



Implementierung von DbC mit Java 5 Annotations, AspectJ und Eclipse

Design by Contract

» HEIKO SEEBERGER UND ANDREAS WAGNER

Sie kennen Design by Contract (DbC)? Dann sind Sie bestimmt zumindest in der Theorie davon überzeugt. Aber setzen Sie DbC in der Praxis ein? Möglicherweise nicht oder nicht konsequent, da Sie die Umsetzung zu mühsam finden. Dieser Artikel beleuchtet einen einfachen Ansatz zur Implementierung von DbC. Beschrieben wird zudem eine Anwendung von Java 5 Annotations, AspectJ und Eclipse Plug-ins im gemeinsamen Konzert.

ContractJ ist eine Implementierung von Design by Contract auf Basis von Java 5 Annotationen, AspectJ und Eclipse Plug-ins. Design by Contract ist eine Methodik, in der öffentliche Schnittstellen, d.h. APIs, in Form von Contracts klare Vorgaben machen, was sie erwarten und was sie liefern. Preconditions definieren die Erwartungshaltung und Postconditions beschreiben die Liefergarantien. Dazu gibt es als unveränderlichen Zustand von Objekten noch Invariants. DbC wurde

von Bertrand Meyer im Rahmen des Designs der Programmiersprache Eiffel eingeführt [1], [2].

Auch Sun orientiert sich in den Hinweisen zu den im JDK 1.4 eingeführten Assertions für Java teilweise an DbC. So wird z.B. empfohlen, eine *IllegalArgumentException* zu werfen, wenn ein Argument eines Methodenaufrufes unerlaubterweise null ist [4].

Soweit die Theorie. In der Praxis stellt sich jedoch schnell heraus, dass dieser Ansatz mühsam ist und stark redundan-

dante Codefragmente erzeugt. Eine immer wieder auftauchende Precondition ist z.B. die oben schon erwähnte Not-Null-Precondition. Diese wird immer auf dieselbe Weise implementiert, wobei sich allenfalls die Message der *IllegalArgumentException* ändert:

```
public void findPerson(String name) {
    if (name == null) {
        throw new IllegalArgumentException(...);
    }
    ...
}
```

Neben dieser praktischen Betrachtung hat dieser Ansatz weitere Nachteile. Erstens geht es bei DbC darum, Contracts auf Basis von APIs zu definieren und nicht innerhalb der Implementierung. Im obigen Beispiel ist die Precondition jedoch in der Implementierung „vergraben“. Zweitens lebt DbC von der klaren Kommunikation der Contracts: Nur wenn die Nutzer von APIs die Contracts kennen und verstehen, können sie diese korrekt anwenden. Daher spielt die Do-

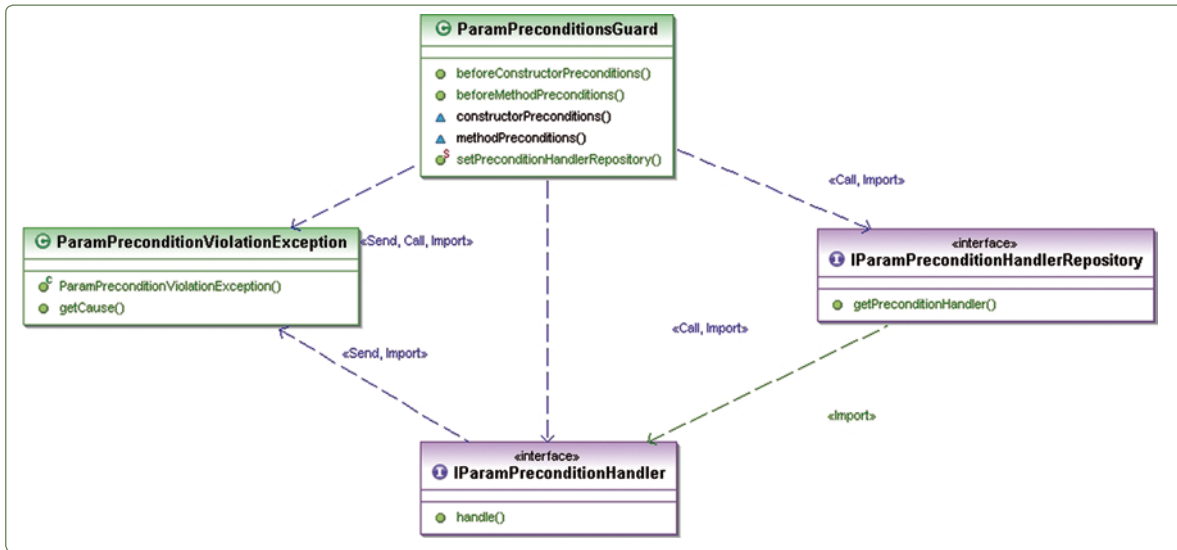


Abb. 1: Zusammenspiel der wesentlichen Klassen für die ContractJ Parameter Preconditions

kumentation der Contracts eine wichtige Rolle. Im obigen Beispiel könnte z.B. mittels JavaDoc der Parameter *name* entsprechend dokumentiert werden.

```

/**
 * @param name The person's name. Must not be null!
 */
public void findPerson(String name) {
    if (name == null) {
        throw new IllegalArgumentException(...);
    }
    ...
}

```

Allerdings birgt diese Aufspaltung des Contract in Dokumentation und Implementierung die Gefahr von Inkonsistenz: Die Dokumentation könnte gänzlich vergessen oder bei Refactorings nicht nachgezogen werden.

Im Folgenden wird mit ContractJ ein Ansatz vorgestellt, der all die aufgeführten Nachteile beseitigt, d.h. sowohl einfach zu implementieren ist, als auch auf die APIs ausgerichtet ist und Dokumentation und Implementierung vereinigt. Doch zunächst werden kurz die dazu eingesetzten Technologien beleuchtet.

Java 5 Annotations

Mit den in Version 5 eingeführten Annotations steht ein Sprachmerkmal zur Verfügung, um Metadaten zu definieren und auf Java-Sprachelemente wie z.B. Klassen, Methoden oder Parameter anzuwenden. Der Nutzen von Annotations gegenüber Kommentaren, XML-Dateien etc. ergibt sich daraus, dass sie selbst auch Java-Sprachelemente sind. Dadurch ist es u.a. möglich, Annotations mittels Reflection zur Laufzeit auszu-

werten, wozu die verwendeten Annotations jedoch selbst mit *@Retention(RetentionPolicy.RUNTIME)* annotiert sein müssen, damit sie es bis in die Java Virtual Machine „schaffen“. Ausführlichere Informationen können der JDK-Dokumentation entnommen werden.

AspectJ

Aspektorientierung ist eine Methodik zur Modularisierung so genannter Cross Cutting Concerns. Letztere sind Belange eines Systems, die sich nicht im Objektmodell modularisieren lassen, typische Beispiele sind Autorisierung, Transaktionen, Logging, Exception Handling etc. Die Objektorientierung kennt mit dem Objektmodell nur eine Ebene der Modularisierung, sodass zumindest die Aufrufe dieser Querschnittsfunktionen über die Klassen verteilt sind. Genau hier greift die Aspektorientierung, indem sie Objektorientierung ergänzt und über Aspekte eine Möglichkeit bietet, diese Cross Cutting Concerns zu modularisieren.

Aus der Helikopterperspektive besteht AOP aus folgenden Bestandteilen:

- Advices, die den modularisierten Cross Cutting Concern darstellen, also den Code, der an bestimmten Stellen mit den Klassen zusammengefügt werden soll,
- Join Points, welche die möglichen Stellen für das Weaving darstellen,
- Pointcuts, die Mengen von Join Points selektieren und
- Weaving, dem Prozess der Zusammenführung von Advices und „normalen“ Klassen.

AspectJ ist eine sehr mächtige AOP-Sprache für Java. Grundsätzlich erweitert AspectJ den Java-Sprachumfang um einige AOP-Konstrukte, z.B. aspect, pointcut etc. Seit Version 5 bietet AspectJ, die Verwendung von Java 5 vorausgesetzt, auch die Möglichkeit, ohne Spracherweiterungen und ausschließlich mit Annotations auszukommen. In diesem Kontext ist hervorzuheben, dass AspectJ seit Version 5 die Möglichkeit bietet, Pointcuts für Methodensignaturen unter Verwendung von Annotations zu definieren. Z.B. kann der Aufruf der Methode

```

@Test
public void test() { ... }

```

mit dem folgenden Pointcut adressiert werden:

```

pointcut test(): call(call(@Test * *(..)));

```

AspectJ ist seit geraumer Zeit ein Eclipse-Projekt [5] und erfreut sich daher mit AJDT [6] einer hervorragenden Integration in die Eclipse-Entwicklungsumgebung [7].

Eclipse Plug-ins

Anwendungen auf Basis der Eclipse-Architektur bestehen aus Modulen, so genannten Plug-ins. Plug-ins können als Enabler auftreten und Extension Points definieren, welche durch andere Plug-ins (Extender) erweitert werden. Neben deklarativen Informationen können Extension Points auch Interfaces oder Klassen enthalten, welche vom Extender implementiert oder erweitert werden müssen. Der Enabler

bietet die Erweiterung seines eigenen Verhaltens an, indem er alle Extender sucht, deren deklarative Informationen nutzt und, falls eine programmatische Erweiterung vorgesehen ist, deren Klassen an den entsprechenden Stellen ruft. Diese Architektur schafft die Basis für modulare und erweiterbare Anwendungen.

ContractJ Postconditions

Wie bringt ContractJ nun diese Technologien zu einer anwenderfreundlichen DbC-Implementierung zusammen? Rekapitulieren wir kurz die Anforderungen: einfach anzuwenden, an den APIs ausgerichtet und die Dokumentation gratis dazu.

Zunächst einmal kommt AspectJ als „Wunderwaffe“ für Contracts ins Spiel. Immer dann, wenn Contracts generisch sind, sodass sie an mehreren Stellen vorkommen, ist es nützlich und sinnvoll, den Contract in einen Aspekt auszulagern. So muss der Contract nur einmal geschrieben werden und kann mittels geeignetem Pointcut an allen nötigen Stellen eingewoben werden. Eine Not-Null-Postcondition könnte z.B. durch folgenden Advice implementiert werden:

```
after() returning(Object o): notNullPostcondition() {
    if(o == null) {
        throw new IllegalArgumentException(...);
    }
    ...
}
```

Der aufmerksame Leser ahnt schon, dass Annotations ein heißer Kandidat für die Definition der Pointcuts sind. Der folgende Pointcut adressiert alle Methoden mit Rückgabe, die mit `@NotNull` annotiert sind:

```
pointcut notNullPostcondition(): call(@NotNull !void
*(..));
```

Somit würde z.B. die folgende Methode mit der Not-Null-Postcondition advised werden:

```
@NotNull
public String info() { ... }
```

Dieses Design ist schon ein Paradebeispiel für AspectJ, wobei die Verwendung von Annotations natürlich erst seit Java 5 und AspectJ 5 möglich ist. ContractJ greift dieses Design auf und liefert in der Version 2.0 die folgenden „ready to use“-Postconditions mit:

- `@NotNull`
- `@NotNegative`
- `@Positive`

Wie sieht es hier mit unseren Anforderungen aus? Einfach ist das bloße Voranstellen von Annotations allemal, ausdrucksstark und selbsterklärend auch, und durch die Verwendung des Call-JoinPoints wirken die Advices auf Interfaces bzw. Typ, der den Contract mittels Annotations deklariert.

ContractJ Preconditions

Was oben für Postconditions geschildert wurde, gilt prinzipiell analog für Preconditions. Allerdings ist es aktuell mit AspectJ nicht möglich, Pointcuts auf Basis von annotierten Parametern zu definieren. Daher kann dieses Design nicht direkt auf Parameter Preconditions, d.h. Preconditions, die sich auf einzelne Methodenparameter beziehen, angewendet werden. Für den folgenden Code kann mit AspectJ kein Pointcut auf Basis der `@NotNull` Annotation definiert werden:

```
public void test(@NotNull Integer i) { ... }
```

Daher erweitert ContractJ für Parameter Preconditions den Ansatz: Die Annotation `@ParamPreconditions` identifiziert diejenigen Methoden, die Parameter Preconditions enthalten. Der Aspekt `ParamPreconditionsGuard` definiert Pointcuts für Methoden und Konstruktoren, die mit `@ParamPreconditions` annotiert sind, und wertet in den verknüpften Advices zur Laufzeit die Annotations der Parameter aus. Zu jeder Parameter Annotation wird im `IParamPreconditionHandlerRepository` ein `IParamPreconditionHandler` gesucht und dessen `handle()`-Methode aufgerufen, welche die zur Annotation gehörende Precondition implementiert. Im folgenden Codefragment wird somit die zur Annotation `@NotNull` gehörende Parameter Precondition geprüft.

```
@ParamPreconditions
public void test(@NotNull Integer i) { ... }
```

Gesucht und gefunden werden diese `IParamPreconditionHandler` prinzipiell in einem beliebigen `IParamPreconditionHandlerRepository`, wobei ContractJ mit dem `EclipseParamPre-`

Beste Bücher für besten Code!



Georg Pietrek,
Jens Trompeter (Hrsg.)

Modellgetriebene Softwareentwicklung

MDA und MDSD in der Praxis
256 Seiten, Hardcover, CD, 39,90 €
ISBN 978-3-939084-11-2

- Etablierung von MDSD in der Softwareentwicklung
- MDSD im objektorientierten Entwicklungsprozess
- Vorgehensmodelle
- Modellierung
- Abstraktion
- Agile Modellierung
- Modellierungssprachen
- Metamodellierung
- Domänenspezifische Modellierung

Weitere Informationen und Bestellmöglichkeiten finden Sie unter www.entwickler.press.de.
Unsere Bücher erhalten Sie auch in jeder gut sortierten Buchhandlung.



conditionHandlerRepository eine Implementierung auf Basis von Eclipse Plug-ins mitliefert. Über den Extension Point *paramPreconditions* können Annotations mit zugehörigen Handlern registriert werden. Natürlich liefert ContractJ einige „ready to use“-Parameter Preconditions, welche dieselben Annotations nutzen wie die Postconditions:

- @NotNull
- @NotNegative
- @Positive

Durch die Verwendung der Eclipse-Plug-in-Architektur ist ContractJ für Erweiterungen offen: Um eigene Parameter Preconditions zu implementieren, muss im eigenen Plug-in nur der genannte Extension Point erweitert werden. Die neuen Parameter Annotations müssen selbst mit *@ParamPrecondition* annotiert werden, damit die Pointcuts des *ParamPreconditionsGuard* „ziehen“:

```
@ParamPrecondition
@Retention(RUNTIME)
@Target({ METHOD, PARAMETER })
public @interface MyParameterAnnotation { ... }
```

Die eigenen *IParamPreconditionHandlers* signalisieren den Verstoß gegen den Contract der Precondition, indem sie eine *ParamPreconditionViolationException* werfen, die als Wrapper für die eigentliche Exception, meist wohl eine *IllegalArgumentException*, dient:

```
public void handle(final IParamPreconditionContext paramPreconditionContext)
    throws ParamPreconditionViolationException {

    if (...) {
        throw new ParamPreconditionViolationException(
            new IllegalArgumentException(...));
        ...
    }
}
```

Wie steht es nun um unsere Anforderungen? Was oben für reine AspectJ Contracts gesagt wurde, gilt weiterhin, obwohl als kleiner Wermutstropfen hinzu kommt, dass die Methode, welche Parameter Preconditions enthält, selbst mit *@ParamPreconditions* annotiert werden muss. Aber das ist nur ein verhältnismäßig kleiner Aufwand im Vergleich zum erzielten Nutzen.

Ein möglicher Nachteil dieses Ansatzes sei nicht verschwiegen: Durch die

Notwendigkeit zur Verwendung von Reflection zur Ermittlung der annotierten Parameter könnte ein negativer Effekt auf die Laufzeitperformance entstehen. Während statische AOP im Wesentlichen nicht mehr kostet, als die entsprechende Funktionalität direkt im Code einzubauen, muss hier das Laufzeitverhalten untersucht werden, am besten mittels Profiling.

ContractJ Architektur

ContractJ kann prinzipiell in jeder beliebigen Java Anwendung verwendet werden, es ist jedoch stark auf die Eclipse-Plug-in-Architektur ausgerichtet.

ContractJ erleichtert den Einsatz von DbC

Das zentrale Plug-in *org.contractj.core* enthält obige Pre- und Postconditions. Da Plug-ins zunächst einmal normale JAR-Archive darstellen, kann ContractJ auch ohne Eclipse verwendet werden, wobei das Parameter-Preconditions-Framework um eine andere konkrete Implementierung des *IPreconditionHandlerRepository* erweitert werden muss.

Für die Integration von ContractJ in die Eclipse Entwicklungsumgebung stehen weitere Plug-ins zur Verfügung, z.B. um im JDt eine *ClasspathVariable* mit dem Pfad zur ContractJ-Installation zu initialisieren und JavaDoc und Source Locations zu spezifizieren.

Installation und Verwendung von ContractJ

ContractJ ist ein Open-Source-Projekt, das bei SourceForge.net [8] gehostet wird. Das aktuelle Release kann als ZIP-Archiv heruntergeladen und manuell installiert werden, indem der Inhalt in das Installationsverzeichnis von Eclipse entpackt wird. Zusätzlich steht als komfortablere Lösung eine Update Site [9] zur Verfügung, sodass die Installation direkt aus Eclipse heraus über den Update Manager erfolgen kann.

Um ContractJ einzusetzen, benötigt man zunächst ein Plug-in-Projekt mit AspectJ Nature. Dann muss das Plug-in *org.contractj.core* in die Liste der abhängigen Plug-ins aufgenommen werden.

Danach gilt es noch, den Aspect Path in den Project Properties um die Classpath-Variable *CONTRACTJ_LIB* zu erweitern und schon kann es losgehen.

Fazit und Ausblick

ContractJ kann den Einsatz von DbC soweit erleichtern, dass diese Methodik Akzeptanz findet und so konsequent eingesetzt werden kann. ContractJ erweist sich als echter Enabler für DbC. Es besteht Grund zur Hoffnung, dass AspectJ zukünftig auch Pointcut Matching auf Basis von Parameter Annotations bietet. Dann könnte man auch die Parameter Preconditions direkt mit AspectJ Pointcuts „fassen“. Damit wäre leider auch ein bedeutender Teil von ContractJ obsolet, aber solch weitreichenden Vereinfachungen soll natürlich nichts im Wege stehen. Eine andere Idee, um den oben beschriebenen Nachteil möglicher Performanceverschlechterung zu umgehen, ist die Verwendung von Annotation Processing, um spezifische Aspekte zu generieren. Unterstützung aus der Community ist hierbei natürlich höchst willkommen.



Heiko Seeberger leitet die Market Unit Enterprise Architecture der metafinanz GmbH (www.metafinanz.de). Er erstellt seit etwa zehn Jahren Enterprise Applications mit Java, wobei sein aktueller Fokus auf Eclipse und AspectJ liegt. Kontakt: heiko.seeberger@metafinanz.de



Andreas Wagner arbeitet als Advanced Consultant für die metafinanz GmbH. Seine Interessen sind unter anderem Eclipse-Plug-in-Entwicklung und hier im Speziellen die Rich Client Plattform. Kontakt: andreas.wagner@metafinanz.de

>>Links & Literatur

- [1] Design by Contract: Eiffel Software: www.eiffel.com
- [2] se.ethz.ch/~meyer/publications/computer/contract.pdf
- [3] archive.eiffel.com/doc/manuals/technology/contract
- [4] java.sun.com/javase/6/docs/technotes/guides/language/assert.html
- [5] www.eclipse.org/aspectj
- [6] www.Eclipse.org/ajdt
- [7] Oliver Böhm: Aspektorientierte Programmierung mit AspectJ 5, dpunkt.verlag
- [8] sourceforge.net/projects/contractj
- [9] contractj.sourceforge.net/updatesite