle(new VerwendeZDLRequest(_this, gibZDL()));
      deactivate();
    }
  });
  ...
}
```

   Beim Erzeugen des Such-Werkzeugs wird die Klasse `HauptWerkzeug` automatisch als dessen Kontextwerkzeug registriert und empfängt daher alle Requests, die von Objekten der Klasse `ZDL-SucheWerkzeug` verschickt werden. Die Methode `doHandle()` dient als Einstiegspunkt für alle eintreffenden Requests. Von dort werden Requests des Typs `VerwendeZDLRequest` an die neue Methode `behandleVerwendeMaterialRequest()` weitergeleitet.

```
private void behandleVerwendeZDLRequest(Request req) {
  VerwendeZDLRequest zdlRequest = (VerwendeZDLRequest) req;
  setMaterial(zdlRequest.gibZDL());    // sich selbst aktualisieren
  _anzeigeWerkzeug.setMaterial(_zdl); // Subwerkzeug aktualisieren
}
```

   Da das Such-Werkzeug nicht eingebettet werden soll, braucht es keinen `ToolViewSlot` in der GUI des Kontextwerkzeugs und kann daher wie folgt erzeugt werden:

```
tryToCreateSubtool(ZDLSucheWerkzeug.class, null, null);
```

   Aufgerufen wird die Methode, wenn der Benutzer auf einen Knopf zum Öffnen des Such-Werkzeugs klickt.

5. Die Tests der Benutzeroberfläche mussten nach Abschluss des Refactorings angepasst werden, weil sich die Eingabefelder in einem anderen Fenster befinden.

Konsequenzen

Da Werkzeugen eine Schnittstelle zum Benutzer haben, also Benutzer direkt mit ihnen arbeiten, bedeutet die Aufteilung eines Werkzeugs meist auch eine Änderung an der Benutzerschnittstelle. Also kann sich die geänderte technische Struktur auch auf die Benutzung des Werkzeugs auswirken. Trotzdem handelt es sich bei *Zerlege Werkzeug in kleinere Werkzeuge* um ein Refactoring gemäß der in Abschnitt 2.2 gebrachten Definition, weil sich die Funktionalität durch das Auslagern nicht ändert.

Zusammenhänge

Handelt es sich bei der zu verschiebenden Funktionalität um mehrere, automatisch hintereinander ausgeführte Arbeitsschritte, ist *Extrahiere Automat* vorzuziehen. Details zur Abwägung der beiden Refactorings sind im Abschnitt 4.3.1 nachzulesen.

Verallgemeinerung

Zwei Verallgemeinerungen des Refactorings sind möglich: Das Abstrahieren von JWAM-spezifischen Teilen und von der Art des Werkzeugs (monolithisch oder komplex). Die Beschreibung der Schritte lässt sich schwerer verallgemeinern als bei den anderen im Katalog beschriebenen Refactorings, weil sie auch Annahmen über die Benutzeroberfläche von Werkzeugen macht. Selbst wenn in einem Projekt Teile der Infrastruktur von JWAM verwendet werden, die Werkzeuge aber anders repräsentiert werden, müssten die Schritte vollkommen neu gesetzt werden.

Mehr Potential zur Wiederverwendung der Beschreibung gibt es beim Einsatz komplexer Werkzeuge in JWAM. In den ersten beiden Schritten muss statt eines monolithischen ein komplexes Werkzeug gebaut werden, die Einbettung funktioniert aber für alle Werkzeuge gleich. Ein Blick ins Rahmenwerk zeigt, dass die benötigten Methoden in der Klasse `AbstractTool` definiert sind, von der die Basisklassen für beide Werkzeugarten erben.

Kapitel 5

Schlusswort

Die in Abschnitt 1.2 präsentierten Fragestellungen bilden den Ausgangspunkt dieser Arbeit. Zum Abschluss werden die gewonnenen Erkenntnisse zusammengefasst und den Fragen gegenübergestellt. Im Lauf der Entstehung des Musterkatalogs kamen weitere Fragen zur Wechselwirkung zwischen WAM-Ansatz und Refactoring auf, die in Abschnitt 5.3 gesammelt wurden. Sie können als Anknüpfungspunkte für weitere wissenschaftliche Betrachtungen von WAM-Ansatz und Refactoring sein.

5.1 Zusammenfassung

Ziel dieser Arbeit war es, die Anwendung von Refactorings in Software-Systemen zu untersuchen, die nach dem WAM-Ansatz entwickelt wurden. Das Hauptaugenmerk lag darauf, WAM-spezifische Refactorings zu finden und so zu beschreiben, dass diese in der Entwicklung gewinnbringend eingesetzt werden können. Nach der Aufarbeitung von Grundlagen wurde eine Musterbeschreibung entwickelt, welche den Einsatz der gefundenen Refactoring-Muster in der Praxis ermöglichen soll. Um die Praxistauglichkeit weiter zu erhöhen, wurde nach Gliederungskriterien für die Muster gesucht. Die Gliederung nach dem Zweck der Refactorings ist wegen ihrer Problemorientierung die beste der diskutierten Möglichkeiten. Der Musterkatalog besteht daher aus den drei Teilen *Modellierung korrigieren*, *Wiederverwendbarkeit erhöhen* und *Komplexität beherrschen*.

Im Musterkatalog finden sich die Antworten auf die grundlegenden Fragestellungen dieser Arbeit. Dass es WAM-spezifische Refactorings gibt, zeigt sich schon durch die Existenz des Katalogs. Eine für die Entwickler nützliche Beschreibung und Gliederung der Refactorings wurde durch die Orientierung an bewährten Musterkatalogen und durch die Erprobung der Refactorings in der Praxis erreicht. Die mögliche Beeinflussung fachlich motivierter Architekturen durch technische Refactorings wurde in der Musterbeschreibung berücksichtigt. Sie beleuchtet unter anderem den Einsatz technischer Refactorings als Alternative zu den jeweiligen WAM-spezifischen Refactorings.

5.2 Bewertung

5.2.1 Nutzen des Musterkatalogs

Bisher wurde der Katalog nur in dem Projekt eingesetzt, aus dem die Muster ursprünglich abstrahiert wurden. Die Hälfte der Muster konnte mehrmals angewendet werden. Dabei traten die in Abschnitt 1.1 beschriebenen potentiellen Vorteile – einfachere, sicherere und effizientere Umsetzung von Refactorings – tatsächlich ein. Meine Erfahrungen mit der Verwendung des Katalogs zeigten, dass durch die in der Musterbeschreibung enthaltene Schrittfolge Refactorings schneller umgesetzt werden konnten, als dies ohne Anleitung der Fall war. Auch unerwünschte Seiteneffekte bei der Durchführung wurden vermieden. Außerdem erleichterte die Kenntnis der Muster die Identifikation möglicher weiterer Refactoring-Potenziale.

Während der letztgenannte Vorteil auch in WAM-Systemen bemerkbar sein sollte, die ohne das JWAM-Rahmenwerk gebaut wurden, ist dies von den anderen Vorteilen nicht im gleichen Ausmaß zu erwarten.

Das liegt an der Bedeutung der Anleitung für die einfache Umsetzbarkeit und an der engen Anknüpfung der Anleitung an JWAM. Zwar wird in der Musterbeschreibung auch auf die Verallgemeinerung der Schrittfolge eingegangen, doch müsste diese ohne den Einsatz von JWAM meist stark angepasst werden.

Mit der Festlegung auf Plattform und Rahmenwerk wurde die allgemeine Anwendbarkeit der Refactorings vermindert. Dadurch verbesserte sich aber die Umsetzbarkeit in einer speziellen Umgebung. Bis jetzt hat JWAM jedoch keine starke Verbreitung in der Praxis gefunden. Das liegt unter anderem daran, dass JWAM zur Konstruktion von Anwendungen mit Java Swing gedacht ist und daher nur in eingeschränktem Maß mit anderen Technologien verwendet werden kann. Daher macht es Sinn, zumindest die Abhängigkeit vom Rahmenwerk aus der Musterbeschreibung zu entfernen.

5.2.2 Grenzen zwischen Refactoring und Erweiterung

Einige WAM-Refactorings dienen zumindest teilweise dazu, unmittelbar nach Abschluss eines Refactorings eine Erweiterung der Funktionalität zu implementieren. Bei einigen ist dies der einzige Grund für das Refactoring (*Ersetze Wert durch Fachwert*, *Ersetze allgemeinen Fachwert durch speziellen Fachwert* und *Lasse Werkzeug mit Aspekt arbeiten*). Das wirft die Frage auf, ob solche Änderungen überhaupt als Refactoring aufgefasst werden sollten. In dieser Arbeit wurden die Änderung der Funktionalität nicht in die Refactoring-Schritte einbezogen, sondern an deren Ende gestellt. Die Schrittfolge wurde entsprechend gewählt. In der Motivation der Refactorings wird ebenfalls auf die Erweiterung eingegangen.

In der Praxis lässt sich die Trennung von Refactoring und Erweiterung nicht immer so einfach umsetzen. Manchmal könnte die Einbeziehung der Erweiterung in die Schrittfolge eine Arbeitsersparnis bringen oder testgetriebene Entwicklung erleichtern.

5.2.3 Technische WAM-Refactorings

Manche WAM-Refactorings betreffen nur die Konstruktion von Elementen der WAM-Modellarchitektur. Ihre Auswirkungen sind im WAM-Modell einer Software nicht zu bemerken. In Abschnitt 2.3 wurde für diese Art von Refactorings der Begriff *technisches WAM-Refactoring* eingeführt. Beispiele für technische WAM-Refactorings sind die Umwandlung von komplexen Werkzeugen in monolithische Werkzeuge sowie die umgekehrte Vorgehensweise (vom monolithischen zum komplexen Werkzeug). Der Katalog beinhaltet nur fachliche WAM-Refactorings, weil sich technische schwerer verallgemeinern lassen. Für fachliche WAM-Refactorings wurde zwar eine plattform- und rahmenwerkspezifische Beschreibung gewählt, doch liegt die Motivation der Refactorings in der WAM-Modellarchitektur begründet. Technische WAM-Refactorings stützen sich hingegen auf die Entwurfsmuster der Modellarchitektur und auf deren plattformspezifische Umsetzung. In [Züllighoven05] werden für einige Entwurfsmuster plattformspezifische Implementierungen diskutiert.

Die Aufarbeitung technischer WAM-Refactorings scheint daher weniger allgemeingültig zu sein als die von fachlichen WAM-Refactorings. Trotzdem kann ein Katalog technischer WAM-Refactorings die Entwicklung von WAM-Systemen vereinfachen und eine dahingehende Beschäftigung mit diesem Thema rechtfertigen.

5.3 Ausblick

5.3.1 Weitere Kategorien und Muster

Die Muster des Katalogs wurden aus Refactorings abgeleitet, die in einem WAM-System vorgenommen wurden. Es ist möglich, dass auch andere als die im Katalog enthaltenen Refactorings der Definition eines WAM-Refactorings entsprechen. Der Katalog kann nicht nur um Muster, sondern auch um Kategorien erweitert werden. Eine solche könnte die Korrektur von Verstößen gegen Architekturregeln sein. Da die Behebung von Architekturverstößen in der Regel die Funktionalität einer Software nicht ändert, aber deren Qualität verbessert, sind solche Maßnahmen Refactorings. Sind die Architekturregeln WAM-spezifisch, handelt es sich um WAM-Refactorings. Eine Erweiterung des Katalogs um Muster dieser Kategorie ist daher denkbar.

5.3.2 WAM-spezifische Smells

Die in Abschnitt 3.2 vorgestellten Maßnahmen zur Identifikation von Refactoring-Potentialen weisen kaum WAM-Spezifika auf. Da es aber WAM-spezifische Refactorings gibt, liegt die Vermutung nahe, dass es auch WAM-spezifische Smells gibt. Diese könnten als Indikatoren für die Notwendigkeit von WAM-Refactorings dienen. Wie schon die WAM-Refactorings würden solche Smells in der WAM-Modellarchitektur begründet liegen. Eine Beschreibung dieser Smells könnte den Einsatz von Refactorings in WAM-Systemen erleichtern.

5.3.3 Automatisierung von WAM-Refactorings

Moderne Entwicklungsumgebungen wie *Eclipse* und *IntelliJ IDEA* unterstützen einen Teil der in [Fowler00] beschriebenen Refactorings. Änderungen im Quelltext können durch solche Automatisierungsmechanismen mit wenigen Benutzereingaben durchgeführt werden. Außerdem wird die Anwendung der Refactorings sicherer, weil eine bestimme Schrittfolge eingehalten wird und damit unerwünschte Nebeneffekte vermieden werden können. Bei manchen Entwicklern sinkt die Akzeptanz von Refactorings, wenn diese nicht durch die Entwicklungsumgebung unterstützt werden ([Lippert05]). Eine Automatisierung von WAM-Refactorings ist daher wünschenswert. Mit dem JWAM-Rahmenwerk steht ein Grundgerüst der WAM-Modellarchitektur als Java-Implementierung zur Verfügung. Wie schon in [Karstens05] könnte JWAM zur maschinellen Identifikation von WAM-Elementen im Quelltext dienen, was eine Voraussetzung für die Automatisierung ist.

Erste Überlegungen in diese Richtung lassen eine Automatisierung von fachlichen WAM-Refactorings als unrealistisch erscheinen. Schon beim Aufstellen von maschinell überprüfbaren Architekturregeln in JWAM (siehe [Karstens05]) kam es zur Problemen, weil nicht alle Konzepte der WAM-Modellarchitektur in Klassen und Interfaces abgebildet sind. Außerdem enthalten viele Refactorings abstrakt formulierte Anweisungen, die Fachwissen und menschliches Denkvermögen verlangen. Ein Beispiel dafür ist der zweite Schritt in *Extrahiere Automat* (siehe 4.3.1): „Kopiere die einzelnen Methoden, die den zu automatisierenden Prozess ausmachen [...] in den Automaten."

5.3.4 Refactorings in Modellarchitekturen

Den Ausgangspunkt dieser Arbeit bilden Refactorings, die in einem Software-Projekt durchgeführt wurden. Eine abstrahierende Sicht auf diese Refactorings zeigte die Zusammenhänge mit der WAM-Modellarchitektur. Verfolgt man diesen Prozess weiter, kommt nach der Abstraktion von Projektspezifika der Schritt hin zur Generalisierung der Modellarchitektur. Modellarchitekturen definieren unter anderem Elemente, Beziehungen zwischen den Elementen und Regeln. Refactorings in Modellarchitekturen stellen strukturelle Veränderungen dieser Artefakte dar. Beispiele für solche Refactorings enthält der Musterkatalog dieser Arbeit. Unter anderem werden Elementtypen ausgetauscht, Aufgaben zwischen Elementen verschoben und Benutzungsbeziehungen entlang von Hierarchien bewegt. Die Beispiele belegen, dass Refactorings in Modellarchitekturen gefunden werden können. Allerdings sind diese eventuell zu abstrakt, um für Entwickler von Nutzen zu sein. Die Kenntnis solcher Refactorings kann aber das Finden von Refactorings in konkreten Modellarchitekturen erleichtert.

Die Beschreibung von Refactorings in Modellarchitekturen wird durch den Abstraktionsgrad verkompliziert, weil oft das Benennen konkreter Schritte schwer fällt. Für diesen Zweck könnten sich die in [Gorts06] beschriebenen *Refactoring Thumbnails* eignen, die eine grafische Beschreibung von Refactorings auf hohem Abstraktionsniveau sind. Das Ziel dieser Beschreibungssprache ist es, die Ideen der Refactorings zu kommunizieren, ohne dabei auf die Details der Umsetzung einzugehen. Eine einfache, an UML-Klassendiagramme erinnernde Notation macht die Thumbnails intuitiv verständlich. Abfolgen von Refactorings und deren Zusammenhänge werden ähnlich wie in [Demeyer03] mit Pfeilen zwischen den Thumbnails dargestellt.

Literaturverzeichnis

[Beck01] Kent Beck, Mike Beedle et al., *Manifesto for Agile Software Development*, http://agilemanifesto.org, 2001 (zuletzt besucht am 16.03.2006)

[Beck04] Kent Beck: *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2004

[Becker-Pechau06] Petra Becker-Pechau, Bettina Karstens, Carola Lilienthal: *Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln*, in: Proceedings Software Engineering 2006, Gesellschaft für Informatik e. V., Bonner Köllen Verlag, 2006

[Demeyer03] Serge Demeyer, Stephane Ducasse, Oscar Nierstrasz: *Object-Oriented Reengineering Patterns*, Morgan Kaufmann Publishers, 2003

[Fowler00] Martin Fowler: *Refactoring: Improving The Design of Existing Code*, Addison-Wesley, 2000

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995

[Gorts06] Sven Gorts: *Refactoring Thumbnails*, http://www.refactoring.be, 2006 (zuletzt besucht am 16.03.2006)

[Hunt99] Andrew Hunt, David Thomas: *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999

[JWAM05] *JWAM Framework Website*, www.jwam.de, 2005 (zuletzt besucht am 16.03.2006)

[Karstens05] Bettina Karstens: *Die Regeln der WAM-Architektur*, Diplomarbeit am Fachbereich Informatik der Universität Hamburg, 2005

[Kerievsky05] Joshua Kerievsky: *Refactoring To Patterns*, Addison-Wesley, 2005

[Lippert05] Martin Lippert, Andreas Havenstein: *Elementares Handwerkszeug: Refactorings in Eclipse schrittweise, effizient und sicher durchführen*, in: Eclipse-Magazin Vol. 4, August 2005, S. 52ff

[Roock04] Stefan Roock, Martin Lippert: *Refactorings in großen Softwareprojekten*, dpunkt.verlag, 2004

[Sotograph06] *Sotograph Website*: www.software-tomography.com, 2006 (zuletzt besucht am 16.03.2006)

[Schulz04] Marco Schulz: *Kleine Refactoring-Muster*, Diplomarbeit am Fachbereich Informatik der Universität Hamburg, 2004

[Wake04] William Wake: *Refactoring Workbook*, Pearson Education Inc., 2004

[Westphal06] Frank Westphal: *Testgetriebene Entwicklung mit JUnit & FIT*, dpunkt.verlag, 2006

[Züllighoven05] Heinz Züllighoven: *Object Oriented Construction Handbook*, dpunkt.verlag, 2005