Vermeidung der Abhängigkeitsdivergenz zwischen Design und Implementierung in Java

Jens Cornelis, Klaus Dorer
Fakultät Elektrotechnik und Informationstechnik
Hochschule Offenburg
Badstr. 24
77652 Offenburg, Germany
mail@jenscornelis.de, klaus.dorer@fh-offenburg.de

Kurzbeschreibung: Die Einhaltung der innerhalb der Designphase festgelegten Architektur eines Softwareprojektes muss während der Entwicklungsphase sichergestellt werden. Dieses Papier beschreibt eine Erweiterung des Eclipse-Plugins JDepend4Eclipse, die die Verwaltung von Regelsätzen erlaubt und die Prüfung auf in einem Projekt vorhandene, unerlaubte Abhängigkeiten auf Knopfdruck innerhalb der Entwicklungsumgebung vornimmt. Die Erweiterung des Plugins wird bereits erfolgreich in internen Projekten der Hochschule Offenburg eingesetzt und soll demnächst öffentlich verfügbar sein.

1. Einleitung

Bei der Definition der Architektur eines Software Systems werden sowohl die Komponenten, aus denen das System bestehen soll, als auch deren Abhängigkeiten festgelegt. Diese werden üblicherweise in einem UML Komponentendiagramm oder Paketdiagramm dargestellt. Die in diesen Diagrammen definierten Abhängigkeiten sind explizit als erlaubt anzusehen. Andere Abhängigkeiten darf es nicht geben, um möglichst loose Kopplung zu erreichen. Diese sind daher als unerlaubt zu betrachten, auch wenn diese unerlaubten Abhängigkeiten nicht explizit in den Diagrammen angegeben sind.

Das in Abbildung 1 gezeigte Model-View-Controller Pattern legt beispielsweise die erlaubten Abhängigkeiten zwischen der graphischen Benutzeroberfläche (View), der Programmlogik (Controller) und dem zugrunde liegenden Datenmodell (Model) fest.

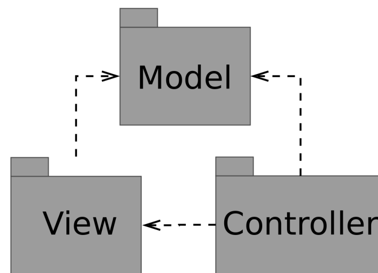

Abbildung 1. Paketdiagramm für die Abhängigkeiten des MVC Patterns

Um zu einem späteren Zeitpunkt aus einer Desktopanwendung zum Beispiel eine Webanwendung machen zu können, soll das Datenmodell keine Abhängigkeiten zu Benutzeroberfläche oder Programmlogik aufweisen. Dadurch können die Klassen für die graphische Benutzeroberfläche leicht ersetzt oder verändert werden, ohne dass dies Auswirkungen auf das Datenmodell des Programmes hat.

Eine Abhängigkeit des Modells zum View ist daher im Paketdiagramm nicht vorgesehen und ist als unerlaubt anzusehen. Eine solche unerlaubte Abhängigkeit muss bei der späteren Implementierung verhindert werden, da ansonsten der ursprüngliche Designansatz und die angestrebte Entkopplung der Komponenten nicht mehr greifen würde.

Eine Abweichung der Implementierung vom ursprünglichen Design kann bereits durch die Sprachunabhängigkeit der UML verursacht werden, die sprachspezifische Implementierungsdetails nicht berücksichtigt und daher durch die Programmiersprache bedingte Abhängigkeiten nicht ausreichend abbildet [2].

In Java ist es zum Beispiel nur sehr bedingt möglich, im Design definierte, unerlaubte Abhängigkeiten programmatisch zu verbieten. Zwar können Klassen und Interfaces als Paket-privat deklariert werden und sind so vom Zugriff außerhalb ihres Pakets geschützt. Allerdings muss dann eine abstrakte Fabrik in jedem Java Paket dafür sorgen, dass man die Implementierung dieser Komponente auch erzeugen kann, was einen nicht unerheblichen Overhead bedeutet, der in der Regel nicht in Kauf genommen wird. Außerdem ist dieser Mechanismus nicht in der Lage, unerlaubte Abhängigkeiten zwischen notwendigerweise nicht Paket-privaten Typen zu kontrollieren.

Daher ist es in der Praxis oft so, dass die entwickelte Java Software vom ursprünglichen Design abweicht (architectural drift [7]) und sich in

diesem Sinn unerlaubte Abhängigkeiten einschleichen. Dieser Effekt wird noch dadurch verstärkt, dass moderne Entwicklungsumgebungen automatisch import statements generieren. Der Nachteil dieser ansonsten sehr angenehmen Einrichtung ist, dass dem Entwickler so kaum noch auffällt, wenn unerlaubte Abhängigkeiten entstehen. Im Einzelfall kann eine Divergenz der Abhängigkeiten von Design und Implementierung natürlich auch auf Defizite im Design hinweisen (architectural erosion [7]). Die Einschätzung ob eine im Design nicht berücksichtigte, in der Implementierung aber vorhandene Abhängigkeit tatsächlich als unerlaubt zu betrachten ist, muss daher im Einzelfall entschieden werden. Wird die Abhängigkeit in der Implementierung als erlaubt angesehen, muss auch das Design entsprechend geändert werden. Wichtig ist dabei, diese Abweichungen überhaupt erst erkennen zu können.

In diesem Papier wird eine Erweiterung des JDepend4Eclipse Plugins vorgestellt, das dieses Problem behebt. Es erlaubt die Spezifikation erlaubter und unerlaubter Abhängigkeiten sowie deren automatische Überwachung. In Kapitel 2 wird das dieser Arbeit zu Grunde liegende Plugin JDepend4Eclipse vorgestellt. Kapitel 3 beschreibt die nötigen Erweiterungen, um unerlaubte Abhängigkeiten verwalten und erkennen zu können. Kapitel 4 stellt einige Ergebnisse bei der Arbeit mit dem neuen Plugin vor. In Kapitel 5 gehen wir auf andere Arbeiten zu diesem Thema ein. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf zukünftige Erweiterungsmöglichkeiten und Trends in Kapitel 6.

2. Hintergrund

Das Eclipse-Plugin JDepend4Eclipse von Andrei Loskutov [4] ist eine Wrapperapplikation von JDepend der Firma Clarkware Consulting. Es ermöglicht dem Anwender durch Auswahl des entsprechenden Eintrags im Kontextmenü einer Ressource, wie etwa einem Java-Paket oder dem gesamten Quelldateiordner eines Eclipse-Projekts, eine Bestimmung von Softwaremaßen zur Designqualität vorzunehmen.

Der Softwareentwickler erhält einen Überblick über die Abhängigkeitsstruktur der von ihm zur Analyse ausgewählten Pakete und Klassen und kann durch Analyse der afferenten (Ca) und efferenten (Ce) Abhängigkeiten eine Aussage über die Unabhängigkeit und Verantwortlichkeit eines Pakets treffen (vgl. [5]). Die von JDepend untersuchten Metriken sind im Einzelnen:

Ca: Anzahl der von dieser Klasse, bzw. den Klassen innerhalb dieses Pakets abhängigen Pakete.(Afferente Abhängigkeiten)

Ce: Anzahl der Pakete, von denen diese Klasse, bzw. die Klassen innerhalb dieses Pakets abhängig sind. (Efferente Abhängigkeiten)

A: Verhältnis der Anzahl abstrakter Klassen zur Anzahl aller Klassen innerhalb dieses Pakets.

I: Anzahl der efferenten Abhängigkeiten im Verhältnis zur Summe der afferenten und efferenten Abhängigkeiten.

D: Distanz von der Ideallinie A + I = 1

Insbesondere können so auch einfach ungewünschte zyklische Abhängigkeiten erkannt und deren Ursache bestimmt werden. Nicht berücksichtigt werden bei dieser Analyse durch JDepend4Eclipse bislang allerdings derlei Abhängigkeiten, die aufgrund der in der Designphase der Softwareentwicklung für dieses Projekt angelegten Architektur als unerlaubt anzusehen sind. Im Rahmen einer an der Hochschule Offenburg durchgeführten Thesis wurde die Funktionalität des Plugins um eine Möglichkeit zur Definition erlaubter Abhängigkeiten und Überprüfung des Projektes auf das Vorhandensein von unerlaubten Abhängigkeiten erweitert.

3. Erweiterung von JDepend4Eclipse

Um JDepend4Eclipse in Softwareprojekten zur einfachen Kontrolle der Umsetzung vereinbarter Architekturansätze bei der Implementierung der Komponenten einsetzen zu können, wurde JDepend4Eclipse sowohl um einen Editor zur Pflege der Regelsätze, als auch um die für die Überprüfung des Projektes auf Regelverletzungen notwendige Anwendungslogik ergänzt.

Damit die erstellten Regeln sowohl mit dem Editor als auch mit externen Anwendungen erstellt und gepflegt werden können, werden die Abhängigkeiten in einer Datei im XML-Format projektbezogen gespeichert. Um die Datei dem Plugin eindeutig zuordnen und diese innerhalb von Eclipse auch direkt mit dem dafür vorgesehenen Rule Editor öffnen zu können, wurde die Dateiendung JDR für diesen Dateityp vorgesehen. Der Pfad und der Dateiname dieser Ressource lassen sich in den Preferences von JDepend4Eclipse angeben.

Die Speicherung der Regelsätze innerhalb des Eclipse Projektes ist insbesondere zur Nachverfolgung von Modifikationen an den erstellten Regeln und zur Verteilung dieser Regeln an die Mitglieder eines Entwicklungsteams von großer Bedeutung. Zudem lassen sich auf diesem Weg auch außerhalb der Eclipse Entwicklungsumgebung Modifikationen am Regelwerk vornehmen und eine weitere Verwendung in anderen Werkzeugen ist möglich. Als

sichtbare und stets verfügbare Datei innerhalb des Projekts erfahren die dort gespeicherten Regeln eine große Aufmerksamkeit und sind im täglichen Entwicklungsprozess präsent.

In den folgenden Abschnitten wird die Benutzung von JDepend4Eclipse anhand der Verwaltung der Abhängigkeiten mit dem Rule Editor und der zur Betrachtung der Analyseergebnisse vorgesehenden Ansicht erläutert.

3.1. Abhängigkeiten verwalten

Zielanwender des Editors ist ein Entwickler, der neu hinzugekommene Abhängigkeiten innerhalb des Projekts mit Hilfe des Editors anhand der Architektur oder in Rücksprache mit dem Software-Architekten prüft und bislang nicht erlaubte und nun im Projekt vorhandene Abhängigkeiten entweder explizit als erlaubt markiert oder deren Regelwidrigkeit bestätigt. Durch die Versionierung der Datei und die damit einhergehende Nachverfolgbarkeit der daran vorgenommenen Änderungen, kann ein Missbrauch des Editors weitestgehend ausgeschlossen werden.

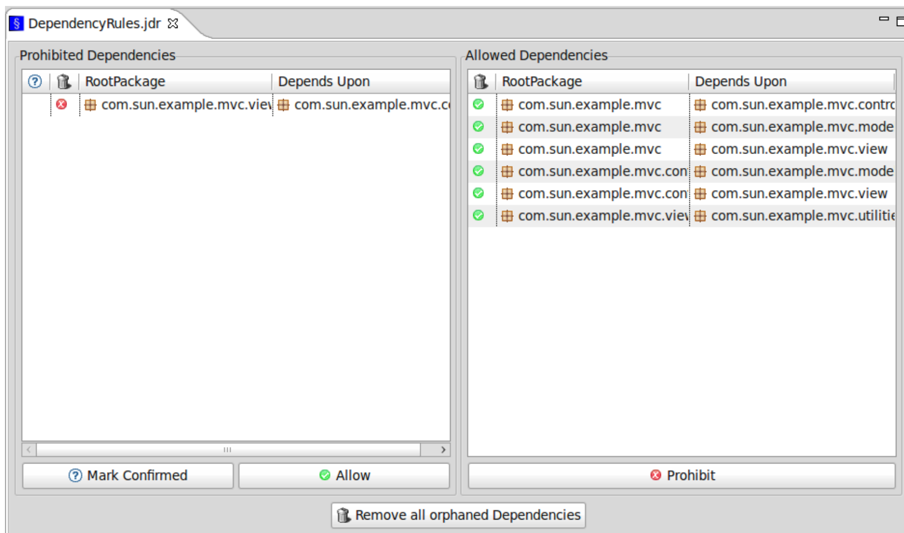

Abbildung 2. JDepend4Eclipse RuleEditor zur Bearbeitung der Regelsätze

Der in Abbildung 2 zu sehende Editor ist in drei Bereiche aufgeteilt. Die linke Tabelle zeigt dem Anwender jene Abhängigkeiten innerhalb seines Projekts, welche nicht als erlaubt markiert wurden und somit verboten

sind. Neue und noch nicht bestätigte Abhängigkeiten sind mit einem Fragezeichen-Symbol gekennzeichnet und können entweder explizit als verboten bestätigt oder den erlaubten Abhängigkeiten hinzugefügt werden. Die Tabelle rechts zeigt die explizit als erlaubt deklarierten Abhängigkeiten an. Die in den beiden Tabellen angezeigten Abhängigkeiten stellen also die Gesamtheit der innerhalb des Projektes vorhandenen Abhängigkeiten zwischen Java Paketen dar.

Durch das Umbenennen oder Löschen von Paketen entstehen in der Regeldatei Einträge, die so im Projekt nicht mehr vorkommen. Verbotene oder erlaubte Abhängigkeiten, die im Projekt nicht mehr existieren jedoch in der Datei noch vorhanden sind, werden mit einem Mülleimer-Symbol gekennzeichnet. Diese können entweder in der Datei belassen oder durch Drücken des entsprechenden Buttons gelöscht werden.

Unter den Tabellen befindet sich der Bereich für die Steuerung des Editors. Um etwa Abhängigkeiten zu erlauben werden diese markiert und durch Auswahl des entsprechenden Buttons den erlaubten Abhängigkeiten hinzugefügt. Entsprechend funktioniert auch das nachträgliche Verbieten bislang erlaubter oder fälschlicherweise als erlaubt markierter Abhängigkeiten oder die Bestätigung neu hinzugekommener Abhängigkeiten.

3.2. Abhängigkeiten erkennen

Die im Eclipse-Umfeld als Perspective bezeichnete, bisherige Ansicht von JDepend4Eclipse wurde um ein zusätzliches Fenster ergänzt, das die unerlaubten Abhängigkeiten der im PackageTreeView durch den Anwender selektierten Pakete bzw. Klassen anzeigt. Bezogen auf diese Selektion, bekommt der Anwender hier die regelwidrigen Abhängigkeiten angezeigt.

Die Projektstruktur lässt sich in dem in Abbildung 3 zu sehenden PackageTreeView ablesen. Der Anwender kann hier die Auswahl von Paketen und Klassen vornehmen, deren Metriken betrachtet werden sollen. Somit ist nicht nur möglich festzustellen, ob ein Paket unerwünschte, efferente Abhängigkeiten aufweist, sondern der Entwickler kann auch feststellen, durch welche Klassen diese verursacht werden und kann den Umstand beheben. In der Praxis sind unerlaubte Abhängigkeiten häufig für das Auftreten zyklischer Abhängigkeiten verantwortlich. Diese Zusammenhänge lassen sich in dieser Ansicht auf einen Blick erkennen.

Sollen nur interne Abhängigkeiten geprüft werden, so können Abhängigkeiten zu externen Bibliotheken durch die bereits in JDepend4Eclipse vorhandenen Filter ausgeblendet werden. Diese können in den Preferences des Plugins eingestellt werden. Geplant ist, diese Funktionalität projektspezifisch zur Verfügung zu stellen.

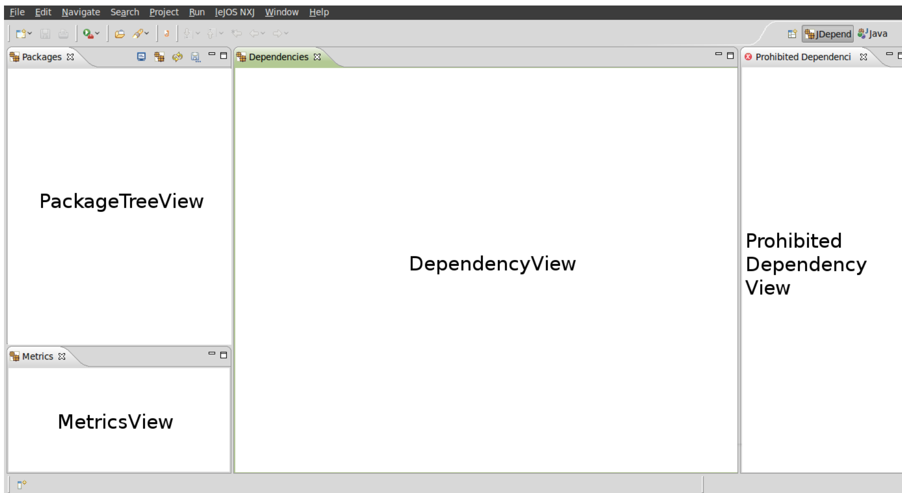

Abbildung 3. JDepend4Eclipse Perspective

4. Ergebnisse

Zur Evaluation wurde das Plugin in zwei Projekten beispielhaft eingesetzt. Die Ergebnisse sind im Folgenden dargestellt.

4.1. SUN MVC Beispiel

Die Funktionsweise von JDepend4Eclipse wurde zunächst in einer überschaubaren Umgebung an einem im Sun Developer Network als Beispiel für die Implementierung des Model-View-Controller Patterns zur Verfügung gestellten Projektes [3] erprobt. Da für den ersten Lauf von JDepend4Eclipse zunächst noch keine Datei mit Regelsätzen zur Verfügung stand, wurden die im Projekt vorhandenen Abhängigkeiten in der JDepend4Eclipse Perspective zunächst alle als unerlaubt angezeigt. Bei Betrachtung der Metriken war hier vor allem auffällig, dass im Projekt zyklische Abhängigkeiten vorhanden waren. Das Erlauben der Abhängigkeiten im Editor von JDepend4Eclipse erfolgte unter Berücksichtigung des in Abbildung 1 gezeigten Entwurfsmusters.

Zusätzlich zu den Java Paketen *model*, *view* und *controller* waren hier noch zwei weitere Pakete zu berücksichtigen. Das Paket *mvc* beinhaltete die main() Methode und wies efferente Abhängigkeiten sowohl zu *model*, *view* und *controller* auf. Da diese Methode zur Erzeugung der Objekte und zur Dependency Injection genutzt wurde, waren diese Abhängigkeiten jedoch als

gerechtfertigt anzusehen und konnten somit erlaubt werden. Das Paket *utilities* beinhaltete dem Entwurfsmuster View Helper entsprechend für die Views notwendige Logik, weshalb die Abhängigkeit des Pakets *view* zu *utilities* ebenfalls als gerechtfertigt betrachtet wurde und diese daher ebenfalls erlaubt wurde. Zuletzt blieb eine Abhängigkeit des *view* Pakets zum *controller* Paket in der Liste der unerlaubten Abhängigkeiten bestehen.

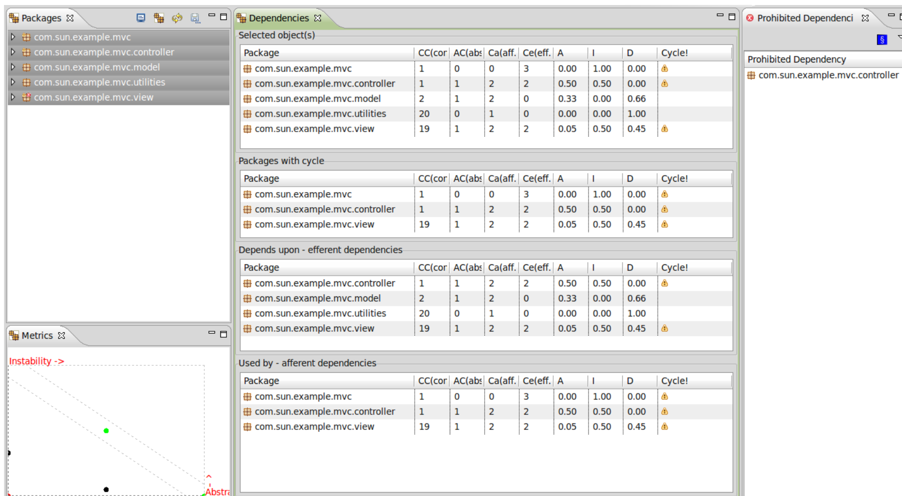

Abbildung 4. JDepend4Eclipse Analyseergebnis des SUN MVC Beispiels

Nach Fertigstellung der Regelsätze wurde ein erneuter Analyselauf von JDepend4Eclipse veranlasst. Wie in Abbildung 4 gezeigt, konnten so die Klassen DisplayViewPanel und PropertiesViewPanel als Urheber der unerlaubten und zyklischen Abhängigkeit in der JDepend4Eclipse Perspective ermittelt und das Projekt durch entsprechendes Refactoring bereinigt werden.

4.2. Projekt Attractive

Der Einsatz in einem mittelgroßen Projekt wurde anhand des an der Hochschule Offenburg durchgeführten Projekts Attractive durchgeführt. In diesem Projekt sind derzeit 230 Java Klassen in 28 Komponenten und 62 Paketen organisiert. Vor der Verfügbarkeit der automatischen Prüfung auf unerlaubte Abhängigkeiten wurde die Übereinstimmung der Implementierung zur vorgegebenen Architektur manuell mit Hilfe des Vorgängers JDepend4Eclipse (Version 1.2.0) geprüft. Dieser Prozess dauerte

ca. 30 Minuten und wurde entsprechend selten ca. alle vier Wochen durchgeführt.

Die Ergebnisse der automatischen Prüfung sind in Tabelle 1 dargestellt. Die erstmalige Definition erlaubter und unerlaubter Abhängigkeiten mit dem in Abschnitt 3.1 vorgestellten Editor dauerte 36 Minuten und damit nur unwesentlich länger als eine manuelle Prüfung vorher. Ab dann dauerte eine automatische Prüfung 2,7 Sekunden und damit unmerklich länger, als ein Lauf der Vorgängerversion ohne automatische Prüfung. Insgesamt wurden 123 Abhängigkeiten zwischen Paketen gefunden, davon waren 119 erlaubte Abhängigkeiten. Interessant ist also, dass bereits bei der erstmaligen Definition der Abhängigkeiten vier unerlaubte Abhängigkeiten gefunden wurden, ein deutliches Zeichen für die Fehleranfälligkeit des zuvor durchgeführten manuellen Prozesses. Mit dem Einsatz des neuen Plugins durch alle Entwickler kann jetzt vor jedem Commit innerhalb weniger Sekunden die Kompatibilität zur Architektur sichergestellt werden.

Klassen	230
Komponenten	28
Laufzeit	2,7 s
Erlaubte Abhängigkeiten	119
Gefundene, unerlaubte Abhängigkeiten	4

Tabelle 1. Ergebnisse der Anwendung auf das Attractive Projekt

5. Andere Arbeiten

Das Problem der Divergenz von Architektur und Implementierung wird vielfach in der Literatur diskutiert (siehe z.B. [7]). Die meisten Lösungsansätze basieren darauf, die Architektur in geeigneten Meta-Sprachen zu beschreiben, um eine automatische Überprüfung der Divergenz von Architektur und Implementierung zu ermöglichen. Schwanke und andere [8] führen die Sprache 'Gestalt' ein mit deren Hilfe Architekturspezifikationen derart formalisiert werden können, dass deren Einhaltung durch die Software automatisch verifiziert werden kann. Die industrielle Einsatzmöglichkeit wird am Beispiel eines großen Projekts in C belegt. Mens [6] zeigt auf ähnliche Weise am Beispiel von Smalltalk, wie mit einer logischen Meta-Programmiersprache (LMP) der Prozess der Verifikation automatisiert werden kann.

Derartige Ansätze sind aber relativ aufwändig in der Definition und Einarbeitung. Insbesondere für bereits vorhandene Systeme wird man diesen

Aufwand nicht leisten wollen. Demgegenüber kann die Definition unerlaubter Abhängigkeiten mit dem hier beschriebenen Plugin in relativ kurzer Zeit erfolgen.

Das Problem der Divergenz von Architektur und Implementierung an der Wurzel packen würde, wenn bereits der Compiler unerlaubte Abhängigkeiten erkennen und anzeigen würde. Dies könnte das für Java7 diskutierte Projekt Jigsaw mit Java Specification Request (JSR) 294 leisten [1]. Dort ist ein Modulsystem vorgesehen, das auf Sprachebene erlaubt, Abhängigkeiten von Modulen zu definieren. Allerdings ist derzeit noch nicht klar, ob JSR 294 tatsächlich Teil von Java7 sein wird. Ebenfalls gibt es Meldungen, dass dieser JSR nicht Teil der Java SE 7 platform specification sein wird und damit nicht zwangsweise von allen Runtimes unterstützt werden würde. In jedem Fall wird JDepend4Eclipse für bestehende Projekte, die nicht auf Java7 umgestellt werden, ein wichtiges Werkzeug zur Sicherstellung der Implementierungsqualität bleiben.

6. Zusammenfassung und Ausblick

Durch die im Produktiveinsatz von JDepend4Eclipse gesammelten Erfahrungen wurden Erkenntnisse über weiteren Funktionsbedarf und wünschenswerte Funktionen gewonnen. Die Überprüfung des Projektes auf das Bestehen unerlaubter Abhängigkeiten erfolgt derzeit lediglich durch manuellen Start durch den Anwender. Wünschenswert wäre eine entsprechende Kontrolle automatisch, beispielsweise durch einen eingesetzten Automation Server erfolgen zu lassen. Ein möglicher Ansatz ist die automatische Generierung eines JUnit-Modultests, der im Rahmen der automatisierten Testläufe die Schnittstellen von JDepend4Eclipse nutzt, um eine Überprüfung erfolgen zu lassen. Bei Vorliegen von unerlaubten Abhängigkeiten könnte der Entwickler, der durch seinen Checkin in die Versionskontrolle die Regelverstöße verursacht hat, entsprechend informiert und zu einer Korrektur veranlasst werden. Die Schwierigkeit bei der Umsetzung besteht hier vor allem darin, dass für die Überprüfung ein Start von JDepend während des Testlaufs notwendig wäre und dies die Installation der Eclipse-RCP-Module zwingend erforderlich macht.

Die Erstellung der JDepend-Regeldatei wird beim ersten Start von JDepend4Eclipse angelegt und behandelt derzeit alle Abhängigkeiten im Projekt zunächst als unerlaubte Abhängigkeiten. Dadurch wird sichergestellt, das eine Auseinandersetzung mit den vorhandenen Abhängigkeiten erfolgt und der Editor genutzt werden muss, um die Abhängigkeiten entweder als erlaubt zu deklarieren, oder aber eine Bestätigung des Regelverstoßes und eine dementsprechende Korrektur der zugrunde liegenden Struktur

vorzunehmen. Die Informationen über erlaubte und somit auch über unerlaubte Abhängigkeiten liegen aber bereits in der Designphase in Form der erstellten UML-Diagramme vor. Um eine Durchgängigkeit zu erreichen und beispielsweise auch bei Durchführung eines Roundtrip-Engineering die Erstellung der Regeldatei automatisieren zu können, ist die Erstellung der Regelsätze durch das zur Erstellung der UML-Diagramme genutzte Werkzeug wünschenswert. Dies könnte ebenfalls in Form eines Plugins für die jeweilige Software erreicht werden.

Nach einer mehrwöchigen Testphase hat das neue Plugin inzwischen einen stabilen Stand erreicht. Wie in Abschnitt 4.2 erläutert wird das Plugin in internen Projekten der Hochschule bereits erfolgreich eingesetzt. Jetzt ist geplant, das Plugin der breiten Öffentlichkeit zugänglich zu machen. Es ist zu erwarten, dass es rege Anwendung finden wird.

Literatur

[1] A. Buckley. Java specification request jsr 294. `http://jcp.org/en/jsr/detail?id=294`, Februar 2010.

[2] Cooper et al. Java implementation verification using reverse engineering, 2004.

[3] R. Eckstein. Java se application design with mvc. `http://java.sun.com/developer/technicalArticles/javase/mvc`, März 2007.

[4] A. Loskutov. Jdepend plugin for eclipse: Jdepend4eclipse. `http://andrei.gmxhome.de/jdepend4eclipse/index.html`, Februar 2010.

[5] R. C. Martin. Oo design quality metrics - an analysis of dependencies, 1994.

[6] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[7] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[8] R. W. Schwanke, V. A. Strack, and T. Werthmann-Auzinger. Industrial software architecture with gestalt. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 176, Washington, DC, USA, 1996. IEEE Computer Society.