Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur

Peter Müller und Arnd Poetzsch-Heffter

Fernuniversität Hagen, D-58084 Hagen, Deutschland
[Peter.Mueller, Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de

Zusammenfassung In Java bestehen enge Wechselwirkungen zwischen den Kapselungskonstrukten und den Regeln für die Bindung von Methoden. Anhand zweier Beispiele zeigen wir, daß die mangelnde Orthogonalität der beiden Sprachkonzepte zu Fehlern in der Sprachspezifikation und Implementierung von Java führte. Der Aufsatz diskutiert mögliche Lösungen, liefert eine korrigierte und konzeptionell vereinfachte Fassung des Methodenauswahlalgorithmus von Java und präsentiert ein erweitertes Konzept zur Vergabe von Zugriffsrechten. Insgesamt wird damit die Orthogonalität der beteiligten Sprachkonzepte wesentlich gesteigert.

1 Einleitung

Moderne objektorientierte Programmiersprachen unterstützen in der Regel eine Vielzahl von Sprachkonzepten wie Subtyping, Vererbung, Kapselung, Modularisierung, dynamische Bindung usw. Ziel jedes Sprachentwurfs sollte es sein, diese verschiedenen Konzepte so orthogonal wie möglich zu realisieren. Insbesondere sollten die semantischen Wechselwirkungen zwischen den resultierenden Sprachkonstrukten klein gehalten werden.

Beim Design der Sprache Java ist dies bei vielen Aspekten gelungen. Die Beziehungen zwischen Kapselungskonstrukten, Paketkonzept und Methodenbindung in Java sind allerdings unnötig komplex und haben zu Unklarheiten und Fehlern in der Sprachspezifikation und bei Sprach-Implementierungen geführt. Beispielsweise können geschützte (protected) Methoden an Programmstellen aufgerufen werden, an denen sie nicht zulässig sind. Damit können Kapselungsschranken durchbrochen werden, auf die der Programmierer sich verlassen muß, um z.B. Klasseninvarianten zu garantieren.

Dieser Aufsatz analysiert die angesprochene Problematik anhand von Beispielen, erläutert mögliche Lösungen und präsentiert ein einheitliches Verfahren zur Methodenauswahl, mit dem ein hohes Maß an Orthogonalität zwischen den beteiligten Sprachkonzepten erreicht wird. Der Aufsatz konzentriert sich zwar auf die Situation in Java, trägt aber auch dazu bei, die grundsätzliche Wechselwirkung zwischen Kapselung und Methodenbindung besser zu verstehen. Zunächst geben wir eine kurze Einführung in diese Konzepte und ihre Beziehung.

Kapselung und Zugriffsrechte. Objektorientierte Programmiersprachen unterstützen Kapselung. Diese ermöglicht es, bestimmten Nutzergruppen einer

Klasse nur eingeschränkten Zugriff auf Attribute der Klasse und ihrer Objekte zu gewähren; unter *Attributen* verstehen wir hier Methoden, Instanz- und Klassenvariablen. Durch diese Schnittstellenbildung kann man Implementierungsdetails verbergen und vor unsachgemäßer Benutzung schützen. Üblicherweise werden drei *Zugriffsmodi* unterstützt: *Private* Attribute dürfen nur für die Implementierung der Klasse selbst verwendet werden. *Geschützte* Atribute dürfen darüber hinaus bei der Implementierung von Subklassen eingesetzt werden. *Öffentliche* Attribute sind für alle Nutzer zugreifbar. In Java werden die Zugriffsmodi durch die *Zugriffsmodifikatoren* `private`, `protected` und `public` deklariert.

Neben diesen Zugriffsrechten, die durch die Klassengrenzen festgelegt sind, unterstützt Java Zugriffsrechte, die sich an den Paketgrenzen orientieren: Attribute können *paketweit zugreifbar* sein, d.h. jede Anweisung im gleichen Paket besitzt das Recht, auf das Attribut zuzugreifen.[1] Öffentliche und geschützte Attribute sind in Java paketweit zugreifbar. Darüber hinaus sind alle Attribute paketweit zugreifbar, die ohne Zugriffsmodifikator deklariert wurden. Dieser Modus heißt in Java *default access*, wir werden ihn als *paketlokal* bezeichnen. (Man beachte: Jedes paketlokale Attribut ist paketweit zugreifbar; aber es gibt paketweit zugreifbare Attribute, die nicht den Zugriffsmodus „paketlokal“ besitzen.)

Methodenbindung. Die Bindung des Methodenaufrufs an die zugehörige Implementierung kann statisch, wie bei klassischen Prozeduren, oder erst zur Programmlaufzeit, d.h. dynamisch, erfolgen. Die auszuführende Methode wird bei der dynamischen Bindung in Abhängigkeit vom Zielobjekt ermittelt. In Java kommen beide Bindungsarten vor. Um auch bei dynamischer Methodenbindung statische Analysen (z.B. Typkorrektheit) durchführen zu können, wird zur Übersetzungszeit jedem Methodenaufruf eine Methodendeklaration zugeordnet; diese nennen wir im folgenden *statische Methodendeklaration*. Zur Laufzeit wird entweder die statisch zugeordnete Methode oder eine diese überschreibende Methode ausgeführt. Eine ausführliche Darstellung des Algorithmus zur Methodenauswahl in Java findet sich in Abschnitt 2.1.

Wechselwirkungen. Zwei Aspekte verursachen eine enge Beziehung zwischen Kapselung und Methodenbindung: 1. Zur Übersetzungszeit muß geprüft werden, ob der Aufrufer einer Methode das Recht besitzt, auf die Methode zuzugreifen. Dabei betrachten wir als *Aufrufer* einer Methode die Klasse, aus der heraus der Aufruf erfolgt. Die Zugriffsprüfung kann nur anhand der statischen Methodendeklaration vorgenommen werden. Um sicherzustellen, daß auch die zur Laufzeit ausgeführte Methode zugreifbar ist, müssen überschreibende Methoden mindestens die gleichen Zugriffsrechte gewähren wie die überschriebene Methode. D.h., die Menge der Klassen, die auf die überschriebene Methode zugreifen dürfen, muß eine Teilmenge der Menge der Klassen sein, die auf die überschreibende Methode Zugriff haben. Dies muß durch syntaktische Kontextbedingungen garantiert werden. 2. Nutzer einer Klasse K haben nur Informationen über die Attribute

[1] Mit diesem Zugriffsmodus läßt sich insbesondere innerhalb von Paketen ein *friend-Mechanismus* wie in C++ nachbilden.

von K, auf die sie zugreifen dürfen. Das bedeutet insbesondere, daß Subklassen nicht von der Existenz einer verborgenen (z.B. privaten) Methode der Superklasse wissen. Daher können sie diese auch nicht überschreiben. Deklariert eine Subklasse dennoch eine Methode mit gleicher Signatur, so handelt es sich dabei nicht um Überschreiben, sondern um eine Neudefinition. Da diese neu definierten Methoden semantisch mit den verborgenen Methoden nichts gemeinsam haben, dürfen sie bei der Methodenauswahl nicht berücksichtigt werden, wenn sich der Aufruf auf die verborgene oder eine sie überschreibende Methode bezieht.

Diese beiden Aspekte spielen für die folgende Analyse der konzeptionell unausgereiften Kombination von Kapselung und Methodenauswahl in Java eine wesentliche Rolle. Die Schwächen dieser Kombination haben sich in der Vergangenheit bereits in etlichen Fehlern in Java Compilern manifestiert (vgl. [Sun], Bug Parade). Eine Durchsicht der dokumentierten Fehler belegt, daß insbesondere der paketweite Zugriff zu etlichen Problemen führt, was in der Auswertung eines Fehlerberichts mit "package-private access is a weird beast" (Bug Id 4094611) kommentiert wurde. Auch im Zusammenhang mit dem Überschreiben und Neudefinieren von paketlokalen Methoden über Paketgrenzen hinweg wurden bereits einige Fehler identifiziert: "... the sort of three-classes-and-two-packages problems that make this so tricky" (Bug Id 4026584).

Gliederung des Aufsatzes. Der Rest dieses Aufsatzes ist wie folgt aufgebaut: Abschnitt 2 erläutert den Methodenauswahlalgorithmus von Java. Anhand zweier Beispiele werden Unklarheiten und Fehler dieses Verfahrens demonstriert. Abschnitt 3 diskutiert mögliche Lösungsansätze, zeigt, wie die Methodenauswahl in Java korrigiert und vereinheitlicht werden kann, und präsentiert ein erweitertes Konzept für Zugriffsrechte. Abschnitt 4 enthält eine Zusammenfassung.

2 Fehler bei der Methodenauswahl in Java

Dieser Abschnitt faßt den von Java verwendeten Algorithmus zur Methodenauswahl zusammen. Anhand zweier Szenarien wird gezeigt, daß das Überschreiben von Methoden mit paketweitem Zugriff zu unklaren und fehlerhaften Situationen in Java führt. Ferner werden die Ursachen hierfür analysiert.

2.1 Methodenauswahl in Java

Um die Probleme des Zusammenspiels zwischen der Methodenauswahl und Zugriffsrechten genau analysieren zu können, werden wir im folgenden den von Java verwendeten Algorithmus zur Methodenauswahl beschreiben. Um die Präsentation zu vereinfachen, abstrahieren wir von denjenigen Aspekten, die für die hiesige Diskussion irrelevant sind, und machen folgende Annahmen:

- Alle Klassen sind öffentlich. (Java sieht auch die Möglichkeit vor, Klassen nur paketlokal zugreifbar zu machen.)
- Methoden haben keine Parameter. (Dadurch braucht das Überladen nicht betrachtet zu werden.)

- Methodenaufrufe haben immer die Form `Primary.Identifier()`.
- Interfaces und statische Methoden werden nicht betrachtet.
- Alle Methoden haben den Rückgabetyp `void`.
- Programme werden nach Modifikationen immer vollständig neu übersetzt. (Dadurch brauchen wir nicht auf die sonst notwendigen Laufzeitprüfungen einzugehen.)

Wir konzentrieren uns also auf den einfachen Standardfall: die Auswahl von parameterlosen Instanzmethoden. Nach der Sprachspezifikation (vgl. [GJS96], Kap. 15.11) erfolgt die Methodenauswahl in zwei Phasen: Zur Übersetzungszeit wird jedem Methodenaufruf statisch eine Methodendeklaration zugeordnet, anhand derer das Vorhandensein einer geeigneten Methode, die Zugreifbarkeit und die Typisierung geprüft werden (*Bindungsvorbereitung*). Zur Laufzeit wird dann in Abhängigkeit des Objekts, auf dem die Methode ausgeführt werden soll, eine Methodenimplementierung ausgewählt. Unter Berücksichtigung der obigen Einschränkungen erfordert die Bindungsvorbereitung drei Schritte:

1. Bestimmung der statischen Methodendeklaration. Diese ergibt sich aus dem Typ des Ausdrucks, auf dem die Methode aufgerufen wird, und aus dem Bezeichner der Methode.
2. Prüfung der Zugreifbarkeit in Abhängigkeit vom Zugriffsmodifikator der Methodendeklaration.
3. Festlegung des *Aufrufmodus*. Private Methoden erhalten den Aufrufmodus *nonvirtual*, was zu statischer Bindung führt; alle anderen hier betrachteten Methoden erhalten den Modus *virtual*.

Zur Laufzeit sind nochmals zwei Schritte notwendig, um die auszuführende Methode zu bestimmen:

4. Auswertung des Ausdrucks zur Bestimmung des sogenannten *Zielobjekts*, d.h. des Objektes, auf dem die Methode ausgeführt werden soll.
5. Lokalisierung der Methode. Besitzt der Aufruf den Modus nonvirtual (vgl. Schritt 3), so wird die statisch gefundene Methodendeklaration verwendet. Andernfalls wird die Methode dynamisch ausgewählt, indem zuerst in der Klasse des Zielobjekts, dann Schritt für Schritt in deren Superklassen nach einer Methode mit passender Signatur gesucht wird. Die erste gefundene Methode wird ausgeführt. Diese Suche terminiert spätestens beim Erreichen der statischen Methodendeklaration.

Das Verfahren zur Methodenauswahl muß zwei Forderungen garantieren: Zum einen soll die statisch gefundene oder ggf. eine diese überschreibende Methode ausgewählt werden. Zum anderen muß die ausgewählte Methode für den Aufrufer zugreifbar sein. In den beiden folgenden Abschnitten werden wir zeigen, daß beide Forderungen von Javas Auswahlverfahren nicht immer erfüllt werden.

2.2 Szenario 1: Überschreiben paketlokaler Methoden

Auf eine paketlokale Methode m darf nur aus Klassen zugegriffen werden, die zum selben Paket wie m gehören. Die Existenz paketlokaler Methoden wird

anderen Paketen gegenüber vollständig verborgen, so daß solche Methoden auch nur innerhalb ihres Pakets überschrieben werden können.

Nehmen wir an, die Klassen S und T befänden sich in verschiedenen Paketen `PS` und `PT` und S sei eine Subklasse von T. Deklariert nun T eine paketlokale Methode m_T und S eine Methode m_S mit identischer Signatur[2], so wird m_T in S nicht überschrieben, sondern es wird eine neue zusätzliche Methode eingeführt. Gestattet die neu definierte Methode geschützten oder öffentlichen Zugriff, so können in `PT` Klassen existieren, die sowohl auf m_T als auch auf m_S Zugriff haben.

```
package PT;

public class T {
  void m() { ... }
}
```

```
package PS;

public class S extends PT.T  {
  public void m() { ... }
}
```

Da es sich bei m_T und m_S um verschiedene Methoden mit gleichem Bezeichner handelt, muß zur Übersetzungszeit festgestellt werden können, ob einer Aufrufstelle m_T oder m_S als Deklarationsstelle zugeordnet werden soll. Dazu wird die statische Typinformation des Methodenaufrufs herangezogen: Ist der Empfänger vom statischen Typ T, so wird m_T als statische Deklaration verwendet, handelt es sich um einen Ausdruck vom Typ S, so wird m_S gewählt. In Subtypen von S — sofern m nicht überschrieben wurde — verdeckt m_S die Methode m_T. Die im Paket `PT` stehende Anweisungsfolge

```
T v = new PS.S();
v.m();
```

führt also zur Auswahl von m_T als statischer Methodendeklaration. Bei genauem Nachvollziehen des oben beschriebenen Algorithmus stellt man jedoch fest, daß dieser dynamisch zum Aufruf von m_S führt, obwohl es sich dabei um eine völlig andere Methode handelt, die m_T nicht überschreibt!

Das Szenario wird noch interessanter, wenn wir das Paket `PT` um folgende Klasse erweitern:

```
package PT;

public class U extends PS.S {
  public void m() { ... }
}
```

Hier stellt sich nämlich die Frage, ob die Methode m_U sowohl m_T als auch m_S überschreibt oder nur eine von beiden. Der Sprachbericht bleibt diesbzgl. unklar. Einerseits steht in Kap. 8.4.6.1: "If a class declares an instance method, then the declaration of that method is said to *override* any and all methods with the same signature in the superclasses ... of the class that would otherwise be accessible to code in the class"; das bedeutet u.E., daß beide Methoden überschrieben werden (auch wenn es semantischen Vorstellungen widerspricht, zwei völlig unabhängige Methoden, die nur zufällig den gleichen Bezeichner haben, mit einer Methode zu

[2] Gemäß der vereinfachenden Annahmen aus Abschnitt 2.1 sind zwei Signaturen von Methoden mit gleichem Bezeichner immer identisch.

überschreiben). Andererseits steht in Kap. 8.4.6.4: "It is not possible for two or more inherited methods with the same signature not to be `abstract`, ...". Dies legt nahe, daß m_T und m_S nicht gleichzeitig geerbt werden können.

Insgesamt liefert obiges Beispiel nur einen kleinen Ausschnitt einer Klasse von unklaren Situationen und Problemen. Neben dem oben beschriebenen Fehler im Methodenauswahlalgorithmus hat das zu etlichen ähnlich gelagerten Fehlern in Java Compilern geführt. So wird z.B. bei Verwendung des Sun Compilers[3] in folgendem Beispiel m_S aufgerufen, obwohl die dynamische Bindung zu einem Aufruf von m_U führen müßte:

```
PS.S v = new U();
v.m();
```

Zusammenfassend kann festgestellt werden, daß der Bezeichner einer Methode alleine nicht ausreicht, um zu entscheiden, ob eine Methode zur Laufzeit aufgerufen werden darf. Zusätzlich wird Information darüber benötigt, ob eine dynamisch gefundene Methode die statisch ausgewählte Methode überschreibt. Dazu muß insbesondere der Begriff des Überschreibens geklärt werden.

2.3 Szenario 2: Überschreiben geschützter Methoden

Auch das zweite Szenario verwendet eine Konstruktion mit drei Klassen und zwei Paketen:

```
package PT;

public class T {
  protected void m() { ... }
}
```

```
package PS;

public class S extends PT.T  {
  protected void m() { ... }
}
```

In diesem Beispiel wird die geschützte Methode m_T von der ebenfalls geschützten Methode m_S überschrieben. Definiert man nun im Paket `PT` eine Klasse C, die keine Subklasse von T oder S ist, so hat C zwar Zugriff auf m_T (beide befinden sich im Paket `PT`), nicht jedoch auf m_S. Nehmen wir an, in C befinden sich folgende Anweisungen:

```
T v = new PS.S();
v.m();
```

Der Methodenauswahlalgorithmus wählt an dieser Stelle m_S zur Ausführung aus, obwohl diese Methode von C aus gar nicht zugreifbar ist! Der Fehler liegt hier darin, daß die Zugreifbarkeit nur anhand der statisch gefundenen Methodendeklaration geprüft wird und gemäß Sprachspezifikation daraus geschlossen wird, daß die überschreibende Methode ebenfalls zugreifbar ist. Der Sprachbericht geht nämlich von der falschen Annahme aus, daß die überschreibende Methode "mindestens so zugreifbar" ist, wie die überschriebene Methode (vgl. [GJS96], Kap. 8.4.6.3). Wie das Beispiel zeigt, ist diese Annahme aber falsch: m_S ist in `PT` im allg. weniger zugreifbar als m_T.

Der folgende Abschnitt behandelt Lösungsansätze für die durch beide Szenarien skizzierten Problemklassen.

[3] sowohl unter JDK 1.1 als auch JDK 1.2 beta

3 Bewältigung der Probleme

Dieser Abschnitt behandelt Lösungen für die oben dargestellten Probleme. Um die Entwurfsentscheidungen für die danach beschriebenen Lösungen vorzubereiten, diskutieren wir zunächst in Kürze zwei Ansätze, die zu adhoc-Lösungen führen können, aber nicht die grundlegende Problematik bewältigen: 1. Spracheinschränkungen und 2. Aufweichen der Kapselung. Die beiden Hauptteile des Abschnitts präsentieren eine korrigierte und vereinheitlichte Fassung des Methodenauswahlalgorithmus und eine verfeinerte Version der Zugriffsrechte, die den neuen Methodenauswahlalgorithmus in geeigneter Weise ergänzt.

Spracheinschränkungen. Die in Szenario 1 skizzierten Situationen kann man verhindern, indem per Kontextbedingung verboten wird, daß eine Klasse Zugriff auf verschiedene Methoden mit gleichem Bezeichner bekommen kann. Abgesehen davon, daß Spracheinschränkungen bereits existierende Java-Programme ungültig machen, trägt diese Einschränkung wenig zur Orthogonalität der Sprache bei: Sie fügt eine neue Regel hinzu, anstatt die existierenden Regeln zu verallgemeinern. Außerdem bietet sie keine Lösung für das Problem aus Szenario 2.

Aufweichen der Kapselung. Abgesehen von Ungenauigkeiten haben die obigen Szenarien eine Inkonsistenz in der Sprachspezifikation von Java freigelegt. Sie entsteht dadurch, daß 1. Zugreifbarkeit definiert wird, 2. ein Methodenauswahlalgorithmus angegeben wird und 3. behauptet wird, daß der Auswahlalgorithmus ein bestimmtes dynamisches Zugriffsverhalten garantiert. Die Inkonsistenz läßt sich beseitigen, indem die Behauptung bzgl. des dynamischen Zugriffsverhaltens modifiziert wird. Beispielsweise könnten Zugriffsmodifikatoren ausschließlich die statische Zugreifbarkeit regeln ohne Einfluß auf die Zugreifbarkeit zur Laufzeit zu haben. Allerdings setzt man damit de facto die Kapselung außer Kraft: Es können nämlich Methoden aufgerufen werden, deren Existenz (und erst recht deren Verhalten) an der Aufrufstelle gar nicht bekannt sind (z.B. die Methode m_S in Szenario 2). Die Modifikation (z.B. Umbenennung) solcher eigentlich gekapselter Methoden verändert das Verhalten der aufrufenden Methode. Somit führt ein Aufweichen der Kapselung zu einer Verletzung der Grundsätze der objektorientierten Programmierung, weshalb wir diesen Ansatz nicht empfehlen.

3.1 Präzisierung und Korrektur der Methodenauswahl

Basierend auf einer Präzisierung der notwendigen Begriffe, einem korrigierten und vereinheitlichten Algorithmus zur Methodenauswahl und einer dynamischen Prüfung der Zugriffsrechte präsentieren wir eine Lösung für die skizzierten Probleme, die den Sprachumfang von Java nicht verändert. Im Anschluß diskutieren wir ihren Nachteil, dessen Überwindung sich der folgende Abschnitt widmet.

Überschreiben. Zunächst müssen wir den Begriff des Überschreibens klären. Sei S eine Klasse, in der eine Methode m deklariert ist, und T die direkte Superklasse von S. Hat S das Zugriffsrecht auf eine Methode m in T mit gleicher Signatur,

dann sagen wir, daß m die von T ererbte Methode *direkt überschreibt*. Andernfalls handelt es sich um eine Neudefinition. (Diese verschattet ggf. eine in S nicht zugreifbare Methode von T mit gleicher Signatur.) Eine Methode m_0 *überschreibt* eine andere Methode m_n, wenn es eine Kette von Methoden $m_0, \ldots, m_n$ gibt, so daß m_i die Methode m_{i+1} direkt überschreibt, d.h., daß zwischen m_0 und m_n keine Neudefinition gleicher Signatur liegt.

Angewandt auf den Fall von Szenario 1 bedeutet diese Definition, daß m_U die Methode m_S überschreibt (sogar direkt), aber nicht die Methode m_T.

Vereinheitlichung. Die Diskussion von Szenario 1 hat gezeigt, daß der Auswahlalgorithmus von Java so modifiziert werden muß, daß eine dynamisch gefundene Methode nur dann ausgewählt wird, wenn sie die statische Deklaration überschreibt. Da private Methoden nicht überschrieben werden können, bedeutet dies insbesondere, daß das modifizierte Verfahren auch für private Methoden funktioniert, da es in diesem Fall stets die statisch gefundene Methode auswählt. Es liegt daher nahe, *konzeptionell* nicht zwischen den Aufrufmodi für private und nicht private Methoden zu unterscheiden, sondern alle dem gleichen Auswahlverfahren zu unterwerfen.[4] Der entscheidende Vorteil dieser Betrachtungsweise liegt darin, daß die Aufrufsemantik einer Methode nun nicht mehr von ihrem Zugriffsmodus abhängt. Somit wurde eine vollständige Orthogonalisierung von Zugriffsrechten und Methodenauswahl erreicht.

Auswahlalgorithmus. Wir schlagen vor, den Methodenauswahlalgorithmus so zu ändern, daß nur Methoden ausgewählt werden, die die statische Deklaration überschreiben. Diese Änderung erlaubt es auch, auf die Unterscheidung der Aufrufmodi virtual/nonvirtual zu verzichten, da private Methoden nicht überschrieben werden können. Damit entfällt der dritte Schritt der Bindungsvorbereitung (vgl. Abschnitt 2.1). Wenn die Zugriffsrechte und deren Kontextbedingungen in Java unverändert bleiben sollen (vgl. Abschnitt 3.2), was wir hier vorausgesetzt haben, benötigen wir zur Laufzeit drei Schritte, die wir zum Vergleich mit dem alten Auswahlalgorithmus mit 4, 5 und 5a bezeichnet haben:

4. Auswertung des Ausdrucks zur Bestimmung des Zielobjekts.
5. Lokalisierung der Methode. Wiederum beginnt die Suche in der Klasse des Zielobjekts und setzt sich dann in deren Superklassen fort. Die erste Methode mit passender Signatur, die *die statisch gefundene Methode überschreibt*, oder die statisch gefundene Methode selbst wird ausgewählt.

5a. Prüfung der Zugriffsrechte: Ist die ausgewählte Methode an der Aufrufstelle zugreifbar, so wird sie ausgeführt. Andernfalls wird ein `IllegalAccessError` erzeugt.

Diskussion. Die Stärke der beschriebenen Lösung besteht darin, daß sie den Sprachschatz von Java unverändert läßt und den Algorithmus konzeptionell vereinfacht, indem auf die Unterscheidung zwischen den Aufrufmodi virtual und

[4] Zu Zwecken der Optimierung können Compiler natürlich private Methoden weiterhin statisch binden.

nonvirtual verzichtet wird. Dies erhöht auch die Orthogonalität, da die Aufrufsemantik nicht mehr vom Zugriffsmodus abhängt. Die skizzierten Unklarheiten der Sprachspezifikation werden beseitigt und illegale Zugriffe abgefangen.

Allerdings entspricht die dynamische Prüfung im Schritt 5a nicht den Erwartungen an eine stark typisierte Sprache, von der man auch erwartet, daß die Zugriffsrechte statisch geprüft werden können. Wegen der in Szenario 2 dargelegten Überlappung der Zugriffsrechte „geschützt“ und „paketlokal“ ist dies jedoch ohne leichte Änderung des Sprachschatzes von Java nicht möglich. Einen Vorschlag für eine derartige Änderung präsentiert der folgende Abschnitt.

3.2 Verfeinerte Zugriffsrechte

In diesem Abschnitt beschreiben wir ein verbessertes Konzept für Zugriffsrechte, das eine statische Prüfung der Zugreifbarkeit ermöglicht, so daß der Schritt 5a in obigem Auswahlalgorithmus entfallen kann. Dieses Konzept verfeinert die aktuellen Zugriffsrechte in Java, da es möglich wird, Subklassen Zugriff auf ein Attribut zu gewähren, ohne dieses gleichzeitig paketweit zugreifbar zu machen.

Statische Prüfung von Zugriffsrechten. Zugriffsrechte von Methoden lassen sich statisch prüfen, wenn die Programmiersprache folgenden Grundsatz garantiert: *Eine überschreibende Methode gewährt die gleichen bzw. mehr Zugriffsrechte wie/als die überschriebene Methode.* Denn statt der statisch gefundenen Methode kann nur eine sie überschreibende Methode ausgeführt werden. Die Zugriffsregeln von Java verletzen diesen Grundsatz: Geschützte Methoden gewähren allen Klassen ihres Pakets und allen ihren Subklassen den Zugriff. Wird eine geschützte Methode m_T einer Klasse T im Paket `PT` von einer geschützte Methode m_S einer Subklasse von T in einem anderen Paket überschrieben, gewährt m_S den Klassen von `PT` keinen Zugriff — im Gegensatz zu m_T (siehe Szenario 2). Außerdem gewährt m_S im Gegensatz zu m_T der Klasse T keinen Zugriff (vgl. [GJS96], Kap. 6.6.2; die JDK Comiler lassen allerdings diese Form des Zugriffs zu).

Neben kleineren Änderungen läßt sich der Grundsatz in Java einhalten, wenn geschützte Methoden in anderen Paketen nur mit öffentlichen Methoden überschrieben werden dürfen. Ohne eine Verfeinerung der Zugriffsrechte führt eine solche Änderung allerdings dazu, daß sich gekapselte Vererbungshierarchien nicht mehr über mehrere Pakete erstrecken könnten. Dies wäre eine unakzeptable Einschränkung. Wir schlagen deshalb vor, die notwendigen Änderungen der Zugriffsregeln mit der Einführung eines neuen Zugriffsrechts zu verbinden.

Verfeinerte Zugriffsrechte und -regeln. Zur Bewältigung der skizzierten Problematik sollte Java die folgenden Zugriffsmodi mit der angegebenen gegenüber der alten Fassung leicht modifizierten Bedeutung bereitstellen:

1. `private`: Private Attribute sind nur für die umfassende Klasse zugreifbar.
2. `private protected`: Gesondert geschützte Attribute gewähren den Super- und Subklassen und der umfassenden Klasse das Zugriffsrecht.[5]

[5] Als Zugriffsmodifikator verwenden wir hier `private protected`, da dieser in einer frühen Versionen von Java schon einmal existiert hat (vgl. [CH96], Seite 201).

3. default access: Paketlokale Attribute gewähren allen Klassen des Pakets das Zugriffsrecht.
4. `protected`: Geschützte Attribute gewähren den Super- und Subklassen und allen Klassen des Pakets das Zugriffsrecht.
5. `public`: Öffentliche Attribute gewähren allen Klassen das Zugriffsrecht.

Der obige Grundsatz gibt dann die Kontextregeln vor, die wir in folgender Tabelle zusammenfassen. Links steht dabei der Zugriffsmodus der überschriebenen Methode, rechts die möglichen Modi der sie direkt überschreibenden Methode:

Überschriebene Methode	Überschreibende Methode
private protected	private protected, protected, public
default access	default access, protected, public (im gleichen Paket)
protected	im gleichen Paket: protected, public in anderen Paketen: public
public	public

Gemäß der Definition von Überschreiben ist es nicht möglich, paketlokale Methoden in anderen Paketen oder private Methoden zu überschreiben.

4 Zusammenfassung

In diesem Aufsatz haben wir gezeigt, daß das komplexe Zusammenspiel von Zugriffsrechten, Paketkonzept und dynamischer Methodenauswahl in Java zu komplizierten syntaktischen Kontextbedingungen und einer oft schwer nachvollziehbaren Aufrufsemantik für Methoden führt. Daraus resultieren häufig unklare Situationen, die etliche Compilerfehler und Fehler in der Sprachspezifikation nach sich ziehen. Letztere haben wir anhand zweier Szenarien aufgezeigt und analysiert.

Darauf aufbauend haben wir beschrieben, wie die bestehende Sprachspezifikation so korrigiert werden kann, daß die Probleme gelöst werden. Dabei haben wir gezeigt, daß die Auswahlverfahren für private und nicht private Methoden konzeptionell vereinheitlicht werden können, wodurch die Aufrufsemantik vereinfacht wird und eine Orthogonalisierung von Aufrufsemantik und Zugriffsrechten erreicht wird. Außerdem haben wir eine Korrektur und Verfeinerung der Zugriffsrechte präsentiert, die eine statische Überprüfung der Zugreifbarkeit ermöglichen.

Literatur

[CH96] G. Cornell and C. S. Horstmann. *Java bis ins Detail.* Heise, 1996.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[Sun] Sun. Java developer connection. Available from `http://java.sun.com/jdc`.