# Law of Demeter

Patricia Maier

Hochschule Esslingen
WS 2014/2015

# 1 Das Gesetz von Demeter

Das **Law of Demeter** (LoD, deutsch "Gesetz von Demeter") oder auch "Principle Of Least Knowledge" [1], "Law of Goodstyle" [1, p. 323] beziehungsweise "Law of Demeter for Functions/Methods" (LoD-F) [2] wurde erstmals im Herbst 1987 von Ian Holland an der Northeastern University in Boston im Rahmen des Forschungsprojektes „Demeter“[1] aufgestellt [3]. Nach K. Lieberherr, I. Holland und A. Riel wird mit dem Gesetz von Demeter eine programmiersprachenunabhängige Richtlinie, welche die Idee der **Modularisierung und Kapselung** in der objektorientierten Softwareentwicklung aufgreift, beschrieben [1, p. 323].

K. Lieberherr, I. Holland und A. Riel formulierten den Zweck des **Law of Demeter** erstmals in ihrer 1988 erschienenen Veröffentlichung mit den Worten "The motivation behind this law is to ensure that the software is as modular as possible." [1, p. 325]. Die Modularisierung von Software geht mit einer schwachen Kopplung der einzelnen Komponenten einher, weshalb das LoD auch als ein **Spezialfall** des Entwurfsprinzips **loose coupling** betrachtet werden kann [3].

Spricht man in der Softwareentwicklung von Modularität und einer schwachen Kopplung, so setzt dies eine **Einschränkung der gegenseitigen Bekanntheit der einzelnen Softwarekomponenten** und damit ihrer gegenseitigen Kommunikation voraus. Karl Lieberherr beschreibt das Principle Of Least Knowledge dementsprechend mit der Formulierung "Each unit should have only limited knowledge about other units: only units **closely** related to the current unit" [3], mit welcher er eine Verbindung zum Entwurfsprinzip **information hiding** herstellt. Die Kommunikation schränkt Lieberherr darüber hinaus mit der Aussage "Each unit should only talk to its friends; **Don‘t talk to strangers**." [3] stark ein. Unter einer "unit" versteht Lieberherr im Zusammenhang mit dem LoD eine Methode [3]. Das Geheimhaltungsprinzip[2] ist folglich bei der Betrachtung des Law of Demeter von großer Bedeutung. Methoden, welche mit dem

[1] Ein Projekt zur Entwicklung von verschiedenen Tools, welche die Softwareentwicklung unter Einhaltung des LoD leichter machen.
[2] Oder auch „Information hiding“

Gesetz von Demeter konform sind, kennen nur die interne Struktur "befreundeter" Klassen. Strukturelle Informationen bezüglich "fremder" Klassen bleiben verborgen.

Wird Lieberherrs Aussage – nur mit Freunden und nicht mit Fremden zu sprechen – beispielsweise auf die in folgendem Bild dargestellte Klassenkonstellation übertragen, so hat dies bezüglich erlaubter Methodenaufrufe die im Folgenden erläuterte Bedeutung. Innerhalb der Methode `test()` der Klasse `A` darf auf die befreundete Klasse `B` über die Referenz `refB` und deren Methoden zugegriffen werden. Ein Aufruf wie beispielsweise der von `refB.doSomethingB()` ist demnach vollkommen legitim, denn `A` spricht nur mit dem Freund `B`. Folgende Abbildung zeigt die Konstellation dieser Klassen in einem Klassendiagramm:

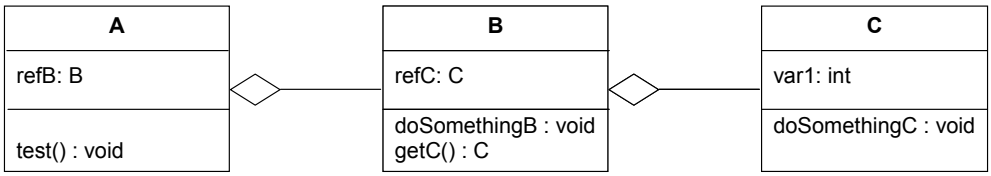


*Bild 1-1: einfaches Klassendiagramm LoD*

Erfolgt innerhalb der Methode `test()` jedoch ein Aufruf von `refB.getC().doSomethingC()`, so wird über `B` hinweg auf eine der Klasse `A` unbekannte Klasse `C` zugegriffen. Das Geheimhaltungsprinzip und folglich das Gesetz von Demeter wird durch einen derartigen Methodenaufruf verletzt. Im Zuge der Einhaltung des Entwurfsprinzips **information hiding** wird durch das LoD somit die **zulässige Verkettung von Methodenaufrufen** drastisch **limitiert**.

Soll das Wissen über andere Komponenten und deren Struktur auf Freunde beschränkt werden, so muss die **Kommunikation zweier Fremder** (hier `A` und `C`) durch eine zusätzliche Methode – in der dazwischenstehenden Klasse `B` – gekapselt werden. Diese Methoden werden auch als **Wrapper-Methoden**[3] bezeichnet [4]. Sie ermöglichen auf indirektem Weg die Kommunikation zwischen zwei nicht explizit befreundeten Objekten.

Ziel des Law of Demeter ist es, Abhängigkeiten zu reduzieren [5] und so ein Softwaresystem mit minimalen Abhängigkeiten bereitzustellen. Welche konkreten Methodenaufrufe im Rahmen des LoD erlaubt sind, wurde von Lieberherr, Holland und Riel in verschiedenen Formulierungen, welche **unterschiedliche Ausprägungen des Gesetzes** beschreiben, konkretisiert.

## 1.1 Formen des Law of Demeter

Wie bei vielen anderen Entwurfsprinzipien gibt es auch beim Gesetz von Demeter verschiedene Ausprägungen, Interpretationen und Entwicklungsstufen. So unterschieden die Erfinder des LoD, K. Lieberherr, I. Holland und A. Riel, im Jahre 1988 in ihrer Veröffentlichung [1] ein "**weak Law of Demeter**" von einem "**strong Law of Demeter**". Im darauffolgenden Jahr sprechen K. Lieberherr und I. Holland hingegen von zwei weiteren Kategorien, einer "**Objektform**" und einer "**Klassenform**" [6]. Die folgenden Unterkapitel befassen sich mit diesen Ausprägungen.

[3] Auch als Vermittler-Methoden bezeichnet

### 1.1.1 Objektform

Die Formulierung des Gesetzes von Demeter in der Objektform ist die am häufigsten verwendete Form. Somit kann diese auch als allgemeine beziehungsweise generelle Ausprägung des Law of Demeter betrachtet werden. Mit der Objektform legen Lieberherr, Holland und Riel die Grenzen fest, innerhalb derer Methodenaufrufe auf Objekten mit dem Law of Demeter konform sind. Es wird definiert, welche Objekte als befreundet betrachtet werden dürfen. Die folgende **Definition der Objektform** ist an [6] angelehnt.

Eine Methode `m` eines Objektes `:A` darf nach dem LoD nur folgende Methoden aufrufen:

- Methoden des Objektes `:A` selbst,
- Methoden der an `m` übergebenen Objekte,
- Methoden von lokal in `:A` neu erzeugten Objekten und
- Methoden eines globalen[4] Objektes von `:A`.

Folgender Programmcode[5] zeigt dies an konkreten Beispielen in der Programmiersprache Java:

```
public class MyLawOfDemeter {
  private A aRef = new A();
  private void doSomething(){}

  public void testMethod(B bRef){
     C cRef = new C();
     this.doSomething();    //(1) -> current class
     bRef.doSomething();    //(2) -> argument class
     cRef.doSomething();    //(3) -> immediate part class
     aRef.doSomething();    //(4) -> immediate part class
  }
}
```

Hiermit einhergehend beschreibt Lieberherr das Law of Demeter auch mit den Worten "[ ] you should only talk to **yourself** (current class), to **close relatives** (**immediate part classes**), and **friends** who visit you (argument classes)" [7, p. 203]. Eine weitere Verkettung wie beispielsweise `aRef.getD().doSomething()` innerhalb der Methode `testMethod()` der Klasse `MyLawOfDemeter` würde gegen das Gesetz von Demeter verstoßen. Denn ein Objekt der Klasse `D`, welches von `getD()` zurückgegeben werden würde, ist der aktuellen Klasse nicht bekannt. Die Klasse `MyLawOfDemeter` würde demnach mit dem "Fremden" `D` "sprechen", was nicht erlaubt ist.

### 1.1.2 Klassenform

Die Klassenform ist der Objektform sehr ähnlich, wird jedoch nicht nur auf Objekte, sondern allgemein auf ganze Klassen bezogen. Hierbei findet eine Unterscheidung in eine **minimierte** und eine **strikte Form** statt. Folgende Definitionen der strikten und minimierten Form sind an [6] und [8] angelehnt.

---

[4] Ein globales Objekt wird auch als „Komponentenobjekt“ bezeichnet [2].
[5] angelehnt an [10]

**Strikte Form:** Eine Methode `m` der Klasse `A` darf folgende Methoden aufrufen:

- Methoden der Klasse `A` selbst,
- Methoden der Klasse eines Übergabeparameters von `m`,
- Methoden von Klassen, deren Instanzen lokal neu erzeugt wurden oder
- Methoden einer Klasse eines globalen[6] Objektes.

**Minimierte Form:** Die minimierte Form lockert die Einschränkungen der strikten Form etwas auf. Sie erlaubt den Zugriff auf Methoden weiterer Klassen, nämlich

- einer Klasse, welche stabil ist oder ein stabiles Interface implementiert[7],
- einer Klasse, auf deren Methoden ein direkter Zugriff aus Laufzeit-Effizienzgründen nötig ist, oder
- einer Klasse eines neu erzeugten Objektes.

Diese Form des LoD ist die schwächste Ausprägung des Gesetzes und bringt somit die wenigsten Einschränkungen für erlaubte Methodenaufrufe mit sich.

### 1.1.3 Weak Law of Demeter

Das "**weak Law of Demeter**" wurde im Jahre 1988 von K. Liebherr, I. Holland und A. Riel wie folgt definiert:

"The Weak Law of Demeter defines instance variables as being BOTH the instance variables that make up a given class AND any instance variables inherited from other classes." [1, p. 329]

Dies bedeutet, dass von einer Klasse aus sowohl auf die eigenen Instanzvariablen, als auch auf die von der Basisklasse **geerbten Variablen** innerhalb von Methoden zugegriffen werden darf. Wird das weak Law of Demeter angewandt, so wirken sich Änderungen der Basisklasse auch auf die Methoden von abgeleiteten Klassen aus, welche diese von Änderungen betroffenen Objekte der Basisklasse direkt verwenden. [1, p. 329]

### 1.1.4 Strong Law of Demeter

Während das weak Law of Demeter den Zugriff der Methoden auf geerbte Variablen zulässt, grenzt das **strong Law of Demeter** den Zugriff auf geerbte Variablen ein.

"The Strong Law of Demeter defines instance variables as being ONLY the instance variables that make up a given class. Inherited instance variable types may not be passed messages." [1, p. 329]

Gemäß dem strong Law of Demeter wird eine Variable nur dann als **Instanzvariable** bezeichnet, wenn sie **direkter Bestandteil der betrachteten Klasse** ist. Ein Zugriff in einer Methode einer abgeleiteten Klasse auf eine Variable, welche durch Vererbung an die abgeleitete Klasse weitergereicht wurde, ist nicht legitim. Deshalb fordert das **strong Law of Demeter** oftmals zusätzliche Methoden (**Wrapper-**

[6] In Java kann ein Objekt mithilfe von `public` und `static` als global definiert werden.

[7] D. h., dass sich die Klasse beziehungsweise das Interface nicht ändert. Methodenaufrufe usw. sind fix.

**Methoden**) [1, p. 329], welche die Abhängigkeit der Methoden der abgeleiteten Klasse von der konkreten Struktur der Basisklasse kapseln. Diese Wrapper-Methoden fungieren als **Vermittler zwischen** den **Attributen der Basisklasse und** den Methoden **der abgeleiteten Klasse**. Dabei ist es wichtig, dass sie in der **Basisklasse implementiert** und **nicht** von den abgeleiteten Klassen **überschrieben** werden. Kommt es zu Änderungen der internen Struktur – also beispielsweise der Attribute – der Basisklasse, so betrifft dies folglich nur die entsprechende Wrapper-Methode, nicht aber die Methoden der abgeleiteten Klassen.

Diese Ausprägung des LoD unterstützt damit vor allem das Prinzip des **information hiding** stärker als das weak Law of Demeter und reduziert somit die Abhängigkeiten zwischen einer Basisklasse und ihren abgeleiteten Klassen.

### 1.1.5 Beispiel für das Weak und Strong Law of Demeter

K. Lieberherr, I. Holland und A. Riel erläutern den Unterschied des strong und weak LoD in "Object-Oriented Programming: An Objective Sense of Style" [1, pp. 329,330] beispielhaft an einem Obstkorb. Hierbei befinden sich in einem Obstkorb (Klasse `Obstkorb`) jeweils ein einziger Apfel, eine einzige Orange und eine einzige Pflaume. Die Klassen `Apfel`, `Orange` und `Pflaume` sind von der Klasse `Frucht` abgeleitet, welche ein Attribut `Gewicht` besitzt.

Zur Berechnung des Gewichts der unterschiedlichen Früchte des Obstkorbs sind unterschiedliche Formeln nötig, da sich der eigentliche Fruchtanteil je nach Sorte unterscheidet. So muss beispielsweise bei einer Pflaume das Gewicht des Kerns abgezogen werden und bei einer Orange das Gewicht der Schale. Es soll das Gewicht jeder einzelnen Fruchtsorte mit Hilfe der entsprechenden Formel ermittelt werden.

Wird das **weak Law of Demeter** angewandt, so besitzen die Klassen `Apfel`, `Orange` und `Pflaume` jeweils eine eigene Methode mit der Bezeichnung `berechneGewicht()`, welche das Gewicht anhand des von der Klasse `Frucht` geerbten Attributes `Gewicht` und eines Prozentsatzes berechnet. Das folgende Beispiel zeigt die Implementierung der Klasse `Frucht`, der Klasse `Obstkorb` und beispielhaft für eine konkrete Frucht, die Klasse `Apfel`, unter Berücksichtigung des weak LoD:

```
public abstract class Frucht{
    float Gewicht = 1.0f;
    public Frucht(){};
    public abstract float berechneGewicht();
}

public class Apfel extends Frucht{
    public float berechneGewicht() {
        return Gewicht*0.85f;
    }
}

public class Obstkorb {

    private ArrayList<Frucht> fruechte = new ArrayList<Frucht>();
```

```
    private float obstkorbGewicht = 0;
    public Obstkorb()   {
        fruechte.add(new Apfel());
        ...
    }

    public float berechneGewicht() {
        for (Frucht elem : fruechte){
            obstkorbGewicht += elem.berechneGewicht();
        }
        return obstkorbGewicht;
    }
}

public abstract class Frucht{
    float Gewicht = 1.0f;
    public Frucht(){};
    public abstract float berechneGewicht();
}

public class Apfel extends Frucht{
    public float berechneGewicht() {
        return Gewicht*0.85f;
    }
}

public class Obstkorb {

    private ArrayList<Frucht> fruechte = new ArrayList<Frucht>();
    private float obstkorbGewicht = 0;
    public Obstkorb()   {
        fruechte.add(new Apfel());
        ...
    }

    public float berechneGewicht() {
        for (Frucht elem : fruechte){
            obstkorbGewicht += elem.berechneGewicht();
        }
        return obstkorbGewicht;
    }
}
```

Der Zugriff der abgeleiteten Klasse `Apfel` auf das Attribut `Gewicht` der Basisklasse `Frucht` ist nach dem strong LoD unzulässig. Wird das strong LoD befolgt, so kapselt eine Wrapper-Methode (**`berechne-ProzentualesGewicht()`**), welche sich in der Basisklasse befindet und von den abgeleiteten Klassen aufgerufen wird, den Zugriff auf das Attribut `Gewicht`. Die abgeleiteten Klassen verfügen nun über keine genauen Informationen bezüglich der Implementierung der Methode beziehungsweise der Attribute der Basisklasse.

Folgender Programmcode zeigt die Implementierung des Obstkorb-Beispiels unter Einhaltung des **strong Law of Demeter**:

```
public abstract class Frucht{
    float Gewicht = 1.0f;
```

```
    public Frucht(){};
    public abstract float berechneGewicht();
    public float berechneProzentualesGewicht(float prozent)    {
        return Gewicht * prozent;
    }
}

public class Apfel extends Frucht{
    public float berechneGewicht() {
        return this.berechneProzentualesGewicht(0.85f);
    }
}

public class Obstkorb {
    private ArrayList<Frucht> fruechte = new ArrayList<Frucht>();
    private float obstkorbGewicht = 0;
    public Obstkorb()   {
        fruechte.add(new Apfel());
        ...
    }
    public float berechneGewicht() {
        for (Frucht elem : fruechte){
            obstkorbGewicht += elem.berechneGewicht();
            System.out.println(obstkorbGewicht);
        }
        return obstkorbGewicht;
    }
}
```

## 1.2 Vorteile

Das Gesetz von Demeter hilft Entwicklern, **eine robuste und gut strukturierte Software** zu erstellen. In Kombination mit der Minimierung von Code-Duplizierungen, der Anzahl an Übergabeparametern von Methoden und der Anzahl der Methoden einer Klasse wird unter anderem eine **geringere Kopplung der Methoden** und verstärktes „**information hiding**“ erreicht. Dies führt zu einer unabhängigen Austauschbarkeit, Testbarkeit und Wiederverwendbarkeit der einzelnen Softwarekomponenten und einer leichteren Wartbarkeit der kompletten Software [1, p. 323]. Anhand der Entkopplung der Beziehungen über mehrere Klassen hinweg kann die Komplexität der Methoden reduziert und eine Verständlichkeit der Software durch eine stärkere Realitätsnähe erzielt werden. Dies ist beispielsweise bei David Bocks Paperboy der Fall. Kein Kunde würde dem Zeitungsjungen den Geldbeutel in die Hand geben, damit dieser sich selbst seinen Lohn entnehmen kann.

Auf Methodenebene führt das LoD zu sogenannten **narrow interfaces**, da jede Methode nur über eine geringe Anzahl Methoden der befreundeten Objekte Bescheid wissen muss [3].

Darüber hinaus deckt das Law of Demeter weitere Entwurfsprinzipien ab, wie beispielsweise **loose coupling** und **information hiding**.

## 1.3 Nachteile

Aufgrund des Verbotes, mit Fremden zu sprechen, sind oft zusätzliche Methoden – **Wrapper-Methoden** – nötig, um die gewünschte Funktionalität zu implementieren. Diese Methoden fordern nicht selten eine große Anzahl an Übergabeparametern [6].

Darüber hinaus führt das Gesetz von Demeter auf Klassenebene zu sogenannten **wide interfaces** (umfangreiche Schnittstellen), da die Einhaltung des LoD, wie bereits erwähnt, zusätzliche Helfer-Methoden, die Wrapper-Methoden, benötigt, um das Verschachteln durch Objektstrukturen hindurch zu vermeiden [3].

Dementsprechend **mächtige Interfaces auf Klassenebene** sind unter Umständen problematisch, da sie gegen einige Entwurfsprinzipien der Softwareentwicklung wie beispielsweise das interface segregation principle, das single responsibility principle oder separation of concerns, verstoßen.

Die Implementierung einer Software unter Einhaltung des Gesetzes von Demeter kann darüber hinaus zu einem geringfügigen Performance-Verlust oder einem etwas erhöhten Speicherbedarf führen [6].

## 1.4 Fazit

Das Gesetz von Demeter setzt sich wie die meisten Software-Entwurfsprinzipien mit der Reduktion von Abhängigkeiten auseinander. Hierbei versucht es durch das Verbot bestimmter Methodenzugriffe, diese Abhängigkeiten zu eliminieren. Somit profitiert ein unter **LoD** entwickeltes Softwaresystem von einer besseren Strukturierung, welche eine **Verringerung der Abhängigkeiten** bewirkt und darüber hinaus die Übersichtlichkeit, Wartbarkeit und Testbarkeit stark erhöht. Passend hierzu trägt das Gesetz von Demeter oftmals auch die Bezeichnung **Law of Goodstyle** [1, p. 323].

Mit der Aussage "[ ] any object-orientated program written in bad style can be transformed systematically into a structured program obeying the Law of Demeter." [1, p. 324] zeigen Lieberherr, Holland und Riel, dass das Law of Demeter allgemein auf jedes schlechte Softwaredesign mit Erfolg angewandt werden kann.

Das Law of Demeter ist im Allgemeinen nicht als überall geltend und unanfechtbar anzusehen. Es ist eher als eine Art Richtlinie zu verstehen [6, p. 13], an die man sich – sofern möglich – halten sollte. Es gibt jedoch auch Fälle, in denen die Anwendung des LoD nicht sinnvoll ist. Beispielsweise, wenn die Anforderungen an die Performance eine höhere Priorität haben als zum Beispiel die Wartbarkeit oder Anpassbarkeit.

Die konkrete Umsetzung des Gesetzes von Demeter variiert je nach verwendeter Programmiersprache. Lieberherr, Holland und Riel formulierten das LoD deshalb gesondert für verschiedene Programmiersprachen (Smalltalk-80, CLOS, Eiffel und C++) [6]. Bei Programmiersprachen, welche als Zugriffsoperator auf Komponenten, wie Methoden oder Attribute, einen Punkt oder auch einen Pfeil verwenden (z. B. Java, C++), wird das LoD oft auf die einfache Vorschrift, nur einen Punkt beziehungsweise Pfeil zu verwenden, beschränkt. Eine Übertragung des Law of Demeter auf die Programmiersprache Java fehlt in den ursprünglichen Formulie-

rungen. Aus diesem Grund wurde in dieser Ausarbeitung versucht, das LoD möglichst im Sinne der Erfinder auf Java zu übertragen.

# Literaturverzeichnis


[1] I. H. A. R. K. Lieberherr, „Object-Oriented Programming: An Objective Sense of Style," [Online]. Available: http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf. [Zugriff am 19 09 2014].

[2] B. Appleton, „Introducing Demeter and its Laws," [Online]. Available: http://www.bradapp.com/docs/demeter-intro.html. [Zugriff am 18 09 2014].

[3] K. Lieberherr. [Online]. Available: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html. [Zugriff am 18 09 2014].

[4] B. Appleton, „(OTUG) Law of Demeter," 24 10 1996. [Online]. Available: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/AppletonExplainsLoD.txt. [Zugriff am 17 09 2014].

[5] I. H. Karl J. Lieberherr, „Formulations and Benefits of the Law of Demeter," [Online]. Available: http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/law-formulations/revision1/ss.tex. [Zugriff am 17 09 2014].

[6] K. J. L. u. I. Holland, 1989. [Online]. Available: http://www-public.it-sudparis.eu/~gibson/Teaching/CSC5021/ReadingMaterial/LieberherrHolland89.pdf.

[7] K. J. Lieberherr, Adaptive Object-Oriented Software - The Demeter Method With Propagation Patterns, PWS Publishing company.

[8] D. Berens, 10 2011. [Online]. Available: http://www.fernuni-hagen.de/imperia/md/content/ps/masterarbeit-berens.pdf.

[9] D. Bock. [Online]. Available: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf.

[10] E. GmbH, „"Das Gesetz von Demeter"," [Online]. Available: http://www.empros.ch/vielfach/faustregeln/502149936a0f4bb0f/dasgesetzvondemeter.