mehr zum thema:
www.refactoring.com

CODE-RECYCLING: REFAKTORISIERUNG EINER ALT-ANWENDUNG IN DER PRAXIS

Über Softwarequalität entscheidet nicht zuletzt die Struktur der einzelnen Module und Komponenten. Neben der Dokumentation ist das eines der Hauptkriterien, ob sich Softwareentwickler in einem Projekt zurechtfinden. Dieser Beitrag erzählt, wie aus einem Legacy-System eine moderne Software mit wartungsfreundlichem Code wurde, ohne die Anwendung von Grund auf neu zu schreiben.

der autor

Mark Paluch
[E-Mail: mpaluch@paluch.biz]
ist seit 12 Jahren in der Softwareentwicklung mit dem Schwerpunkt „Java Enterprise Applications" tätig.

In vielen Unternehmen sind noch Legacy-Anwendungen im Betrieb. Das sind Systeme aus einer Zeit, in der schnelle Ergebnisse zählten. Oftmals wusste nur der Softwareautor, wie das System funktioniert. Dieser Fakt alleine ist noch keine Motivation für strukturelle Änderungen der Software. Sobald aber größere Anpassungen an der Software erforderlich werden, steigt die Angst vor Risiken. Der ursprüngliche Urheber der Software ist nicht mehr im Unternehmen und neue Anforderungen müssen realisiert werden.

In diese unangenehme Situation geriet der verantwortliche IT-Manager eines unserer Kunden. Er war für eine Reporting- und Antragsplattform eines Fuhrparkmanagement-Systems zuständig. Neben dem Fachbereich meldete nun auch der IT-Betrieb technische Anforderungen, die berücksichtigt werden mussten. Kaum jemand traute sich an den Code oder gar an ein neues Release. Nachdem mehrere neue Releases gescheitert waren, entstand so das Projekt „Wartbarkeit durch Refactoring" (siehe Kasten 1).

> Das in diesem Artikel beschriebene Projekt wurde innerhalb eines Monats abgewickelt. Das Team bestand aus drei Personen die insgesamt neun Personenwochen (Code-Review, Entwicklung, Dokumentation und Qualitätssicherung) beschäftigt waren. Die reine Entwicklungszeit betrug fünf Personenwochen. Ursprünglich wurde das Legacy-System vor sechs Jahren in ca. 14 Personenwochen entwickelt. Vor der Refaktorisierung bestand die Java-Codebasis aus 66 Klassen mit 9.893 ELOC (effektiven Codezeilen). Durch die Refaktorisierung wurde der Code in 169 Klassen mit 17.226 ELOC transformiert.

Kasten 1: *Eckdaten des Projekts „Wartbarkeit durch Refactoring".*

Das Projekt

Nach [Fow99] beschreibt Refaktorisierung strukturelle Verbesserungen an vorhandenem Code, ohne sein Verhalten zu ändern. Das kann sowohl automatisiert durch Tools als auch manuell erfolgen. Im Laufe der Zeit hat sich ein großer Katalog an Refaktorisierungsmaßnahmen (vgl. [Ref]) etabliert. Viele Refaktorisierungen sind heute ein wichtiger Teil des Softwareentwicklungsprozesses und tragen zur stetigen Verbesserung der Softwarestruktur bei.

Für das in diesem Artikel beschriebene Projekt gab es im Vorfeld zwei Optionen:

1. Die bestehende Codebasis wegwerfen und durch eine Neu-Implementierung ersetzen.
2. Das System per Refaktorisierung neu ordnen.

Die erste Variante einer kompletten Neu-Implementierung wäre nur dann sinnvoll gewesen, wenn bereits alle Anforderungen in Dokumentationen manifestiert wären. Zudem stellte für uns der Fremd-Code eine neue und unbekannte Software dar. Es mangelte an fachlicher und technischer Dokumentation. Durch die bereits festgelegte Terminleiste konnte die fachliche Konzeption nicht im Vorfeld durchgeführt werden. Auch war unser Kunde nicht bereit, die zusätzlichen Kosten für ein Requirements-Engineering zu tragen. Die zweite Variante, eine Refaktorisierung der bestehenden Anwendung, wurde durch die äußeren Umstände interessant. Letztendlich fiel die Entscheidung für eine Refaktorisierung.

Die Grundlage des Legacy-Systems bildete eine Java-Entwicklung, bestehend aus Web- und Server-Komponenten. „Struts 1.1" wurde als Aktions- und Präsentations-Framework genutzt. Struts sollte auch weiterhin bestehen bleiben, da die Migration auf ein anderes Framework zu kostenintensiv gewesen wäre. Deshalb wurde folgende Vorgehensweise für das Projekt festgelegt:

- Code-Review
- Erstellung des Soll-Konzeptes
- Absicherung des Codes durch neue Unit-Tests
- Refaktorisierung und Implementierung
- Qualitätssicherung und Dokumentation

Das vorangestellte Code-Review zeigte viele prozedurale Abläufe, die zu einem kaum wartbaren Code führten. Der Software mangelte es an objektorientierten Ansätzen. Im Code fanden sich viele Redundanzen (oftmals durch *Copy&Paste* entstanden) sowie *Magic Values* und hart codierte *Token* – sowohl im Java- als auch im JSP-Code. Diese strukturellen Probleme bildeten einen Alptraum für jeden, der die Software warten sollte. Es war eine Software, deren Entwicklung sich durch viele Anti-Patterns auszeichnete (vgl. [Bro98], siehe Kasten 2).

Die Code-Metriken (siehe Abb. 1) zeigten die größten Aufgabenfelder: 99 Klassen verteilten sich auf 6 Pakete. Die Klassen waren sehr komplex gestaltet (darunter

> *Anti-Patterns* (Anti-Muster) bezeichnen überholte oder schlechte Lösungen für bestimmte Probleme. Durch Anti-Patterns leidet häufig oft die Qualität der Anwendung oder des Quellcodes. Durch eine bewusste Softwareentwicklung mit im Team festgelegten Prinzipien können diese Anti-Patterns vermieden werden. So entsteht *Clean Code* (vgl. [Mar08]): ein Quellcode, der leicht verstanden und somit ideal gewartet werden kann.

Kasten 2: *Anti-Patterns.*

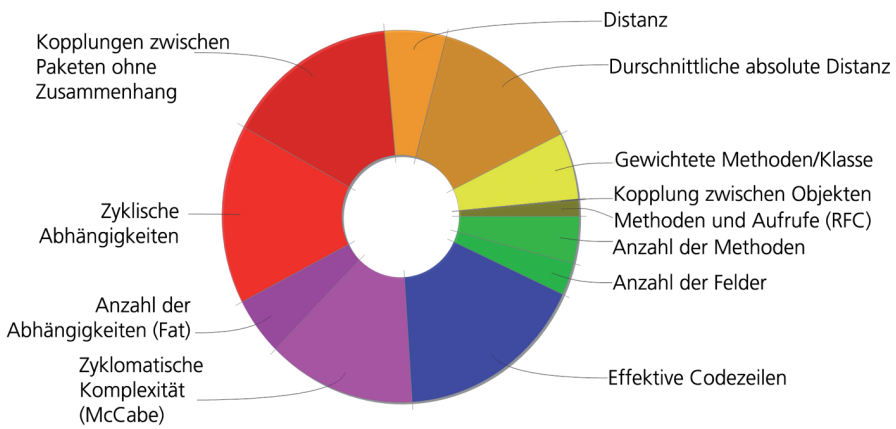

Abb. 1: Darstellung der Codestruktur-Probleme vor der Refaktorisierung.

eine Klasse mit 270 Methoden). Lange Methoden machten 80 % des Codes aus. Zur Vorbereitung wurden zahlreiche Unit-Tests implementiert. Gerade bei bestehenden Anwendungen darf sich die Business-Funktionalität nicht durch technische Verbesserungen ändern. Die Absicherung des Codes fand sowohl auf der Methoden- als auch auf der Web-Ebene statt. Durch das Web-Interface konnten die Tests ungeachtet des zu Grunde liegenden Codes implementiert werden – die Anwendung musste auch nach der Refaktorisierung dieselben Ausgaben liefern (vgl. [Coh04]).

Definition der Maßnahmen

Die hauptsächlichen Schwächen waren nach dem Review identifiziert. Diese sollten mit den folgenden Refaktorisierungen (vgl. [Fow99]) behoben werden:

- Verteilung der Klassen auf neue Pakete
- Änderung von Klassen- und Methodennamen
- Methoden-Extraktion
- Extraktion von Super-Klassen
- Verteilung bestehender Methoden auf neue Klassen
- Entfernung von doppeltem Code
- JSP-Refaktorisierung

Als Entwicklungsumgebung diente Eclipse im aktuellen Release 3.5. Eclipse bietet für Java-Entwickler eine große Sammlung von Refaktorisierungswerkzeugen an. Wollte man zum Beispiel vor einigen Jahren eine Methode einer Klasse umbenennen, war das eine aufwändige Aufgabe: Alle Aufrufe zu dieser Methode mussten manuell gesucht und ersetzt werden. Oftmals blieb das schlechte Gefühl, etwas übersehen zu haben. Diese Aufgabe und viele weitere übernimmt Eclipse mit seinen Refaktorisierungswerkzeugen. Es identifiziert automatisch alle Abhängigkeiten und ändert den Code entsprechend ab. Bei einigen Refaktorisierungen, wie z. B. der Methodenextraktion, funktioniert dies nicht immer; manchmal muss der Code erst von Hand umgestaltet werden, damit die automatischen Refaktorisierungen greifen können.

Gerüstet mit dem Projektplan und den Unit-Tests stand dem Beginn der Programmierung nun nichts mehr im Weg.

Ordnung innerhalb der Module

Die Legacy-Anwendung war mit einem Master-Paket und fünf kleineren Paketen für Hilfsklassen einfach strukturiert. Zur Neustrukturierung sollten neue Pakete erstellt werden. Die bestehenden Java-Klassen konnten danach verschoben und gegebenenfalls umbenannt werden. Die entsprechende Refaktorisierung dafür heißt *Move* oder *Rename*.

Bei dieser Aufgabe war es wichtig, dass nur ein Entwickler die Refaktorisierung durchführte. In unserem Fall konnte das restliche Team an anderen Projekten arbeiten. Der Grund für diese Vereinzelung ist folgender: Sobald zwei Entwickler parallel an einer Klasse arbeiten, führt dies zu automatischen Zusammenführungen in der Quellcode-Verwaltung. Ändert ein Entwickler eine Klasse und ein anderer verschiebt die Klasse, entstehen Konflikte. Wir haben mit der beschriebenen Vorgehensweise diese Konflikte innerhalb der Quellcode-Verwaltung vermieden.

Eclipse hat bei der Reorganisation die meiste Arbeit übernommen. Die integrierte Entwicklungsumgebung (IDE) analysiert bei jedem *Move* die Abhängigkeiten. Alle gefundenen Stellen werden im Anschluss automatisch angepasst. Bei dieser Refaktorisierung wird schnell deutlich, dass die Semantik im explizit manifestierten Code beibehalten wird. Dadurch ist das Risiko von Fehlern an dieser Stelle nicht vorhanden (vgl. [Opd92]). Risiken beim Verschieben oder Umbenennen von Modulen treten immer dort auf, wo dynamische Typpreferenzen vorhanden sind (z. B. Zugriff via *Reflection*). Diese Stellen findet die IDE nicht.

Pro Modul wurde jeweils ein eigenes Projekt angelegt. Durch die Trennung der Module konnten ungewollte Abhängigkeiten direkt im Ansatz erkannt und vermieden werden. Anschließend wurde die neue Paketstruktur erzeugt. Durch *Drag&Drop* verschoben wir die einzelnen Klassen in die zugehörigen Pakete. Diese Aufgabe war schnell bewältigt, schon am Nachmittag rückte das Team zur nächsten Aufgabe an.

Kleinere Verschiebeaktionen können auch durchgeführt werden, wenn andere Entwickler am Projekt beteiligt sind. Jedoch sollten größere Aktionen im Team abgesprochen werden, denn Konflikte mag keiner.

Methoden aus Spaghetti-Code extrahieren

Jeder kennt ihn, niemand mag ihn: unübersichtlichen Code. Seitdem die Legacy-Anwendung im Einsatz war, kamen mehr und mehr Anforderungen auf das Team zu. Durch die fortwährende Weiterentwicklung entstand ein Code, der sich durch eine hohe Komplexität auszeichnete. Heute helfen Initiativen (vgl [Mar08] und [CCD]) und Softwarearchitekten bei der Entwicklung, damit der Code wartbar bleibt. Code-Reviews und häufige Retrospektiven tragen ebenfalls dazu bei, problematischen Code zu minimieren.

Vor uns lagen etliche Klassen mit übergroß geratenen Methoden – darunter 80 %, die nur eine einzige Methode besaßen. Innerhalb dieser Klassen erreichten Methoden teilweise kriminelle Längen von über 800 Codezeilen. Nun war es an der Zeit, den vorhandenen Code aufzuräumen und in übersichtliche Methoden zu verpacken. Dies erfolgte über die Refaktorisierung *Methode extrahieren*.

Die Extraktion einer Methode erstellt eine neue Methode aus einem ausgewählten

Codeblock. Die neue, extrahierte Methode enthält den ausgewählten Code, die vorherige Auswahl wird durch einen Aufruf der neuen Methode ersetzt. Die Umwandlung eines Codefragments in eine eigene Methode schafft die Möglichkeit, den Code schnell und exakt neu zu ordnen. Taucht in der Java-Klasse der gleiche Code mehrmals auf (also Duplikate), kann Eclipse diesen Code mit dem Methodenaufruf ersetzen.

Die bestehenden Codeblöcke, die extrahiert werden sollen, mussten vor Beginn der Refaktorisierung identifiziert werden. Dabei folgten wir folgendem Konzept: Jede Methode hat ihren eigenen Zweck (*Separation of Concerns*, vgl. [Rea89]), der möglichst klein und atomar gehalten werden soll. Alle Methoden sollten auf den gleichen Abstraktionsgrad gebracht werden.

Die Reorganisation der Importe übernahm ebenfalls die IDE. Diese Aktion ist bei der Refaktorisierung häufig erforderlich: Nach dem Verschieben von Codeteilen verbleiben oft ungenutzte Importe in den Klassen. Diese verwaisten Importe können schnell zu langen Kompilierungszeiten führen.

Der Code wurde vor der Änderung technisch geprüft. Wichtig dabei waren Abhängigkeiten zu lokalen Variablen. Im Anschluss an die Analyse wurde der Code so umgestaltet, dass Codesequenzen als eigenständige Methoden extrahiert werden konnten. Die Methoden-Extraktion in Eclipse half uns, benötigte Parameter zu identifizieren. Nach den Änderungen sicherten die Unit-Tests das korrekte und unveränderte Verhalten des Codes ab.

Im unstrukturierten Code tauchten auch Programmteile auf, die mehrere lokale Variablen änderten. In diesen Fällen kann eine Methode nicht ohne Weiteres extrahiert werden – sie muss vorher angepasst werden. Möglichkeiten, diesem Problem zu begegnen, sind entweder eine Neustrukturierung des Codes oder die Konstruktion von *Data Transfer Objects* (vgl. [Fow02]).

Neue Klassen extrahieren

Aus zwei ähnlichen Klassen werden die gemeinsamen Teile in eine dritte Klasse, die Basisklasse, überführt. Anschließend werden die zwei ähnlichen Klassen von der neuen Basisklasse abgeleitet. Auf diesem Weg wird eine Abstraktion geschaffen. Gleichzeitig ist dies das klassische Beispiel (vgl. [Fow99]) für die Refaktorisierung „Basisklasse extrahieren".

In unserem Umfeld hatten wir es vorrangig mit großen Klassen zu tun. Die Methoden-Extraktion hat zwar viele Codesequenzen in

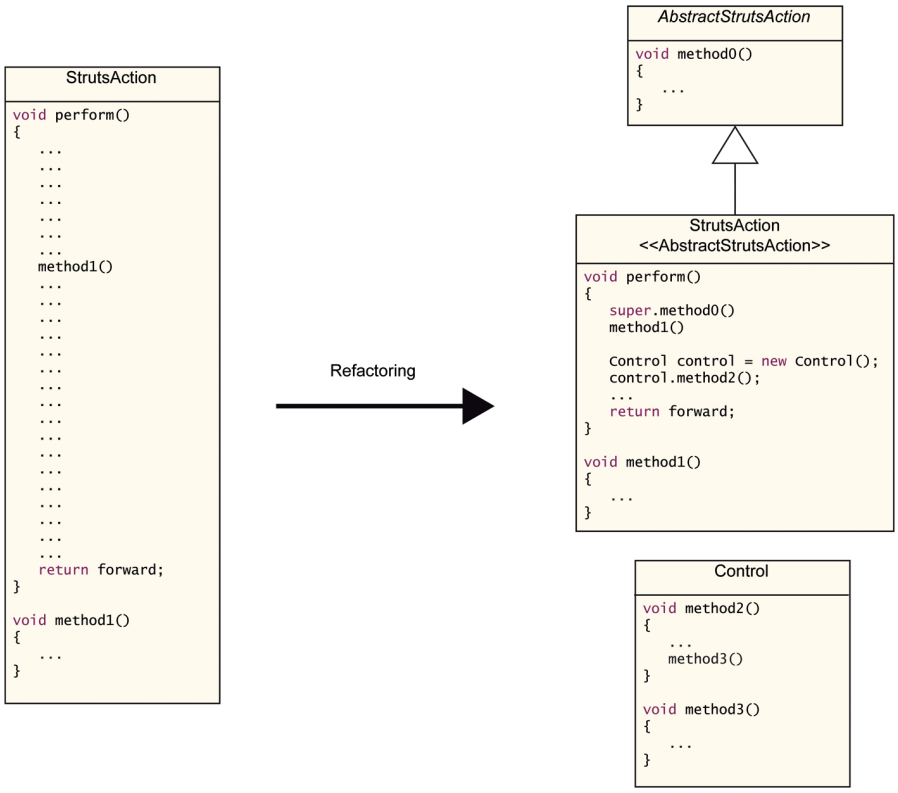

Abb. 2: Transformation der Strukturen.

übersichtliche Methoden verpackt, dadurch wurden die Klassen aber nicht kleiner. Die durchschnittliche Klassengröße (*ELOC* – effektive Codezeilen) betrug vor der Refaktorisierung knapp 180 Zeilen. Nach der Refaktorisierung konnte die durchschnittliche Klassengröße auf 102 Zeilen verringert werden. Das nächste Ziel war es daher, mehrere Abstraktionsstufen einzuführen. Dies wurde über die Extraktion von Basisklassen erreicht (siehe Abb. 2).

Struts Form-Beans

Struts verwendet *Form-Beans* für die Übergabe der Benutzereingaben an die Geschäftslogik. Die *Form-Beans* enthalten sowohl einzelne Eigenschaften (*Member*-Variablen mit *Gettern* und *Settern*) als auch Funktionen zum Zurücksetzen oder Validieren der Daten. Um diesen Aufgabenmix aufzulösen, wurden die Eigenschaften in eine abstrakte Basisklasse extrahiert. Von dieser abstrakten Basisklasse wurde das konkret implementierte *Form-Bean* abgeleitet. Diese Trennung abstrahiert die Datenhaltung von der *Struts Form-Logik*.

> Das Refaktorisierungsspektrum für Klassenhierarchien enthält neben der Extraktion von Basisklassen, Interfaces und Subklassen auch das Verschieben von Feldern, Konstruktoren und Methoden. Diese können in der Hierarchie nach oben (*Pull Up*) in die Basisklasse, ein Interface oder nach unten (*Push Down*) in die abgeleiteten Subklassen verschoben werden. Eine umfangreiche Liste der Refaktorisierungen stellt Martin Fowler unter [Ref] bereit.

Kasten 3: Weitere Refaktorisierungen für Basis- und Subklassen.

Struts-Geschäftslogik

Ein weiterer Bereich für die Extraktion von Basisklassen waren die *Struts Action*-Klassen (also die Logik hinter einem Seitenaufruf). Die fehlende Abstraktion und Wiederverwendung von Funktionalität führte zur redundanten Implementierung von Methoden (z. B. Blätterfunktion (*Paging*) oder Formatierungen). Dies fiel uns besonders bei den Reports auf.

Die Reports boten dem Anwender eine Paging-Funktion. Zusätzlich konnten die Reports die Daten in unterschiedlichen Formaten (z. B. HTML, CSV, Druckansicht) ausgeben. Aus der ersten Report-Aktion wurden die redundant implementierten Methoden in eine abstrakte Basisklasse extrahiert. Anschließend wurden die weiteren Reports manuell angepasst und somit die Coderedundanzen beseitigt.

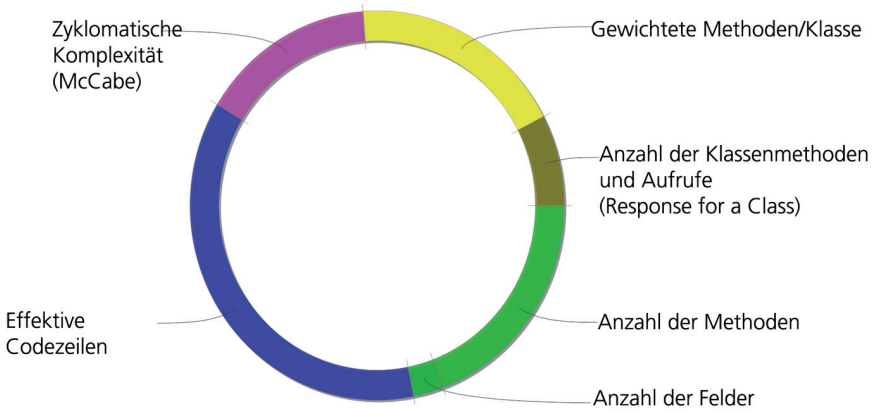

Abb. 3: Darstellung der Codestruktur-Probleme nach der Refaktorisierung.

Von prozedural zu objektorientiert

Jetzt galt es, aus prozeduralem Code echten, objektorientierten Code zu schaffen (vgl. [Mar07]). In unserem Fall lag die Business-Logik in der Präsentationsschicht, den *Struts-Action*-Klassen. Diese wurden in vorhergehenden Schritten bereits in mehrere Klassen und Methoden aufgespalten. Die Organisation in Methoden war die eigentliche Grundlage, um das objektorientierte Konzept durchgängig zu implementieren.

Die Business-Logik sollte auf neue *Application Controller* verteilt werden. Die *Application Controller* waren im zuvor erstellten Konzept (siehe Abb. 3) definiert worden. Die Verteilung der Methoden erfolgte durch Verschieben.

Das Verschieben von Methoden aus einer Quell- in eine Zielklasse gestaltet sich oft als eine einfache Refaktorisierung. Dazu bietet Eclipse zwei Wege: einen manuellen und einen automatisierten. Der manuelle Weg kann über *Drag&Drop* bzw. *Cut&Paste* der Methoden ausgeführt werden. Dabei ist der Entwickler in der Pflicht, denn er muss die Abhängigkeiten beachten.

Der zweite Weg, mit dem man in Eclipse Methoden verschieben kann, ist automatisiert. Eclipse schlägt mögliche Ziele für die Methode vor und ändert alle aufrufenden Stellen. Methoden ohne Abhängigkeiten ihrer Klasse lassen sich mit Eclipse problemlos verschieben. Methoden mit Klassenabhängigkeiten haben unter Umständen Anpassungsbedarf: Um die Abhängigkeiten zu erfüllen, ändert Eclipse die Methoden-Signatur. Die zuvor verwendeten Klassen-*Member* aus der Quellklasse werden über eine referenzierte Instanz verwendet. Die Instanz der Quellklasse wird als Methoden-Parameter zusätzlich übergeben. Bei statischen Aufrufen wird direkt die Quellklasse referenziert statt der Instanz. Diese Vorgehensweise kann ohne eine Nachbearbeitung zu ungewollten Abhängigkeiten führen.

Dieser Schritt war sicherlich der bedeutendste in Richtung einer wartbaren und testbaren Software. Durch die Organisation der Methoden in *Application Controllern* konnten viele Abhängigkeiten auf ein Minimum reduziert werden.

Auflösung von Code-Duplikaten

Coderedundanzen sind ein in Software häufig anzutreffendes Problem. Zwar stellt die redundante Implementierung von Methoden keinen Fehler dar, jedoch wirkt sich doppelter Code negativ auf die Wartbarkeit aus, denn der gleiche Code muss an mehreren Stellen geändert werden. Oft sind *Copy&Paste* sowie Zeitdruck an diesem Anti-Pattern schuld.

Die Redundanzen im Projekt wurden durch ein Code-Review aufgedeckt. Beim Review waren uns redundante Implementierungen aufgefallen, die durch das Kopieren von Klassen entstanden waren. Dabei hat der ursprüngliche Softwareautor bei der Entwicklung vorhandene Klassen als Vorlage für neue Klassen verwendet, manchmal mit dem Effekt, dass im kopierten Code Änderungen durchgeführt wurden.

Rückblickend auf das Projekt war die Auflösung von Duplikaten eine der größten Fehlerfallen. Ohne Codekommentare gibt es kaum eine Möglichkeit zu erfahren, was der Grund für ein Duplikat oder seine Änderung war. Daher mussten abgeänderte Codesequenzen inhaltlich untersucht werden. Erst nach einer solchen Klärung konnten die geänderten Duplikate zusammengeführt werden.

JSP-Refaktorisierung

In der gängigen Literatur zu Refaktorisierungen wird der Fokus vor allem auf den reinen Code gelegt. Es werden *Best Practices* zur Code-Reorganisation angeboten, jedoch werden keine Refaktorisierungsvorgehensweisen zu Meta-Sprachen wie JSP (Java Server Pages) genannt. Für diesen Bereich sind zudem nur wenige Tools verfügbar (z. B. [Jet10], [Sou10]). Die wenigen Refaktorisierungen im Bereich JSP sind hauptsächlich dafür bekannt, dass Java-Code aus den Server-Pages in *Tags* verschoben wird. Das entspricht dem Paradigma der Methodenextraktion. In unserem Kontext haben wir jedoch die nächste Generation der Anti-Pattern vorgefunden: Duplikate statt Templates und feste Pfadangaben.

In unserem Refaktorisierungsprojekt galt es daher, die JSP-Code-Redundanzen zu entfernen und feste Pfadangaben dynamisch zu gestalten. Gerade im Bereich von Software in Unternehmen kommt es häufig vor, dass Systeme auf unterschiedliche Hardware migriert werden. Ein Risikofaktor für Migrationen sind daher Pfade oder *Token* in der Software.

Zur Beseitigung von Redundanzen (z. B. *JSP-Header*-Definitionen) haben sich in unserem Projekt *Include*-Anweisungen bewährt. Diese werden ähnlich wie gemeinsame Methoden referenziert. Auf diese Weise kann der Code zentralisiert werden und es gibt nur eine Stelle für die Wartung.

Den Pfadangaben begegneten wir auf eine andere Weise, nämlich mit *JSP-Tags*. Für die Steuerung der Pfad-Variablen wurden eigene *Tags* entwickelt, die mit Hilfe der Funktionen zum Suchen&Ersetzen der IDE in die einzelnen Seiten eingebunden wurden. Leider gibt es noch keine Werkzeuge, die eine automatisierte Funktion zum Extrahieren von *Tags* anbieten.

Refaktorisierung als Weg zu mehr Wartbarkeit?

Hauptbestandteil dieses Projektes war Refaktorisierung. Aus vorhergehenden Projekten war uns bekannt, dass oft kein Verständnis für präventive Maßnahmen am

Code vorhanden ist. Statt Features wird ohnehin funktionierender Code verändert, eventuell mit der Folge, dass sich neue Fehler einschleichen. Warum soll man also in maroden Code investieren? Die Antwort ist ganz einfach: Weil sich oftmals eine Umstrukturierung rechnet.

Durch die gezielte Refaktorisierung wurde aus einer monolithischen Legacy-Anwendung ein System geschaffen, das mit den neueren Entwicklungstechniken und -standards mithalten kann. Auch beweist [Opd92] in seiner Untersuchung, dass die Refaktorisierung unter gewissen Vorbedingungen das Verhalten der Software beibehält. Daher war das Risiko von Fehlern eher gering. Zudem war die Codebasis durch zahlreiche Tests abgesichert.

Spürbar gesenkt wurde auch die Einarbeitungszeit für das Projekt. Hatte ein neuer Entwickler zuvor über zwei Wochen zur Einarbeitung gebraucht, so ist nach der Refaktorisierung die Einarbeitungszeit um 60 % gesunken. Aus dubiosem Code (mit *Code Smells* behaftet, vgl. [Fow99]) wurde übersichtlicher Code (siehe Abb. 3). Damit hat das Projekt sein Ziel erreicht. Die Wartbarkeit und Erweiterbarkeit wurden wieder auf ein reguläres Niveau gebracht und die Risiken des Legacy-Systems wurden entschärft (vgl. [Bom08]).

Zum Ende des Projekts fand eine erneute Codeanalyse statt. Diese zeigte, dass die Codeprobleme um 87 % verringert wurden – und es gibt sicherlich noch Möglichkeiten zur Optimierung. Und auch der Kunde ist zufrieden: Durch die Vereinfachungen für den Betrieb wurde der Release- und Update-Prozess deutlich vereinfacht. Die Dokumentation profitiert nicht zuletzt davon, dass unser Team die Software verstanden hat.

Fazit

In dem hier vorgestellten Projekt hat sich gezeigt, dass Refaktorisierungen eine sinnvolle Zukunftsinvestition sind. Es ist auch eine Alternative zum Wegwerfen der bestehenden Codebasis, die sich in unserem Fall gerechnet hat. Durch die Refaktorisierung und Untersuchung des Codes konnte das gesamte Team die Software kennenlernen. So stellt die Codebasis eine gut wartbare Software dar.

Die aktuellen Werkzeuge des Eclipse-Projekts haben sich als mächtiges Instrument in der Refaktorisierung erwiesen. Diese Automatisierung führt automatisch zu mehr Produktivität, indem sie den Entwicklungsprozess unterstützt. In einigen Anwendungsfällen liefern die Werkzeuge jedoch nicht zu 100 % das gewünschte Ergebnis. So bleibt ein kleiner Teil immer noch manuell zu erledigen. Auf jeden Fall können Flüchtigkeitsfehler durch die IDE stark eingeschränkt werden.

Das Projekt hat allen Beteiligten gezeigt, dass aus einer Menge unübersichtlichem Code eine moderne Anwendung entstehen kann. Der gezielte Einsatz von Refaktorisierungen macht den Softwarecode verständlicher, leichter zu lesen und bereit für die Zukunft. ■

Literatur & Links

[Bom08] C. Bommer, M. Spindler, V. Barr, Software-Wartung: Grundlagen, Management und Wartungstechniken, dpunkt.verlag, 2008

[Bro98] W.J. Brown, R.C. Malveau, W. Hays, S. McCormick, T.J. Mowbray, T. Hudson (Hrsg.), AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc, 1998

[CCD] Clean Code Developer (CCD), siehe: www.clean-code-developer.de/

[Coh04] F. Cohen, Java Testing, Design, and Automation. A Guide to Maximizing Performance: From Unit Testing to Automated Web Tests, Prentice Hall International, 2004

[Fow02] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman, 2002

[Fow99] M. Fowler, K. Beck, J. Brant, W. Opdyke, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman, 1999

[Jet10] JetBrains, IntelliJDEA, 2010, siehe: www.jetbrains.com/idea/features/jsp_editor.html

[Mar07] R.C. Martin, Working Effectively with Legacy Code, Prentice Hall International, 2007

[Mar08] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall International, 2008

[Opd92] W.F. Opdyke, Refactoring Object-Oriented Frameworks, Doktorarbeit, Universität von Illinois, 1992, siehe http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.688

[Rea89] C. Reade, Elements of Functional Programming, Addison-Wesley Educational Publishers Inc., 1989

[Ref] Refactorings in Alphabetical Order, siehe: www.refactoring.com/catalog/index.html

[Sou10] SourceForge.net, Eclipse JSP Refactoring Tools, 2010, siehe: http://sourceforge.net/projects/jsprtools/