Refactorings mit Eclipse schrittweise, effizient und sicher durchführen

Elementares Handwerkzeug

VON MARTIN LIPPERT UND ANDREAS HAVENSTEIN

Seitdem der Smalltalk-Refactoring-Browser die ersten kleinen Refactorings, damals noch für Smalltalk, automatisierte, hat sich eine Menge getan. Heutige Entwicklungsumgebungen unterstützten in der Regel eine ganze Reihe von Refactorings, die die Entwickler mit wenigen Tastendrücken schnell und effizient ausführen können. Das Eclipse-SDK bietet einen sehr umfangreichen Satz solcher automatisierten Refactorings an. Dennoch werden die Möglichkeiten dieser Refactorings nur selten wirklich ausgenutzt. Stattdessen werden viele Refactorings immer noch „per Hand" ausgeführt – obwohl man es sich deutlich einfacher machen könnte. Wie? Das zeigen wir in diesem Artikel.

Refactorings gehören mittlerweile zum elementaren Handwerkszeug eines jeden Entwicklers. Sie dienen dazu, das softwaretechnische Design einer Software zu verbessern, damit neue Anforderungen einfacher implementiert werden können und die Software auch in Zukunft änderbar bleibt. Eine wichtige Eigenschaft von Refactorings ist, dass sie das beobachtbare Verhalten der Software nicht verändern. Damit unterscheiden wir Refactorings klar von neuen Anforderungen, die in ein System integriert werden und somit das beobachtbare Verhalten verändern. Auch in der täglichen Entwicklung trennen wir diese unterschiedlichen Aktivitäten strikt voneinander.

Refactorings können prinzipiell auch manuell durchgeführt werden. Allerdings bergen manuell durchgeführte Refactorings immer die Gefahr, unbeabsichtigt, aber fälschlicherweise das Verhalten der Software zu verändern. Die Gefahr, durch eine Umstrukturierung neue Fehler einzubauen, existiert unbestritten. Um dieses Risiko zu reduzieren, sollten manuelle Refactorings allein mit einer entsprechend umfangreichen automatisierten Unit-Test-Suite in der Hinterhand durchgeführt werden.

Moderne integrierte Entwicklungsumgebungen machen uns den Umgang mit Refactorings deutlich einfacher. Sie sichern uns zu, dass automatisierte Refactorings das Verhalten der Software nicht verändern oder weisen uns ggf. auf mögliche Proble-

me hin. Dazu führen automatisierte Refactorings in der Regel alle nötigen Anpassungen am Sourcecode vollständig durch. Wird beispielsweise ein *Rename* auf einer Methode durchgeführt, werden automatisch alle Aufrufe dieser Methode angepasst.

Während das *Rename* (von Methoden und Klassen) sicherlich zu den prominentesten Beispielen automatisierter Refactorings zählt, existieren in Eclipse eine Vielzahl weiterer automatisierter Refactorings. Im Folgenden werden wir uns ansehen, wie diese Refactorings eingesetzt werden können und welche Möglichkeiten sich hinter diesen Features der IDE verbergen.

Die Klassiker: Move und Rename

Zu den Klassikern unter den automatisierten Refactorings gehören das Verschieben von Klassen und das Umbenennen von Bezeichnern. Sie zählen sicherlich zu den am meisten genutzten automatisierten Refactorings in IDEs.

Mit dem *Move*-Refactoring (ALT + SHIFT + V) können Klassen in ein anderes Package verschoben werden. Dabei werden alle Referenzen auf diese Klasse (typischerweise Imports) automatisch angepasst. Existiert in dem Ziel-Package bereits eine Klasse mit dem gleichen Namen, weist Eclipse den Entwickler darauf hin und fragt, ob die bereits existierende Klasse überschrieben werden soll.

Das *Rename*-Refactoring (ALT + SHIFT + R) lässt sich sowohl auf Klassen als auch auf Methoden, Variablen, Parameter oder Packages anwenden. Auch hierbei werden alle Referenzen auf den entsprechenden Bezeichner automatisch angepasst. Wird eine Klasse umbenannt, wird auch automatisch die entsprechende Source-Datei mit umbenannt. Bei einem Parameter wird der Name des Parameters auch innerhalb des Methodenkommentars (bei Verwendung des @*param*-Tags) angepasst.

Erkennt Eclipse mögliche Konflikte, wird der Benutzer wiederum darauf hingewiesen. Besonders interessant wird dieses Refactoring, wenn eine Methode aus einem Interface umbenannt werden soll. In einem solchen Fall passt Eclipse auch die implementierenden Klassen automatisch an. Auch wenn die Methode in der implementierenden Klasse umbenannt wird, passt Eclipse automatisch das entsprechende Interface an, weist den Entwickler vorher jedoch auf diese Änderung hin. Hier ist aber Vorsicht geboten: Implementiert eine Klasse mehrere Interfaces und besitzen diese Interfaces gleichnamige Methoden, verändert Eclipse den Namen der Methode in allen implementierten Interfaces. Würde Eclipse die Methode nur in einem Interface und der implementierenden Klasse verändern, hätte das Refactoring ein Problem verursacht, weil die Klasse dann die alte Methode nicht mehr implementiert, diese aber von einem anderen Interface noch erwartet würde.

Eine Sonderrolle spielen Sourcecodes, die nicht in Java implementiert sind. Klassennamen beispielsweise werden auch häufig in *plugin.xml*-Dateien von Plug-in-Projekten oder JSP-Dateien innerhalb von Webanwendungen benutzt. Eclipse ist in der Lage, beim Umbenennen einer Klasse auch diese Source-Dateien mit zu berücksichtigen. Dazu muss der Entwickler beim Aufruf des *Rename*-Refactorings im erscheinenden Dialog angeben, in welchen Dateien dieses Refactoring durchgeführt werden soll (beispielsweise **.xml* für alle XML-Dateien im Workspace). Außerdem muss die Klasse in den entsprechenden Dateien voll qualifiziert genannt sein (das bedeutet, mit dem kompletten Package-Namen). Mehr dazu siehe im nebenstehenden Kasten.

Kleine Helferlein für lokale Variablen

Obwohl lokale Variablen nur einen sehr beschränkten Gültigkeitsbereich genießen (wie der Name ja auch schon sagt), existieren auch für sie eine Reihe von aus unserer Sicht äußerst nützlicher Refactorings. Allen voran *Extract Local Variable* (ALT + SHIFT + L). Mit diesem Refactoring lassen sich auf Knopfdruck Ausdrücke in lokale Variablen umwandeln. Das Besondere bei diesem Refactoring ist, abgesehen davon, dass es sich sehr einfach und schnell mit der Tastatur ausführen lässt, dass es nicht nur den selektierten Ausdruck in eine lokale Variable umwandelt, sondern auch alle weiteren Duplikate des Ausrucks im aktuellen Sourcecode-Block durch die lokale Variable ersetzt.

Sollte man feststellen, dass eine lokale Variable falsch benannt ist, lässt sie sich natürlich mit einem *Rename*-Refac-

Refactorings für Java- und Nicht-Java-Quellen

In Entwicklungsprojekten werden immer häufiger Java-Quellen gemeinsam mit Nicht-Java-Quellen verwendet, die aber auf Elemente aus den Java-Quellen referenzieren. Beispiele hierfür sind: Bei der Entwicklung von Plug-ins für Eclipse werden in den *plugin.xml*-Dateien häufig Klassen referenziert (für Extensions und die Plug-in-Klasse beispielsweise). Ähnliches passiert bei der Entwicklung von Webanwendungen mit JavaServer Pages. Auch hier werden häufig Java-Klassen innerhalb der JSP-Dateien referenziert.
Das Standard-Rename-Refactoring von Eclipse kann allerdings nur die Typinformationen des Java-Codes „verstehen“ und darauf aufbauend die Umbenennung eines Bezeichners in referenzierenden Quellen anpassen. Nicht-Java-Quellen „versteht“ das Java-Tooling nicht und kann deshalb auch nicht entscheiden, ob ein Textausschnitt aus einer Nicht-Java-Quelle auf den umzubenennenden Bezeichner verweist.

Abhilfe bei voll qualifizierten Klassennamen

Werden in Nicht-Java-Quellen ausschließlich Klassennamen voll qualifiziert angegeben, kann der Refactoring-Support des Java-Tooling diese Klassennamen auch in Nicht-Java-Quellen aufspüren und anpassen. Das funktioniert beispielsweise gut für die *plugin.xml*-Dateien bei der Plug-in-Entwicklung. Bei Methodennamen und Import-ähnlichen Konstrukten, bei denen Klassennamen dann nicht mehr voll qualifiziert verwendet werden können, funktioniert dies allerdings nicht mehr.

Refactoring Participants

Da das Java-Tooling prinzipiell nur mit Java-Quellen umgehen kann, stellt sich die Frage, wie das Java-Tooling bei Refactoring-Operationen auch Quellen anpassen kann, die nicht der Java-Syntax entsprechen. Um dieses Problem zu lösen, ist die Refactoring-Funktionalität von Eclipse erweiterbar gestaltet worden. Seit der Version 3.0 können Plug-ins die Refactoring-Funktionalität erweitern. Eine solche Erweiterung könnte beispielsweise die Syntax von JSP-Dateien kennen und dafür sorgen, dass bei einem *Rename*-Refactoring aus dem Java-Tooling heraus auch JSP-Dateien entsprechend angepasst werden. Damit steht für Plug-in-Entwickler die Möglichkeit offen, ihre Quellen auch an den normalen Java-Refactorings partizipieren zu lassen.

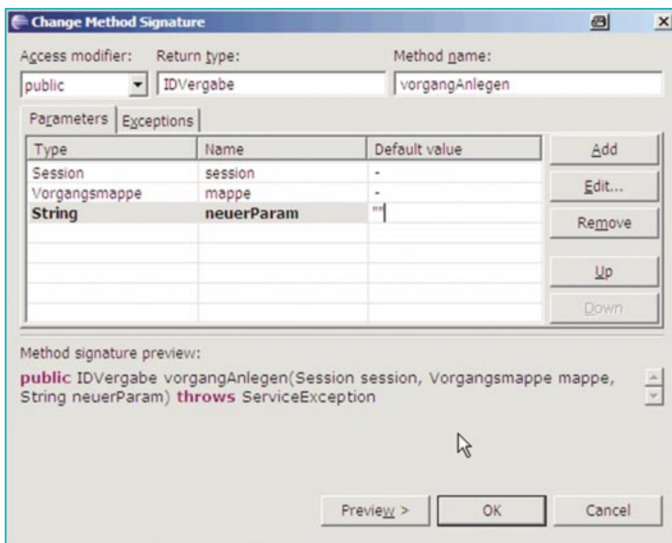

Abb. 1: Extrem mächtig: der Change-Method-Signature-Dialog

toring umbenennen. Schneller und visuell ansprechender ist es bei Bezeichnern, deren Gültigkeit auf das aktuelle Source-File beschränkt ist, den Bezeichner mit einem *Rename-In-File* umzubenennen (CTRL + 2 + R). Aber Vorsicht: Hierbei handelt es sich nicht um ein echtes Refactoring. Eclipse prüft hier nicht, ob der Bezeichner tatsächlich nur lokal sichtbar ist, sondern benennt den Bezeichner einfach nur innerhalb der aktuellen Datei um.

Besonders einfach und nützlich ist zudem das Refactoring *Convert Local Variable into Field* (ALT + SHIFT + F), mit dem sich eine beliebige lokale Variable zu einem Attribut der Klasse umwandeln lässt. Auch hierbei sorgt der Refactoring-Support von Eclipse dafür, den Entwickler darauf hinzuweisen, wenn es zu möglichen Konflikten mit bereits existierenden Attributen der Klasse kommt.

Extrem mächtig: Methoden-Signaturen verändern

Ein sehr mächtiges und aus unserer Sicht oft unterschätztes Refactoring in Eclipse ist das *Change-Method-Signature*-Refactoring (ALT + SHIFT + C). Damit lassen sich nahezu alle Eigenschaften einer Methodensignatur ändern: Die Sichtbarkeit, der Rückgabetyp, der Name der Methode und auch alle Parameter und Exceptions. Die Sichtbarkeit kann natürlich nur verändert werden, wenn es sich bei der Methode nicht um eine Methode eines Interface handelt.

Für das Refactoring spielt es keine Rolle, ob die Methode in dem Interface oder einer implementierenden Klasse ausgewählt wurde. Das Umbenennen der Methode sowie einzelner Parameter funktioniert bei diesem Refactoring analog zu dem *Rename*-Refactoring. Es werden bei Parametern also auch wieder entsprechende Methoden-Parameter-Kommentare mit angepasst.

Am besten kann man dieses Refactoring jedoch einsetzen, wenn ein oder mehrere Parameter zu einer Methode hinzukommen, Parameter wegfallen oder sich ihre Reihenfolge beim Methodenaufruf verändern soll. Alle drei Varianten lassen sich in dem entsprechenden Dialog des Refactorings (Abb. 1) einstellen. Führt Eclipse das Refactoring aus, werden wiederum alle Referenzen auf diese Methode automatisch angepasst. Parameter werden ggf. weggelassen oder in der neuen definierten Aufrufreihenfolge übergeben. Da Eclipse bei neuen Parametern natürlich nicht wissen kann, welcher Wert bei einem Aufruf der Methode übergeben werden soll, kann der Entwickler im Refactoring-Dialog für diesen Parameter einen Default-Wert einstellen. Dieser Default-Wert wird dann überall dort, wo die Methode gerufen wird, eingesetzt.

Auch wenn dies auf den ersten Blick nicht besonders spektakulär klingt, so hat sich dieses Refactoring in der Praxis als sehr nützlich erwiesen. Nahezu alle Änderungen an Methodensignaturen können mit diesem Refactoring soweit automatisiert werden, dass das System sofort wieder lauffähig ist. Eine Eigenschaft, die wir in Projekten sehr schätzen.

Nützliche Inlines

Denkt man über typische Code-Smells nach, fällt einem sehr schnell duplizierter Code ein. Ein Klassiker unter dem Code-Smells. Vielleicht schrecken gerade deshalb viele Entwickler vor Inline-Refactorings zurück, lassen sie doch geradezu vermuten, dass durch sie Code-Duplizierung entsteht.

Grundsätzlich stimmen wir diesen Bedenken natürlich zu. Häufiges Inlining kann schnell zu dupliziertem Code führen – gerade, wenn dies beispielsweise nur zur Performance-Optimierung genutzt wird. Diese Art von Inlining kann man getrost der Java Virtual Machine überlassen, die dafür sehr effiziente Mechanismen mitbringt.

Allerdings erweisen sich einige Inline-Refactorings in bestimmten Situationen als äußerst hilfreich. Wollen wir beispielsweise Verweise auf eine als deprecated markierte Methoden entfernen, können wir das *Inline-Method*-Refactoring (ALT + SHIFT + I) sehr gewinnbringend einsetzen. Dazu implementieren wir die als deprecated markierte Methode so, wie es der referenzierende Code in Zukunft tun sollte (beispielsweise indem eine andere Methode mit einer anderen Signatur gerufen wird und ggf. Werte konvertiert werden). Im zweiten Schritt führen wir ein *Inline-Method*-Refactoring auf der als deprecated markierten Methode aus. Alle Referenzen auf diese Methode werden nun ersetzt durch den Code, den wir übergangsweise in diese Methode implementiert haben. Dadurch werden alle Aufrufe der als deprecated markierten Methode vollkommen automatisch auf eine neue Methode umgestellt.

Ähnlich erleichtern können wir uns die Umstellung von Konstruktoren. Soll beispielsweise der Aufruf eines Konstruktors komplett aus dem Code verschwinden und durch den Aufruf eines anderen, bereits existierenden Konstruktors ersetzt werden, können wir uns wieder des *Inline-Method*-Refactorings bedienen. Allerdings ist das *Inline-Method*-Refactoring nicht per se auf Konstruktoren aus-

führbar. Wir müssen uns also etwas Besonderes einfallen lassen.

Wir bedienen uns dazu eines Tricks, der bei etwas komplizierteren Refactorings gerne angewandt wird: Wir kombinieren eine Reihe von kleinen Refactorings zu einem größeren. Um den Konstruktur einem Inline zu unterziehen, führen wir zunächst das *Introduce-Factory*-Refactoring auf dem Konstruktor aus. Als Ergebnis generiert uns Eclipse eine statische *Factory*-Methode, deren Implementierung lediglich den alten Konstruktor aufruft, der dann seinerseits als *private* deklariert wird. Die *Factory*-Methode dient uns nur als Zwischenschritt, wir werden sie zu einem späteren Zeitpunkt wieder entfernen.

Nun verändern wir die Implementierung der *Factory*-Methode so, dass sie nicht mehr den alten Konstruktur nutzt, sondern direkt den Code des alten Konstruktors enthält. Dies sollte in der Regel nur ein *this*-Aufruf eines neueren Konstruktors sein. Jetzt sollten wir in der Lage sein, den alten Konstruktor zu entfernen, da er im gesamten System nicht mehr verwendet wird. Eclipse hat ja im *Introduce-Factory*-Refactoring alle direkten Aufrufe des Konstruktors durch Aufrufe der *Factory*-Methode ersetzt.

Im letzten Schritt wenden wir das *Inline-Method*-Refactoring auf der *Factory*-Methode an. Als Ergebnis werden alle Aufrufe der *Factory*-Methode durch den Code ersetzt, der in der *Factory*-Methode stand – in der Regel nur der Aufruf des neuen Konstruktors. Und schon haben wir in drei kurzen Schritten ein Inline auf einem Konstruktor durchgeführt.

Interfaces sind immer gut

Eine Aufgabe, mit der wir uns ebenfalls häufiger konfrontiert sehen, ist es, ein Interface aus einer Klasse herauszulösen und an möglichst vielen Stellen im Code nur noch das Interface anstelle der Klasse zu verwenden. Während diese Aufgabe manuell unter Umständen nur sehr mühsam zu erledigen ist, hilft uns der Refactoring-Support von Eclipse auch in dieser Situation. Das Zauberwort heißt: *Extract-Interface*.

Dazu selektieren wir die Klasse, von der ausgehend wir ein neues Interface erzeugen möchten und wählen das *Extract-Interface*-Refactoring aus. In dem entsprechenden Refactoring-Dialog können wir sowohl den Namen des Interfaces definieren als auch angeben, welche Methoden der Klasse in das Interface übernommen werden sollen. In diesem Dialog können wir Eclipse auch sagen, dass es alle Referenzen auf die Klasse durch Referenzen auf den neuen Interface-Typ ersetzen soll, sofern dies möglich ist. Eine Änderung, die uns manuell unter Umständen sehr viel Zeit kosten würde.

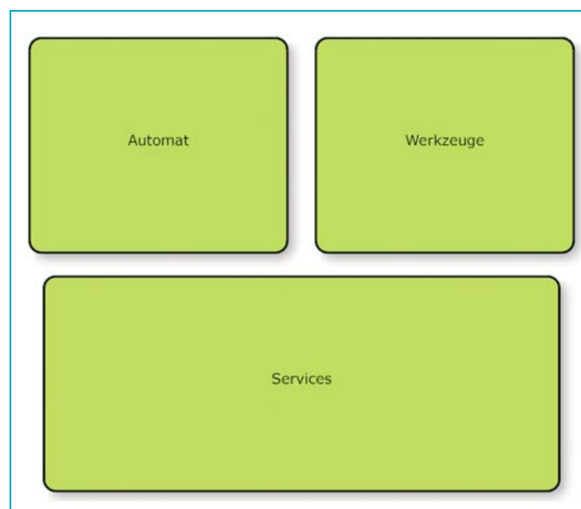

Abb. 2: XRadar-generierte Schichtenansicht

Java 5-Refactorings

Seit mit der Version 3.1 das Java-Tooling von Eclipse Java 5 unterstützt, erwartet man natürlich, dass auch alle bisher implementierten Refactorings mit den neuen Features von Java 5 umgehen können. In der Tat ist es ohne Probleme möglich, Typparameter umzubenennen und Ähnliches. Allerdings sind in Eclipse 3.1 auch Java 5-spezifische Refactorings hinzugekommen. Unter ihnen ist das *Infer-Generic-Type-Arguments*-Refactoring aus unserer Sicht besonders interessant. Dieses Refactoring kann sehr einfach genutzt werden, um alten Java-Code auf die Generics von Java 5 umzustellen. Besonders hilfreich ist dies bei den Collection Libraries. Betrachten wir uns ein kurzes Beispiel: Der folgende Code könnte so oder so ähnlich mit Java 1.4 implementiert worden sein:

```
List list = new ArrayList();
list.add("Hallo");
list.add("Foo");
```

Mit Java 5 könnten wir den Typ des Bezeichners *list* mittels der neuen Typparameter von List und ArrayList auf String einschränken. Dazu nutzen wir das *Infer-Generic-Type-Arguments*-Refactoring. Als Ergebnis setzt Eclipse automatisch den entsprechenden Typparameter ein:

```
List<String> list = new ArrayList<String>();
list.add("Hallo");
list.add("Foo");
```

Ein sehr nützliches Refactoring, wenn man vorhandenen Code auf die Nutzung von Generics umstellen möchte.

Das große Aufräumen beginnt: Move Revisited

Zum Abschluss kommen wir noch einmal auf das Refactoring zurück, mit dem wir diesen Artikel begonnen haben: *Move*. So einfach das Verschieben von Klassen mit diesem Refactoring auch geworden ist, die Entscheidung, in welches Package die Klasse verschoben wird, bleibt bei uns. Und diese Entscheidung kann in größeren Systemen keineswegs trivial zu treffen sein. Meist existieren architekturelle Vorstellungen von Schichten oder Komponenten, die ihrerseits mehrere Packages zusammenfassen und untereinander nur bestimmte Zugriffe erlauben sollen. Die Sichtbarkeiten von Packages untereinander ist in Java allerdings nicht definierbar. So können Klassen, die in Packages unterer Schichten angeordnet sind, ungehindert auf Klassen in Packages oberer Schichten zugreifen, ohne dass wir es bemerken.

Eine gute Unterstützung für die Erkennung und Beseitigung derartiger Architekturverletzungen bietet beispielsweise das XRadar-Framework [1]. XRadar generiert neben vielen anderen hilfreichen Metriken eine Visualisierung der Schichtenarchitektur mit den darin enthaltenen Modulen. Auf Architekturverletzungen wird über farbliche Markierungen hingewiesen. Die Erzeugung der XRadar-Analysedaten wird über Ant gesteuert und lässt sich somit nahtlos in Eclipse integrieren. Wir nutzen dieses Tool gerne, um uns Feedback über die tatsächlich existierenden Abhängigkeiten liefern zu lassen, und lassen uns davon wichtige Impulse liefern, ob Klassen ggf. verschoben werden müssen.

Zur Verdeutlichung verwenden wir im Folgenden ein kleines Beispielsystem, das aus zwei Schichten mit insgesamt drei Modulen besteht (Abb. 2). Die untere Serviceschicht stellt Basisdienste zur Verfügung. Die obere Schicht enthält die Module *Automat* und *Werkzeug*. Die obere Schicht nutzt die Klassen der unteren Schicht, umgekehrt darf die untere Schicht aber keine Klassen der oberen Schicht referenzieren.

Die Schichten und Module der Architektur werden über eine einfach strukturierte XML-Datei definiert (Listing 1). Der Ausschnitt der XRadar-Definitionsdatei verdeutlicht, wie einfach die Architektur für XRadar beschrieben werden kann: Ein *<subsystem>*-Element beschreibt jeweils ein einzelnes Modul:

- Das *level*-Attribut gibt numerisch die Schicht an, in der das Modul angeordnet ist.
- Im *<included-packages>*-Element werden die konkreten Java-Packages benannt, die zu dem Modul gehören.
- In den *<legal-subordinates>*-Elementen werden die erlaubten Beziehungen zu anderen Modulen beschrieben. Im Beispiel dürfen die Automaten- und Werkzeugklassen auf die Services-Klassen zugreifen, die Klassen im Services-Modul dürfen aber keine Klassen anderer Module referenzieren.

Listing 1

```
<radar-config>
 <subsystems>
  <subsystem id="Automat" level="1">
   ...
   <included-packages>
    <package-root value="myproject.automat"
                                      recurse="true"/>
   </included-packages>
   <legal-subordinates>
    <subsystem id="Services"/> ...
   </legal-subordinates>
  </subsystem>

  <subsystem id="Werkzeuge" level="1">
   ...
   <included-packages>
    <package-root value="myproject.werkzeug"
                                      recurse="true"/>
   </included-packages>
   <legal-subordinates>
     <subsystem id="Services"/> ...
   </legal-subordinates>
  </subsystem>

  <subsystem id="Services" level="2">
   ...
   <included-packages>
    <package-root value="myproject.service"
                                      recurse="true"/>
   </included-packages>
   ...
  </subsystem>
 </subsystems>
</radar-config>
```

Per Ant-Target-Aufruf wird die Visualisierung der Architektur von XRadar generiert. Als Beispiel würde während der Entwicklung vielleicht eine Darstellung wie in Abbildung 3 generiert werden.

Ist ein Modul rot markiert, so sind in den Klassen des Moduls unerlaubte Beziehungen enthalten. Fährt man mit der Maus über das Modul, so wird erkennbar, wie viele illegale Beziehungen es zu anderen Modulen enthält. Im Beispiel ist sichtbar, dass in dem Services-Modul ein nicht erlaubter Bezug auf eine Klasse des Werkzeug-Moduls existiert. Die so von XRadar generierten Grafiken und Statistiken weisen also auf Stellen im System hin, die einem Refactoring unterzogen werden sollten.

Die zur Behebung der Architekturverletzung notwendigen Refactorings lassen sich mit Eclipse bequem durchführen. Die Klassen, die von der Services-

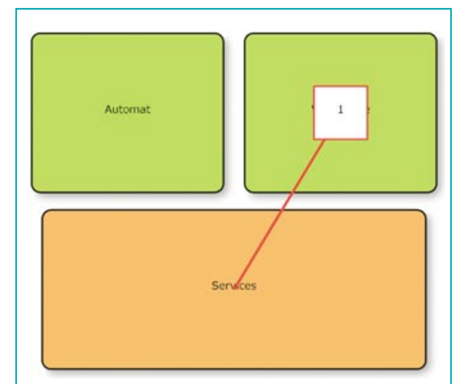

Abb. 3: XRadar-Darstellung einer Architekturverletzung

Schicht im Werkzeuge-Modul referenziert werden, könnten beispielsweise über einfaches Drag-and-Drop im Eclipse-Package-Explorer in das Services-Modul verschoben werden. Diese Art von *Move*- Refactorings lässt sich in Eclipse komfortabel mit der Maus durchführen, ohne auch nur einmal die Tastatur bemühen zu müssen.

Generiert man über das XRadar-Ant-Target nach diesen *Move*-Refactorings wieder die XRadar-Reports, so erhält man unmittelbar ein visuelles Feedback, ob die Refactorings eine saubere Architektur als Ergebnis haben. Sind die XRadar-Module wieder grün eingefärbt, wie in Abbildung 1, so wurde das Ziel des Refactorings erreicht.

Fazit

Wir haben hier längst nicht alle automatisierten Refactorings des Java-Toolings von Eclipse vorgestellt, sondern uns bewusst auf einige beschränkt, die sich in unserer Praxis täglich bewährt haben. Und wir können nur eins sagen: Ohne diese automatisierten Refactorings möchten wir nicht mehr an Java-Quelltexten arbeiten.

Martin Lippert und **Andreas Havenstein** arbeiten als Berater, Coaches und Entwickler bei it-agile. Ihr Schwerpunkt liegt in der Softwareentwicklung mit agilen Methoden und Techniken. Sie sind zu erreichen unter: {martin.lippert, andreas.havenstein}@it-agile.de.

Links & Literatur

[1] XRadar: xradar.sourceforge.net