Bachelorarbeit

Untersuchung von öffentlichen Java-Schnittstellen auf öffentlich nicht sichtbare Typen – Fallstudie

Mathias Weber

25. Januar 2010

TU Kaiserslautern
Fachbereich Informatik

Betreuer:
Prof. Dr. Arnd Poetzsch-Heffter
M. Sc. Yannick Welsch

Erklärung

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit mit dem Thema

„Untersuchung von öffentlichen Java-Schnittstellen auf öffentlich nicht sichtbare Typen – Fallstudie“

selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum)	(Unterschrift)

Untersuchung von öffentlichen Java-Schnittstellen auf öffentlich nicht sichtbare Typen – Fallstudie

Mathias Weber

25. Januar 2010

Zusammenfassung

Öffentliche Schnittstellen spielen im Software-Engineering eine wichtige Rolle. Mit ihrer Hilfe können abstrakte Datentypen und Komponenten erzeugt werden, die einfach zu benutzen sind, deren interne Implementierung jedoch geschützt ist. Werden jedoch öffentlich nicht sichtbare Typen in diesen Schnittstellen verwendet, so sind diese unter Umständen nicht so zu verwenden, wie in der Dokumentation angegeben. Dies kann zu falscher Verwendung einer Bibliothek führen und den Benutzer der Bibliothek verwirren. In dieser Ausarbeitung gehe ich auf die Verwendung von öffentlich nicht sichtbaren Typen in öffentlichen Schnittstellen in Java ein. Ich werde Kategorien für die verschiedenen Fälle einführen und mein Tool vorstellen, das die Untersuchung anhand von Jar-Archiven durchführt. In einer Fallstudie zeige ich, welche Funde in benutzten Bibliotheken vorkommt und dass die Verwendung von öffentlich nicht sichtbaren Typen in öffentlichen Schnittstellen vermieden werden kann. Am Ende diskutiere ich, ob die Menge an Kombinationen der Sichtbarkeit von Elementen in Java nicht für viele Programmierer zu kompliziert sein könnte.

Abstract

Public interfaces are very important in software engineering. They are needed to design abstract datatypes and components, which are easy to use and whose internal representation is protected. But if public not accessible types are used in these interfaces, they can probably not be used as stated in the documentation. This can lead to wrong usage of a library and the user of this library can get confused. In this paper I address the usage of public not accessible types in public interfaces. I will introduce categories of finds and my tool, which analyzes Jar-archives. In a case study, I will show which finds I have made in used libraries and that the usage of public not accessible types in public interfaces can be avoided. In the end I discuss if there are to many combinations of visibility modifiers, so it could be to complex for programmers to handle.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Bei der Entwicklung von Software kommt es je nach Größe auf unterschiedliche Aspekte an. Bei kleinen Tools für den einmaligen Gebrauch wird man nicht so sehr auf die öffentliche Schnittstelle achten, sondern mehr auf die Funktionalität. Die Wartbarkeit und Verständlichkeit des Quellcodes spielen nur eine untergeordnete Rolle. Bei der Softwareentwicklung im Großen spielen eben jene Aspekte jedoch eine übergeordnete Rolle. Geringe Kopplung werden wichtig, unabhängige Komponenten sollen entwickelt werden. Information Hiding ist ein wichtiges Mittel, um den Code robust zu machen. Diese Konzepte werden mit der öffentlichen Schnittstelle geregelt, die eine Klasse oder ein Paket besitzen.

Durch Einschränken der Sichtbarkeit von Methoden und Variablen auf ein Minimum soll sicher gestellt werden, dass die Komponente korrekt und sicher verwendet wird. Abstrakte Datentypen können erstellt werden, welche eine Schnittstelle nach außen zur Verfügung stellen, die interne Implementierung jedoch schützen. Eine Änderung der internen Implementierung hat somit keine Auswirkungen auf Programme, in denen die Datenstruktur verwendet wurde, soweit die Schnittstelle gleich bleibt. Bei allen diesen Konzepten stellt sich immer die Frage, ob eine Methode nach außen zugreifbar und verwendbar ist. Dabei besteht ein Unterschied zwischen zugreifbar und verwendbar. Nicht jede Methode die zugreifbar ist kann auch uneingeschränkt verwendet werden. In dieser Ausarbeitung werde ich statt von zugreifbar von sichtbar sprechen, um eine bessere Abgrenzung zur Verwendbarkeit zu erhalten.

Diese Arbeit beschäftigt sich mit öffentlichen Java-Schnittstellen und der Verwendung von öffentlich nicht sichtbaren Typen darin. Dabei geht es darum heraus zu finden, welchen Einfluss das Auftauchen von öffentlich nicht sichtbaren Typen auf die Verwendbarkeit von Methoden hat. Ich werde zeigen, dass es in vielen Fällen möglich ist, die Sichtbarkeit von Methoden und ihrer Parameter entsprechend an zu passen, sodass öffentlich nicht sichtbare Parameter in öffentlichen Schnittstellen vermieden werden. Somit werde ich zeigen, dass es mit relativ wenig Aufwand möglich ist, die Sichtbarkeit und die Verwendbarkeit von Methoden der öffentlichen Schnittstelle in Deckung zu bringen und somit besser verständliche Schnittstellen zu gestalten.

1.2 Java Typsystem

Java ist statisch typisiert. Jede Variable und jeder Ausdruck hat somit einen bestimmten statischen Typ, der schon zur Übersetzungszeit feststeht. Ein Typ kann entweder ein primitiver Typ, ein Klassentyp, ein Interface-Typ oder ein Array-Typ sein[4, Kapitel 4]. Die primitiven Typen sind in Java vordefiniert und unveränderlich, Klassen und Interfaces dagegen können vom Programmierer selber definiert werden. Eine Klasse kann aus Attributen, Methoden und inneren Klassen sowie inneren Interfaces bestehen. Dies sind die Elemente einer Klasse. In Java gibt es Modifikatoren, die Elemente einer Klasse vor Zugriffen schützen sollen. Unter anderem sind in Java folgende Zugriffsmodifikatoren vorhanden: *public*, *private*, *protected* und *default* oder *package*. Hierbei gilt folgende Konvention: Ist kein expliziter Zugriffsmodifikator angegeben, so gilt implizit die *default* oder auch *package* genannte Zugriffsbeschränkung.[4, Kapitel 6.6] Dabei gilt für die Sichtbarkeit eines Elementes E folgendes:

- Trägt E den *public* Modifikator, so ist es sowohl innerhalb als auch außerhalb des aktuellen Pakets voll sichtbar.
- Trägt E den *private* Modifikator, so ist es nur innerhalb der aktuellen Klasse sichtbar, nach außen hin ist es jedoch nie sichtbar.
- Trägt E den *protected* Modifikator, so ist es innerhalb des aktuellen Pakets sichtbar. Vereinfacht gesagt ist es außerhalb des Pakets innerhalb einer Subklasse sichtbar, sonst nicht. (Dies stellt eine Vereinfachung dar. Für genauere Informationen siehe [4, Kapitel 6.6.2])
- Trägt das Element keinen Modifikator, so trägt es automatisch den *default* Modifikator. Dieser schränkt die Sichtbarkeit auf das aktuelle Paket ein.

Des Weiteren sind geschachtelte Klassen in Java erlaubt. Das heißt, in einer Klasse können weitere Klassen deklariert werden. Es gibt mehrere Arten von verschachtelten Klassen (sei C die äußere Klasse, in der die Klasse D deklariert wurde)[4, Kapitel 8.1.3]:

- statische Klassen. D muss mit dem *static* Modifikator deklariert werden. Eine Nutzung der Klasse D ist auch ohne eine Instanz der Klasse C möglich.
  Ein Beispiel zeigt Abbildung 1 Zeilen 7 und 21. Die Klasse E wird in Zeile 7 als statische Klasse der Klasse C definiert. Die Instanziierung sieht man in Zeile 21.
- anonyme innere Klassen. D ist hierbei aus der Superklasse S entstanden, indem die Unterschiede zu S an der Stelle der Instanziierung eingeflochten wurden. D kann von außen nicht verwendet werden.

Ein Beispiel zeigt Abbildung 1 Zeilen 1 bis 5 und 10 bis 14. In den Zeilen 1 bis 5 wird die Klasse S definiert. In den Zeilen 10 bis 14 erbt die anonyme Klasse von der Klasse S und überschreibt die Methode *S.doit()*.

- lokale Klassen. D wurde lokal in einer Methode deklariert und kann von außen nicht direkt verwendet werden.
  Ein Beispiel zeigt Abbildung 1 Zeilen 16 bis 19. Hier wird innerhalb der Methode C.m() zunächst die Klasse L definiert und anschließend instanziiert. Die Klasse L ist hierbei nur innerhalb der Methode C.m() sichtbar.

- sonstige innere Klassen. D muss hierzu innerhalb der Klasse C, ohne static Modifikator deklariert sein und durch keinen der oberen Fälle abgedeckt sein. Eine Nutzung der Klasse D ist nur über eine Instanz der Klasse C möglich.
  Ein Beispiel zeigt Abbildung 1 Zeilen 7 sowie 24 und 25. Die Klasse D ist als innere Klasse der Klasse C definiert (Zeile 7). Die Instanziierung der Klasse D ist somit nur mit Hilfe einer Instanz der Klasse C möglich, da von innerhalb der Klasse D Zugriff auf alle Instanzvariablen der Klasse C besteht. Dies ist in den Zeilen 24 und 25 zu sehen.

Statische und nicht-statische innere Klassen können wie alle Elemente einer Klasse die vorgestellten Modifikatoren tragen. Ein Beispiel zeigt Abbildung 2:

In diesem Beispiel definiert die Klasse C die inneren Klassen I1, I2 und I3. Diese können Zugriffsmodifikatoren haben, wie man an dem Beispiel sieht. Äußere Klassen können nur entweder *public* oder default deklariert werden, sie können also nur entweder öffentlich sichtbar oder paketweit sichtbar sein. Innere Klassen dagegen werden wie die restlichen Elemente einer Klasse behandelt und können daher auch die Modifikatoren *private* und *protected* tragen.

Außerdem kann eine Klasse von einer anderen Klasse erben sowie eines oder mehrere Interfaces implementieren. Erbt eine Klasse von einer anderen so führt dies dazu, dass die neue Klasse bestimmte Elemente der Oberklasse übernimmt. Dabei gilt folgendes[4, Kapitel 8.2]:

- Elemente, die *private* deklariert sind, werden nie vererbt.
- Elemente mit *package* Modifikator werden nur an Klassen innerhalb des selben Pakets vererbt.
- *protected* und *public* Elemente werden auch an Klassen außerhalb des selben Pakets vererbt.

```
public class S {
    public void doit() {
        //tue etwas sinnvolles
    }
}
public class C {
    public class D {...}
    public static class E {...}
    public void m() {
        S s = new S() {
            public void doit() {
                //tue etwas anderes
            }
        };
        ...
        class L {
            ...
        }
        L myl = new L();
        ...
    }
}
...
    C c = new C();
    D d = new c.D();
    E d = new C.E();
...
```

Abbildung 1: Beispiel: Geschachtelte Klassen und ihre Verwendung

```
public class C {
    protected class I1 {}
    public class I2 extends I1 {}
    private class I3 {}
}
```

Abbildung 2: Beispiel: Innere Klassen

Außerdem ist die neue Klasse auch vom Typ der Superklasse. Da *java.lang.Object* keine Superklasse besitzt, somit also die Wurzel des Vererbungsbaumes darstellt, erbt jede Klasse, die von keiner anderen Klasse erbt, automatisch von *java.lang.Object*. Somit ist jede Klasse vom Typ *java.lang.Object*. Ein Subtyp kann an jeder Stelle verwendet werden, wo sein Supertyp erwartet wird.[4, Kapitel 8.1.4] Methoden, die von einer Superklasse übernommen wurden, können in einer Subklasse überschrieben werden. Wird nun die Methode mit diesem Namen auf einer Referenz mit statischem Typ der Superklasse, aber dynamischem Typ der Subklasse, aufgerufen wird statt der Methode der Superklasse die Methode der Subklasse aufgerufen. Ein Beispiel zeigt Abbildung 3.

```
public class C {
    public void m(){
        System.out.println("hello␣from␣C");
    }
}
public class D extends C {
    public void m() {
        System.out.println("hello␣from␣D");
   }
}
...
C c = new D();//erlaubt, da D Subtyp von C
c.m();
...
```

Abbildung 3: Beispiel: Überschreiben von Methoden

In diesem Beispiel wurde die Klasse C definiert, welche eine Methode m() beinhaltet. Die Klasse D erbt von C und somit auch die Methode m(). Diese wird jedoch in D überschrieben. In Klasse C gibt die Methode m() „hello from C“ auf die Konsole aus, in D gibt sie „hello from D“ aus. Im letzten Abschnitt wird eine Variable vom Typ C deklariert, welche mit einem neuen Objekt vom Typ D initialisiert wird. Dies ist erlaubt, da D ein Subtyp von C ist und somit jedes D dort verwendet werden kann, wo ein C erwartet wird. Bei Aufruf von c.m() wird nicht die Methode C.m() aufgerufen sondern D.m(), da D die Methode überschreibt und die Variable zur Laufzeit ein Objekt vom Typ D referenziert. Es wird folglich „hello from D“ auf die Konsole ausgegeben.

Methoden können Parameter haben, welche im Methodenrumpf verwendet werden können. Somit können den Methoden Daten zur Verarbeitung übergeben werden. Diese Parameter haben, wie Variablen auch, einen zur Übersetzungszeit festgelegten Typ. Im Prinzip können die Parameter einer Methode mit jedem an dieser Stelle sichtbaren Typ deklariert werden.

1.3 Nicht öffentliche Typen in öffentlichen Schnittstellen

In dieser Arbeit gehe ich auf eine interessante Möglichkeit der Kapselung in Java ein. Neben den bekannten Modifikatoren *public*, *protected* und *private* für Methoden und Klassen gibt es auch Möglichkeiten die Zugreifbarkeit zu steuern, die aus speziellen Kombinationen dieser Modifikatoren resultieren. Dabei lässt Java auch Kombinationen zu, die zu eventuell unerwarteten Ergebnissen führen. Hier betrachtet werden die folgenden drei Varianten:

Variante 1

```
package p;
public class A{
    public void m(B b) {...}
}
class B {...}
public class C extends B {...}

package q;
public class D extends A {
    //public void m(B b) {...}    //Fehler, da B nicht sichtbar
    public void m(C b) {...}      //überschreibt die Methode A.m()
    public void m(Object b) {...}//nicht, da anderer Parametertyp
}
```

Abbildung 4: Beispiel: Variante 1

Verwendung von öffentlich nicht sichtbaren Typen in als *public* markierten Methoden als Parameter. Hierbei werden als *protected*, *private* oder paketweit sichtbar deklarierte Typen in als *public* deklarierten Methoden verwendet. Diese Methoden müssen nicht unbedingt in einer öffentlich sichtbaren Klasse deklariert sein, sie können auch in einer Superklasse einer öffentlich sichtbaren Klasse deklariert und somit nach unten vererbt worden sein. Eine Subklasse erbt automatisch die kapselnden Methoden. Die Kapselungseigenschaft tritt jedoch erst zu Tage, wenn diese Methoden in einer Klasse eines anderen Pakets überschrieben werden soll oder versucht wird, die Methode aufzurufen. Dazu müsste als Parameter ein nicht sichtbarer Typ verwendet werden. Abbildung 4 zeigt ein Beispiel. Die Methode A.m() entspricht dem beschriebenen Muster. Der Versuch, in Zeile 10 die Methode in Klasse D zu überschreiben, schlägt fehl, da der Typ B in diesem Paket nicht sichtbar ist. Da in Java die Parameter-Invarianz implementiert ist und somit beim Versuch, als Parameter den nicht sichtbaren Typen durch eine Sub-

oder Superklasse zu ersetzten, eine neue Methode entsteht, jedoch keine Überschreibung stattfindet, ist ein Überschreiben der Methode nicht möglich. Dies zeigt Abbildung 4. Der Versuch in Zeile 11 und 12 die Methode A.m() zu überschreiben schlägt fehl, da durch diese Deklarationen die Methode nur überladen wird. Das heißt, die Methode kann zwar eventuell verwendet werden, soweit man den Typen des Parameters erreichen kann oder ein öffentlich sichtbarer Subtyp existiert, da hierbei auch eine Subklasse des deklarierten Typs übergeben werden kann, jedoch ist das Überschreiben außerhalb des Pakets ausgeschlossen.

Variante 2

```
package p;
public class A {
    protected void m(B b) {}
}
class B {}
```

Abbildung 5: Beispiel: Variante 2

Verwendung von öffentlich nicht sichtbaren Typen in als *protected* markierten Methoden einer *public* Klasse, die nicht als *final* deklariert ist. Hierbei entsteht eine ähnliche Problematik wie bei Variante 1. Wird eine als *protected* markierte Methode vererbt, so wird sie innerhalb der Subklasse damit sichtbar. Wird in öffentlich sichtbaren Methoden einer öffentlich sichtbaren Klasse Typen verwendet, die nicht öffentlich zugänglich sind, so kommt es ebenso zu dem oben beschriebenen Kapselungseffekt. Dazu muss es möglich sein, von der Klasse zu erben, sie darf also nicht final sein, da somit die Weitervererbung und somit auch das Überschreiben der Methode ausgeschlossen ist. Ist die Klasse jedoch nicht final deklariert kommt es wieder zu dem Effekt, dass der öffentlich nicht sichtbare Typ, der in der Deklaration der Methode verwendet wurde auch zum Überschreiben der Methode benötigt wird. Die Situation ist wieder die selbe wie in Variante 1.

Auch im Beispiel von Abbildung 5 ist ein Überschreiben der Methode A.m() nur innerhalb des Pakets p möglich. Außerhalb des Pakets ist der Typ B nicht sichtbar und verhindert somit das Überschreiben der Methode. Ebenso kann die Methode nicht aufgerufen werden, solange nicht ein sichtbarer Subtyp von B existiert oder B anderweitig erreichbar ist.

```
package p;
public abstract class A{
    abstract void m();
}

package q;
public class B extends A{
    //void m(){}  //Hier ist die Implementierung nicht moeglich,
                  //da die Methode nicht sichtbar ist
}
```

Abbildung 6: Beispiel: Variante 3

Variante 3

Verwendung von nur paketweit sichtbaren abstrakten Methoden in öffentlichen abstrakten Klassen. Diese Variante baut auf einem anderen Problem auf: Auch Methoden können als nur paketweit sichtbar deklariert werden. Somit ist klar, dass diese Methoden auch nur innerhalb des Pakets der deklarierenden Klasse überschrieben werden kann. Jedoch kann es dazu kommen, dass die ganze Klasse außerhalb des Pakets nicht mehr spezialisiert werden kann. Das ist dann der Fall, wenn die Klasse eine abstrakte Methode besitzt, selbst also auch abstrakt ist, und diese Methode als nur paketweit sichtbar deklariert ist, die Klasse selbst jedoch öffentlich sichtbar ist. Dies führt dazu, dass die Klasse prinzipiell zur Vererbung in anderen Paketen herangezogen werden kann, jedoch verhindert die nicht zugängliche abstrakte Methode das. Alle abstrakten Methoden einer Klasse müssen bei einer konkreten Klasse implementiert werden. Jedoch ist mindestens eine der Methoden, die in diesem Fall implementiert werden müssten, nicht sichtbar. Somit ist es auch nicht möglich, von der Klasse in einer konkreten Implementierung zu erben.

Wie im Beispiel von Abbildung 6 zu sehen erfüllt die Klasse A die oben beschriebenen Kapselungseigenschaften. Die Klasse B kann nicht von der Klasse A erben, da sie zwar für die Methode A.m() eine konkrete Implementierung angeben müsste, diese Methode aber nicht sichtbar ist.

Diesen drei Varianten gemeinsam ist, dass das Verhalten, das Java in diesen Fällen zeigt, nicht den Erwartungen entspricht. Man würde erwarten, dass bei Klassen, die sichtbar sind, es auch möglich ist, die Klasse und all ihre sichtbaren Methoden zu überschreiben. Dies ist, wie wir gesehen haben, nicht immer der Fall. Diese Spezialfälle ermöglichen es zwar, bei Kenntnis der oben beschriebenen Eigenschaften von Java, Methoden und Klas-

se zu schützen. In dieser Hinsicht könnten die beschriebenen Sachverhalte eine elegante Möglichkeit der Kapselung darstellen. Allerdings bietet es auch Spielraum für Fehler, zum Beispiel im Design, die sich erst sehr spät herausstellen.

1.4 Kategorien 1 bis 6

Wie oben schon erwähnt behandle ich hier drei Varianten von nicht öffentlichen Typen in öffentlichen Schnittstellen:

1. öffentlich nicht sichtbare Parameter in öffentlichen Methoden (Methode *public*, Klasse *public*)
2. öffentlich nicht sichtbare Parameter in *protected* Methoden (Klasse public und nicht final)
3. paketweit sichtbare abstrakte Methoden in öffentlichen Klassen

Diese Varianten habe ich nochmals unterteilt nach der Sichtbarkeit der Parameter der Methoden beziehungsweise der Konstruktoren. Die Typen der Parameter können jeweils *private*, *protected* oder *package* deklariert sein. Wären alle Parameter *public*, so wären bestimmte Kapselungseigenschaften nicht erfüllt. Variante drei zerfällt in zwei mögliche Fälle: Ist mindestens einer der Konstruktoren öffentlich zugänglich (also *public* oder *protected*) so tritt das unerwartete Verhalten auf, dass die Klasse nur auf Grund der nicht nach außen Sichtbaren Methoden nicht erweitert werden kann. Ist jedoch keiner der Konstruktoren öffentlich zugänglich, so ist die Klasse auf Grund dessen schon nicht erweiterbar und die Funde der kapselnden Methoden spielen nur noch eine untergeordnete Rolle. Somit ergeben sich folgende Kategorien:

1. Variante 1 mit *protected* Parameter
2. Variante 1 mit *package* Parameter
3. Variante 1 mit *private* Parameter
4. Variante 2 mit *protected* Parameter, der in einer anderen Klasse deklariert ist
5. Variante 2 mit *package* Parameter
6. Variante 2 mit *private* Parameter
7. Variante 3 mit öffentlich sichtbarem Konstruktor
8. Variante 3 mit öffentlich *nicht* sichtbarem Konstruktor

Wobei hier „Variante x mit y Parameter“, x aus {1,2}, y aus {*protected*, *package*, *private*} nichts anderes bedeutet, als dass der Typ des restriktivsten Parameters (das heißt der am wenigsten sichtbare) mit dem Modifikator y deklariert wurde. Hierbei ist ein Typ als *package* deklariert, wenn er keinen der Modifikatoren *public*, *private* oder *protected* trägt (siehe Einführung Kapitel 1.2).

Die Konsequenz aus der Bildung dieser Kategorien ist nun, dass der Modifikator der Methoden einer bestimmten Kategorie geändert werden kann, ohne dabei die Verwendbarkeit der Methoden einzuschränken. Dies folgt aus der Tatsache, dass die Methoden nur aufgerufen werden können, wenn Zugriff auf ihre Parameter besteht. Außerdem sind alle Methoden dieser Kategorien nicht überschreibbar.

Die folgenden Aussagen beziehen sich auf Klassen, die außerhalb des untersuchten Projektes erstellt werden, das heißt nicht innerhalb des selben Pakets wie die untersuchte Java-Klasse liegen.

Kategorie 1 In Kategorie 1 sind die Methoden selbst als *public* deklariert (von Variante 1). Mindestens ein Parameter ist jedoch als *protected* deklariert. Somit ist ein uneingeschränkter Zugriff von außen nicht möglich (außer, es handelt sich um einen bedingten Fund, siehe Kapitel 1.5). Die Methode kann nur in der Klasse und ihren Subklassen verwendet werden. Somit kann der Modifikator der Methode nach *protected* geändert werden, ohne dabei die Verwendbarkeit der Methode einzuschränken. Ein Beispiel ist in Abbildung 7 zu sehen.

```
package p;
public class C {
    protected class D {
        ...
    }
    public void m(D d) {...}
}
```

Abbildung 7: Beispiel: Kategorie 1

Kategorie 2 In Kategorie 2 sind die Methoden ebenfalls *public* deklariert. Mindestens ein Parameter ist jedoch als *package*, das heißt paketweit sichtbar, deklariert. Somit können Methoden dieser Kategorie nur innerhalb des selben Pakets sinnvoll genutzt werden. Diese könnten somit als *package* deklariert werden, ohne ihre Verwendbarkeit

zu ändern. Ein Beispiel erhält man, wenn man in Abbildung 7 den Modifikator der Klasse D weg lässt (und somit den impliziten *package* Modifikator verwendet).

Kategorie 3 In Kategorie 3 sind die Methoden ebenfalls als *public* deklariert. Mindestens ein Parameter ist jedoch als *private* deklariert. Somit sind Methoden dieser Kategorie nur innerhalb der deklarierenden Klasse sinnvoll verwendbar und könnten somit auch als *private* deklariert werden. Ein Beispiel erhält man, wenn man in Abbildung 7 den Modifikator der Klasse D von *protected* auf *private* ändert.

Man sieht also, dass alle Methoden der Variante 1 in ihrer Sichtbarkeit eingeschränkt werden könnten, ohne dabei die Verwendbarkeit, das heißt das Verhalten des Pakets, zu ändern.

Kategorie 4 Die Methoden der Kategorie 4 sind *protected* deklariert. Der restriktivste Modifikator der Parameter ist ebenfalls *protected*. Hier sind jedoch nur Parameter interessant, deren Typen nicht in der selben Klasse wie die Methode deklariert sind. Da *protected* auch die Eigenschaft besitzt, dass innerhalb des Pakets auf die Typen und Methoden zugegriffen werden kann, kann *protected* als eine erweiterte Art des *package* Modifikators angesehen werden. Es kann also den Fall geben, dass ein Parameter einer Methode C.m() des Typs 2 als *protected* deklariert ist, jedoch der Parameter nicht in C sondern in einer anderen Klasse D definiert wurde. Der Zugriff kann in dem Fall, dass C nicht eine Subklasse von D ist, nur von innerhalb des Pakets erfolgen, wie bei einem als *package* deklarierten Parameter. Diesen Fall deckt die Klasse 4 ab. Jedoch kommt dieser Fall nur selten vor, da er sehr harte Restriktionen stellt. Der Modifikator der Methoden der Klasse 4 könnte auf *package* geändert werden, ohne die Verwendbarkeit einzuschränken. Ein Beispiel sieht man in Abbildung 8.

```
package p;
public class C {
    protected void m(DInner d) {...}
}
public class D {
    protected class DInner {
        ...
    }
}
```

Abbildung 8: Beispiel: Kategorie 4

Kategorien 5 und 6 In Kategorie 5 sind die Methode *protected* deklariert, also innerhalb der Subklassen sichtbar. Mindestens einer der Parameter ist jedoch *package* deklariert, also nur paketweit sichtbar und zugreifbar. Somit könnten die Methoden ebenfalls *package* deklariert werden, ohne dabei die Verwendbarkeit zu ändern, da ein Zugriff auf die Methoden nur innerhalb des Pakets möglich ist. Ein Beispiel hierfür erhält man, in dem man in Abbildung 8 den Modifikator der inneren Klasse DInner weg lässt (und somit den impliziten Modifikator *package* verwendet).

Für Kategorie 6 gilt dies analog mit *private* statt *package*. Somit könnten die Methoden der Kategorie 6 als *private* deklariert werden, ohne die Verwendbarkeit zu beeinflussen. Ein Beispiel zur Kategorie 6 ist in Abbildung 9 zu sehen.

```
package p;
public class C {
    private class I {
        ...
    }
    public void m(I i) {
        ...
    }
}
```

Abbildung 9: Beispiel: Kategorie 6

1.5 Bedingte Funde

Bei der Einteilung der Methoden in die verschiedenen Kategorien anhand der Zugriffsmodifikatoren kann es zu Funden kommen, für die nicht alle Kapselungsregeln gelten. Dies ist immer dann der Fall, wenn es mindestens eine öffentlich sichtbare Methode gibt, die den scheinbar nicht zugänglichen Parameter zurück gibt. Ein Beispiel Abbildung 10.

```
package p;
class C {}
public class D {
    public C retC() {...}
    public void useC(C c) {...}
}
```

Abbildung 10: Beispiel: Bedingter Fund durch Rückgabe des Parameters

In diesem Beispiel könnte von außen ein Aufruf der Methode *D.useC()* erfolgen, indem die Rückgabe der Methode *D.retC()* weitergeleitet wird. Dies sähe dann so aus wie in Abbildung 11.

```
package q;
...
D d = new D();
d.useC(d.retC());
...
```

Abbildung 11: Beispiel: Verwendung bedingter Funde

Die Eigenschaft hierbei ist, dass die Rückgabe (in diesem Fall der Methode *retC()*) nicht zwischengespeichert werden kann, da der Typ nicht öffentlich zugänglich ist und somit auch keine Deklaration einer Variable mit diesem Typ (in diesem Fall C) möglich ist.

Ein weiterer Fall, in dem ein bedingter Fund vorkommt ist der Fall, wenn es eine öffentlich sichtbare Subklasse des öffentlich nicht sichtbaren Typs gibt. Ein Beispiel zeigt Abbildung 12.

```
package p;
class C{
    class InnerC{}
    public class InnerCPub extends InnerC{}
    ...
    public m(InnerC ic){}
}
```

Abbildung 12: Beispiel: Bedingter Fund durch Subklasse

In diesem Fall wird die Methode *m()* als Fund erkannt, da ihr Parameter mit *package* Modifikator deklariert ist, sie somit in die Klasse 2 fällt. Nach meiner Überlegung könnte somit die Methode selbst mit *package* Modifikator deklariert werden, ohne die Verwendung einzuschränken. Das ist hier jedoch nicht der Fall, da die Klasse *InnerCPub* existiert, die öffentlich zugänglich ist und als Vertreter von InnerC mit der Methode *m()* verwendet werden kann. Somit würde der *package* Modifikator die Methode soweit einschränken, dass sie nicht mehr von außen benutzbar wäre und somit auch nicht mit dem Typ *InnerCPub* verwendet werden kann. Hierbei sei darauf hingewiesen, dass die Untersuchung des EncapsAnalyzers nicht vollständig ist. Näheres im Kapitel 2.2 über das Tool.

Bei beiden Fällen von bedingten Funden handelt es sich um Funde, bei denen die Änderung des Modifikators der Methode eine Einschränkung bedeuten könnte. Nicht betroffen ist jedoch die Überschreibbarkeit der Methode, obwohl die Methode sehr wohl nutzbar ist, nicht gegeben ist. Somit ist bedingter Fund hier so anzusehen, dass das Urteil über

die Sichtbarkeit der Methode zu falschen Ergebnissen führen würde. Das Aussortieren und Verwalten dieser Funde ist im EncapsAnalyzer im Schritt drei bei der Verarbeitung der Methoden implementiert. Die bedingten Funde werden im Ergebnis angegeben, können aber von Funden mit voller Aussagekraft unterschieden werden. Mehr dazu im Kapitel 2.1 über das Tool EncapsLogAnalyzer.

Wie schon erkannt gelten für bedingte Funde nicht alle Eigenschaften, die nötig wären um die Zugreifbarkeit der entsprechenden Methoden einzuschränken. Ich werde sie trotzdem in die entsprechenden Kategorien einordnen mit dem Hinweis, dass die Funde nur bedingt aussagekräftig sind. Dies rührt aus der Tatsache, dass bedingte und normale Funde beide die Eigenschaft haben, dass sie ein Überschreiben verhindern und daher bei der Beurteilung dieses Sachverhalts zusammen betrachtet werden müssen.

1.6 Kategorien 7 und 8

Da für die Betrachtung der Funde der Kategorien 7 und 8 nicht die Methoden selbst ausschlaggebend sind, sondern die Klasse und ihre Konstruktoren, sollen diese anschließend betrachtet werden. In diesen beiden Kategorien gibt es keine bedingte Funde, da es nicht die Parameter der gefundenen Methoden sind, die die Kapselung hervorrufen, sondern alleine die Existenz der gefundenen Methoden.

Methoden der Kategorie 7 sind von Typ 3 (also abstrakt und nur paketweit sichtbar) und in einer öffentlich sichtbaren Klasse deklariert. Gleichzeitig hat diese Klasse auch einen öffentlich sichtbaren Konstruktor, von ihr kann also geerbt werden. Da die Methoden jedoch nach außen nicht sichtbar sind und somit ein Überschreiben verhindert wird könnte die Klasse eben so gut einen paketweit sichtbaren Konstruktor besitzen, ohne dabei die Vererbbarkeit oder den Zugriff zu verändern. Dies folgt aus der Tatsache, dass die betrachtete Methode abstrakt ist und somit in einer konkreten Implementierung überschrieben werden muss. Dies ist nicht möglich, da die Methode selbst nicht sichtbar ist. Um diesen Effekt aufzulösen ist es nötig,

1. entweder die Klasse selbst paketweit sichtbar zu machen, oder
2. ihr nur öffentlich nicht sichtbare Konstruktoren zu geben, oder
3. die Methoden, die als Funde auftauchen, öffentlich sichtbar zu machen.

Mit Möglichkeiten 1 und 2 würde ein Vererben nach außen komplett ausgeschlossen. Wobei hier zu beachten wäre, dass Möglichkeit 2 die Funde in die Kategorie 8 verschieben würde. Möglichkeit 3 würde einfach das öffentliche Interface der Klasse erweitern.

Somit könnte danach von der Klasse geerbt werden, da alle Methoden, die eine konkrete Implementierung verlangen, sichtbar sind.

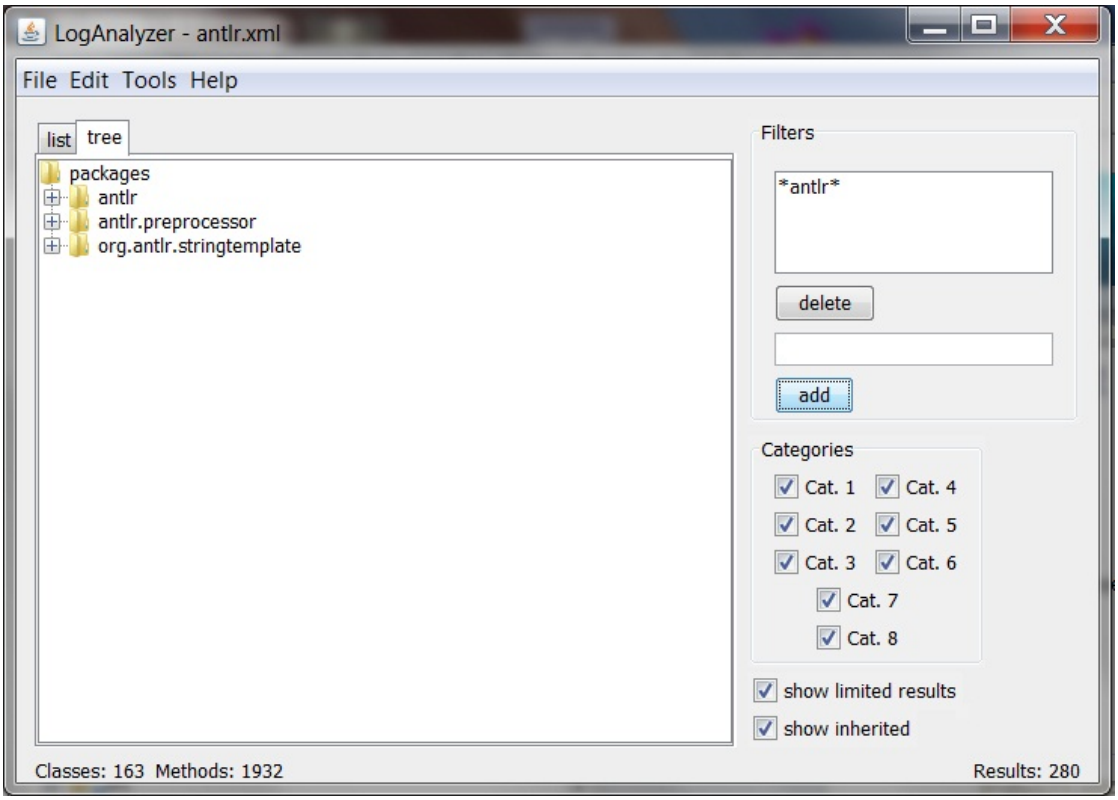
Bei Methoden der Kategorie 8 besitzt die deklarierende Klasse einen öffentlich nicht sichtbaren Konstruktor. Dadurch ist eine Vererbung nach außen von vorne herein ausgeschlossen. Der Modifikator der Methode könnte also von *package* auf *protected* geändert werden, ohne dass dabei ein verändertes Verhalten auftritt. Jedoch würde dadurch eventuell wahrscheinlicher, dass die betreffende Methode in der API-Dokumentation auftaucht, obwohl die Klasse selbst nicht überschrieben werden kann. Diese Kategorie ist die schwächste der hier untersuchten. Die Funde können meist so belassen werden, da durch eine Änderung keine weitere Einschränkung der Sichtbarkeit erfolgt und gleichzeitig eine Änderung des Sichtbarkeitsmodifikators zu einer Verschlechterung der Dokumentation führen kann.

Die hier untersuchten Kapselungseigenschaften beeinflussen also zum einen die Möglichkeit, Methoden in ihren Subklassen zu überschreiben. Zum anderen haben sie jedoch auch Einfluss darauf, ob die Modifikatoren der Methoden mit maximaler Restriktion gewählt wurden.

Die Frage, die sich jetzt stellt, ist, ob diese Kapselungen in realen und eingesetzten Programmen Anwendung findet, und ob diese Vorkommen nicht auf Design und andere Fehler zurück zu führen sind. Zur Untersuchung auf diese Phänomene reicht es allerdings nicht, die Schnittstellen jeder Klasse isoliert zu betrachten. Vielmehr ist eine weiträumige Untersuchung aller öffentlich zugänglicher Schnittstellen inklusive der Vererbungshierarchie und aller nicht öffentlich zugänglicher Typen erforderlich. Der erste Schritt in der Untersuchung dieser Frage ist also die Entwicklung eines Tools, das diese Aufgabe voll automatisch für große Mengen an Code übernehmen kann und anschließend die benötigten Informationen ausgibt.

2 Tool

2.1 GUI: EncapsLogAnalyzer



Die Kernkomponente EncapsAnalyzer ist für einfachere Benutzbarkeit in eine grafische Oberfläche gefasst. Diese nennt sich EncapsLogAnalyzer. Von ihr aus können Untersuchungen eines Ordners mitsamt seinen Unterordnern oder Untersuchungen einer oder mehrerer Jar-Archive gestartet werden. Die Ergebnisse werden in Listen- oder Baumform dargestellt und können über verschiedenen Kriterien gefiltert und aufbereitet werden.

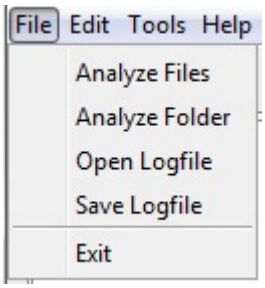


Abbildung 13: Das File-Menü

Menü Datei Über „File → Analyze folder“ kann ein Ordner angegeben werden, der rekursiv nach Jar-Dateien untersucht wird. Alle so gefundenen Java Archive werden in die Untersuchung mit einbezogen. So können komplette Projekte ohne manuelles suchen der Jar-Archive untersucht werden. Das Finden der Jar-Archive kann je nach Ordnerstruktur lange Zeit in Anspruch nehmen. Die zweite Option ist über „File → Analyze files“ abrufbar. In dem sich öffnenden Dialog können eine oder mehrere Jar-Dateien über Mehrfachselektion zur Untersuchung ausgewählt werden. So können zum Beispiel einzelne Teilkomponenten aus einem Projekt separat untersucht werden.

Die Ergebnisse der Untersuchung werden direkt aus der Komponente EncapsAnalyzer ausgelesen und die relevanten Informationen aufbereitet. Im Speziellen werden sowohl Informationen über die gefundenen Methoden, die die geforderten Kapselungseigenschaften besitzen, extrahiert, als auch Informationen über die Pakete, die untersucht wurden. Es stehen somit auch Informationen über die Anzahl der insgesamt untersuchten Klassen und Methoden zur Verfügung. Die so gewonnenen Daten können permanent auf die Festplatte gespeichert werden. Dies erfolgt über „File → Save logfile“. Die Ausgabe erfolgt in XML, welches alle aufbereiteten Informationen enthält, sodass nach dem Laden einer solchen Sicherung alle Informationen wiederhergestellt werden können, die nach der Untersuchung vorlagen. So kann die Untersuchung einmal durchgeführt werden und das Resultat abgespeichert und zur Archivierung, Vergleich oder paralleler Auswertung verwendet werden. Das Laden einer zuvor gespeicherten Log Datei erfolgt über „File → Open logfile“.

Ergebnisanzeige Die Anzeige der Analysedaten erfolgt sowohl in einer sortierten Liste, als auch in einer Baumdarstellung. Der Wechsel der Darstellung erfolgt durch Auswahl des Reiters *list* oder *tree* in der rechten oberen Ecke. In der Listendarstellung werden alle gefundenen Methoden sortiert untereinander dargestellt. In der Baumdarstellung werden die Funde zu Paketen zusammengefasst dargestellt. Dabei wird das Paket berücksichtigt, in dem sich die Klasse befindet, in der die Methode deklariert ist. In der Listendarstellung lassen sich schneller Muster erkennen, da die Methoden untereinander aufgelistet sind. So fällt es schnell auf wenn eine Methode sehr oft in der Liste auftaucht, da die Liste sortiert ist und somit diese Funde direkt untereinander stehen. In der Baumdarstellung lässt sich schneller ein Überblick über die Pakete mit Funden gewinnen. Diese Darstellung eignet sich sehr gut, um Pakete zu finden, die herausgefiltert werden sollten, um die Ergebnisse zu bereinigen. So sieht man hier sehr schnell Muster in den Paketnamen, die zum Beispiel auf interne Implementierungen hinweisen. Außerdem erkennt man so schnell, welche Pakete zu dem untersuchten Softwareprojekt gehören,

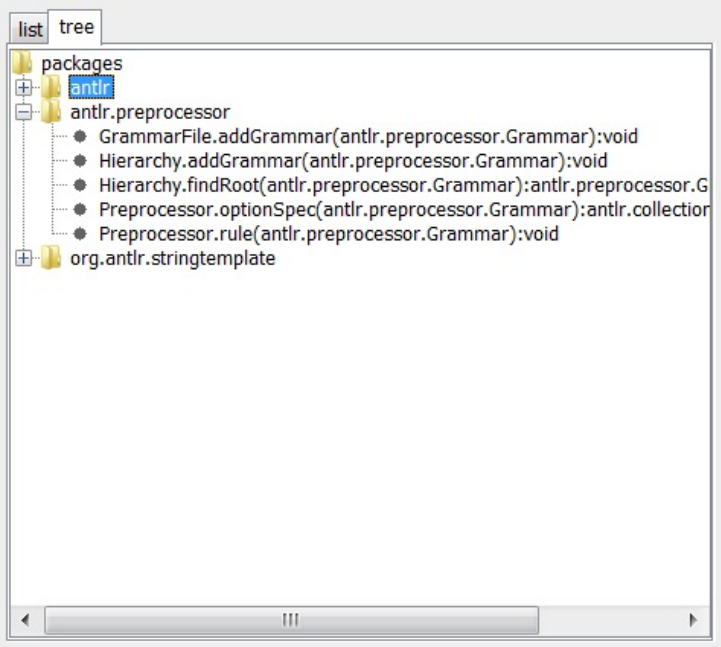


Abbildung 14: Die Ergebnisanzeige zeigt die Funde der Analyse (die gefundenen Methoden) an

und welche zu den Bibliotheken, die in dem Projekt verwendet wurden. So taucht bei der Untersuchung der freien Java Runtime Apache Harmony Funde auf, deren Paketnamen mit *org.apache.harmony* beginnen. In den anderen Funden taucht *harmony* nicht in den Paketnamen auf. Wir haben also ein Muster gefunden, welches uns die interessanten Funde gibt. Dieses können wir jetzt als Filter ansetzen (siehe weiter unten).

Mit einem Doppelklick auf einen Listen- oder Baumeintrag der eine Methode darstellt öffnet sich ein Fenster, in dem Details zu dem gewählten Eintrag aufgeführt sind. Zu den Details zählen die Kategorie der Kapselung, das Paket, in dem sich der Typ der Methode befindet, der Name des Typs, in dem die Methode deklariert wurde und der Name der Methode inklusive deren Parameter und Rückgabetyp, sowie Informationen über die nicht öffentlich sichtbaren Parameter. Hinter den Parametern wird in Klammern angegeben, welche Sichtbarkeit der Parameter besitzt. Dies ist vor allem interessant, wenn mehrere nicht öffentlich sichtbare Parameter auftauchen. So könnte eine Methoden in der Kategorie 3 (öffentliche Methode mit privatem Parameter) auftauchen, weil einer ihrer Parameter privat ist. Jedoch könnte die Methode gleichzeitig noch einen oder mehrere Parameter besitzen, die *protected* oder *package* ist.

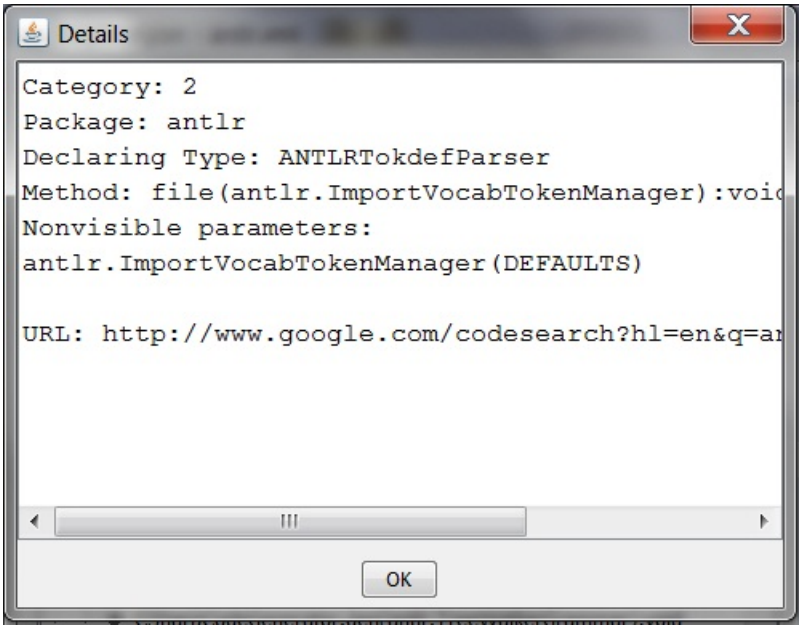


Abbildung 15: Der Details Dialog zeigt die Details der in der Ergebnisanzeige mit Doppelklick ausgewählten Methode an

Filter Auf der rechten Seite befinden sich verschiedene Filtermöglichkeiten. Im oberen Bereich können verschiedene Arten von Ausdrücken zu einer Liste von Filterausdrücken hinzugefügt werden:

Dabei gilt folgende Konvention:

- Ein * steht für beliebig viele beliebige Zeichen
- Steht kein * an erster Stelle, so wird von vorne ab geglichen
- Steht kein * am Ende, so muss das Paket mit dem Filterausdruck enden
- Beginnt der Ausdruck mit ^, so wird der Filterausdruck negiert.

Jeder Eintrag, der der Filterliste hinzugefügt wird wird mit UND verbunden. Ein ODER steht ebenfalls zur Verfügung und wird als (ausdruck1 | audruck2) dargestellt.

Beispiele:

Filter:

```
^*impl*
^*intern*
```

Entfernt alle Funde aus der Anzeige, die *impl* oder *internal* innerhalb des Paketnamens

tragen. Somit würden *org.eclipse.compare.internal* sowie *org.eclipse.core.internal.dtree* nicht mehr angezeigt, *org.eclipse.draw2d* ist jedoch weiterhin sichtbar.

Filter:

```
(com.sun*|sun*)
```

Zeigt nur Pakete an, deren Paketnamen mit *com.sun* oder *sun* beginnen. Alle anderen Pakete werden ausgeblendet.

Die Filterfunktion kann somit verwendet werden, um sich nur Pakete und deren Methoden anzeigen zu lassen, die zu dem eigentlichen Projekt gehören. Ein Filter von *org.netbeans** sorgt somit dafür, dass nur die Pakete von Netbeans berücksichtigt werden und nicht die in Netbeans verwendeten Bibliotheken, wie Apache-Derby und Ant. Weiterhin können somit auch implementierungsspezifische Pakete ausgefiltert werden, soweit diese einen speziellen Marker im Namen haben, wie dies in Eclipse der Fall ist. Alle internen Pakete haben dort die Form **internal**. Mit dem Filter ^**internal** werden somit die internen Klassen ignoriert.

Eine weitere Möglichkeit der Filterung findet sich im unteren Teil des rechten Bereichs. Dort finden sich Checkboxes für die verschiedenen Kategorien der Kapselung, wie sie in diesem Dokument definiert sind. Ist die jeweilige Checkbox selektiert, so werden die Methoden dieser Kategorie angezeigt, andernfalls werden die Methoden dieser Kategorie nicht angezeigt. Unterhalb der Auswahl der Kategorien befinden sich zwei weitere Checkboxes, die mit „show limited results“ und „show inherited“ gekennzeichnet sind. „show limited results“ dient dazu, die bedingten Funde anzuzeigen oder auszublenden. Diese wurden im Kapitel 1.5 besprochen. Möchte man Aussagen über die Einschränkbarkeit der Sichtbarkeit von Methoden treffen, können die bedingten Funde das Ergebnis verzerren, da bei ihnen keine Aussage über diesen Sachverhalt getroffen werden kann. Deshalb sollte man bei dieser Auswertung diese nicht aussagekräftigen Ergebnisse ausblenden. Möchte man jedoch eine Aussage darüber machen, welche Methoden von außen nicht mehr überschrieben werden können, so müssen auch die bedingten Ergebnisse berücksichtigt werden, da für sie diese Eigenschaft auch zutrifft.

Die Checkbox „show inherited“ dient dazu, die von den Superklassen geerbten Methoden, die nicht überschrieben werden, auszublenden. Es kommt öfters vor, dass in einer Basisklasse eine Referenzimplementierung einer Methode gemacht wird, die den Kapselungseigenschaften entspricht. Wird diese jedoch in den Subklassen nicht überschrieben taucht genau diese Methode in jeder Subklasse als Fund auf mit dem Hinweis, dass die Methode von einer anderen Klasse geerbt wurde. Für Untersuchungen einzelner Klasse

ist dies sehr hilfreich. Sollen jedoch nur quantitative Aussagen gemacht werden, also wie viele Methoden tatsächlich die Kapselungseigenschaften erfüllen, so kann eine große Zahl an Subklassen das Ergebnis in hohem Maße beeinflussen. In diesem Fall kann die Anzeige der geerbten Methoden ausgeschaltet und somit die Ergebnisse bereinigt werden.

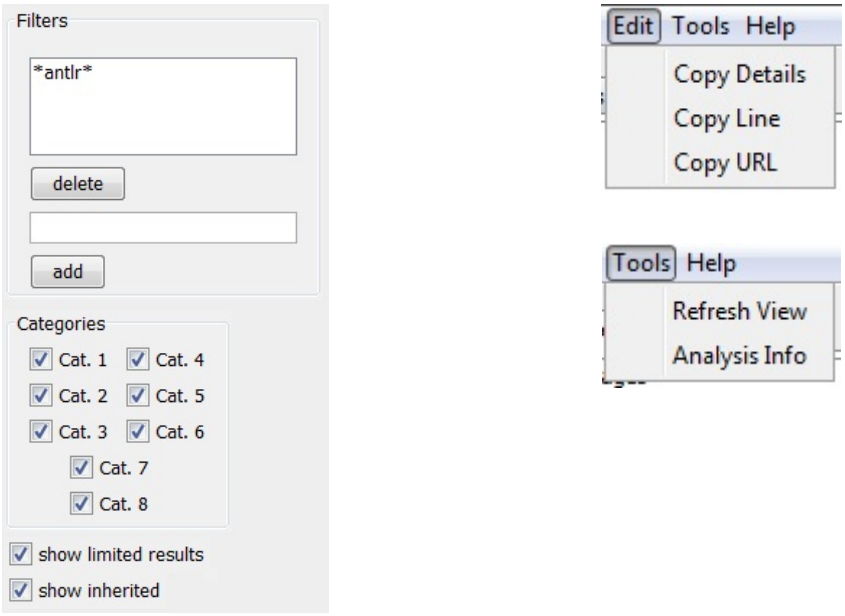


Abbildung 16: Das Filter Bedienfeld (links) dient dem Einstellen der Filterkriterien; Die-Menüs Edit und Tools (rechts)

Statusleiste In der unteren rechten Ecke wird die Anzahl der Funde gezeigt, die den aktuell gewählten Filtern entsprechen. In der linken unteren Ecke werden Statistiken zu den aktuell in der Baumansicht gezeigten Paketen angezeigt. Dies ist nicht die Anzahl der Klassen und Methoden, die in den Paketen untersucht wurden, die den Filtern entsprechen, sondern nur die Anzahl an Klassen und Methoden der Pakete, in der auch Funde entstanden sind. Im Menü gibt es auch eine Option, die eine komplette Statistik der aktuellen Untersuchung anzeigt. Diese besitzt auch die Anzahl der Methoden und Klassen, die dem aktuellen Paketfiltern entsprechen.

Menü Edit In der Menüleiste befindet sich ein Menü Edit. Ist in der der Listen- oder der Baumansicht eine Methode markiert, so kann über „Edit → copy details“ die Details zur Methode und über „Edit → copy line“ die Zeile, wie sie in der Listenansicht auftaucht, in die Zwischenablage kopiert werden. „Edit → copy url“ kopiert die URL zu

Google-Codesearch, womit der Quelltext der aktuell markierten Methode gesucht werden kann. Dies erleichtert bei fremdem Code das Auffinden des Quelltextes, um genauere Untersuchungen, die nicht automatisch durchgeführt werden können, zu machen.

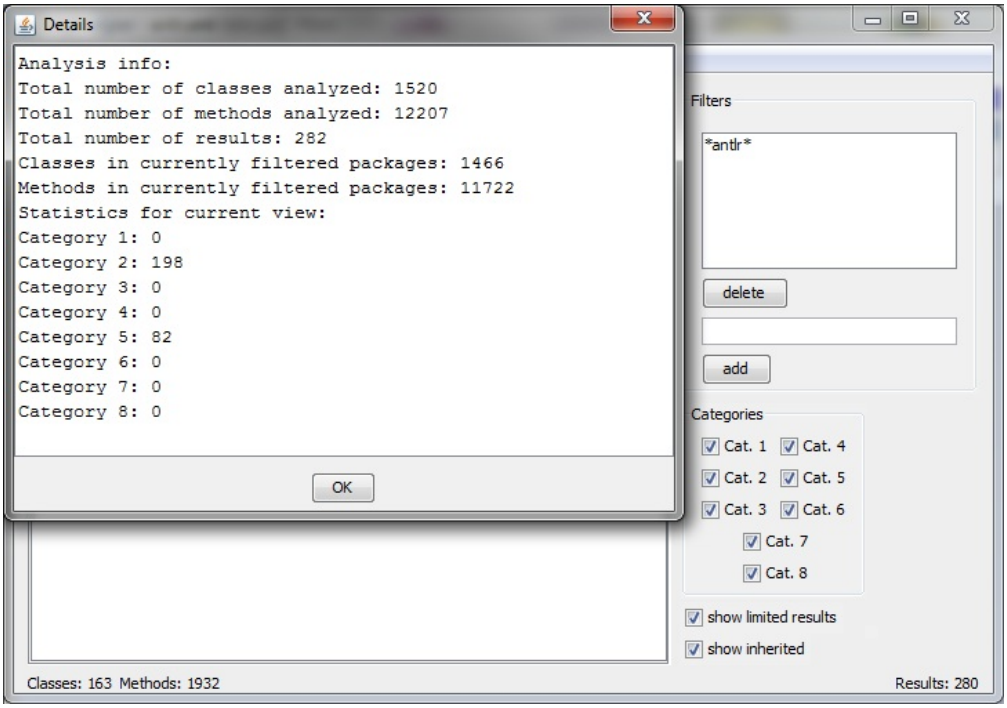


Abbildung 17: Die Analysis-Info zeigt allgemeine Informationen zu der aktuellen Analyse

Menü Tools Ein weiteres hilfreiches Feature ist „Tools → analysis info“. Hier werden einige Statistiken zur aktuellen Untersuchen aufgeführt. Darunter auch die Anzahl der gesamten untersuchten Methoden und Klassen, sowie die Gesamtzahl der Funde der aktuellen Analyse. Darunter befinden sich die Anzahlen der Klassen und Methoden, die sich in den gefilterten Paketen befinden. Hierbei werden die Filter, die unter anderem auch die Anzeige der Funde steuern, auch auf die Pakete angewendet und die Anzahl der Klassen und Methoden in diesen Paketen bestimmt. So können unerwünschte Pakete (wie zum Beispiel interne Pakete) aus der Statistik ausgeschlossen werden. Darunter befindet sich die Statistik für die aktuelle Anzeige. Hierbei werden die aktuell angezeigten Funde zu Kategorien zusammengefasst und gezählt. Dies ermöglicht einen schnellen Überblick über die Verteilung der Funde auf die Problemklassen.

2.2 Kernkomponente: EncapsAnalyzer

Da die Analyse von Programmen und Bibliotheken auf die in der Einleitung beschriebenen Eigenschaften zu aufwändig ist, um sie per Hand durchzuführen, untersucht dieses Tool den Code der Untersuchungsobjekte vollautomatisch und stellt die Ergebnisse zur Verfügung. Diese können dann von anderen Programmen ausgelesen, dargestellt und weiter ausgewertet werden, in diesem Fall von der Komponente EncapsLogAnalyzer. Die Kernkomponente bewertet dabei die Funde nicht. Die weitere Untersuchung, also die Bewertung, ob es sinnvoll oder notwendig ist, eine Änderung des Sourcecodes des Fundes vorzunehmen, ist dem Benutzer vorbehalten.

Da unter anderem auch recht große Bibliotheken und Programme untersucht werden sollen muss vor allem darauf geachtet werden, dass der Speicherverbrauch im Rahmen bleibt. Weiter wäre es vorteilhaft, wenn das Tool statt auf dem Sourcecode auf den im Bytecode vorliegenden .class-Dateien operieren könnte, um genauer zu sein, auf Java-Archiven, da praktisch alle Java-Programme in dieser Form verteilt werden. Zu diesem Zweck verwende ich in meinem Tool das ASM-Framework.

ASM-Framework dient dazu, Java-Klassen auf Bytecode-Ebene zu analysieren. Dabei können die Klassen als Byte-Array oder als Stream angegebenen werden. Die Analyse kann in zwei verschiedenen Modi erfolgen:

Der erste Modus ist die Tree API und ähnelt dem Ansatz des Dom-Toolkits zur Verarbeitung von XML-Dateien. Dabei wird die Klasse komplett eingelesen und aus den so gewonnen Informationen ein Baum erstellt. In ihm werden die Informationen der Klasse strukturiert dargestellt, Klassen sind Knoten, die als Unterknoten die Methoden haben. Diese Knoten beinhalten die Informationen über die Elemente, wie Zugriffsmodifikatoren, Name der Klasse oder Methode, die Signatur und Vieles mehr. Dabei sind sehr viele Informationen, die für die Analyse, die dieses Tool durchführt, nicht relevant sind. Dadurch würde sehr viel Overhead entstehen beim extrahieren der Informationen aus der Klassendatei, da alle Objekte, auch die, die für die Analyse unwichtige Informationen tragen, Speicher verbrauchen. Aus diesem Grund habe ich mich dazu entschlossen, in diesem Tool die Core API zu benutzen, die ohne diese Baumstruktur auskommt.

Der zweite Modus ist die Core API und ähnelt dem Ansatz des SaX-Toolkits. Dabei wird die Klassen-Datei sequenziell abgearbeitet und beim Erreichen der verschiedenen Komponenten der Klasse, zum Beispiel Annotationen, Methoden, Attribute und Codestellen, signalisierende Methoden aufgerufen, in denen die Analyse erfolgen kann. Als Parameter

erhalten diese Signalisierungsmethoden die Eigenschaften der untersuchten Komponente. Bei Klassen ist dies zum Beispiel deren Version, deren Zugriffsmodifikatoren wie *public* oder *private*, der Name der Klasse, deren Signatur, die Superklasse und deren implementierte Interfaces. So können die relevanten Informationen erfasst und weiterverarbeitet werden, ohne eine speicherraubende Baumstruktur mit für die Analyse nicht relevanter Daten im Speicher halten zu müssen. Eine effiziente Baumstruktur mit den wichtigen Informationen baut mein Tool während der sequenziellen Analyse selbst auf.

Beiden Ansätzen gemeinsam ist jedoch, dass die Vererbungshierarchie nicht berücksichtigt wird. Das heißt, von einer Superklasse geerbte Methode erscheinen nicht als Methoden in der Unterklasse. Es werden nur Methoden erfasst, die in der Klassendatei selbst deklariert sind. Das heißt, die vererbten Methoden müssen separat untersucht werden, indem die Vererbungshierarchie als Baumstruktur aufgebaut wird und danach die relevanten Methoden den Unterklassen hinzugefügt werden.

Öffentlich sichtbare Klassen Interessant für die Analyse sind jedoch nur die Klassen, die öffentlich sichtbar sind, und deren Superklassen. Die öffentlich sichtbaren Klassen sind jedoch nicht auf Anhieb zu erkennen und müssen zuerst gefunden werden. Die Analyse teilt sich also in drei Prozesse auf:

1. Analyse aller Klassen und Einteilung in öffentlich sichtbar und nicht öffentlich sichtbar
2. Aufbauen der Vererbungshierarchie durch Analyse der öffentlich sichtbaren Klassen und deren Superklassen
3. Weiterleiten der vererbten Methoden an die Subklassen innerhalb der Vererbungslinien und anschließende Auswertung

Zur Feststellung der öffentlich sichtbaren Klassen gibt es nun mehrere Möglichkeiten:

1. Die Festlegung kann während der Analyse der Klasse erfolgen. Dann müssen jedoch alle Methoden in den Baum aufgenommen werden, da noch nicht festgestellt werden kann, welche der Methoden die Kapselungseigenschaften erfüllen. Erst am Ende der Untersuchung stehen die öffentlich nicht sichtbaren Klassen fest und die Signatur der Methoden kann nach ihnen untersucht werden. Diese Methode ist jedoch in hohem Maße speicherintensiv.
2. Die Festlegung kann während der Untersuchung bei Bedarf erfolgen. Dabei wird die Analyse durchgeführt und während der Untersuchung der Methoden die in der Signatur auftauchenden Klassen untersucht. Ist der als Parameter auftauchende Typ

noch nicht untersucht, so wird die Untersuchung dieses Typs angestoßen. Dadurch können zwar eventuell öffentlich nicht sichtbare Klassen, von denen nicht geerbt wird und die ebenso nicht als Methodenparameter auftauchen, von der Analyse ausgeschlossen werden, jedoch ist dieser Spezialfall vermutlich selten. Dies würde bedeuten, dass diese nicht untersuchte Klasse nach außen hin nicht verwendet werden kann. Durch diese Art der Untersuchung kann es jedoch vorkommen, dass Klassen mehrmals in den Jar-Dateien gesucht werden müssen, einmal zur Überprüfung auf öffentliche Sichtbarkeit und ein zweites mal zur Untersuchung auf kapselnde Methoden. Außerdem ist diese Methode die komplexeste, da vieles in einem Schritt erledigt werden muss.

3. Die Festlegung kann vor der Untersuchung der Methoden erfolgen. Dabei werden zunächst alle Klassen in die Kategorien öffentlich sichtbar und nicht öffentlich sichtbar eingeteilt. Bei der ersten Untersuchung werden alle Jar-Dateien durchgehend abgearbeitet. Da dadurch die Jar-Dateien sequenziell gelesen werden können ist diese Untersuchung relativ effizient. Danach können die öffentlich sichtbaren Methoden und deren Superklassen auf Methoden mit den Kapselungseigenschaften untersucht werden. Die Parameter können direkt geprüft werden, da die Listen mit den öffentlich sichtbaren und den öffentlich nicht sichtbaren Typen schon vorliegen. Diese Methode scheint mir ein guter Kompromiss zwischen Speicherverbrauch und Geschwindigkeit zu sein. Deshalb habe ich diesen Ansatz in meinem Tool implementiert.

Spezialfall innere Klassen Bei der Untersuchung auf öffentlich sichtbare Klassen existiert ein Spezialfall: Innere Klassen. Bei den öffentlich nicht sichtbaren Typen muss noch unterschieden werden, welche Zugriffsmodifikatoren sie tragen, da anhand dieser Modifikatoren die Einteilung in die einzelnen Kategorien erfolgt. Die inneren Klassen müssen separat untersucht werden, da ihre Sichtbarkeit sich unter anderem auch nach der Sichtbarkeit der äußeren Klasse richtet. Der einzige Fall, bei dem die innere Klasse direkt eingeordnet werden kann ist der Fall, dass die innere Klasse selbst *private* ist. In diesem Fall ist die Sichtbarkeit immer *private*, da sie schon maximal eingeschränkt ist. Ansonsten muss die Schachtelung also von außen nach innen aufgelöst werden, um die tatsächliche Sichtbarkeit zu erhalten. Von einer inneren Klasse wird während der Analyse der Modifikator gelesen, der bei der Deklaration der Klasse verwendet wurde. Nach der kompletten Untersuchung liegen auch die Informationen für die äußeren Klassen vor. So gibt es nun mehrere denkbare Kombinationen, die zu verschiedener Sichtbarkeit führen:

Für die vorliegende Analyse stellt sich die Frage nach der Ordnung der Sichtbarkeiten. Klar ist: Die maximale Sichtbarkeit ist *public*, da diese Elemente von innerhalb wie von außerhalb des Pakets ohne Einschränkung zugänglich sind. Klar ist auch die minimale Sichtbarkeit, diese ist *private*, da die Elemente nur innerhalb der deklarierenden Klasse zugänglich sind, nach außen nicht. Dazwischen liegt *protected* und *package*. Für diese Untersuchung ist dabei *package* die größere Einschränkung, da wir die Elemente aus der Sicht von außerhalb des Pakets betrachten. Das bedeutet, dass für uns Elemente der Sichtbarkeit *package* nicht sichtbar sind, ähnlich wie *private*. Somit ergibt sich folgende Ordnung der Sichtbarkeiten:

$private < package < protected < public$

Für die Sichtbarkeit einer inneren Klasse ergeben sich somit folgende Fälle:

1. Die innere Klasse ist *public*
   Die Sichtbarkeit der inneren Klasse richtet sich jetzt voll nach der Sichtbarkeit der äußeren Klasse, da die innere nicht zugreifbarer sein kann als die äußere.
2. Die innere Klasse ist paketweit sichtbar (*package*)
   Die Sichtbarkeit der inneren Klasse kann in diesem Fall nie höher sein als paketweit sichtbar. Jedoch kann sie noch durch eine *private* äußere Klasse auf *private* eingeschränkt werden.
3. Die innere Klasse ist *protected*
   Die innere Klasse kann in diesem Fall nicht zugreifbarer werden als *protected*. Ist die äußere Klasse also *public*, so erhält die innere Klasse trotzdem *protected* Sichtbarkeit. Ansonsten richtet sie sich nach der Sichtbarkeit der äußeren Klasse.

Allgemein kann also festgestellt werden, dass die Sichtbarkeit einer inneren Klasse sowohl durch ihren eigenen Modifikator als auch den ihrer äußeren Klasse beschränkt wird und die reale Sichtbarkeit der restriktivsten dieser beiden Sichtbarkeiten entspricht.

Dieser Schritt kann jedoch nach gelagert werden, da die wichtigen Daten der inneren Klassen schon während des Durchlaufs gesammelt werden können und nur die Einordnung potenziell später erfolgen muss. Dies ist in dem Tool implementiert und findet bei der Untersuchung auf öffentlich sichtbare Klassen statt.

Es gibt die Möglichkeit, dass eine als *public* deklarierte innere Klasse einer nicht sichtbaren äußeren Klasse nachträglich sichtbar wird. Dies geschieht, wenn eine solche innere Klasse an eine öffentliche Subklasse vererbt wird. Dies kann jedoch nur nachvollzogen werden, wenn man die komplette Vererbungshierarchie betrachtet. Da dies in aller Regel

eine große Ausnahme darstellen wird und das Tool dadurch wesentlich komplexer werden würde habe ich auf die Implementierung dieser Untersuchung verzichtet.

Die eigentliche Suche Nach der Einteilung aller zu untersuchenden Klassen in öffentlich sichtbar und öffentlich nicht sichtbar sowie Feststellung der Modifikatoren der Typen erfolgt die Suche nach Methoden, die den gesuchten Mustern entsprechen. Zu Beginn der Untersuchung einer Klasse wird ein Knoten im Baum der Vererbungshierarchie erzeugt, der die Klasse darstellt. In ihm werden die Informationen gespeichert, die die Klasse betreffen. Darunter befindet sich auch die Liste der möglichen Funde. In diesem Schritt werden auch die noch zu untersuchenden Klassen identifiziert, diese sind die noch nicht untersuchten Superklassen der schon untersuchten Typen. Beim erreichen einer Methode innerhalb der aktuell untersuchten Klassendatei wird die entsprechende signalisierende Methode aufgerufen. Hierbei werden Informationen zu den Modifikatoren, der Name der Methode und deren Signatur übergeben. Die Signatur wird zerlegt und so die Typen der Parameter ermittelt. Da die Sichtbarkeit der Parameter im vorherigen Schritt festgestellt wurde und die Listen vorliegen kann nun untersucht werden, ob die aktuelle Methode einem der gesuchten Muster entspricht. Ist dies der Fall, so wird sie in die Liste möglichen Funde aufgenommen. Dies geschieht für jede Methode jeder Klasse die untersucht wird.

Bei der Untersuchung wurden Klassen gefunden, die noch untersucht werden müssen, da sie als Superklassen ebenso Methoden der gesuchten Muster enthalten können, die so an die sichtbaren Klassen vererbt wurden. Die Untersuchung wird mit diesen Typen fortgesetzt. Da am Ende jeder Vererbungshierarchie irgendwann die Klasse *java.lang.Object* steht endet die Untersuchung nach endlicher Zeit, ohne dass dabei neue Superklassen angefallen wären. In der Theorie würde also bei der Untersuchung immer genau ein einzelner Baum entstehen, der als Wurzel *java.lang.Object* hat. In der Praxis wird sich die Untersuchung jedoch oft mit einer Teilmenge der gesamten Typen befassen. Typen, die nicht in dieser Teilmenge vorkommen, von denen jedoch geerbt wird, bleiben somit als Wurzeln zusätzlicher Bäume übrig. Jedoch endet die Untersuchung immer damit, dass es eine endliche Menge von Bäumen gibt, die die Vererbungshierarchie darstellen. Im nächsten Schritt werden die vererbten Methoden in der Vererbungshierarchie berücksichtigt. Dabei werden alle gefundenen Methoden in der Baumstruktur nach unten kopiert. Ist eine Methode mit der selben Signatur schon vorhanden, so ist diese durch überschreiben der Methode der Superklasse entstanden. Somit ist das Kopieren an dieser Stelle nicht nötig. In diesem Schritt werden auch die öffentlich sichtbaren Klassen festgestellt da diese es sind, aus denen zum Schluss die interessanten Methoden extrahiert werden können.

Als letzter Schritt werden die Methoden aus den öffentlich sichtbaren Klassen extrahiert und in die in einem vorherigen Kapitel festgelegten Kategorien eingeteilt. Dabei erfolgt auch die Feststellung, ob eine Methode ein beschränktes Ergebnis liefert (siehe Kapitel 1.5 über nicht sichtbare Typen in öffentlichen Schnittstellen), zum Beispiel weil ihre Parameter öffentlich sichtbare Subklassen besitzen oder die Typen von öffentlich sichtbaren Methoden zurückgeliefert werden. Außerdem werden in diesem Schritt Methoden ausgefiltert, die keine wirklichen Funde sind, jedoch erst aufgrund der jetzt vorhandenen Klassenhierarchie als solche falschen Treffer erkannt werden können. Dies ist zum Beispiel in der Kategorie 4 der Fall. Hier werden Methoden betrachtet, die *protected* sind und *protected* Parameter besitzen. Interessant sind jedoch nur die Methoden, deren Parameter nicht in der selben Klasse wie die Methode sichtbar sind. Dies kann jedoch bei einigen Parametern erst mit vorhandener Klassenhierarchie festgestellt werden. So können innere Klassen an Subklassen vererbt werden und somit diese Typen sichtbar werden. Diese Methoden werden ausgefiltert und nicht in die Ergebnisliste aufgenommen. Nach diesem Schritt liegt die komplette Liste der Ergebnisse der Analyse vor und kann weiter verarbeitet werden.

Bei der Untersuchung der Methoden existieren ebenfalls Spezialfälle, die separat betrachtet werden müssen. Zum einen werden Konstruktoren wie normale Methoden behandelt. Da sie allerdings nicht vererbt werden können kann es nicht zu einem ungewollten Verhalten kommen, selbst wenn sie nicht öffentlich sichtbare Typen als Parameter besitzen. Damit fallen sie nicht unter Variante 1 und 2. Da Konstruktoren nicht abstrakt sein können fällt auch Variante 3 als mögliche Kapselung weg. Somit können Konstruktoren während der Untersuchung komplett ignoriert werden. Diese erscheinen in der signalisierenden Methode mit Namen „<init>“. Beim Auftauchen eines Konstruktors wird die Methode vorzeitig beendet.

Ein weiterer Spezialfall sind Interfaces und die Deklaration ihrer Methoden. Da bei Java die Methoden aller Interfaces immer öffentlich sichtbar sind, auch ohne den Modifikator *public*, scheinen diese bei der Untersuchung nur paketweit sichtbar zu sein. Da all diese Methode auch abstrakt sind, das heißt keine Implementierung enthalten, fallen sie scheinbar unter Variante 3 (abstrakte Methoden in einer öffentlich sichtbaren abstrakten Klasse). Dass dies nicht der Fall ist ist klar. Somit wird intern die Definition der Variante 3 noch erweitert und Interfaces aus dieser Variante ausgeschlossen.

Performance Aspekte Da es bei der Implementierung des Tools vor allem auch darauf ankommt, große Datenmengen untersuchen zu können, ist eine wichtige Voraussetzung

der effiziente Umgang mit Speicher. Das bedeutet, nur die für die Untersuchung wichtigen Daten abzuspeichern und die Rohdaten der Klassendateien nur so lange wie nötig im Speicher zu halten. Der erste Durchlauf speichert die öffentlich sichtbaren und öffentlich nicht sichtbaren Typen in HashMap's von Typnamen auf die Typ Informationen ab. Das stellt die kompakte Speicherung der Daten sicher, da nur die benötigten Typ Informationen gespeichert werden. Die HashMap's ermöglichen außerdem einen schnellen Zugriff und vor allem eine schnelle Prüfung auf vorhanden sein eines Eintrages. Dadurch wird die Überprüfung der Methodenparameter, die in der anschließenden Analyse erfolgt, zu einer Überprüfung mit fast konstanter Zeit pro Parameter. Dadurch hat das Tool auch in der Laufzeit ein gutes Verhalten. Das äußert sich unter anderem darin, dass die Untersuchungsdauer fast linear in Bezug auf die Anzahl der Untersuchten Methoden skaliert.

Die Erkennung von bedingten Funden ist nicht ganz vollständig möglich. So ist ein Funde unter anderem bedingt, wenn eine öffentlich sichtbare Subklasse existiert. Ist die Subklasse als *public* deklariert, so wird der bedingte Fund erkannt, da die Vererbungshierarchie für diese Java-Klassen vorliegt. Die komplette Vererbungshierarchie mit *protected* Klassen liegt jedoch nicht vor, da diese einen enormen Mehrverbrauch an Speicher zur Folge hätte. Somit kann zum Beispiel nicht erkannt werden, dass eine als *protected* deklarierte Subklasse existiert und der fragliche Parameter somit auch in Subklassen sichtbar ist. Jedoch scheint dies ein seltener Randfall zu sein weshalb auf die Untersuchung derartiger bedingter Funde zu Gunsten eines geringeren Speicherbrauchs verzichtet wurde.

Für die Untersuchung von großen Datenbeständen ist genügend Heapspeicher von Nöten. Es ist daher anzuraten, dem Programm mehr als die standardmäßigen 64MB zuzugestehen. Dies wird dadurch erreicht, dass das Hauptprogramm mit den folgenden Parametern gestartet wird:

```
java -Xms64m -Xmx256m -jar <Haupt-Jar-Archiv>
```

Dadurch wird dem Programm eine initiale Heapgröße von 64MB und eine maximale Heapgröße von 256MB zugewiesen. So gehen die meisten Untersuchungen zügig und ohne Speicherprobleme vonstatten.

Ich empfehle grundsätzlich, dem Analyzer ein wenig mehr Speicher zur Verfügung zu stellen, da dadurch die Laufzeit verringert werden kann. Dies trifft vor allem bei der Untersuchung größerer Projekte zu. Sobald der Analyzer bei der Untersuchung an die Speichergrenzen stößt, erhöht sich der Anteil der Speicherverwaltung an der CPU-Auslastung drastisch, was zu starker Verzögerung der Untersuchung oder gar zu deren Abbruch führen kann.

Fehlerprotokoll Sollten Fehler während der Analyse aufgetreten sein, so findet sich im Verzeichnis, in dem das Hauptprogramm gestartet wurde, eine Datei error.log. Ein wichtiger Fehler ist der Umstand, dass innere Klasse nicht zugeordnet werden können. Dies hat folgenden Hintergrund:

Während der Untersuchung werden Daten über alle in den Jar-Archiven befindlichen Bytecode-Dateien gesammelt. In ihnen befinden sich Informationen über die Java-Klassen, die in ihnen implementiert sind, über die verschachtelten Klassen (also Klassen, die innerhalb äußerer Klassen oder in Methode deklariert sind) aber auch Informationen über innere Klassen, die nicht direkt in der dazugehörigen Java Source Datei deklariert wurden. Manche dieser Informationen beziehen sich auf innere Klassen, deren äußere Klassen nicht untersucht wurden, da sie aus anderen Projekten stammen, deren Jar-Archive nicht bei der Untersuchung mit angegeben wurden. Somit können diese auch nicht in die Kategorien öffentlich zugänglich oder öffentlich nicht zugänglich eingeteilt und deren reale Sichtbarkeit festgestellt werden. Diese werden im Fehlerlog angegeben unter der Überschrift „Following classes could not be checked“. Während meiner Untersuchung ist mir durch das Fehlen dieser Informationen keine nachteilige Wirkung aufgefallen, da diese Klassen meist öffentlich sichtbar und zugänglich sind und somit keine der hier untersuchten Phänomene auslösen können.

3 Fallstudie

Bei der Fallstudie geht es nun darum herauszufinden, welche der Kategorien von Methoden in Bibliotheken vorkommen. Dabei versuche ich ein möglichst breites Anwendungsfeld zu überdecken.

Die vielleicht wichtigste Bibliothek ist das Java Runtime Environment selbst. Es ist die am häufigsten verwendete Java-Bibliothek überhaupt. Dabei kann man davon ausgehen, dass das JRE von Sun[11] als der de facto Standard gelten kann. Daneben habe ich auch die freie Implementierung Apache Harmony[14] überprüft.

Interessant sind Bibliotheken, die eine breite öffentliche Schnittstelle besitzen und weit verbreitet sind. Dies trifft unter anderem auch auf JUnit[5] zu, da es der Standard für Unit-Tests unter Java ist. Ebenfalls weit verbreitet ist ActiveMQ[13] im Bereich der asynchronen Kommunikation im Enterprise Bereich. Es implementiert die Java Message Service API[1] und wird unter anderem in Enterprise Service Bus (ESB) Implementierungen und SOA Infrastrukturen eingesetzt.

Des weiteren wurden Bibliotheken untersucht, die auf einem tieferen Level arbeiten und von Fachleuten erstellt wurden. Dazu zählen die beiden Java bytecode manipulation frameworks Apache BCEL[3] und ASM[8]. Diese Frameworks dienen dazu, Java bytecode, wie er nach dem Kompilieren der Sourcedateien entsteht, zu analysieren (ASM wurde unter anderem in dem hier vorgestellten Tool EncapsAnalyzer verwendet) und eventuell die Klassendateien auf bytecode-Ebene zu verändern.

Ein weiteres Framework, das wohl hauptsächlich von professionellen Softwareentwicklern erstellt und von professionellen Softwareentwicklern verwendet wird ist das Tool ANTLR[9]. Mit seiner Hilfe können kontextfreie Grammatiken geparsed und Syntaxbäume erzeugt werden. ANTLR kann Code in Java, und noch einigen anderen Sprachen generieren.

Ein bekanntes Framework, um eine Suche innerhalb einer Menge von Dokumenten zu realisieren ist das Framework Lucene[16] von der Apache Software Foundation. Dokumente können hierbei zur Analyse angegeben werden, woraus eine Indexstruktur erstellt wird. Mit Hilfe dieser Indexstruktur ist es danach möglich, eine effiziente Suche innerhalb der indexierten Dokumente durchzuführen. Diese Schritte sind in Lucene abstrahiert und zu einer einfach zu verwendenden API zusammengefasst.

Im Bereich der persistenten Speicherung von Daten in Java-Systemen wurde die beiden

[1] `http://java.sun.com/products/jms/`

Projekte OpenJPA[15] der Apache Software Foundation und Hibernate[10] untersucht. Zur Verarbeitung und Speicherung der Daten verwendet der Open Source EJB-Server OpenEJB standardmäßig OpenJPA, die Java Persistence API. Dabei handelt es sich um ein Object-Relational Mapping Framework. Java Objekte können dabei in eine relationale Datenbank gespeichert, in ihr gesucht und aus ihr gelesen werden, ohne dabei auf SQL oder eine andere relationale Zugriffssprache zurück greifen zu müssen. Ein mit OpenJPA vergleichbares Framework ist Hibernate, welches ebenfalls untersucht wurde. Hierbei ist Hibernate der noch weiter verbreitete Ableger der Gattung der Object-Relational Mapper für Java.

Eine andere Art der Speicherung der Daten wurde mit Apache Xalan[17] und Apache Xerces[18] untersucht. Hierbei handelt es sich um Frameworks zur Verarbeitung und Modifikation von XML-Dokumenten. Diese sind ein weiteres Mittel der persistenten Speicherung von Daten und ebenfalls weit verbreitet. Dabei handelt es sich bei Xalan um einen XSLT Processor, eine Möglichkeit, XML-Dateien zu transformieren, zum Beispiel durch ersetzen von bestimmten Tags durch gleichwertige Repräsentationen. Dadurch können unter anderem XML-Dateien eines Formats in ein nicht kompatibles anderes Format überführt werden oder eine Transformation von XML in HTML zur Anzeige erfolgen. Xerces ist ein XML parsing framework, womit XML-Dateien eingelesen und Datensätze ausgelesen werden können. Dabei kann zwischen einer Behandlung als Baum oder der sequenziellen Abarbeitung gewählt werden.

Zur Speicherung im PDF-Format können Java-Programmierer die Bibliothek iText[1] verwenden. Diese erlaubt es, einen großen Funktionsumfang von PDF auszunutzen und so Daten in benutzerlesbarer Form auszugeben.

Als Letztes wurde das Framework Tigris Graph Editing Framework (GEF)[19] untersucht. Mit Hilfe dieser Bibliothek kann auf einfache Art und Weise eine grafische Oberfläche erstellt werden, die in Eclipse integriert ist. Somit lassen sich relativ einfach grafische Plugins für das Eclipse Framework erstellen.

3.1 Untersuchung im Detail

Bei der Untersuchung wurde darauf geachtet, die Ergebnisse auf die Klassen der untersuchten Bibliothek beziehungsweise des untersuchten Programms zu beschränken. So wurden bei der Untersuchung von Netbeans nur die Klassen in den Paketen betrachtet, die mit „org.netbeans“ beginnen. Außerdem ausgeschlossen wurden Pakete, die als interne Implementierung zu erkennen waren. So wurden Pakete mit „impl“ oder „internal“

im Namen heraus gefiltert. Außerdem heraus gefiltert wurden die doppelten Ergebnisse welche dadurch zustande kommen, dass Methoden einer Superklasse an die Subklasse vererbt werden, ohne dabei überschrieben zu werden. Für die Betrachtung hier reicht es, sich die Methode in der Superklasse anzusehen.

3.1.1 Sun JRE vs. Harmony

Untersucht wurde in diesem Fall die rt.jar aus dem Java Development Kit Sun Java Version 1.6.0_16-b01 (Windows Version) und das komplette Apache Harmony 5.0M11 JDK 32bit for Windows mit Ausschluss der internen Implementierung. Dabei wurden insgesamt 189 Funde beim Sun JRE gemacht, 91 davon in der öffentlichen Schnittstelle (also nach Ausschluss der internen Pakete). Bei Apache Harmony wurden 55 Funde gemacht, davon keiner in der internen Implementierung.

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
JRE	82992	0	9	0	0	2	0	41	39
Harmony	48733	0	5	0	0	4	1	22	23

Vergleich zwischen JRE von Sun (nur rt.jar) und dem freien Harmony von Apache.

Da beides Implementierungen der Java Runtime sind würde man erwarten, dass die Ergebnisse sehr ähnlich sind, was auf den ersten Blick auch der Fall ist. Schaut man sich jedoch die Ergebnisse im Detail an, so sieht man, dass die Funde teilweise an völlig verschiedenen Stellen auftreten.

Ergebnisse der Kategorie 2:

Die beiden Implementierungen haben ein gemeinsames Ergebnisse: Einmal in *sun.awt.color* und einmal in *org.apache.harmony.awt.gl.color*. Beide beziehen sich auf CMM, das Color Management Module, zum konvertieren einer Farbe. Bei Sun heißt die Methode

```
CMM.cmmColorConvert(long, CMMImageLayout, CMMImageLayout):int
```

Bei Harmony heißt die Methode

```
NativeCMM.cmmTranslateColors(long, NativeImageFormat,
    NativeImageFormat):void
```

Dabei handelt es sich um native Methoden, das heißt Methoden, die eine C++ Funktion darstellen. Da es JNI erlaubt, Methoden als paketweit sichtbar zu deklarieren und eine

Implementierung in einer anderen Sprache wie C oder C++ zu schreiben wäre es in diesem Fall möglich, die Methoden als paketweit sichtbar zu deklarieren.

In Harmony jedoch ist auch die Methode ICC_Transform.cmmTranslateColors(...) ein Fund, da auch hier das *NativeImageFormat* verwendet wird. Auch diese Methoden könnte als paketweit sichtbar deklariert werden, da sie von außen nicht verwendet werden kann. In der Implementierung von Sun ist dies besser gelöst, hier wird mit öffentlich zugänglichen Formaten wie *BufferedImage* und short[] gearbeitet.

In der **Implementierung von Sun** gibt es im Paket *sun.awt.geom* in der Klasse *AreaOp* und ihren Subklassen *AreaOp$CAGOp, AreaOp$EOWindOp* und *AreaOp$NZWindOp* die Methode *classify(Edge):int*. Diese wird von den Subklassen (welche innere Klassen der Klasse *AreaOp* sind) jeweils überschrieben. Die Methode *classify()* wird nur in *AreaOp* und seinen Subklassen aufgerufen, wobei *Edge* nur paketweit sichtbar ist. Die minimale Sichtbarkeit, wobei die Verwendbarkeit nicht geändert wird, ist hiermit paketweit sichtbar, da von Außen nicht auf *Edge* zugegriffen werden kann.

```
sun.java2d.loops.GeneralRenderer.doDrawRect(PixelWriter,...):void
```

Alle Methoden der selben Gattung wie *doDrawLine()* und *doDrawPoly()* sind paketweit sichtbar. Die einzige Methoden, die öffentlich sichtbar ist, jedoch auch von außen nicht aufgerufen werden kann, da *PixelWriter* nicht sichtbar ist, ist *doDrawRect()*. Auch diese Methode könnte paketweit sichtbar gemacht werden, ohne dabei die Verwendbarkeit einzuschränken. Es scheint sich hierbei um eine Nachlässigkeit des Programmierers zu handeln.

```
sun.nio.ch.DatagramSocketAdaptor.create(DatagramChannelImpl):
    DatagramSocketAdaptor
sun.nio.ch.ServerSocketAdaptor.create(ServerChannelImpl):
    ServerSocketAdaptor
sun.nio.ch.SocketAdaptor.create(SocketChannelImpl):SocketAdaptor
```

Das Pattern der drei Methoden ist das selbe. Sie werden nur aus dem jeweiligen *ChannelImpl*-Klassen heraus aufgerufen, also zum Beispiel wird *SocketAdaptor.create()* nur aus *SocketChannelImpl* heraus aufgerufen. Somit ist es nicht sinnvoll, die Methoden als *public* zu deklarieren. Sie könnten ohne Einschränkung auch als paketweit sichtbar deklariert werden.

In Harmony treten ebenfalls weitere Funde auf. Einer davon ist

```
org.apache.harmony.pack200.Segment.pack(Archive$SegmentUnit,...):void
```

Die Methode *pack()* ist öffentlich sichtbar, wird jedoch nur in *Archive* verwendet, um den Packvorgang der *SegmentUnit*'s des Archives zu realisieren. Somit hat die Methode nach außen keine Bedeutung und könnte auf paketweit sichtbar geändert werden.

```
org.apache.harmony.tools.jarsigner.JSSigner.signJar(JSParameters):
    void
org.apache.harmony.tools.jarsigner.JSVerifier.verifyJar(JSParameters)
    :void
```

Die Methoden sind nicht implementiert, es steht nur ein leerer Methodenrumpf. Außerdem wird die Methode nur in der Main-Klasse des Pakets *org.apache.harmony.tools.jarsigner* benutzt und kann somit auch als paketweit sichtbar deklariert werden. Ob nach der Implementierung jedoch auch ein Aufruf von außen möglich sein soll ist nicht zu erkennen.

Wie man sieht unterscheiden sich die Probleme, die in Sun JRE und Apache Harmony gefunden wurden, in erheblichem Maße.

Auch die **Funde der Kategorie 5** unterscheiden sich komplett. Zuerst die Funde im **Sun JRE**:

```
javax.swing.text.ParagraphView.adjustRow(ParagraphView$Row, int, int)
    :void
```

Diese Methode hat einen leeren Methodenrumpf. Außerdem scheint bekannt zu sein, dass die Methode nicht sinnvoll oder veraltet ist. Zitat aus der Source-Datei ParagraphView.java[2]:

```
@specnote This method is specified to take a Row parameter, which is
    a private inner class of that class, which makes it unusable from
    application code. Also, this method seems to be replaced by {@link
     FlowStrategy#adjustRow(FlowView, int, int, int)}.
```

```
sun.awt.image.ImageDecoder.nextConsumer(ImageConsumerQueue):
    ImageConsumerQueue
```

[2] http://www.google.com/codesearch/p?hl=en#K93FVcDDrEc/libjava/classpath/javax/swing/text/ParagraphView.java&q=javax/swing/text/ParagraphView&sa=N&cd=1&ct=rc

ImageConsumerQueue ist paketweit sichtbar. Nach meinen Untersuchungen wird jedoch die Methode *nextConsumer()* nur innerhalb der Klasse *ImageDecoder* benutzt. Die Methode könnte also auch als *private* deklariert sein. Jedoch müsste zumindest erwogen werden, die Methode als paketweit sichtbar einzustufen, da ein Zugriff von außen wegen *ImageConsumerQueue* als Parameter ausgeschlossen werden kann.

Auch **in Harmony** existieren Funde der Kategorie 5:

```
java.awt.GridBagLayout.GetMinSize(Container, GridBagLayoutInfo):
    Dimension
java.awt.GridBagLayout.getMinSize(Container, GridBagLayoutInfo):
    Dimension
```

Die erste Methode entspricht nicht den Konventionen zur Namensgebung von Methoden in Java. Laut Java Language Specification[4] sollen Methoden mit einem kleinen Buchstaben beginnen. Dazu kommt, dass die Methode mit kleinem Anfangsbuchstaben die mit großem Anfangsbuchstaben einfach aufruft, um das Ergebnis zu ermitteln. Alles in Allem wirkt diese Konstellation recht unprofessionell. Noch dazu kommt die Tatsache, dass *GridBagLayoutInfo* nur paketweit sichtbar ist und somit die Methoden nur innerhalb des Pakets aufgerufen werden können. Somit könnten die Methode auch paketweit sichtbar sein.

```
javax.management.loading.MLet.check(String, URL, String, MLetContent)
    :URL
```

Die Methode gehört nicht direkt zu Harmony, sondern zu MX4J, einer freien Implementierung der Java Management Extension. Schon der Kommentar, der in der Source-Datei zu finden ist spricht das Problem an:

```
/**
* This method is called when an MLet file has been parsed and before
* its information is used by this MLet.
* By overriding this method subclasses have the possibility to
    perform
* caching and versioning of jars,
* but unfortunately it contains as parameter a package private class,
* that thus forbids overriding: a big
* mistake in the JMX 1.2 specification.
*
* @since JMX 1.2 */
```

Das Problem ist also bekannt und durch die Spezifikation der Java Management Extension Version 1.2 bedingt.

```
org.apache.harmony.lang.management.DynamicMXBeanImpl.
    getPresentAttribute(String,
DynamicMXBeanImpl$AttributeAccessType):MBeanAttributeInfo
```

Wie schon am Namen zu erkennen ist handelt es sich hierbei um eine Klasse der internen Implementierung. Somit wird eine Benutzer der Bibliothek nicht oder nur indirekt mit ihr in Kontakt kommen.

In **Harmony** ist Fund der **Kategorie 6** aufgetreten:

```
javax.swing.text.ParagraphView.adjustRow(ParagraphView$Row, int, int)
    :void
```

Hier scheint ein eindeutiger Designfehler vorzuliegen. In der Dokumentation steht folgendes:

```
/**
 * This method does nothing and is not supposed to be called.
 * The functionality described for this method in the API
     Specification
 * is equivalent to that of
 * @link FlowView.FlowStrategy#adjustRow(FlowView, int, int, int)
 */
```

Warum jedoch in der API Spezifikation eine Methode auftaucht, die in der Implementierung einfach ignoriert wird kann ich mir nur so erklären, dass der Designfehler in der Implentierungsphase entdeckt wurde, die Spezifikation jedoch nicht mehr geändert werden konnte.

Die **Funde der Kategorie 7** geben an, dass von der Klasse, in der die Funde gemacht wurden, nicht öffentlich geerbt werden kann. Somit betrachte ich hier ganze Klassen und nicht mehr einzelne Methoden.

Zunächst Funde im **JRE**:

```
com.sun.media.sound.AbstractMidiDeviceProvider
```

In dieser Klasse sind sieben Methode *abstract*, aber nur paketweit sichtbar.

```
createDevice(...), createInfo(...), getDeviceCache(...), getInfoCache
    (...), getNumDevices(), setDeviceCache(...), setInfoCache(...)
```

Die Klasse besitzt keinen explizit definierten Konstruktor, darum wurde ein Standard-Konstruktor, der *public* ist, hinzugefügt. Da die Methoden jedoch nicht überschrieben werden können könnte die Klasse auch zumindest einen paketweit sichtbaren Konstruktor definieren um anzuzeigen, dass von der Klasse nur innerhalb des Pakets geerbt werden kann. Alternativ könnten die Methoden *protected* gemacht werden, was es erlauben würde, von außen von der Klasse zu erben. In diesem Fall wäre es eher angebracht, der Klasse einen paketweit sichtbaren Konstruktor zu definieren und die Methoden danach *protected* zu machen, da die Klasse eine paketweit sichtbare innere Klasse *Info* besitzt und nur als Grundgerüst für *MidiInDeviceProvider* und *MidiOutDeviceProvider* dient.

```
sun.awt.im.InputMethodManager
```

Auch hier gibt es vier Methoden, die *abstract*, aber nur paketweit sichtbar sind.

```
findInputMethod(...), getDefaultKeyboardLocale(...),
    hasMultipleInputMethods(), setInputContext(...)
```

Die einzige Klasse, die von *InputMethodManager* erbt ist *ExecutableInputMethodManager*. Eine solche Instanz wird auch von *InputMethodManager.getInstance()* zurückgegeben. Es liegt also nahe auch hier einen paketweit sichtbaren Konstruktor zu definieren und die Methoden dafür *protected* zu machen. Somit würde die Verwendbarkeit beider Klassen nicht eingeschränkt.

```
sun.awt.image.InputStreamImageSource
```

In dieser Klasse gibt es nur genau eine Methode, die *abstract* und paketweit sichtbar ist.

```
checkSecurity(Object, boolean):boolean
```

Da die Klasse in keiner Dokumentation auftaucht scheint es sich hierbei um eine Klasse zu handeln, die nicht für die direkte Vererbung bestimmt ist. Dies legt die Maßnahme nahe, einen paketweit sichtbaren Konstruktor zu definieren und die Methode *checkSecurity(...)* als *protected* zu deklarieren.

```
sun.awt.image.VSyncedBSManager
```

Diese Klasse besitzt nur statische Methoden außer den Methoden

```
checkAllowed(...), relinquishVsync(...)
```

Sie besitzt jedoch keinen expliziten Konstruktor, weshalb die Klasse einen Standardkonstruktor erhält, welcher öffentlich zugänglich ist. Die Klasse hat zwei Subklassen, *NoLimitVSyncBSMgr* und *SingleVSyncedBSMgr*, beides innere private Klassen. Da dies die einzigen Subklassen sind und diese von außen nicht sichtbar sind und nur über die statischen Methoden zugreifbar sind, wäre es ratsam, einen expliziten privaten Konstruktor zu definieren. Die inneren Klassen könnten so immer noch erben, nach außen würde dafür kommuniziert werden, dass die Klassen abgeschlossen ist.

```
sun.font.Font2D
```

Durch die beiden Methoden

```
createStrike(...), getMapper()
```

ist es nicht möglich, öffentlich von dieser Klasse zu erben. Außerdem werden die Subklassen offensichtlich nur indirekt über die Klasse *FontManager* verwendet. Es wäre hier also eine gute Lösung, einen paketweit sichtbaren Konstruktor zu definieren und somit zu Kennzeichnen, dass die Klasse nur innerhalb des Pakets vererbt werden soll.

```
sun.font.FontStrike
```

Die Klasse *FontStrike* wird in Verbindung mit der Klasse *Font2D* verwendet. Auch hier gilt, dass die Klasse nur innerhalb des Pakets Subklassen hat und der Zugang nur indirekt über *Font2D* und somit über *FontManager* erfolgt. Somit ist es auch hier möglich, einen paketweit sichtbaren Konstruktor einzuführen und somit die Verwendbarkeit klar zu machen.

```
sun.font.PhysicalFont
```

Dies ist eine Erweiterung von *Font2D*. Sie wird also auch über die Klasse *FontManager* verwaltet und zur Verfügung gestellt. Somit kann auch hier die Klarheit durch einen

paketweit sichtbaren Konstruktor erhöht werden.

```
sun.management.MappedMXBeanType
```

Die möglichen Subklassen von *MappedMXBean* sind als innere Klassen in *MappedMXBean* deklariert. Die Klasse ist somit in sich abgeschlossen. Zugriff auf diese Subklassen ist nur über die öffentlich sichtbaren statischen Methoden möglich. Somit wäre es in diesem speziellen Fall möglich, einen privaten Konstruktor zu deklarieren, um den Umstand des Abschlusses auch nach außen sichtbar zu machen.

```
sun.management.Sensor
```

Die Methoden

```
clearAction(), triggerAction(), triggerAction(...)
```

sorgen durch ihre paketweite Sichtbarkeit dafür, dass die Klasse nur innerhalb des Pakets erweitert werden kann. Der einzige Konstruktor ist jedoch *public* deklariert. Dies steht im Widerspruch zueinander. Der Konstruktor kann in paketweit sichtbar geändert werden, ohne dadurch die Möglichkeiten einzuschränken.

Auch in **Apache Harmony** sind Funde der Kategorie 7 aufgetaucht.

```
javax.management.monitor.Monitor
```

Der Funde gehört nicht zum eigentlichen Harmony, sondern vielmehr zu MX4J[3], der Open Source Java Management Extension. Auch hier gilt wieder, die Klasse ist nur innerhalb des Pakets erweiterbar, somit kann ein paketweit sichtbarer Konstruktor definiert werden, um dies auch nach außen sichtbar zu machen.

```
org.apache.harmony.awt.gl.font.TextRunSegment
```

Die Klasse besitzt ausschließlich abstrakte Methoden, die alle paketweit sichtbar sind. Sie dient also als eine Art Interface Ersatz, um ein nur paketweit sichtbares Interface zu definieren. Daher verwundert es, dass die Klasse keinen paketweit sichtbaren Konstruktor besitzt. Dieser würde an der Erweiterbarkeit nichts ändern und die Sichtbarkeit nach außen klar hervorheben.

[3] `http://mx4j.sourceforge.net/`

```
org.apache.harmony.xnet.provider.jsse.HandshakeProtocol
```

Durch die Methoden

```
makeFinished(), receiveChangeCipherSpec()
```

ist die Klasse nur innerhalb des Pakets erweiterbar. Beide Methoden sind paketweit sichtbar. Die einzigen Erweiterungen der Klasse, die ich finden konnten, sind *ServerHandshakeImpl* und *ClientHandshakeImpl*. In beiden Fällen ist die Methode *makeFinished() protected* und die Methoden *receiveChangeCipherSpec() public* deklariert. Hier wäre also eine Möglichkeit, die Methoden in der Klasse *HandshakeProtocol* auch schon mit diesen Sichtbarkeiten zu belegen oder den Konstruktor der Klasse auf paketweit sichtbar zu ändern.

```
org.apache.harmony.xnet.provider.jsse.Message
```

Die Klasse hat zwei paketweit sichtbare Methoden:

```
getType(), send(...)
```

In allen gefundenen Subklassen haben die Methoden der Modifikator *public*. Somit könnte man die Modifikatoren auch schon in der Klasse *Message* einführen oder der Klasse einen paketweit sichtbaren Konstruktor hinzufügen, um nach außen zu zeigen, dass Vererbung nicht möglich ist.

Wenden wir uns der letzten Klasse von **Funden** zu, **der Kategorie 8**. Auch hier sind einige Funde im **Sun JRE** aufgetaucht.

```
java.awt.geom.Path2D
```

Path2D besitzt einen paketweit sichtbaren Konstruktor. Somit können nur Klassen innerhalb des selben Pakets von Path2D erben. Somit würde es ausreichen, wenn die Methoden

```
append(...), cloneCoordsDouble(...), cloneCoordsFloat(...),
getPoint(...), needRoom(...), pointCrossings(...), rectCrossings(...)
```

, die nur paketweit sichtbar sind, als *protected* deklariert werden würden.

```
java.nio.ByteBuffer
```

Die Methoden

```
get(int), put(int, byte)
```

sind paketweit sichtbar. Sie werden angeblich in der Klasse *ByteBufferAs-X-Buffer* verwendet. Jedoch habe ich in den Quellen[4], die ich gefunden habe, keinerlei Aufrufe dieser beiden Methoden registrieren können. Es stellt sich mir also die Frage, ob diese Methoden überhaupt noch Verwendung finden. Ein Deklarieren als *protected* würde die Methoden in der Dokumentation auftauchen lassen, ansonsten würde sich die Verwendbarkeit nicht ändern.

```
java.nio.CharBuffer
```

Die Methode

```
toString(int,int)
```

ist paketintern. Diese wird von der Klasse *CharBuffer* verwendet, um die Methode *toString()* zu implementieren. Um dieses Implementierungsdetail zu verbergen wäre es in diesem Fall nicht schlecht, die Methode so zu belassen wie sie ist, und mit dieser scheinbaren Anomalie zu leben. Die Methode *toString(int, int)* macht keine Probleme, da die Klasse *CharBuffer* wegen den nur paketweit sichtbaren Konstruktoren nur innerhalb des Pakets erweitert werden kann.

```
java.util.EnumSet
```

Die Methoden

```
addAll(), addRange(...), complement()
```

sind paketweit sichtbar. Da diese Methoden als internes Implementierungsdetail angesehen werden können und die Methoden nur in *RegularEnumSet* und *JumboEnumSet* überschrieben werden wäre es nicht schlecht, die Methoden paketweit sichtbar zu lassen. Durch die Einführung des Modifikators *protected* würde nicht mehr klar werden, dass die Methoden nur zum internen Gebrauch dienen und nach außen keine Bedeutung haben.

[4] `http://www.google.com/codesearch/p?hl=en#guzVIsuIArI/generated/java/nio/ByteBuffer.java&q=java/nio/ByteBuffer.java&d=3`

```
sun.net.www.protocol.http.AuthCacheValue
```

Außer der öffentlichen statischen Methode *setAuthCache(...)* befinden sich in der Klasse nur paketweit sichtbare abstrakte Methoden. Die Klasse kann man somit als Ersatz für ein Interface, jedoch mit nur paketweit sichtbaren Methoden ansehen. Somit wäre in diesem Fall das Umschreiben der Methoden nicht angebracht.

```
sun.security.provider.certpath.Builder
```

Diese Klasse besitzt nur paketweit sichtbare Methoden. Fünf davon sind abstrakt. Somit würde man mit dem Umschreiben der Methode auf *protected* nur die Unifomität zerstören, ohne dabei etwas zu gewinnen.

```
sun.security.validator.Validator
```

Die Klasse besitzt mehrere statische Methoden *getInstance(...)*. Mit ihrer Hilfe kann man an Instanzen der Subklassen gelangen. Die Menge der Subklassen ist somit von vorne herein beschränkt. Daher würde es nicht mehr Klarheit bringen, die Methode *engineValidate(...)* zu einer *protected* Methode umzuschreiben.

Ähnliche Argumentationen ließen sich auch für die restlichen Funde der Klasse 8 finden. Es müsste im Einzelfall vom Entwickler der Methoden beziehungsweise der Klassen entschieden werden, ob eine weitere Einschränkung oder eine Aufweichung des Zugriffsschutzes möglich wäre. Im Großen und Ganzen ziehe ich an dieser Stelle das Resümee, die Funde der Klasse 8 im JRE können unangetastet bleiben, da durch die Änderung der Zugriffsmodifikatoren von *package* in *protected* weder eine mögliche Einschränkung der Sichtbarkeit ausgenutzt werden kann, noch kann dadurch die Verwendbarkeit klarer ausgedrückt werden.

Auch in **Apache Harmony** sind Funde der Kategorie 8 aufgetreten. Diese lassen sich in zwei Gruppen einteilen: Die verschiedenen *Buffer* unter *java.nio* und die Klasse *java.util.EnumSet*.

Die Buffer

```
ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer,
    LongBuffer, ShortBuffer
```

im Paket *java.nio* habe folgende Methoden gemeinsam, die paketweit sichtbar und abstrakt sind:

```
protectedArray(), protectedArrayOffset(), protectedHasArray()
```

Wie die Namen schon andeuten sind die Methoden nur für interne Verwendung gedacht und werden von den verschiedenen möglichen Buffer-Implementierungen überschrieben. Da diese Methoden auf keinen Fall in der öffentlichen Dokumentation auftauchen sollen, sollten diese weiterhin als paketweit sichtbar deklariert werden.

```
java.util.EnumSet
```

Die Methoden

```
complement(), setRange(...)
```

sind paketweit sichtbar. Da die einzigen existierenden Unterklassen *MiniEnumSet* und *HugeEnumSet* beide im Paket implementiert sind und beide final sind liegt es nahe, die Methoden so zu belassen, da keine weitere Vererbung mehr möglich ist und somit die Methoden nicht überschrieben werden können. Die Verwendbarkeit muss somit nicht unbedingt klar gemacht werden, da von außen niemand auf die Methoden zugreifen kann.

Auch hier lässt sich argumentieren, dass keine weitere Einschränkung der Sichtbarkeit ohne Einbußen bei der Verwendbarkeit gemacht werden können. Da bei der Änderung der Modifikatoren von *package* auf *protected* die Wahrscheinlichkeit steigt, dass die Methoden in der öffentlichen Dokumentation auftauchen, sollte im Allgemeinen darauf verzichtet werden.

Im Ganzen betrachtet scheint mir die Implementierung des JRE von Sun an einigen Stellen ausgereifter, wenn auch an manchen Stellen noch Verbesserungen möglich wären. In Apache Harmony habe ich die eine oder andere Stelle gefunden, an der die Implementierung doch recht ungewöhnlich, wenn nicht sogar unprofessionell aussah. Die eklatanteste Stelle ist wohl *java.awt.GridBagLayout.GetMinSize(...)*, wo sogar die Java Coding Conventions[12, Kapitel Naming Conventions] verletzt wurden.

3.1.2 JUnit

Untersucht wurde in diesem Fall JUnit Version 4.7.

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
JUnit	966	0	2	0	0	0	0	0	0

Wie zu sehen ist wurden nur zwei Funde der Kategorie 2 gemacht. Diese sind wie folgt:

```
org.junit.runners.model.FrameworkField.isShadowedBy(FrameworkMember):
    boolean
```

FrameworkField erbt von der abstrakten Klasse FrameworkMember. Die Methode *isShadowedBy(...)* ist in FrameworkMember definiert und wird von FrameworkField überschrieben. Die Methode ist als *public* deklariert, kann jedoch verwendet werden, da FrameworkField eine öffentlich sichtbare Subklasse von FrameworkMember ist. Die Vererbung spielt in diesem Fall auch keine Rolle, da *FrameworkField* einen öffentlich nicht zugänglichen Konstruktor hat, wodurch eine Vererbung ausgeschlossen ist. Somit braucht die Sichtbarkeit der Methoden nicht weiter eingeschränkt zu werden.

```
org.junit.runners.model.FrameworkMethod.isShadowedBy(FrameworkMember)
    :boolean
```

Hier gilt das Gleiche wie bei dem vorherigen Fund. Jedoch mit einem Unterschied: Die Klasse besitzt einen öffentlich zugänglichen Konstruktor. Somit könnte von der Klasse geerbt werden. Das Beispiel von vorher zeigt jedoch, dass die FrameworkMember's nicht dazu gedacht sind, von ihnen zu erben. Somit wäre es hier das Beste, den Konstruktor von FrameworkMethod paketweit sichtbar zu machen, wodurch sich dieser Fund erledigen würde und Einheitlichkeit geschaffen würde.

3.1.3 ActiveMQ

Untersucht wurde in diesem Fall Apache ActiveMQ 5.3.0.

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
ActiveMQ	2067	0	3	0	0	3	0	0	0

Funde der Kategorie 2:

```
org.apache.activemq.filter.MultiExpressionEvaluator
```

Methoden:

```
addExpressionListner(Expression,
    MultiExpressionEvaluator$ExpressionListener):void
removeEventListner(String,MultiExpressionEvaluator$ExpressionListener
    ):boolean
```

Die Methoden dienen dazu, einen Listener an den Evaluator zu binden beziehungsweise ihn wieder zu lösen. Dabei ist der *ExpressionListener* eine inneres Interface, das nur paketweit sichtbar ist. Damit kann aber auch nur eine Klasse auf das Event hören, wenn sie innerhalb des Pakets deklariert ist, denn nur dann kann sie auch das Listener-Interface implementieren. Somit können die Methoden, die den Listener übernehmen, als paketweit sichtbar deklariert werden, ohne die Verwendbarkeit zu beeinträchtigen.

```
org.apache.activemq.jmdns.ServiceInfo.updateRecord(JmDNS, long,
    DNSRecord):void
```

Wie auch schon vom Entwickler so treffend festgestellt[5]:

```
// REMIND: Oops, this shouldn't be public!
```

Ich muss dem Entwickler zustimmen. Da die Klasse *DNSRecord* nur paketweit sichtbar ist, kann diese Methode auch paketweit sichtbar deklariert werden.

Funde der Kategorie 5:

```
org.apache.activemq.store.amq.reader.AMQReader
```

Methode:

```
getNextMessage(MessageLocation):MessageLocation
```

Die Methode übernimmt den paketweit sichtbaren Parameter *MessageLocation* und die Klasse *AMQReader* hat einen öffentlich sichtbaren Konstruktor.Somit könnte ein Programmierer auf die Idee kommen, die Methode überschreiben zu wollen. Da dies nicht möglich ist sollte die Methode besser als paketweit sichtbar deklariert werden, um diesen Umstand nach außen zu kommunizieren.

[5] http://www.google.com/codesearch/p?hl=en#coOE0tRomSs/trunk/activemq-jmdns_1.0/src/main/java/org/apache/activemq/jmdns/ServiceInfo.java&q=org/apache/activemq/jmdns/ServiceInfo&sa=N&cd=1&ct=rc

```
org.apache.activemq.transport.fanout.FanoutTransport
```

Methode:

```
restoreTransport(FanoutTransport$FanoutTransportHandler):void
```

Bei *FanoutTransport$FanoutTransportHandler* handelt es sich um einen paketweit sichtbare innere Klasse, nicht statisch deklariert ist. Es drängt sich mir daher der Gedanke auf, es könnte sich hierbei um einen Designfehler handeln. Sollte die Methode in allen Subklassen verwendbar sein, so müsste die innere Klasse *public* und besser auch statisch deklariert sein. Bei der jetzigen Lage kann man den Modifikator der Methode auf paketweite Sichtbarkeit ändern, ohne dass dadurch die Verwendbarkeit beeinträchtigt würde. Jedoch sollte man auf jeden Fall nochmals darüber nachdenken, die innere Klasse *FanoutTransportHandler* statisch zumachen.

```
org.apache.activemq.transport.stomp.ProtocolConverter.sendToActiveMQ(
    org.apache.activemq.command.Command, ResponseHandler):void
```

ResponseHandler ist zwar eine paketweit sichtbare Klasse, jedoch existiert in ProtocolConverter die *protected* Methode createResponseHandler(StompFrame), wobei StompFrame eine öffentlich sichtbare Klasse ist. Somit kann die Methode benutzt werden, da man durch die Methode createResponseHandler() an eine Instanz kommt. Jedoch kann man die Methode nicht überschreiben. Das wäre nur möglich, wenn die Klasse ResponseHandler öffentlich zugänglich wäre.

Auch hier wurde zumindest in einem Fall vom Entwickler erkannt, dass die Entscheidung für einen Modifikator (in diesem Fall *public*) nicht einschränkend genug war. In einem Fall ist es sehr wahrscheinlich, dass es sich um einen Designfehler handelt, in einem zweiten (add- und remove-Listener) kann ich es mir gut vorstellen.

3.1.4 BCEL & ASM

Untersucht wurde hier Apache BCEL Version 5.2 (Byte Code Engineering Library) und ASM Version 3.2. Zum Vergleich hier die ermittelten Werte:

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
BCEL	2628	0	0	0	0	0	0	0	0
ASM	1323	0	0	0	0	0	0	0	0

Wie man sieht, wurden weder in BCEL noch in ASM Methoden gefunden, die den geforderten Kriterien entsprechen. Das ist nicht sonderlich verwunderlich, da es sich bei beiden Bibliotheken um das Erzeugnis von professionellen Programmieren handelt, die sich sehr weitgehend mit Java, der Sichtbarkeit von Methoden und Klassen, sowie den Auswirkungen von zu hoher und zu niedriger Sichtbarkeit beschäftigt haben. Somit wird bei der Entwicklung dieser Bibliotheken wohl genauer auf eine sinnvolle Sichtbarkeit von Methoden und Klassen geachtet, als dies bei vielen anderen Bibliotheken der Fall ist.

3.1.5 ANTLR

Hier untersucht wurde ANTLR Version 3.2 vom 23 September 2009. Da es sich bei dieser Bibliothek um einen Syntaxparser handelt, der wohl hauptsächlich im Compiler-Bau oder Ähnlichem eingesetzt wird, also hauptsächlich von professionellen Programmierern erstellt und eingesetzt wird, würde man hier, ähnlich wie bei BCEL und ASM erwarten, auf keine Ergebnisse zu stoßen. Doch hier der Überblick.

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
ANTLR	11722	0	197	0	0	75	0	0	0

Die Anzahl der Funde ist überdimensional hoch. Dafür gibt es ausschließlich Funde der Klassen 2 und 5.

Funde der Kategorie 2:

Die Funde der Kategorie 2 sind ausschließlich aus ANTLR Version 2. Diese Version ist aus Gründen der Rückwärtskompatibilität im Java-Archiv der Version 3 enthalten. Die Funde teilen sich in zwei Gruppen: Die Codegeneratoren und die Hilfsklassen.

```
antlr.CodeGenerator
```

Dies ist die Basisklasse für alle Codegenerators. In ihr erfüllen folgende Methoden die Kapselungseigenschaften:

```
gen(...), getASTCreateString(...)
```

Hierbei übernimmt die Methode *gen()* verschiedene paketweit sichtbare Parameter. Diese stellen die Bausteine einer Sprache dar. Da all diese Methoden abstrakt sind und alle Codegenerators von dieser Klasse erben müssen, wiederholen sich all diese Funde in allen Codegeneratoren. Insgesamt sind dies 16 *gen**-Methoden und die Methode *getASTCreateString()*, die auch einen paketweit sichtbaren Parameter übernimmt. Bei insgesamt

8 Generatoren (inklusive der Superklasse Codegenerator) summiert sich dies auf 136 Funde, die nur aus diesem Grund gefunden wurden.

Da alle Bestandteile einer Sprache paketweit sichtbar sind folgt daraus, dass Codegeneratoren für neue Sprachen nur innerhalb des Pakets *antlr* erstellt werden konnten. In der Version 3 wurde dieses Konzept komplett überarbeitet, weshalb dort keine kapselnden Methoden gefunden wurden.

```
CSharpCodeGenerator
CppCodeGenerator
DiagnosticCodeGenerator
DocBookCodeGenerator
HTMLCodeGenerator
JavaCodeGenerator
PythonCodeGenerator
```

Diese speziellen Klassen von Codegeneratoren besitzen neben den Methoden, die sie durch die Vererbung aus *CodeGenerator* implementieren müssen auch noch andere Methoden:

```
genRule(RuleSymbol,...), genCommonBlock(AlternativeBlock,...),
genLookaheadSetForBlock(AlternativeBlock,...), genInclude(...),
    genLineNo(...),
genRuleHeader(...)
```

Diese Methode besitzt nicht jeder Codegenerator, sie sind speziell für die jeweilige Sprache. Jedoch übernehmen auch diese Methoden paketweit sichtbare Parameter, die Sprachelemente, und können somit auch nicht von außerhalb des Pakets verwendet werden. Dies ist wiederum ein Hinweis darauf, dass Codegeneratoren nur innerhalb des Pakets *antlr* deklariert werden konnten.

Da die anderen Funde dem gleichen Muster folgen, diese also auch Sprachelemente übernehmen, die nur paketweit sichtbar sind, und die Methoden teilweise das gleiche Schema wie die *gen()* Methode aufweisen, dass es also Methoden mit dem selben Namen für alle Sprachelemente gibt, verzichte ich an dieser Stelle auf eine genauere Analyse. Klar wird jetzt durch dieses Muster jedoch die große Anzahl an Funden.

Restliche Funde:

```
antlr.ANTLRTokdefParser.file(antlr.ImportVocabTokenManager):void
antlr.ANTLRTokdefParser.line(antlr.ImportVocabTokenManager):void
```

```
antlr.Grammar.define(antlr.RuleSymbol):void
antlr.Grammar.setTokenManager(antlr.TokenManager):void
antlr.LLkAnalyzer.FOLLOW(int, antlr.RuleEndElement):antlr.Lookahead
antlr.LLkAnalyzer.deterministic(...):boolean
antlr.LLkAnalyzer.look(int, ...):antlr.Lookahead
antlr.LLkAnalyzer.subruleCanBeInverted(antlr.AlternativeBlock,
    boolean):boolean
antlr.MakeGrammar.setBlock(antlr.AlternativeBlock, antlr.
    AlternativeBlock):void
antlr.RuleBlock.addExceptionSpec(antlr.ExceptionSpec):void
antlr.RuleBlock.setEndElement(antlr.RuleEndElement):void
antlr.preprocessor.GrammarFile.addGrammar(antlr.preprocessor.Grammar)
    :void
antlr.preprocessor.Hierarchy.addGrammar(antlr.preprocessor.Grammar):
    void
antlr.preprocessor.Hierarchy.findRoot(antlr.preprocessor.Grammar):
    antlr.preprocessor.Grammar
antlr.preprocessor.Preprocessor.optionSpec(antlr.preprocessor.Grammar
    ):antlr.collections.impl.IndexedVector
antlr.preprocessor.Preprocessor.rule(antlr.preprocessor.Grammar):void
```

An dieser Stelle sei nochmals angemerkt, dass es keine Funde der Klasse 2 aus der Version 3 von ANTLR gibt.

Funde der Kategorie 5:

Auch in dieser Klasse kommen viele Funde durch die Codegeneratoren zustande.

Die folgenden Methoden kommen in fast jedem Codegenerator vor und tauchen dort als Funde auf:

```
genASTDeclaration(...):void
genAlt(antlr.Alternative, antlr.AlternativeBlock):void
genBlockInitAction(antlr.AlternativeBlock):void
genBlockPreamble(antlr.AlternativeBlock):void
genMatch(antlr.GrammarAtom):void
genMatchUsingAtomText(antlr.GrammarAtom):void
genMatchUsingAtomTokenType(antlr.GrammarAtom):void
genSynPred(antlr.SynPredBlock, java.lang.String):void
genTokenTypes(antlr.TokenManager):void
getLookaheadTestExpression(antlr.Alternative, int):java.lang.String
lookaheadIsEmpty(antlr.Alternative, int):boolean
```

Durch diese Methoden kommen alleine schon 70 Funde zustande. Sie hängen wieder damit zusammen, dass die Sprachelemente alle paketweit sichtbar sind.

In der abstrakten Klasse Codegenerator selbst ist auch eine Methode zu finden, die die Kapselungseigenschaften erfüllt.

```
antlr.CodeGenerator.genTokenInterchange(antlr.TokenManager):void
```

Die Methode ist *protected* deklariert, wird aber von keinem der Codegeneratoren überschrieben, jedoch wird sie von ihnen verwendet. Wie jedoch schon festgestellt können Codegeneratoren nur innerhalb des Pakets *antlr* deklariert werden, und somit würde paketweite Sichtbarkeit ausreichen.

```
antlr.LLkAnalyzer.altUsesWildcardDefault(antlr.Alternative):boolean
```

Die Methode wird nach meiner Untersuchung nur innerhalb der Klasse *LLkAnalyzer* verwendet. Es kann also sein, dass es ausreichen würde, die Methode als *private* zu deklarieren. Auf jeden Fall kann man die Methode auf paketweite Sichtbarkeit reduzieren, da der Parameter nur innerhalb des Pakets zugänglich ist.

Das gilt auch für die verbleibenden Methoden aus der Version 2. Diese werden ebenfalls nur innerhalb der Klasse verwendet, in denen sie deklariert sind. Deshalb werde ich auf sie nicht weiter eingehen und sie nur auflisten.

```
antlr.MakeGrammar.addElementToCurrentAlt(antlr.AlternativeElement):
    void
antlr.TokenStreamRewriteEngine.addToSortedRewriteList(antlr.
    TokenStreamRewriteEngine$RewriteOperation):void
antlr.TokenStreamRewriteEngine.addToSortedRewriteList(java.lang.
    String, TokenStreamRewriteEngine$RewriteOperation):void
```

In der Kategorie 5 gibt es auch einen Fund in der neueren Version 3. Jedoch ist dieser Fund nicht in ANTLR selbst, sondern in der angegliederten StringTemplate-Bibliothek, die mit ANTLR verwendet werden kann.

```
org.antlr.stringtemplate.StringTemplateGroupInterface.
    getTemplateSignature(
    StringTemplateGroupInterface$TemplateDefinition):String
```

Da TemplateDefinition nicht *protected* sondern *package* deklariert ist ist es sinnlos, die Methode getTemplateSignature *protected* zu deklarieren. Da keine Methode innerhalb

des Pakets die Klasse TemplateDefinition zurück gibt kann die Methode außerhalb des Pakets nicht verwendet werden.

Abschließend lässt sich zu ANTLR sagen, dass die neue Version wesentlich weniger Funde von Anomalien aufweist, als dies bei Version 2 noch der Fall war. Die Entscheidung, die Sprachelemente nur paketweit sichtbar zu machen hatte dazu geführt, dass sehr viele der Methode nicht außerhalb des Pakets verwendet werden konnten. Somit traten hier sehr viele Funde mit Kapselungseigenschaften auf. In der neuen Version wurde die Architektur offensichtlich komplett überarbeitet, womit jetzt so gut wie keine Funde mehr auftreten.

3.1.6 Lucene

Untersucht wurde Lucene 3.0.0 vom 25. Nov. 2009. Zunächst ein Überblick über die Funde:

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
Lucene	8299	0	2	0	0	1	0	2	0

Wie man sieht ist die Zahl der Funde relativ gering.

Funde der Kategorie 2:

```
org.apache.lucene.analysis.ru.RussianStemFilter.setStemmer(
    RussianStemmer):void
```

Die Klasse *RussionStemmer* ist in diesem Fall nur paketweit sichtbar. Die restlichen *XYStemmer*-Klassen sind dagegen öffentlich sichtbar. Da in allen *XYStemFilter*-Klassen auch immer eine Methode existiert, mit der man einen alternativen Stemmer angeben kann ist diese Methode bei den restlichen StemFiltern kein Fund, jedoch bei der russischen Variante schon, da deren Stemmer als einziger aus der Reihe fällt und paketweit sichtbar ist. Die Lösung wäre hier ganz klar, die russische Stemmer-Klasse ebenso öffentlich sichtbar zu machen.

```
org.apache.lucene.queryParser.precedence.PrecedenceQueryParser.
    setDefaultOperator(PrecedenceQueryParser$Operator):void
```

Mit dieser Methode wird der standardmäßige Operator gesetzt, also der Operator der gilt, wenn zwischen zwei Ausdrücken kein expliziter Operator steht. Der Zugriff auf die

Operatoren ist einfach über zwei Konstanten in der Klasse *PrecedenceQueryParser* möglich, OR_OPERATOR und AND_OPERATOR. Somit ist die Methode uneingeschränkt nutzbar, kann jedoch von außen nicht überschrieben werden.

Beide Funde der Klasse 2 befinden sich nicht im offiziellen Code des Frameworks Lucene, sondern in den von der Community beigetragenen Ergänzungspaketen.

Funde der Kategorie 5:

```
org.apache.lucene.search.FieldValueHitQueue.lessThan(
    FieldValueHitQueue$Entry, FieldValueHitQueue$Entry):boolean
```

Die einzigen existierenden Subklassen der abstrakten Klasse FieldValueHitQueue sind private Subklassen, die in der selben Java Klasse deklariert wurden und über die öffentliche statische Methode *create(SortField[], int)* bezogen werden können. Somit kann die Methode *lessThan()* paketweit sichtbar oder sogar *private* gemacht werden, ohne die Verwendbarkeit zu ändern, da die Methode nur zur Realisierung der Subklasse benötigt wird.

Dieser Fund ist im Hauptprogramm aufgetreten, nicht im von der Community beigetragenen Code.

Funde der Kategorie 7:

```
org.apache.lucene.index.MergeScheduler
```

Methoden:

```
close():void merge(org.apache.lucene.index.IndexWriter):void
```

In der Javadoc Dokumentation steht an dieser Stelle:

```
...
* This class typically requires access to
* package-private APIs (eg, SegmentInfos) to do its job;
* if you implement your own MergePolicy, you'll need to put
* it in package org.apache.lucene.index in order to use
* these APIs.
...
```

Es scheint also in diesem Fall bekannt und gewollt zu sein, dass die Subklassen tatsächlich in diesem Paket abgelegt werden. Mir erschließt sich nicht ganz die Logik dahinter, Teile des Programms so zu gestalten, dass man von außerhalb des Pakets nicht mehr darauf

zugreifen kann und dann in der Dokumentation darauf hinzuweisen, dass Subklassen in das eigentlich zum Programm selbst gehörende Paket abgelegt werden sollen. Ein Hinweis jedoch findet sich in der Dokumentation: Dieser Teil des Programms ist experimentell und kann sich jederzeit ändern.

3.1.7 OpenJPA & Hibernate

Untersucht wurden OpenJPA Version 2.0.0M3 und Hibernate Core Version 3.3.2, Hibernate Annotations 3.4.0, Hibernate EntityManager 3.4.0, Hibernate Validator 4.0.2 und Hibernate Search 3.1.1. Hier eine Übersicht der Ergebnisse:

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
OpenJPA	27467	0	202	1	0	4	0	0	0
Hibernate	15573	0	0	3	0	0	0	2	0

Wenden wir uns zunächst OpenJPA zu. Was sofort ins Auge fällt ist die extrem hohe Zahl an Funden der **Kategorie 2**. Diese werden zum größten Teil von einer Vererbungshierarchie verursacht. An der Wurzel der Vererbungshierarchie steht die Klasse *org.apache.openjpa.persistence.query.AbstractVisitable*. In ihr sind folgende Methoden deklariert:

```
asExpression(AliasContext):String
asProjection(AliasContext):String
getAliasHint(AliasContext):String
asJoinable(AliasContext):String
```

Sowohl die Klasse *AbstractVisitable* als auch die Klasse *AliasContext* sind paketweit sichtbar. Von *AbstractVisitable* erben nun direkt oder indirekt einige Klassen:

```
AbsExpression
AbstractDomainObject
ArrayExpression
AvarageExpression
BetweenExpression
BinaryOperatorExpression
CaseExpressionImpl
ConcatExpression
CountExpression
CurrentTimeExpression
DistinctExpression
```

```
ElseExpression
EqualExpression
ExistsExpression
FetchPath (nur 1)
GreaterEqualExpression
GreaterThanExpression
InExpression
IndexExpression
IsEmptyExpression
IsNullExpression
JoinPath
LengthExpression
LessEqualExpression
LessThanExpression
LikeExpression
LiteralExpression
LocateExpression (nur 1)
LogicalExpression
LowerExpression
MaxExpression
MemberOfExpression
MinExpression
NewInstance
NotEqualExpression
OperatorPath
OrderableItem
ParameterExpression
QueryDefinitionImpl
RangeExpression (nur 1)
RootPath
SizeExpression
SquareRootExpression
SubStringExpression
SumExpression
TrimExpression
TypeExpression
UnaryMinusExpression
UpperExpression
VarArgsExpression
```

Insgesamt also 47 Klassen, die wegen den oben genannten vier Methoden als Fund er-

kannt werden. Drei weitere Klassen implementieren nur eine der vier Methoden. Das macht 191 Funde, die nur durch die vier oben genannten Methoden entstehen.

Weitere Funde:

```
org.apache.openjpa.persistence.query.WhenClause.toJPQL(AliasContext):
    String
org.apache.openjpa.persistence.query.QueryDefinitionImpl.fillBuffer(
    java.lang.String, StringBuffer, AliasContext, Predicate):void
org.apache.openjpa.jdbc.kernel.exps.PCPath.addVariableAction(org.
    apache.openjpa.jdbc.kernel.exps.Variable):void
org.apache.openjpa.kernel.DetachedStateManager.attach(AttachManager,
    Object, org.apache.openjpa.meta.ClassMetaData, org.apache.openjpa.
    enhance.PersistenceCapable, OpenJPAStateManager, org.apache.
    openjpa.meta.ValueMetaData, boolean):Object
org.apache.openjpa.lib.util.Localizer.addProvider(org.apache.openjpa.
    lib.util.ResourceBundleProvider):void
org.apache.openjpa.lib.util.Localizer.removeProvider(org.apache.
    openjpa.lib.util.ResourceBundleProvider):boolean
org.apache.openjpa.persistence.criteria.
    CriteriaExpressionVisitor$AbstractVisitor.exit(CriteriaExpression)
    :void
org.apache.openjpa.persistence.criteria.
    CriteriaExpressionVisitor$AbstractVisitor.getTraversalStyle(
    CriteriaExpression):CriteriaExpressionVisitor$TraversalStyle
org.apache.openjpa.persistence.criteria.
    CriteriaExpressionVisitor$AbstractVisitor.isVisited(
    CriteriaExpression):boolean
org.apache.openjpa.persistence.criteria.
    CriteriaExpressionVisitor$ParameterVisitor.enter(
    CriteriaExpression):void
org.apache.openjpa.persistence.util.SourceCode$Class.addField(java.
    lang.String, SourceCode$ClassName):SourceCode$Field
```

In der **Kategorie 3** gibt es einen interessanten Fund:

```
org.apache.openjpa.persistence.query.QueryDefinitionImpl.fillBuffer(
    String, StringBuffer, AliasContext, java.util.List,
    QueryDefinitionImpl$Visit):void
```

Dieser desshalb so interessant, da diese Methode in überladener Form schon als Fund in der Kategorie 2 aufgetaucht ist. In dieser Form wird die Methode jedoch in Kategorie 3 eingestuft, da *QueryDefinitionImpl$Visit* eine private Klasse ist.

Zur Vervollständigung noch die Funde der **Kategorie 5** im Überblick:

```
org.apache.openjpa.kernel.DetachedStateManager.getDetachedObjectId(
    org.apache.openjpa.kernel.AttachManager, java.lang.Object):java.
    lang.Object
org.apache.openjpa.lib.graph.BreadthFirstWalk.enqueue(java.lang.
    Object, org.apache.openjpa.lib.graph.NodeInfo):void
org.apache.openjpa.lib.graph.BreadthFirstWalk.visit(java.lang.Object,
     org.apache.openjpa.lib.graph.NodeInfo):void
org.apache.openjpa.persistence.util.SourceCode$Class.addMethod(java.
    lang.String, SourceCode$ClassName):SourceCode$Method
```

Man sieht also, dass sehr viele der Funde durch ein einzelnes Pattern verursacht wurden. Lediglich 16 Funde wurden ohne Einfluss des Patterns gemacht.

Bleibt noch die Ergebnisse in **Hibernate** zu besprechen.

In **Kategorie 3** fallen die folgenden Funde:

```
org.hibernate.loader.custom.
    CustomLoader$NonScalarResultColumnProcessor.performDiscovery(
    CustomLoader$Metadata, java.util.List, java.util.List):void
org.hibernate.loader.custom.CustomLoader$ResultRowProcessor.
    prepareForAutoDiscovery(CustomLoader$Metadata):void
org.hibernate.loader.custom.CustomLoader$ScalarResultColumnProcessor.
    performDiscovery(CustomLoader$Metadata, java.util.List, java.util.
    List):void
```

Auffällig ist, dass alle drei Funde in inneren Klassen der Klasse CustomLoader auftauchen. In dieser Klasse befinden sich recht viele Unterklassen, viele davon sind auch *public*. Offensichtlich wurde dabei jedoch nicht darauf geachtet, dass die innere Klasse *CustomLoader$Metadata* privat ist und somit von außen nicht zugreifbar. Dies sind die einzigen Funde der Kategorien 1 bis 6.

Daneben existieren noch zwei Funde der **Kategorie 7**:

```
org.hibernate.mapping.Collection.createPrimaryKey():void
org.hibernate.mapping.PersistentClass.nextSubclassId():int
```

Im Vergleich zu OpenJPA sind in Hibernate wenige kapselnde Methoden gefunden worden. Daraus könnte man schließen, dass sich die Programmierer von Hibernate mehr Gedanken über die öffentliche Schnittstelle gemacht haben.

3.1.8 Xalan & Xerces

Untersucht wurden Apache Xalan Version 2.7.1 vom 27. November 2007 und Apache Xerces Version 2.9.0 vom 23. November 2006. Hier ein Überblick über die Ergebnisse:

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
Xalan	5125	0	21	0	0	5	0	0	0
Xerces	3520	0	0	0	0	1	0	0	0

In Xalan fallen die 21 Funde der **Kategorie 2** auf. Bei genauerem Hinsehen finden sich 20 davon im Paket *org.apache.xalan.xsltc.compiler*.

```
LocationPathPattern.typeCheck(SymbolTable):util.Type
Parser:
    addParameter(Param):void
    addVariable(Variable):void
    getQName(QName, QName):QName
    lookupVariable(QName):VariableBase
    removeVariable(QName):void
    setOutput(Output):void
Pattern.typeCheck(SymbolTable):util.Type
Stylesheet:
    addParam(Param):int
    addVariable(Variable):int
    getMode(QName):Mode
    typeCheck(SymbolTable):util.Type
SyntaxTreeNode.typeCheck(SymbolTable):util.Type
Template:
    addParameter(Param):void
    setName(QName):void
    typeCheck(SymbolTable):util.Type
XPathParser.insertStep(Step, RelativeLocationPath):
   RelativeLocationPath
XSLTC:
    registerAttribute(QName):int
    registerElement(QName):int
    registerNamespacePrefix(QName):int
```

Wie man sieht sind es hier die wichtigen Bestandteile eines Compilers (*SymbolTable*, *Param*, *QName*, *Output*, *Variable*, *Step*), die nur paketweit sichtbar sind. Die Methoden, die diese Daten verarbeiten können somit nur innerhalb des Pakets verwendet werden. Über

öffentliche Methoden kann man sicher an die Typen *Symboltable*, *QName* und *Output* (jeweils über Methoden in der Klasse *Parser*). Es bleiben also noch 6 Methoden davon übrig, die zu paketweit sichtbar geändert werden könnten, da die Klassen *Param* und *Variable* nicht zugänglich sind. Hinzu kommt noch im Paket *org.apache.xalan.xsltc.dom* der Fund

```
NodeSortRecord.initialize(int, int, org.apache.xalan.xsltc.DOM,
    SortSettings):void
```

welcher auch nicht bedingt ist.

Die Funde der **Kategorie 5** teilen sich auch wieder in zwei Gruppen auf, die Funde im Compiler-Paket *org.apache.xalan.xsltc.compiler* und die Funde im Template-Paket *org.apache.xalan.templates.* Dabei sind die Funde im Paket *org.apache.xalan.xsltc.compiler* alle bedingt, da sie wieder *Symboltable* und *QName* verwenden.

```
SyntaxTreeNode.setQName(QName):void
SyntaxTreeNode.typeCheckContents(SymbolTable):util.Type
```

Die Funde im Paket *org.apache.xalan.templates* dagegen sind nicht bedingt, an ihre Parameter kann man außerhalb des Pakets nicht gelangen.

```
RedundentExprEliminator.findCommonAncestor(
    RedundentExprEliminator$MultistepExprHolder):ElemTemplateElement
RedundentExprEliminator.isNotSameAsOwner(
    RedundentExprEliminator$MultistepExprHolder, ElemTemplateElement):
    boolean
RedundentExprEliminator.matchAndEliminatePartialPaths(
    RedundentExprEliminator$MultistepExprHolder,
    RedundentExprEliminator$MultistepExprHolder, boolean, int,
    ElemTemplateElement):RedundentExprEliminator$MultistepExprHolder
```

Hier scheint es mir verwunderlich, dass zwar sehr viele *protected* Methoden die Klasse *MultistepExprHolder* als Parameter verwenden, die Klasse selbst jedoch nur paketweit sichtbar ist. Die Frage ist nun also, ob die Autoren der Klasse *RedundentExprEliminator* die Methoden aus Versehen *protected* gemacht haben. obwohl diese eigentlich nur innerhalb des Pakets überschrieben werden sollen, oder ob sie die Klasse *MultistepExprHolder* nur aus versehen paketweit sichtbar gemacht haben und sie statt dessen *protected* sein sollte.

Der einzige Fund in **Xerces** befindet sich im Paket *org.apache.xerces.dom* und ist in die Kategorie 5 eingestuft:

```
DocumentImpl.dispatchAggregateEvents(NodeImpl,
    DocumentImpl$EnclosingAttr):void
```

Wie die Namen sowohl der deklarierenden Klasse als auch der Parameter suggeriert handelt es sich hierbei um eine interne Implementierung. Daher kann man davon ausgehen, dass ein Benutzer der Bibliothek nicht mit dieser Klasse direkt in Kontakt kommen, sondern diese Klasse über das Interface *Document* verwenden wird. Daher muss man diesen Fund nicht zum öffentlichen Interface zählen.

3.1.9 iText

Untersucht wurde iText Version 5.0.0. Hier ein Überblick über die Ergebnisse:

	Methoden	Kat1	Kat2	Kat3	Kat4	Kat5	Kat6	Kat7	Kat8
iText	4588	0	1	0	0	0	0	3	0

Alle Funde wurden im selben Paket gemacht: *com.itextpdf.text.pdf*.

Der Fund der **Kategorie 2**:

```
PdfLister.listStream(PRStream, PdfReaderInstance):void
```

Das bemerkenswerte an diesem Fund ist die Tatsache, dass genau die Stelle in der Methode, in der die *PdfReaderInstance* verwendet wurde, auskommentiert wurde. Somit ist der Parameter nicht mehr verwendet und kann entfernt werden. Somit würde der Fund aufgelöst.

Die Funde der **Kategorie 7**:

```
BaseFont.getFullFontStream():com.itextpdf.text.pdf.PdfStream
BaseFont.getRawWidth(int, java.lang.String):int
BaseFont.writeFont(PdfWriter, PdfIndirectReference, Object[]):void
```

Alle drei Funde beziehen sich auf die Klasse *BaseFont*. Der Konstruktor dieser abstrakten Klasse ist als *protected* deklariert. Es hat den Anschein, als wollte der Autor die Klasse nach außen schützen und den Konstruktor nur intern verwendbar machen. Dazu wäre es nötig den Konstruktor als paketweit sichtbar, also ohne Modifikator, zu deklarieren. Hier scheint die genaue Bedeutung von *protected* nicht ganz klar gewesen zu sein.

3.1.10 GEF (Graph Editing Framework)

Untersucht wurde das Framework GEF-0.13 vom 15. Feb. 2009. Dabei tauchten zwei Funde der Kategorie 3 auf, die sich gegenseitig bedingen:

```
org.tigris.gef.presentation.FigEdge
```

Es existieren folgende Methoden:

```
getPathItemFig(FigEdge$PathItem,...)
removePathItemFig(FigEdge$PathItem,...)
```

Das Interessante an diesen Funden ist folgendes:

Sowohl *getPathItemFid()* als auch *removePathItemFig()* übernehmen als Parameter ein *PathItem*. Diese Klasse ist jedoch eine private innere Klasse der Klasse *FigEdge*. Das ungewöhnliche daran ist jetzt, dass die *PathItem*'s im Vektor *_pathItems* gespeichert werden, der nicht generisch ist. Das heißt, dass dieser Vektor Objekte vom Typ *java.lang.Object* speichert. Es existiert eine Methode *getPathItemsRaw()*, welche eben diesen Vektor zurück gibt. Man kann also von außen an die *PathItem*'s herankommen. Das Problem dabei: Man kann außerhalb der Klasse *FigEdge* nicht von *java.lang.Object* nach *FigEdge.PathItem* casten. Die Kapselung kommt also nur dadurch zu Stande, dass keine Generics verwendet werden.

3.2 Zusammenfassung

Wie sich in der Fallstudie gezeigt hat tauchen in vielen Programmen und Bibliotheken Stellen in der öffentlichen Schnittstelle auf, an denen öffentlich nicht sichtbare Typen verwendet werden. Diese Typen führen jedoch oft dazu, dass die Methoden nicht so verwendet werden können, wie sie auf Grund ihrer Zugriffsmodifikatoren definiert sind. Die öffentliche Schnittstelle ist an dieser Stelle also inkonsistent. Diese Inkonsistenzen kommen in vielen Bibliotheken vor, die nicht von den absoluten Profis in Java geschrieben wurden, die sich sehr intensiv mit dem Java und dessen Zugriffsbeschränkungssystem beschäftigt haben. Dass jedoch selbst diesen Profis Fehler unterlaufen, die weitreichende Folgen haben können, zeigt das Beispiel ANTLR. Der Designfehler, die Sprachelemente nur paketweit sichtbar zu machen, führte zu mehreren hundert Funden. Dass dies in der neuen Version nicht mehr der Fall ist zeigt die Einsicht der Programmierer. Doch selbst bei den Designern der Sprache, nämlich in der Java Laufzeitumgebung, sind solche Funde zu finden. Nicht zuletzt das zeigt, dass das Zugriffssystem von Java relativ mächtig,

jedoch auch sehr komplex ist. Die korrekte und konsistente Verwendung scheint vielen Programmierern nicht klar und Toolsupport, wenn überhaupt, so doch kaum gegeben.

Kommen zudem noch andere Faktoren dazu, wie zum Beispiel Generics im Falle GEF (Kapitel 3.1.10), ist extreme Vorsicht geboten. Die öffentliche Schnittstelle sollte mit äußerster Sorgfalt gestaltet und überprüft werden, ob die Schnittstellenmethoden auch tatsächlich so verwendet werden können, wie dies angedacht war. Außerdem stellte sich mir bei der Untersuchung oft die Frage, warum an vielen Stellen in den Bibliotheken nicht entsprechende Einschränkungen bezüglich der Zugriffsmodifikatoren bei den Methoden gemacht wurden. Eine zu weite öffentliche Schnittstelle mit Methoden, die zwar als public deklariert sind, jedoch weder dokumentiert sind noch auf Grund von öffentlich nicht zugänglichen Typen verwendet werden können, bläht die Dokumentation unnötig auf und kann zu großer Verwirrung bei Programmierern und Designern führen.

4 Verallgemeinerung und Fazit

Wie in dieser Arbeit gezeigt ermöglicht Java Kombinationen der Sichtbarkeit von Methoden und Klassen, die auf den ersten Blick zu falschen Aussagen führen können. Nicht jede Methode, die als *public* deklariert ist und in der öffentlichen Schnittstelle auftaucht kann auch verwendet und überschrieben werden. Somit können in der Dokumentation Methoden auftauchen, die nicht verwendet werden können. Die Klarheit der Dokumentation ist dadurch gefährdet. Für Java zeigt sich, dass das Zugriffssystem zu viele Kombinationen erlaubt, die zu inkonsistenten öffentlichen Schnittstellen führen. Es lässt Methoden zu, die zwar öffentlich sichtbar, jedoch nicht verwendbar sind. Die Frage stellt sich also, ob Zugreifbarkeit nicht einfacher verständlich realisiert werden kann oder zumindest Inkonsistenzen vermieden werden können.

4.1 Zugreifbarkeit in anderen Programmiersprachen

In der Programmiersprache C# von Microsoft, die recht große Ähnlichkeit mit Java aufweist, gibt es äquivalente Zugriffsmodifikatoren wie in Java. Jedoch ist in der Sprachspezifikation geregelt, dass so genannte Accessibility constraints eingehalten werden müssen.[6, Kapitel 3.5.4] Zusammenfassen lassen sich diese Beschränkungen wie folgt: Subklassen, Variablen, Rückgabewerte, Parameter und so weiter dürfen maximal so zugreifbar sein, wie ihr Typ. Somit lassen sich als Parameter und Rückgabewert einer *public* Methode nur *public* Typen verwenden. Werden die Beschränkungen nicht eingehalten führt dies zu einem Fehler beim Übersetzen des Programms. Somit sind die in dieser Arbeit aufgeführten Problem in C# von vorne herein ausgeschlossen. Andere objektorientierte Sprachen wie Oberon und Component Pascal haben ein wesentlich einfacheres System der Zugriffssteuerung. Soll ein Element eines Moduls nach außen sichtbar sein, so muss es als Namenspostfix einen Stern (*) oder ein Minus (-) haben. Dabei bedeutet Stern read-write Zugriff und Minus read-only Zugriff von außen.[20, Kapitel 11][7, Kapital 4] Die reale Zugreifbarkeit eines Elements ist somit sehr gut überschaubar. Entweder, es ist nach außen sichtbar und somit zugreifbar, oder es ist nach außen nicht sichtbar und somit auch nicht zugreifbar. Dadurch entstehen viele Probleme die hier angesprochen wurden deshalb nicht, weil es keine paketweite Sichtbarkeit gibt und auch Sichtbarkeit in Subklassen nicht von dem Zugriffssystem erfasst werden. Die Anzahl der möglichen Kombinationen sinkt. Ob ein solches Zugriffssystem auch für größere Systeme ausreichend ist lasse ich an dieser Stelle offen. In Eiffel wurde ein komplett anderes System gewählt. Hier kann angegeben werden, welche Elemente (in Eiffel *features* genannt) für

die angegebenen Klassen zugreifbar sein sollen. Es können also für verschiedene Klassen unterschiedliche *features* zugreifbar gemacht werden.[2, Kapitel 14.2] Die in Java vorhandenen Zugriffsmodifikatoren lassen sich somit weitestgehend simulieren. Soll ein *feature* als *public* deklariert werden, so gibt man ihm die Sichtbarkeitsliste *{ALL}*. Somit ist das *feature* für die Klasse *ALL* (welche die Wurzel des Vererbungsbaumes darstellt) und seine Subklassen, somit also für alle Klassen sichtbar. Will man das entsprechende Element private deklarieren, so gibt man ihm die Sichtbarkeitsliste *{NONE}*. *NONE* ist das Gegenteil von *ALL*, diese Klasse hat keine Subklassen, somit ist das *feature* für keine andere Klasse sichtbar. Für paketweite Sichtbarkeit müsste man eine Liste aller Klassen im aktuellen Paket angeben. Da die Klasse jedoch meist nur von ganz bestimmten Klassen im aktuellen Paket verwendet wird kann die Liste auf diese Klassen beschränkt werden. Somit schränkt man die Sichtbarkeit der *feature* auf ein Minimum ein. Auf die selbe Weise kann man in Eiffel auch für jede Klasse festlegen, welche *features* in welcher Subklasse sichtbar sein sollen. Dies ist mit der *export*-Deklaration möglich und genau so feingranular wie die Sichtbarkeit nach außen. Trotz der höheren Komplexität des Ansatzes kann somit im Detail die Sichtbarkeit der *features* pro Klasse festgelegt werden. Somit werden Kompromisse vermieden, wie sie bei Java über die Sichtbarkeit *package* oder *protected* nötig sind. So ist es in Java, selbst wegen weniger Ausnahmen, in denen der Zugriff von wenigen Klassen aus gestattet sein soll, eine Deklaration als *private* nicht möglich.

4.2 Fazit für Java

Die richtige Sichtbarkeit von Klassen und Methoden ist essentiell für eine klare öffentliche Schnittstellen. Sind diese zu breit gewählt, so ist das Prinzip des Information Hidings nicht erfüllt. Es kann somit dazu kommen, dass Teile der internen Repräsentation falsch oder sogar in gefährlicher Weise verwendet oder beeinträchtigt werden. Dies kann zu katastrophalen Fehlern im Programm führen. Sicherheitsprobleme sind oft eine Folge von Nichteinhaltung des Prinzips des Information Hiding oder falscher Benutzung von Komponenten des Programms. Dem kann vorgebeugt werden, indem die öffentliche Schnittstelle so eingeschränkt wie möglich gehalten wird. Jedoch ist eine zu große Einschränkung wiederum hinderlich. Viele der Funde, die in der Fallstudie gemacht wurden, weisen auf Fehler beim Design der öffentlichen Schnittstelle hin. Werden öffentlich nicht sichtbare Klassen in öffentlichen Schnittstellen verwendet, so hat dies automatisch zur Folge, dass dieser Teil der öffentlichen Schnittstelle inkonsistent ist. Die Methoden können im schlimmsten Fall nicht verwendet werden, im besten Fall können Teile einer

Klasse nicht mehr angepasst, sprich überschrieben, werden. Beides sind Fehler, die sich in den meisten Fällen vermeiden lassen. Wie in den Beispielen gesehen gibt es meist die Möglichkeit, die Sichtbarkeit so anzupassen, entweder der Klasse oder der Methode, dass die öffentliche Schnittstelle von öffentlich nicht sichtbaren Typen bereinigt wird. Dazu kann es nötig sein, weitere Einschränkungen zu machen oder die Schnittstelle zu erweitern.

Die Untersuchung hat jedoch auch gezeigt, dass es teilweise Unterschiede in den Funden gab, die sich auf die Hintergründe der Programmierer zurückführen lassen. So lässt sich die These bilden, dass Programmierer, die als Hintergrund das Sprachdesign oder die Untersuchung von Sprachen haben, sich also intensiv mit der Zugriffskontrolle und ihren Eigenheiten unter Java beschäftigt haben, sich mehr Gedanken über die Gestaltung der öffentlichen Schnittstelle machen. Dies zeigt sich darin, dass es in Bibliotheken wie ASM, BCEL und dem neuen ANTLR kaum bis keine Funde gibt. Dies könnte auch darauf hinweisen, dass das System der Sichtbarkeit unter Java relativ komplex ist, so komplex, dass es ein fundiertes Wissen benötigt, um die verschiedenen möglichen Kombinationen zu überblicken und richtig anzuwenden beziehungsweise zu vermeiden. Es stellt sich also die Frage, ob bestimmte Kombinationen nicht so exotisch und komplex anzuwenden sind, dass sie in einer Sprache wie Java nicht erlaubt werden sollten.

Alle Wege der Zugriffskontrolle haben Vor- und Nachteile. Ebenso scheint es, als seien nicht alle Wege gleich mächtig. Vor allem das System von Eiffel geht weit über viele andere Systeme hinaus. Wie kompliziert ein System zur Zugriffskontrolle im Endeffekt wirklich sein muss, kann an dieser Stelle nicht beantwortet werden. Jedoch zeigt schon der Vergleich zwischen Java und C#, dass mit wenigen zusätzlichen Beschränkungen viele der in Java vorhandenen Kombinationen von Zugriffsmodifikatoren ausgeschlossen werden können. Somit kann das System zur Zugriffsbeschränkung stark vereinfacht und auch für Programmierer, die keine Profis auf dem Gebiet der Zugriffskontrolle unter Java sind, verständlich und handhabbar gemacht werden.

Literatur

[1] 1T3XT BVBA. 1T3XT: Product. Online Stand Dezember 2009 `http://itextpdf.com/`.

[2] Andreas Hohmann. Programming Languages at a Glance, 2003. Online Stand Januar 2010 `http://www.minimalprogramming.org/html/index.html`.

[3] Apache Software Foundation. BCEL, 2006. Online Stand Dezember 2009 `http://jakarta.apache.org/bcel/`.

[4] J. Gosling, B. Joy, B. Steele, and G. Bracha. The Java Language Specification - Third Edition. Online Stand Dezember 2009 `http://java.sun.com/docs/books/jls/`.

[5] JUnit.org. JUnit.org Resources for Test Driven Development. Online Stand Dezember 2009 `http://www.junit.org/`.

[6] Microsoft Corporation. C# Language Specification, 2007. Online Stand Januar 2010 `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`.

[7] Oberon microsystems, Inc. Component Pascal Language Report, 2001. Online Stand Januar 2010 `http://www.oberon.ch/pdf/CP-Lang.pdf`.

[8] OW2 Consortium. ASM - Home Page. Online Stand Dezember 2009 `http://asm.ow2.org/`.

[9] T. Parr. ANTLR Parser Generator. Online Stand Dezember 2009 `http://www.antlr.org/`.

[10] Red Hat Middleware, LLC. hibernate.org - Hibernate. Online Stand Dezember 2009 `https://www.hibernate.org/`.

[11] Sun Microsystems, Inc. Java SE Overview. Online Stand Dezember 2009 `http://java.sun.com/javase/`.

[12] Sun Microsystems, Inc. Code Conventions for the Java Programming Language, 1999. Online Stand Dezember 2009 `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`.

[13] The Apache Software Foundation. Apache ActiveMQ. Online Stand Dezember 2009 `http://activemq.apache.org/`.

[14] The Apache Software Foundation. Apache Harmony - Open Source Java Platform. Online Stand Dezember 2009 `http://harmony.apache.org/`.

[15] The Apache Software Foundation. Apache OpenJPA. Online Stand Dezember 2009 `http://openjpa.apache.org/`.

[16] The Apache Software Foundation. Welcome to Lucene! Online Stand Dezember 2009 `http://lucene.apache.org/`.

[17] The Apache Software Foundation. Xalan-Java Version 2.7.1. Online Stand Dezember 2009 `http://xml.apache.org/xalan-j/`.

[18] The Apache Software Foundation. Xerces Java Parser Readme. Online Stand Dezember 2009 `http://xerces.apache.org/xerces-j/`.

[19] Tigris.org. gef Project home. Online Stand Januar 2010 `http://gef.tigris.org/`.

[20] N. Wirth. Oberon Language Report, 1990. Online Stand Januar 2010 `http://www-old.oberon.ethz.ch/oreport.html`.