Zwei Metriken zum Messen des Umgangs mit Zugriffsmodifikatoren in Java

Christian Zoller
Axel Schmolitzky

Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg
christian.zoller@informatik.uni-hamburg.de
schmolit@informatik.uni-hamburg.de

Abstract: Wie viele objektorientierte Programmiersprachen bietet Java die Möglichkeit, über Modifikatoren die Zugreifbarkeit von Typen, Methoden und Feldern in mehreren Stufen einzuschränken. So können für unterschiedliche Gruppen von Klienten differenzierte Schnittstellen definiert werden. Es zeigt sich jedoch, dass in der Praxis die gebotenen Möglichkeiten nicht voll ausgeschöpft werden. Wir beschreiben zwei neue Metriken, mit denen sich der angemessene Umgang mit Zugriffsmodifikatoren in Java messen lässt, sowie ein Werkzeug, das diese Metriken berechnet und beim Einschränken von Schnittstellen hilfreich sein kann. Wir haben unseren Ansatz in zwei kommerziellen Projekten und zwölf Open-Source-Projekten erprobt. Dabei wurde deutlich, dass Zugriffsmodifikatoren oft großzügiger gewählt werden als notwendig.

1 Einleitung

Das Definieren von Schnittstellen und die Kapselung von Implementationsdetails sind zentrale Bestandteile eines jeden Softwareentwurfs. Eine hohe Kapselung erleichtert nicht nur die Wiederverwendbarkeit und Änderbarkeit von Software, da unabhängige, klar abgegrenzte Module sich leichter austauschen lassen, sondern auch die Verständlichkeit, denn je weniger Informationen Module untereinander austauschen, umso leichter ist ihr Zusammenspiel zu verstehen. Moderne objektorientierte Sprachen unterstützen diese Prinzipien durch die Mechanismen der Zugriffskontrolle. Bei der Deklaration von Elementen wie Klassen, Methoden oder Variablen kann die Menge der Klienten festgelegt werden, aus denen auf das deklarierte Element zugegriffen werden darf. Je kleiner diese Menge ist, desto höher ist die Kapselung [Sny86], [Mey97, S. 47: „Few Interfaces Rule“].

1.1 Typen-Schnittstellen in Java

Typen werden in Java durch Klassen und Interfaces definiert. Die Zugriffskontrolle wird über die Zugriffsmodifikatoren `private`, `protected` und `public` geregelt [GJSB05, § 6.6]. Auf ein `private` deklariertes Klassen-Member darf nur innerhalb der eigenen Top-Level-Klasse zugegriffen werden. `protected` erlaubt den Zugriff innerhalb des eigenen Pakets und zusätzlich durch Unterklassen außerhalb des Pakets. `public` deklarierte Elemente dürfen von allen Klienten der Klasse verwendet werden. Lässt man den Zugriffsmodifikator ganz weg, darf das deklarierte Member nur im eigenen Paket verwendet werden. Diese Zugriffsebene wird auch default oder *package-private* genannt.
Die zugreifbaren Member einer Klasse bilden deren Schnittstelle. Aufgrund der Möglichkeit die Zugreifbarkeit von Membern in den genannten Stufen einzustellen, besitzen Klassen nicht nur jeweils *eine* Schnittstelle, sondern verschiedene für unterschiedliche Mengen von Klienten: eine für Klienten innerhalb des eigenen Pakets, eine für Unterklassen außerhalb des eigenen Pakets und eine für alle restlichen Klienten.
Der Begriff der Schnittstelle ist somit von dem Java-Konstrukt Interface zu unterscheiden. Doch auch Interfaces definieren Schnittstellen. Diese bestehen aus allen ihren Membern, da Interface-Member immer `public` sind.

1.2 Paket-Schnittstellen in Java

In Java verfügen nicht nur Typen, also Klassen und Interfaces, über Schnittstellen, sondern auch Pakete. Diese bestehen aus den enthaltenen, zugreifbaren Top-Level-Typen und deren zugreifbaren Membern. Für Top-Level-Typen gibt es jedoch nur zwei mögliche Zugriffsstufen: `public` und default. Die Zugriffsmodifikatoren `private` und `protected` sind Klassen-Membern vorbehalten.
Verfolgt man das Prinzip größtmöglicher Kapselung, sollten sowohl die Schnittstellen von Typen als auch die von Paketen so klein wie möglich sein [Mey97, S. 48: „Small Interfaces Rule“].

1.3 Beobachtungen aus der Praxis

In der Praxis lässt sich jedoch feststellen, dass der Umgang mit Zugriffsmodifikatoren nicht so restriktiv gehandhabt wird, wie es vielleicht möglich wäre. Während die Regel, dass Exemplarvariablen grundsätzlich `private` sein sollten, allgemein anerkannt ist, wird z.B. das Kapseln von paketinternen Klassen oder Methoden deutlich seltener angewendet. Insbesondere bei Top-Level-Klassen scheint das einleitende `public`-Schlüsselwort ein Quasi-Standard zu sein.
Dies hat verschiedene Gründe. Schon ein Blick in einschlägige Lehrbücher (z.B. [BK09], [HC07], [RSSW10], [Sav09]) zeigt, dass das Schnittstellen-Konzept, das auf Klassenebene noch ausführlich mit der Unterscheidung zwischen `public` und `private` erläutert

wird, nur selten auf Paketebene übertragen und die default-Zugreifbarkeit oft nur am Rande erwähnt wird. In dem verbreiteten Quelltextanalyse-Werkzeug *PMD*[1] findet sich sogar eine Regel, die von der Verwendung der default-Zugreifbarkeit abrät – warum diese Empfehlung ausgesprochen wird, lässt sich dabei jedoch nicht nachvollziehen. Darüber hinaus scheuen sich Entwicklerinnen und Entwickler möglicherweise davor, Zugreifbarkeiten zu restriktiv zu wählen, um sich nicht in den eigenen Möglichkeiten zu beschneiden. Dadurch können jedoch ungewollte Abhängigkeiten entstehen, worunter letztlich die Qualität der Software leidet.

1.4 Unser Ansatz

Softwaremetriken sind ein Weg die Qualität von Software automatisiert und objektiv zu bewerten. Es liegt nahe, auch den Umgang mit Zugriffsmodifikatoren einer solchen Bewertung zugänglich zu machen. Zu diesem Zweck haben wir zwei Java-Metriken entwickelt, die den Anteil derjenigen Typen und Methoden messen, denen ein unnötig großzügiger Zugriffsmodifikator zugewiesen ist. Um zu großzügige Zugriffsmodifikatoren zu ermitteln und die Metriken zu berechnen, haben wir ein Werkzeug entwickelt und dieses an zwei kommerziellen und zwölf Open-Source-Projekten erprobt. Dabei stellten wir fest, dass ein großer Teil der verwendeten Zugriffsmodifikatoren großzügiger ist als eigentlich notwendig.
Die Zugreifbarkeit von Feldern lassen wir in unseren Betrachtungen außen vor, da die Regel, dass diese immer `private` sein sollten, hinreichend einfach ist und bereits mit Hilfe bestehender Werkzeuge überprüft werden kann (z.B. *PMD*, *FindBugs*[2]).
Im folgenden Abschnitt 2 führen wir zunächst einige Begriffe ein und definieren dann die beiden Metriken. Außerdem stellen wir das Werkzeug *AccessAnalysis* vor, ein Plug-In für die Entwicklungsumgebung *Eclipse*. Anschließend folgen unter 3. die Ergebnisse unserer exemplarischen Untersuchungen. In Abschnitt 4 diskutieren wir Einsatzmöglichkeiten und Grenzen der Metriken. In Abschnitt 5 gehen wir kurz auf ähnliche Arbeiten in diesem Bereich und mögliche Anschlussarbeiten ein. Am Ende folgt eine Zusammenfassung.

2 Zwei neue Metriken für Java

2.1 Begriffe

Das Nichtvorhandensein eines Zugriffsmodifikators, also die default-Zugreifbarkeit, wird im Folgenden zur sprachlichen Vereinfachung ebenfalls als Zugriffsmodifikator bezeichnet.
Ein Zugriffsmodifikator heißt *strenger* als ein anderer, wenn er weniger Klienten den Zugriff gewährt. Andersherum heißt er *großzügiger*. Da in Java die Zugriffsebenen klar hier-

[1] http://pmd.sourceforge.net
[2] http://findbugs.sourceforge.net

archisch gegliedert sind, ergibt sich die Reihenfolge `private` → default → `protected` → `public`, in der die Zugriffsmodifikatoren großzügiger und entgegengesetzt strenger werden.
Legt man die tatsächliche Verwendung der Typen (Klassen und Interfaces) und Methoden innerhalb eines Java-Systems zu Grunde, ergibt sich auf Basis der Sprachdefinition für jedes dieser Elemente ein strengster Zugriffsmodifikator, der ausreicht, um alle Benutzungen des Elements innerhalb des Systems zu gewährleisten. Diesen Zugriffsmodifikator nennen wir *minimalen Zugriffsmodifikator* des Typs bzw. der Methode. Der Zugriffsmodifikator, der einem Element im Quelltext tatsächlich zugeordnet ist, nennen wir *tatsächlicher Zugriffsmodifikator*.
Ist der tatsächliche Zugriffsmodifikator eines Elements großzügiger als sein minimaler, bezeichnen wir diesen in diesem Zusammenhang als *zu großzügig*.

2.2 Inappropriate Generosity with Accessibility of Types (IGAT)

Die Metrik „Inappropriate Generosity with Accessibility of Types“ (IGAT) sei folgendermaßen definiert:

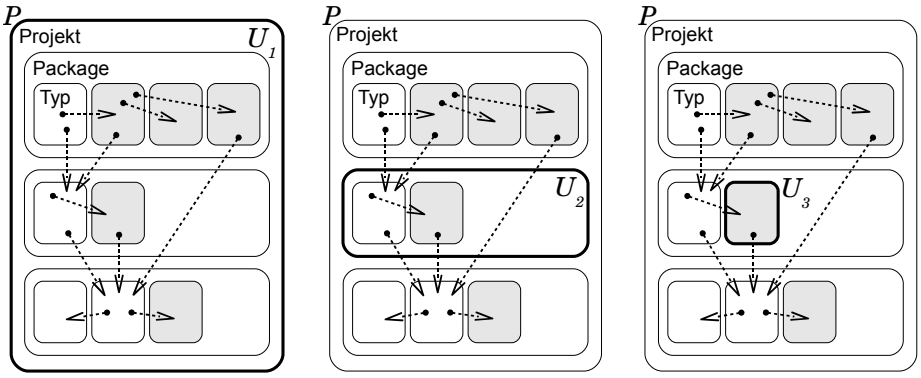
$$IGAT(U,P) = \begin{cases} 0, & \text{wenn } |T(U)| = 0 \\ \dfrac{|T^*(U,P)|}{|T(U)|}, & \text{sonst} \end{cases} \tag{1}$$

wobei

- P der Quelltext aller Übersetzungseinheiten eines Java-Programms ist,
- U mit $U \subseteq P$ eine Quelltext-Teilmenge aus P (z.B. der gesamte Quelltext, ein Paket oder eine Typdeklaration),
- $T(U)$ die Menge aller in U deklarierten Typen und $|T(U)|$ deren Anzahl,
- $T^*(U,P)$ mit $T^*(U,P) \subseteq T(U)$ die Menge aller in U deklarierten Typen mit zu großzügigem Zugriffsmodifikator und $|T^*(U,P)|$ deren Anzahl. Zu großzügig bedeutet hier im Vergleich zu den minimalen Zugriffsmodifikatoren, die auf Basis der Benutzung der Typen innerhalb von P zu bestimmen sind.

Am Anfang der Berechnung steht also die Bestimmung der minimalen Zugriffsmodifikatoren auf Basis des gesamten Quelltextes P. Danach wird dann für die zu vermessende Quelltext-Teilmenge U der Anteil der Typen berechnet, deren Zugriffsmodifikator zu großzügig ist.

Beispiel:



$|T(U_1)| = 9$
$|T^*(U_1, P)| = 5$
$IGAT(U_1, P) = \frac{5}{9}$

$|T(U_2)| = 2$
$|T^*(U_2, P)| = 1$
$IGAT(U_2, P) = \frac{1}{2}$

$|T(U_3)| = 1$
$|T^*(U_3, P)| = 1$
$IGAT(U_3, P) = \frac{1}{1}$

Abbildung 1: IGAT-Berechnung

Abb. 1 zeigt dreimal die schematische Darstellung eines Java-Projekts P, das in drei Pakete mit insgesamt neun Typdeklarationen aufgeteilt ist. Die grau hinterlegten Typen sind diejenigen, deren Zugriffsmodifikator nach Analyse der Benutztbeziehungen (Pfeile) innerhalb von P als zu großzügig erkannt wurde. Je nachdem für welche Untermenge U_i von P der IGAT-Wert berechnet werden soll, ergibt sich die Gesamtzahl der Typen $|T(U_i)|$ und die Anzahl $|T(U_i, P)|$ derjenigen Typen mit zu großzügigem Zugriffsmodifikator. Die Mengen P ist in allen drei Fällen dieselbe.

2.3 Inappropriate Generosity with Accessibility of Methods (IGAM)

Analog zur Typen-Metrik IGAT sei für Methoden die Metrik „Inappropriate Generosity with Accessibility of Methods“ (IGAM) definiert:

$$IGAM(V, P) = \begin{cases} 0, & \text{wenn } |M(V)| = 0 \\ \dfrac{|M^*(V, P)|}{|M(V)|}, & \text{sonst} \end{cases} \tag{2}$$

wobei

- P auch hier der Quelltext aller Übersetzungseinheiten eines Java-Programms ist,
- V mit $V \subseteq P$ eine Quelltext-Teilmenge aus P (z.B. der gesamte Quelltext, ein Paket, eine Typ- oder Methodendeklaration),

- $M(V)$ die Menge aller in V deklarierten Methoden und $|M(V)|$ deren Anzahl,
- $M^*(V,P)$ mit $M^*(V,P) \subseteq M(V)$ die Menge aller in V deklarierten Methoden mit zu großzügigem Zugriffsmodifikator und $|M^*(V,P)|$ deren Anzahl. Zu großzügig bedeutet auch hier im Vergleich zu den minimalen Zugriffsmodifikatoren, die auf Basis der Benutzung der Methoden innerhalb von P zu bestimmen sind.

2.4 Das Werkzeug *AccessAnalysis*

Um die vorgestellten Metriken berechnen zu können, haben wir *AccessAnalysis* [3] entwickelt, ein Plug-In für die Entwicklungsumgebung *Eclipse* [4]. Es kann auf Grundlage eines oder mehrerer Projekte die minimalen Zugriffsmodifikatoren aller enthaltenen Typen und Methoden ermitteln und anhand dieser IGAT und IGAM berechnen. In einer baumartigen Tabelle (Abb. 2) werden die Metriken sowohl auf Projektebene als auch für jedes Paket, jeden Typ und jede Methode (nur IGAM) ausgegeben. Zudem werden die tatsächlichen und minimalen Zugriffsmodifikatoren der Typen und Methoden gegenübergestellt.
Für den Fall, dass ein Typ oder eine Methode innerhalb der analysierten Projekte gar nicht benutzt wird, wird der Pseudo-Zugriffsmodifikator not used als minimaler angegeben.

AccessAnalysis

Element	IGAT	IGAM	Minimal Access	Actual Access
findbugs-1.3.9	0,51	0,38		
edu.umd.cs.findbugs	0,33	0,31		
edu.umd.cs.findbugs.annotations	0,36	0,50		
edu.umd.cs.findbugs.anttask	1,00	0,93		
AbstractFindBugsTask	1,00	0,92	default	public
ClassLocation	1,00	0,67	private	public
getLocation()		1,00	not used	public
setLocation(File)		1,00	not used	public
toString()		0,00	public	public
ComputeBugHistoryTask	1,00	1,00	not used	public
ConvertXmlToTextTask	1,00	1,00	not used	public
ConvertXmlToTextTask()		1,00	not used	public
afterExecuteJavaProcess(int)		1,00	default	protected
beforeExecuteJavaProcess()		1,00	default	protected

Abbildung 2: Ergebnisausgabe des *Eclipse*-Plug-Ins *AccessAnalysis*

[3] http://accessanalysis.sourceforge.net
[4] http://eclipse.org

3 Erprobung

3.1 Einsatz in kommerziellen Projekten

Die ersten Messungen unter realen Bedingungen haben wir in zwei Softwareunternehmen durchgeführt, bei denen wir jeweils ein Projekt analysieren durften. Die Ergebnisse waren noch nicht sehr aussagekräftig, da wir feststellen mussten, dass durch den Einsatz verschiedenster Frameworks, wie *JUnit*, *Hibernate*, *Spring*, *EJB* etc., der Anteil der Typen und Methoden, die per Reflection aufgerufen werden, extrem hoch ist. Hinzu kommen viele Elemente, die nur für die Benutzung in JSPs bestimmt sind. Da *AccessAnalysis* bisher jedoch nur einfachen Java-Quelltext analysieren kann und so viele Benutzungen nicht registriert werden konnten, führte dies zu einer großen Anzahl falscher Meldungen von zu großzügigen Zugriffsmodifikatoren.
Doch bestärkten diese ersten Versuche unsere These, dass die default-Zugreifbarkeit so gut wie nie eingesetzt wird. So waren im ersten Projekt von insgesamt 2.613 Typen nur sechs default deklariert und damit im eigenen Paket gekapselt; von 22.210 Methoden waren es lediglich 29. Im zweiten, deutlich kleineren Projekt sah das Bild ähnlich aus. Hier waren von 355 Typen nur zwei default und von 4.143 Methoden lediglich eine.

3.2 Vermessung von Open-Source-Software

Um trotz der in den ersten Testläufen aufgezeigten Schwierigkeiten zu verwertbaren Ergebnissen zu kommen, haben wir zwölf Open-Source-Projekte ausgewählt, bei denen Falschmeldungen von zu großzügigen Zugriffsmodifikatoren weitestgehend vermieden werden konnten. Bedingung war auch hier, dass es sich um eigenständige Anwendungen handelt und nicht um Bibliotheken oder Frameworks. Darüber hinaus sollten die Projekte keine JSPs enthalten und Reflection, sei es im eigenen Code oder durch eingesetzte Frameworks, durfte keine große Rolle spielen. Weiterhin war gefordert, dass sich der Quelltext nach dem Import in *Eclipse* und dem Einbinden aller benötigten Bibliotheken ohne weitere Modifikation übersetzen ließ. Die in Tab. 1 aufgeführten, von uns ausgewählten Anwendungen erfüllten weitestgehend diese Forderungen.
Die Größe der Projekte reicht von 29 Typen in vier Paketen (*JDepend*) bis zu 1.319 Typen in 53 Paketen (*FindBugs*). Nach dem Import der Quelltexte in *Eclipse* haben wir die enthaltenen *JUnit*-Testklassen entfernt. Da diese sowie die in ihnen deklarierten Testmethoden immer `public` sein müssen, obwohl sie anderweitig im Quelltext nicht verwendet werden, hätten auch sie das Ergebnis verfälscht. Zwar fielen so auch die Benutztbeziehungen zwischen den Testklassen und den zu testenden Elementen weg, doch zeigte ein Vergleich, dass sich das Entfernen der Tests nur wenig auf die minimalen Zugriffsmodifikatoren der anderen Typen und Methoden auswirkte. Bei einigen Projekten enthielt der verfügbare Quelltext von vornherein keine Tests. Neben den *JUnit*-Testklassen haben wir ebenso Beispielcode, der bei Quelltextanalyse-Tools, wie z.B. *PMD*, mitgeliefert wurde, entfernt.

BlueJ	3.0.0	Java-IDE	http://bluej.org
Cobertura	1.9.4.1	Java-Test-Coverage-Tool	http://cobertura.sourceforge.net
DoctorJ	5.1.2	Javadoc-Analysetool	http://www.incava.org/projects/java/doctorj
FindBugs	1.3.9	Java-Fehleranalysetool	http://findbugs.sourceforge.net
FreeCol	0.9.3	Strategiespiel	http://www.freecol.org
FreeMind	0.8.1	Illustrationsprogramm	http://freemind.sourceforge.net
JabRef	2.6	Literaturverwaltung	http://jabref.sourceforge.net
JDepend	2.9	Java-Metriktool	http://www.clarkware.com/software/JDepend.html
jrDesktop	0.3.1.0	Fernwartungstool	http://jrdesktop.sourceforge.net
PDFsam	2.2.0	PDF-Werkzeug	http://www.pdfsam.org
PMD	4.2.5	Java-Fehleranalysetool	http://pmd.sourceforge.net
Sweet Home 3D	2.5	Illustrationsprogramm	http://www.sweethome3d.com

Tabelle 1: Analysierte Open-Source-Projekte.

3.3 Ergebnisse

Abb. 3 zeigt die IGAT- und IGAM-Werte für die gesamten Quelltexte der einzelnen, vermessenen Projekte. Man sieht, dass bei den meisten ein nicht unerheblicher Teil der Zugriffsmodifikatoren zu großzügig ist. Der Anteil der Methoden mit zu großzügigem Zugriffsmodifikator bewegt sich dabei relativ eng in einem Bereich von 25 bis 44 %. Nur das Projekt *DoctorJ* liegt mit einem IGAM-Wert von 71 % deutlich außerhalb dieses Bereichs. Die IGAT-Werte zeigen eine deutlich stärkere Schwankung und insgesamt höhere Werte. Die auffälligsten Ergebnisse haben hier *JabRef* und *JDepend*, bei denen jeweils 59 % der Typen einen zu großzügigen Zugriffsmodifikator haben, sowie auf der anderen Seite *PDFsam*, bei dem es nur 16 % sind.
Tab. 2 stellt die Verteilung der tatsächlichen und minimalen Zugriffsmodifikatoren gegenüber. Auch hier zeigt sich, dass die default-Zugreifbarkeit so gut wie nie eingesetzt wird. Durch die Aufschlüsselung der Zahlen der Typen in Top-Level- und Membertypen wird deutlich, dass dies besonders die Top-Level-Typen betrifft. Hier sind lediglich 104 von 4.570 Typen default deklariert, obwohl es 1.259 sein könnten. Doch auch bei den Membertypen, wo der default-Zugriffsmodifikator einen deutlich höheren Anteil hat, scheint dieser eher zufällig zum Einsatz zu kommen. Denn beim größten Teil der default-Membertypen ist `private` der minimale Zugriffsmodifikator, stattdessen könnten auch hier einige der `public`-Typen default sein.

4 Bewertung

Unsere Grundannahme ist, dass Zugriffsmodifikatoren, die sich an der tatsächlichen Benutzung orientieren, die Benutzungsebenen explizit machen und so die Verständlichkeit

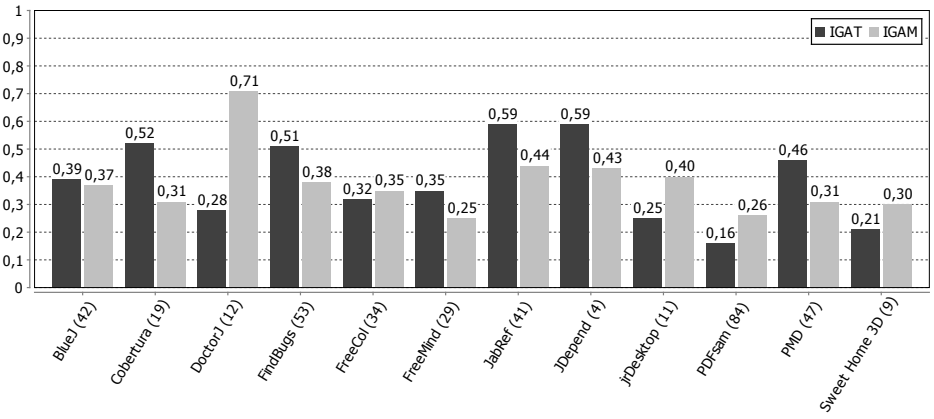


Abbildung 3: IGAT- und IGAM-Gesamtergebnisse der untersuchten Open-Source-Projekte. In Klammern: Anzahl der Pakete.

	Alle Typen		Nur Top-Level-Typen		Nur Membertypen		Methoden	
	tats.	min.	tats.	min.	tats.	min.	tats.	min.
`public`	4.966	2.926	4.466	2.770	500	156	39.060	22.751
`protected`	40	7			40	7	2.347	1.646
default	556	1.391	104	1.259	452	132	1.084	7.325
`private`	592	1.251			592	1.251	8.704	13.438
not used		579		541		38		6.035
gesamt	6.154		4.570		1.584		51.195	

Tabelle 2: Verteilung der Zugriffsmodifikatoren in den analysierten Projekten.

von Software erhöhen. Ein strenger Umgang mit Zugriffsmodifikatoren schützt vor unnötigen Abhängigkeiten und erleichtert so die Änderung einzelner Teile einer Software. Negative Auswirkungen könnten minimale Schnittstellen auf Erweiterbarkeit, Wiederverwendbarkeit und Testbarkeit haben, wenn auf vorhandene Funktionalität ohne Änderung am Quelltext nicht zugegriffen werden kann.
In realen Softwareprojekten kann es etliche Gründe geben, weshalb für einen Typ oder eine Methode nicht immer der minimale Zugriffsmodifikator gewählt wird. Ein naheliegender Grund, weshalb ein Element einen zu großzügigen Zugriffsmodifikator haben kann, ist beispielsweise, dass ein Klient erst noch geschrieben werden muss. Auch wenn ein Projekt nicht als eigenständige Anwendung, sondern als Bibliothek oder Framework entwickelt wird, werden mit großer Wahrscheinlichkeit viele Schnittstellen Elemente enthalten, die innerhalb des Projekts selbst gar nicht benutzt werden. Ebenso spielen bei der Entwicklung eines stark komponentenbasierten Systems, in dem einzelne Komponenten in anderen Pro-

jekten wiederverwendet werden sollen, beim Entwurf der Schnittstellen sicher nicht nur Minimierungsaspekte eine Rolle.
Desweiteren ist auch denkbar, dass der minimale Zugriffsmodifikator eines Elements dessen aus softwaretechnischer Sicht angemessene Zugriffsebene sogar überschreitet. Dies ist dann der Fall, wenn mangelhafte Kapselung bereits durch Klienten ausgenutzt wird.
Der minimale Zugriffsmodifikator kann daher nur als Annäherung für den „richtigen" Zugriffsmodifikator angesehen werden. Seine Bestimmung auf Basis einer abgeschlossenen Quelltextmenge (P) ergibt nur Sinn, wenn sich die Benutzung des zugehörigen Elements auf diese Quelltextmenge beschränkt. In einigen Fällen kann es sinnvoll sein, P auf den Quelltext verwandter Anwendungen zu erweitern.
Der Einsatz der Metriken IGAT und IGAM bietet sich daher vor allem in Projekten an, in denen solch eine abgeschlossene Quelltextmenge angenommen werden kann. Dies haben wir bei der Wahl der von uns untersuchten Anwendungen entsprechend berücksichtigt. In heutigen Systemen müssen neben dem reinen Java-Quelltext auch andere Dokumente bei der Analyse berücksichtigt werden, etwa JSPs oder Konfigurationsdateien von eingesetzten Frameworks. Sind diese Voraussetzungen gegeben, könnten die Metriken in individuellen Qualitätsmodellen als Maß für die Kapselung eingesetzt werden. Ob ein strengerer Umgang mit Zugriffsmodifikatoren tatsächlich Einfluss auf übergeordnete Qualitätsmerkmale wie Änderbarkeit oder Verständlichkeit hat, müsste jedoch empirisch noch nachgewiesen werden.
Im Rahmen des Refactorings können Quelltexteinheiten, die durch hohe IGAT- oder IGAM-Werte auffallen, einer gesonderten Überprüfung ihrer Schnittstellen unterzogen werden. Eine automatische Minimierung der Zugreifbarkeiten erscheint zwar grundsätzlich möglich, jedoch nicht angebracht. Wie dargestellt kann es viele Gründe für einen großzügigeren Zugriffsmodifikator als den minimalen geben. Der minimale Zugriffsmodifikator kann somit als nützlicher Orientierungspunkt dienen, letztlich sollte aber immer eine Entwicklerin oder ein Entwickler über die tatsächliche Zugreifbarkeit entscheiden.
Die Ergebnisse der Untersuchung zeigen, dass solch eine Überarbeitung der Schnittstellen in vielen Projekten angebracht erscheint. Dabei könnten vor allem Pakete als Abstraktionseinheit ein höheres Gewicht bekommen. Die Betrachtung von Paket-Schnittstellen ist nur dann nützlich, wenn Typen bewusst hinzugefügt oder ausgelassen werden. Solange alle Top-Level-Typen wahllos `public` deklariert werden, ist das Schnittstellen-Konzept auf Paketebene wertlos.

5 Verwandte Arbeiten und Ausblick

Der MOOD-Metriken-Katalog von Brito e Abreu [AC94] enthält zwei Metriken, die die Kapselung von Methoden und Feldern zum Gegenstand haben: „Method Hiding Factor" und „Attribute Hiding Factor". Diese geben den Anteil der nicht-öffentlichen Methoden bzw. Felder aller Klassen an. Cao und Zhu [CZ08] erweitern diese Metriken, indem sie die Anzahl der Klassen, die auf eine Methode oder Attribut Zugriff haben, mit einrechnen.
Bouillon et al. beschreiben in [BGS08] ein *Eclipse*-Plug-In, das ebenfalls minimale Zu-

griffsmodifikatoren ermitteln kann, allerdings nur von Methoden. Dieses berechnet keine Metriken, sondern bietet Werkzeug-Unterstützung für die Wahl der Zugriffsmodifikatoren während der Entwicklung.
Eine detailliertere Darstellung der hier vorgestellten Metriken IGAT und IGAM, ihrer Berechnung und der Bestimmung der minimalen Zugriffsmodifikatoren sowie dem *Access-Analysis*-Plug-In und der durchgeführten Fallstudien findet sich in [Zol10]. Darin wird auch ein alternativer Ansatz zur Berechnung der Metriken vorgestellt, bei dem der Anteil der Typen und Methoden mit zu großzügigem Zugriffsmodifikator nicht im Verhältnis zu allen Typen bzw. Methoden angegeben wird, sondern nur zu denjenigen, bei denen sich die Zugreifbarkeit überhaupt einschränken lässt, also zur Menge der Typen und Methoden, deren minimaler Zugriffsmodifikator strenger ist als `public`.

Um z.B. auch JSPs und Konfigurationsdateien von Frameworks untersuchen zu können, müsste *AccessAnalysis* um einen Erweiterungsmöglichkeit ergänzt werden, so dass weitere Analyse-Module als Plug-In hinzugefügt werden könnten.
Eine Übertragung der hier vorgestellten Konzepte auf andere objektorientierte Programmiersprachen wäre wünschenswert, ist aber nur unter bestimmten Voraussetzungen möglich. Zunächst muss das Typsystem der Sprache die vollständige statische Analyse der Benutztbeziehungen zulassen. Darüber hinaus beruht das Prinzip des minimalen Zugriffsmodifikators auf der klaren hierarchischen Gliederung der Zugriffsebenen in Java und der sich daraus ergebenden Ordnung der Zugriffsmodifikatoren. Solch eine Ordnung lässt sich nicht in allen objektorientierten Sprachen ableiten. So gibt es beispielsweise in C# die orthogonalen Zugriffsebenen `internal` und `protected`. `internal` erlaubt den Zugriff innerhalb der gleichen Assembly und `protected` durch alle Unterklassen [ECM06, § 10.5]. Es bedürfte somit einer geeigneten Konvention, ob für Elemente, die nur durch Unterklassen innerhalb der gleichen Assembly benutzt werden, `internal` oder `protected` als minimaler Zugriffsmodifikator gilt.

6 Zusammenfassung

Wir haben in diesem Artikel das Konzept des minimalen Zugriffsmodifikators für Java vorgestellt und diskutiert. Der minimale Zugriffsmodifikator eines Typs oder einer Methode entspricht dem strengsten Zugriffsmodifikator, der ausreicht, um alle Benutzungen des Typs bzw. der Methode innerhalb des Quelltextes eines Systems zu erlauben. Darauf aufbauend haben wir die Metriken IGAT und IGAM als Maße der Übereinstimmung von minimalen und tatsächlichen Zugriffsmodifikatoren definiert. Die Bestimmung der minimalen Zugriffsmodifikatoren und die Berechnung der Metriken haben wir mit dem *Eclipse*-Plug-In *AccessAnalysis* realisiert.
Die Analyse von zwölf Open-Source-Projekten zeigte, dass oft großzügigere Zugreifbarkeiten gewährt werden als eigentlich notwendig. Der Einsatz in zwei kommerziellen Projekten machte allerdings deutlich, dass die Bestimmung der minimalen Zugriffsmodifikatoren in heutigen Systemen nur dann zuverlässig stattfinden kann, wenn neben dem eigentlich Java-Quelltext auch andere Dokumente wie JSPs oder Konfigurationsdateien von Frameworks berücksichtigt werden. Dennoch ließ sich auch hier erkennen, dass besonders

von der Kapselung von Typen innerhalb von Paketen, also der default-Zugreifbarkeit für Java-Klassen und -Interfaces, so gut wie nie Gebrauch gemacht wird.
Wir haben die Vor- und Nachteile einer hohen Kapselung abgewogen und festgestellt, dass das Konzept der Schnittstelle nur dann wertvoll ist, wenn mit Zugreifbarkeiten maßvoll umgegangen wird. Das Ziel der Minimalität ist dabei nicht in allen Fällen zu verfolgen, dennoch kann der minimale Zugriffsmodifikator als Orientierungspunkt für die Wahl des tatsächlichen Zugriffsmodifikators dienen.

Literatur

[AC94] Fernando Brito e Abreu und Rogério Carapuça. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In *Proceedings of the 4th International Conference on Software Quality (QSIC)*, McLean, VA, USA, 1994.

[BGS08] Philipp Bouillon, Eric Großkinsky und Friedrich Steimann. Controlling Accessibility in Agile Projects with the Access Modifier Modifier. In *Proceedings of TOOLS (46)*, Seiten 41–59, 2008.

[BK09] David J. Barnes und Michael Kölling. *Java lernen mit BlueJ – Eine Einführung in die objektorientierte Programmierung*. Pearson Studium, München, 4. Auflage, 2009.

[CZ08] Yong Cao und Qingxin Zhu. Improved Metrics for Encapsulation Based on Information Hiding. In *The 9th International Conference for Young Computer Scientists (ICYCS 2008)*, Seiten 742–747, Los Alamitos, USA, 2008. IEEE Computer Society.

[ECM06] ECMA International. *Standard ECMA-334 – C# Language Specification*. Genf, Schweiz, 4th edition. Auflage, June 2006.

[GJSB05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Upper Saddle River, USA, 3. Auflage, 2005.

[HC07] Cay S. Horstmann und Gary Cornell. *Core Java – Volume I: Fundamentals*. Prentice Hall, Upper Saddle River, USA, 8. Auflage, 2007.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition. Auflage, 1997.

[RSSW10] Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger. *Grundkurs Programmieren in Java*. Carl Hanser Verlag, München, 5. Auflage, 2010.

[Sav09] Walter Savitch. *Absolute Java*. Pearson Education International, Boston, USA, 4. Auflage, 2009.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, Seiten 38–45, New York, USA, 1986. ACM.

[Zol10] Christian Zoller. Ein Ansatz zur Messung der Kapselung in Java-Systemen. Diplomarbeit, Universität Hamburg, 2010.