Systematische Integration von Refactoring und Test im Extreme Programming

Stefan Dissmann, Michael Schlüter

Lehrstuhl Software-Technologie
Universität Dortmund
44221 Dortmund
stefan.dissmann@udo.edu
michael.schlueter@udo.edu

Abstract: Refactoring und Testen bilden wesentliche Bausteine des Extreme Programming. In diesem Beitrag wird der konzeptionelle Zusammenhang zwischen beiden Praktiken herausgearbeitet. Insbesondere wird die Abhängigkeit zwischen der Durchführung eines Refactorings und den sich daraus ergebenden notwendigen Änderungen der Testumgebung betrachtet. Durch eine geeignete Typisierung von Refactorings werden diese Änderungen übersichtlich zusammengefasst und beschrieben. Die Beschreibung bildet die Grundlage für die Gestaltung eines systematischen Vorgehens, welches in weiten Teilen durch Werkzeuge unterstützt und damit effizient in der Entwicklung eingesetzt werden kann.

1 Refactoring und Testen in agilen Prozessen

Agile Entwicklungsvorgehen versuchen, Flexibilität während der Entwicklung durch einen Verzicht auf eine aufwändige Vorplanung zu gewinnen [HR02]. Sie können kurz und prägnant beschrieben werden, da wesentliche Teile ihrer Ausgestaltung den ausführenden Entwicklern überlassen werden. In dem hier vorliegenden Beitrag wird am Zusammenwirken zweier Praktiken des Extreme Programming (XP) [Be00] gezeigt, dass für agile Vorgehensweisen trotzdem sehr wohl präzise und damit durch Werkzeuge unterstützbare technische Abläufe formuliert werden können. Generell werden für agile Entwicklungsvorgehen u.a. die Einfachheit der erstellten Software und die Berücksichtigung von sich ständig ändernden Anforderungen gefordert [Ma01]. Die Umsetzung dieser Forderungen führt dazu, dass die entstehende Software während der Entwicklung laufend überarbeitet und umstrukturiert werden muss. Refactoring [Op92, Fo00] bezeichnet das technische Vorgehen, um unter Erhaltung der realisierten Funktionalität qualitativ bessere Softwarestrukturen zu gestalten. Refactoring bildet eine zentrale Technik für agile Vorgehensweisen, insbesondere auch für XP [Be00]. Werden jedoch in XP durch Refactoring Schnittstellen von Klassen oder Methoden verändert, so scheitern anschließend die zuvor im Rahmen der ebenfalls zentralen Praktik Test-First [Be00, Be03] erstellten und eingesetzten Testfälle. Refactoring erfordert daher neben der Änderung der Software in vielen Fällen auch eine Anpassung der zugehörigen Tests.

Refactoring wird immer auf eine bereits existierende Software $\boldsymbol{S}$ angewandt und führt zu einer umstrukturierten Software $\boldsymbol{S}^{ref}$. Als Folge wird in der Regel der zu $\boldsymbol{S}$ gehörende Test $\boldsymbol{T}$ unbrauchbar und muss in einen Test $\boldsymbol{T}^{ref}$ transformiert werden. Mit $\boldsymbol{T}^{ref}$ kann dann $\boldsymbol{S}^{ref}$ überprüft werden. Dieses Anpassen des Tests an die umstrukturierte Software (*Test-Later-Refactoring*) entspricht herkömmlichen Entwicklungsvorgehen.

Für XP stellt Test-Later-Refactoring jedoch ein fragwürdiges Vorgehen dar, da im Sinne des strikten Test-First-Ansatzes zunächst das anzuwendende Refactoring ausgewählt werden muss, ohne aber direkt die Software zu verändern. Statt dessen müssen zunächst die betroffenen Testfälle entsprechend umgestaltet werden [Pi02], die dann im Sinne des Test-First-Ansatzes die Anpassung der Software an die neu spezifizierten Testfälle erzwingen. Bei diesem Vorgehen ist nicht das Refactoring sondern der daraus abgeleitete Test bestimmend für die Umstrukturierung der Software. Es erfolgt also eine Analyse von $\boldsymbol{S}$, in der das geeignete Refactoring ***ref*** ausgewählt wird. Auf der Basis dieser Auswahl wird dann $\boldsymbol{T}$ zu $\boldsymbol{T}^{ref}$ umgestaltet und der Test $\boldsymbol{T}^{ref}$ wird anschließend im Sinne des Test-First-Ansatzes benutzt, um die Änderung von $\boldsymbol{S}$ in die umstrukturierte Software $\boldsymbol{S}^{ref}$ zu erzwingen. Ein solches *Test-First-Refactoring* unterscheidet sich vom Test-Later-Refactoring durch die Reihenfolge der Änderungen an Software und Tests. Test-First-Refactoring setzt allerdings voraus, dass für jedes Refactoring bekannt ist, welche Änderungen es an den vorliegenden Tests $\boldsymbol{T}$ erfordert. Da [Fo00] nur die Änderungen an $\boldsymbol{S}$ angibt, ist der Einsatz von Test-First-Refactoring unmittelbar nicht möglich.

Unberücksichtigt bleibt sowohl beim Test-Later- als auch beim Test-First-Refactoring eine wesentliche Forderung an das Refactoring, die Erhaltung der in $\boldsymbol{S}$ bereits realisierten Funktionalität auch in $\boldsymbol{S}^{ref}$. Die Erfüllung dieser Forderung könnte geprüft werden, wenn der Test $\boldsymbol{T}$ mit zur Prüfung von $\boldsymbol{S}^{ref}$ herangezogen werden könnte. Da aber Änderungen an Schnittstellen vorgenommen werden, kann die Ausführbarkeit von $\boldsymbol{T}$ auf der Basis von $\boldsymbol{S}^{ref}$ nicht garantiert werden. Ein Einbeziehen des Tests $\boldsymbol{T}$ erfordert daher Eingriffe in den Ablauf des Refactorings. Soll $\boldsymbol{T}$ als Referenz für die in $\boldsymbol{S}^{ref}$ zu erhaltende Funktionalität aus $\boldsymbol{S}$ dienen, so darf $\boldsymbol{T}$ nicht geändert werden. Um $\boldsymbol{T}$ unverändert mit $\boldsymbol{S}^{ref}$ auszuführen, muss $\boldsymbol{S}^{ref}$ – zumindest zeitweilig – in einem Zwischenprodukt geeignet umgestaltet werden. Für die Erstellung solcher Zwischenprodukte von $\boldsymbol{S}^{ref}$ reicht es nicht, nur die Auswirkungen eines Refactorings auf $\boldsymbol{S}$ zu kennen. Vielmehr müssen auch hier die notwendigen Änderungen an dem zu $\boldsymbol{S}$ gehörenden Test $\boldsymbol{T}$ mit einbezogen werden.

2 Wirkung von Refactorings auf die Brauchbarkeit von Tests

Eine erste Analyse der Wirkung von Refactorings auf die zugehörigen Tests der umstrukturierten Software bietet [DM02]. Hier erfolgt eine Einteilung der Refactorings aus [Fo00] in fünf Typen. Refactorings eines Typs zeigen bezüglich der Konsequenzen für die betroffenen Tests eine vergleichbare Wirkung. Die in [DM02] vorgenommene Typisierung ist jedoch unzureichend, da für zwanzig Refactorings Auswirkungen nicht angegeben werden können. Zudem beschäftigt sich [DM02] nur mit der Änderungen von Tests und lässt den Aspekt der Erhaltung bereits realisierter Funktionalität unbeachtet. Ausgehend von [DM02] haben wir daher für die Refactorings aus [Fo00] eine andere Typisierung mit sechs Typen entwickelt [Sc03]. Refactorings eines Typs fordern jeweils

eine gleichartige Umgestaltung der Tests T^{ref} und zeigen auch ein gleichartiges Verhalten bei der Nutzung des existierenden Tests T zur Absicherung der im Rahmen des Refactorings notwendigen Funktionserhaltung. Diese Typisierung führt dann im folgenden Abschnitt zu einem Vorgehen, das Refactoring im Einklang mit der Praktik Test-First durchführt und bei dem in größtmöglichem Maße bereits vorhandene Testfälle weitergenutzt werden. Wir unterscheiden die folgenden sechs Typen von Refactorings:

Identisches Refactoring (Typ IR) Ein identisches Refactoring verändert die Schnittstellen der betroffenen Klassen nicht. Dementsprechend kann der existierende Test unverändert übernommen werden, es gilt $T^{ref}=T$. Die Umstrukturierung der Software lässt sich somit für identische Refactorings nicht durch den Test-First-Ansatz erzwingen, sondern muss explizit durchgeführt werden. Dem Typ IR lassen sich 14 Refactorings aus [Fo00] zuordnen, ein Beispiel ist das Refactoring Substitute Algorithm.

Erweiterndes Refactoring (Typ ER) Ein erweiterndes Refactoring fügt neue Bestandteile zu Klassen hinzu, so dass gilt: T^{ref} enthält alle Testfälle aus T. T wird also zu einem Teil des neuen Tests, der Erhalt der durch T gegebenen Funktionalität ergibt sich zwangsläufig. Erweiternde Refactorings lassen sich als reine Test-First-Refactorings realisieren. Diesem Typ können 16 Refactorings aus [Fo00] zugeordnet werden, ein offensichtliches Beispiel ist das Refactoring Extract Method.

Adaptierbares Refactoring (Typ AR) Ein adaptierbares Refactoring ändert zwar die Schnittstellen von Klassen, die ursprünglichen Schnittstellen lassen sich aber in der umstrukturierten Software vorübergehend durch geeignete Adapter erhalten. Statt S^{ref} wird eine Software $S^{ref+adap}$ entwickelt, für die gilt: S^{adap} enthält nur solche Softwareadapter, die benötigt werden, um T auf der Basis von $S^{ref+adap}$ ausführen zu können. Adaptierbare Refactorings können als reine Test-First-Refactorings realisiert werden, wenn T^{ref} und T gemeinsam als spezifizierende Vorgabe verwendet werden. Nachdem T erfolgreich für $S^{ref+adap}$ ausgeführt wurde, wird nachfolgend zunächst $S^{ref+adap}$ auf S^{ref} reduziert und anschließend S^{ref} anhand von T^{ref} getestet. Dem Typ AR lassen sich 29 Refactorings aus [Fo00] zuordnen, ein Beispiel ist das Refactoring Replace Error Code with Exception. Hier fängt ein Adapter S^{adap} die Exception ab und gibt den ersetzten Fehlercode zurück, um den existierenden Test T auch für $S^{ref+adap}$ ausführbar zu machen.

Kürzendes Refactoring (Typ KR) Ein kürzendes Refactoring strukturiert Klassen so um, dass Schnittstellen entfallen. T ist daher nicht länger ausführbar. S^{ref} könnte durch einen Adapter S^{adap} um Platzhalter für die entfallenden Teile erweitert werden. Pragmatischer ist aber die Forderung, dass bei kürzenden Refactorings der Test T^{ref} ausschließlich aus dem Teil der Testfälle von T besteht, mit dem die aus S zu erhaltende Funktionalität spezifiziert wird. Kürzende Refactorings können dann als reine Test-First-Refactorings realisiert werden, wobei T^{ref} als spezifizierende Vorgabe das Streichen der nicht gewünschten Schnittstellen erzwingt. Dem Typ KR lassen sich sechs Refactorings aus [Fo00] zuordnen, ein offensichtliches Beispiel ist das Refactoring Hide Method.

Veränderndes Refactoring (Typ VR) Diese Refactorings lassen eine Ausführung von T mit S^{ref} nicht zu. Zu diesem Typ zählen jedoch nur die drei Refactorings aus [Fo00], die aus objektorientierter Sicht eher fragwürdige öffentliche Attributen betrachten. Die Unterstützung dieses Typs im Rahmen eines Vorgehens kann daher entfallen.

Zusammengsetztes Refactoring (Typ ZR) Dieser Typ umfasst vier Refactorings aus [Fo00], die sich aus mehreren einfachen Refactorings zusammensetzen. Damit kann das Test-First-Vorgehen für diese Refactorings auf diese einfachen Refactorings zurück geführt werden. Auch dieser Typ kann in weiteren Betrachtungen vernachlässigt werden.

3 Test-First-Refactoring im XP

Die Typisierung bildet die Grundlage für ein systematisches Vorgehen für Test-First-Refactoring im XP. Es besteht aus sechs Aktivitäten, deren Folge für die Refactorings der relevanten Typen IR, ER, AR und KR gleich bleibt. Die Aktivitäten werden für die verschiedenen Typen unterschiedlich ausgefüllt, um ihre spezifischen Eigenschaften berücksichtigen zu können. Wir stellen hier das Vorgehen und die zugehörigen Aktivitäten nur sehr knapp in einer Tabelle dar. Die Folge der Aktivitäten ist durch die Zeilenfolge der Tabelle gegeben. Auf Rücksprünge und Zyklen, die beim Misslingen von Aktivitäten notwendig werden, wird verzichtet, da hier prinzipielle Fragestellungen im Vordergrund stehen, die bereits am Beispiel der erfolgreichen Durchführung eines Refactorings deutlich werden.

XP [Be00] fordert von den Entwicklern die fortlaufende Integration von Änderungen in die erstellte Software. Auch kleine Änderungen müssen unmittelbar durch Tests überprüft werden, um ständig eine ausführbare Software vorliegen zu haben. Treten Fehler auf, so ist deren Lokalisierung bei einem kleinschrittigen Vorgehen einfach und schnell möglich. Auch Refactorings müssen im Sinne dieses Vorgehens einzeln durchgeführt und überprüft werden. Die Folge von sechs Aktivitäten muss daher für jedes Refactoring individuell eingehalten werden und erfordert Disziplin auf Seiten der Entwickler.

Aktivität	**Typen**	**Beschreibung**
Analyse der Software	IR, ER, AR, KR	Analysiere S und bestimme das anzuwendende Refactoring ***ref***
Erstellung des Tests		Leite aus ***ref*** die Änderungen an T ab und erstelle T^{ref} als Vorgabe für S^{ref}.
	IR	T^{ref} ist gleich T.
	ER	T^{ref} besteht aus T und zusätzlichen Testfällen.
	AR	T^{ref} ist gegenüber T geändert.
	KR	T^{ref} ist eine Teilmenge der Testfälle von T.
Prüfung der Vorgabe	ER, AR, KR	Versuche, T^{ref} auf der Basis von S auszuführen. T^{ref} muss Fehler aufzeigen, da eine Änderung der Software notwendig ist.
Umstrukturierung der Software	IR	Führe die Änderung von S gemäß ***ref*** durch.
	ER, AR, KR	Ändere S^{ref} derart, dass T^{ref} erfüllt wird.
Durchführung des Tests	IR, ER, AR, KR	Führe T^{ref} auf der Basis von S^{ref} aus. T^{ref} muss fehlerfrei ablaufen.
Erhaltung der Funktionalität	ER	Führe T auf der Basis von S^{ref} aus. T muss fehlerfrei ablaufen.
	AR	Erweitere S^{ref} derart um Softwareadapter zu $S^{ref+adap}$, dass T erfüllt wird. Führe T auf der Basis von $S^{ref+adap}$ aus. T muss fehlerfrei ablaufen. Reduziere anschließend $S^{ref+adap}$ wieder auf S^{ref}.

Die über die Typisierung entwickelte Systematik bietet einen guten Ansatzpunkt für eine gezielte Werkzeugunterstützung des Test-First-Refactorings. Offensichtlich werden für alle Refactorings nur vier zu unterstützende Handlungsmuster benötigt, die durch die

vier Typen IR, ER, AR und KR bestimmt werden. Diese Handlungsmuster können für konkrete Refactorings weiter präzisiert werden. Beispielsweise wird bei Anwendung des adaptierbaren Refactorings Introduce Parameter Object die Paramterliste einer Methode verändert. Für die Prüfung, ob die vor dem Ausführen des Refactorings bestehende Funktionalität erhalten wurde, muss dann ein Adapter erstellt werden, der die frühere Schnittstelle der Methode bereit stellt. Das Einführen dieses Adapters in die Testumgebung kann werkzeugunterstützt erfolgen, das Implementieren des Adapters muss durch den Entwickler realisiert werden. Das abschließende Reduzieren der Testumgebung um den zusätzlich eingefügten Adapter kann als eine an der Syntax orientierte Aktion wiederum werkzeugunterstützt ablaufen. Voraussetzung für eine solche Werkzeugunterstützung ist die gemeinsame Verwaltung von Software und zugehörigen Tests. Experimentelle Ansätze für solche Werkzeuge sind bereits geschaffen worden [MS04].

4 Zusammenfassung

Wir haben den Zusammenhang zwischen Refactoring und Testen im Rahmen des agilen Vorgehens XP untersucht und dabei sechs Typen von Refactorings identifiziert, die sich in der Wirkung auf den zur Software gehörenden Test unterscheiden. Wir haben uns auf die 72 in [Fo00] gleichartig dokumentierten Refactorings beschränkt, es ist aber zu erwarten, dass sich auch andere Refactorings in die Typisierung einordnen lassen. Anschließend haben wir ein Vorgehen aus sechs Aktivitäten für die vier Typen der identischen, erweiternden, adaptierbaren und kürzenden Refactorings entwickelt, das es erlaubt, für diese Refactorings die Praktik Test-First anzuwenden und zugleich durch die vorübergehende Ausführbarkeit der vor der Anwendung eines Refactorings vorliegenden Tests sicher zu stellen, dass die bereits realisierte Funktionalität erhalten bleibt. Dazu folgen die Aktivitäten klar beschriebenen Handlungsanweisungen. Die Routineaufgaben des so normierten Vorgehens lassen sich einfach an Werkzeuge delegieren, so dass durch die hier vorgestellte Systematik sowohl die Qualität als auch die Handhabbarkeit des Entwicklungsvorgehens XP verbessert werden können.

Literaturverzeichnis

[Bec00] Beck, K. Extreme Programming. Addison Wesley, 2000

[Be03] Beck, K. Test-Driven Development. Addison Wesley, 2003

[DM02] Deursen, A. van, L. Moonen. The Video Store Revisited - Thoughts on Refactoring and Testing. in: Proceedings XP2002, 2002.

[Fo00] Fowler, M. Refactoring - Improving the Design of Existing Code. Addison Wesley, 2000

[HR02 Hruschka, P., C. Rupp. Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML. Carl Hanser Verlag, 2002

[Ma01] -. Manifesto for Agile Software Development. 2001. URL 13.5.2004: agilemanifesto.org

[MS04] Müller, D., Schlich, B. Konzeption und Realisierung eines Werkzeugs zur Unterstützung des Test-Driven Developments (Diplomarbeit). Universität Dortmund 2004

[Op92] Opdyke, W. Refactoring Object-Oriented Frameworks. University of Illinois, 1992

[Pi02] Pipka, J.U. Refactoring in a »Test First«-World. in: Proceedings XP2002, 2002.

[Sc03] Schlüter, M. Ein Konzept für die systematische Kombination von Refactoring und Test (Diplomarbeit). Universität Dortmund 2003