DAS GROSSE ALM-SPEZIAL FÜR DIE ECLIPSE-WELT AUF ÜBER 50 HEFTSEITEN

eclipse MAGAZIN

3.13 S&S

eclipse MAGAZIN

www.eclipse-magazin.de

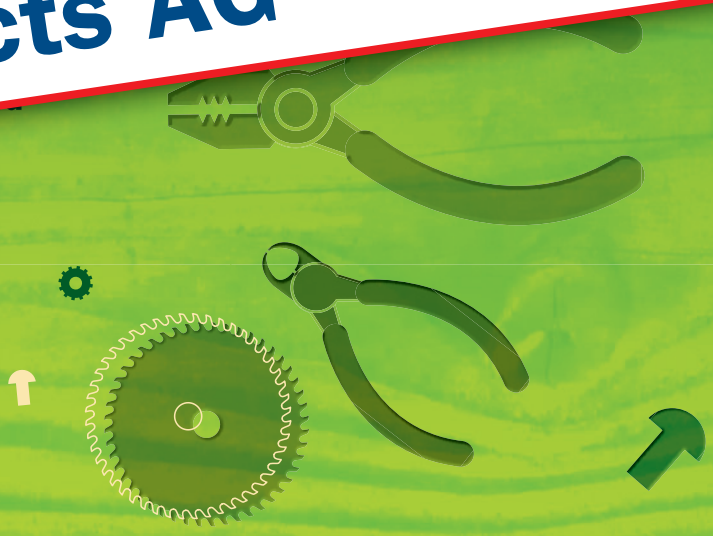
jax 2013 hier im Heft! › 51

ECLIPSE & ALM

ALM TOOLCHAIN | EGIT/JGIT | COLLABORATION | TESTING |
ECLIPSE LYO | SMART ECOSYSTEMS

Sonderdruck für andrena objects AG

andrena OBJECTS
Experts in agile software engineering



Contracts, Java und Eclipse

Mehr als Assert Statements

Agiles Programmieren und formale Verträge, die mit Schnittstellen oder Klassen verknüpft werden – das scheinen Gegensätze zu sein, die sich nicht vereinen lassen. Dabei haben Verträge den Charme, die Softwareentwicklung effizienter zu machen. Mehr noch: Mit ihnen lassen sich auch abstrakte Datentypen (ADT) vollständig mit den Mitteln von Java beschreiben. Wie das? Mit dem neuen Release 6.0 von C4J, Contracts for Java [1], das nun auch als Eclipse-Plugin [2] zur Verfügung steht, wurde die gedankliche Verbindung „Verträge – Forward Engineering“ endgültig gekappt. C4J ist ein unter der EPL stehendes freies Framework und erweitert die Möglichkeiten des Assertion-Konzepts in Java entscheidend. Die definierten Vertragsbeziehungen bleiben auch bei intensivem Refactoring erhalten. Und auch nachträglich lassen sich z. B. für Legacy-Systeme Softwareverträge formulieren. Dieser Artikel demonstriert die Anwendung des neuen C4J-Plugins anhand eines Bankkontos, das über seine Schnittstelle und den zugehörigen Vertrag in seiner Syntax und Semantik beschrieben wird.

von Hagen Buchwald und Florian Meyerer

„Softwareverträge“ ist eine andere Bezeichnung dafür, Vor- und Nachbedingungen sowie Klasseninvarianten systematisch zu definieren. Der große Vorteil dieser Vorgehensweise: Wer Schnittstellen mit in Java formulierten „Vertragsklauseln“ verknüpft, ermöglicht es, die Einhaltung dieser Zusicherungen zur Laufzeit zu überprüfen. Nebenbei sind präzise definierte Schnittstellen eine wichtige Basis für objektorientiertes Programmieren. Warum also schütteln viele Entwickler erst mal den Kopf, sobald das Stichwort „Softwarevertrag“ fällt?

Der Grund liegt bei einem anderen Stichwort: „Refactoring“. Agiles Programmieren und Refactoring gehören zusammen wie Pferd und Kutsche, das Vertragskonzept jedoch, so die Annahme, schließe Refactoring praktisch aus. Außerdem schien es gar nicht nötig, sich mit Softwareverträgen zu befassen, seit Java 1.4 erlaubte, Zusicherungen im Quellcode zu formulieren. „Assert Statements“ heißt diese Lösung.

Nur löst diese Lösung nicht alle Probleme. Man mag es als Ironie betrachten, dass es gerade das – vermeintlich dem Forward Engineering – zuzuordnende Konzept der Softwareverträge ist, das dazu beiträgt, in Java eine der wichtigsten Bedingungen des objektorientierten Programmierens zu erfüllen. Denn als deren Definition [3] gilt die Erstellung von Softwaresystemen als strukturierte Sammlung von Implementierungen abstrakter Datentypen (ADTs). Genau das gelingt allein mit Assert Statements nicht immer. Denn Asserts können lediglich im Methodenrumpf verwendet werden, sprich, im Quellcode. Sobald eine ererbte Methode überschrieben wird, sind die Assert Statements im Zweifelsfall verschwunden. Einer Definition als abstrakte Datentypen verschließen sie sich völlig, weil Interfaces eben gar keinen Quellcode haben, in dem der Entwickler Vor- und Nachbedingungen festschreiben könnte.

Diese „objektorientierten Mängel“ von Java gleicht die Version 6.0 von C4J, „Contracts for Java“, aus. Sie setzt auf der von Jonas Bergström [4] entwickelten ersten Generation von C4J auf und ist das Ergebnis einer Initiative der C4J Special Interest Group des Vereins Karlsruher Software Ingenieure [5] (VKSI), die von Ben Romberg, einem agilem Softwareingenieur von andrena objects, und den Autoren dieses Artikels vorangetrieben wird.

Um C4J wirklich agil werden zu lassen, trennt die Version 6.0 die Vertragsklauseln vom Quellcode, indem sie Vertragsklassen einführt. Sobald die Klasse geladen wird, fügt C4J per Bytecode-Instrumentation die Vorbedingungen direkt zu Beginn der zu schützenden Methoden ein, die Nachbedingungen und Klasseninvariante folgen am Ende. Damit werden die Bedingungen Bestandteil der Methoden – innerhalb der Bytecode-Ebene – und daher während der Laufzeit überprüfbar.

Für nicht finale Klassen gilt der Grundsatz, dass die Vertragsklassen in einer direkten Vererbungsbeziehung zu denjenigen Klassen stehen, die sie schützen, der so genannten Target-Klasse. Dank dieser Vererbungsbeziehung wendet Eclipse alle Refactoring-Mechanismen automatisch auf die zugehörige Vertragsklasse an. Die Folge: Selbst bei intensivem Refactoring entstehen keine Inkonsistenzen zwischen Vertrags- und Target-Klasse und damit keine aufwändigen Nacharbeiten für die Programmierer.

Die Vererbungsbeziehung hat noch einen weiteren, sehr wichtigen Vorteil: Sie macht Anpassungen im Quellcode der zu schützenden Klasse unnötig, wenn ihr nachträglich eine Vertragsklasse zugeordnet werden soll. Stattdessen erlaubt C4J 6.0, Verträge rein von außen anzuhängen. Was so schlicht klingt, öffnet die Tür zu einem sehr großen Anwendungsbereich: Der Sanierung von Altsystemen (Legacy-Code).

Das Erstellen von Softwareverträgen mit dem neuen C4J-Eclipse-Plugin sei hier am Beispiel eines Bankkontos illustriert, das Teil eines Pakets eines für missionskritische Einsätze entwickelten Java-Banking-Frameworks (JBF) sein soll. Bei Bankkonten und -automaten ist es häufig so, dass diejenigen, die den Bankomat programmieren, nicht diejenigen sind, die zentral die Programme in der Bank betreuen. Wir gehen nun von zwei Annahmen aus: Erstens wird das fragliche Konto bereits durch ein Interface beschrieben. Zweitens erhält ein Programmierer jetzt den Auftrag, das dynamische Verhalten des Bankkontos vollständig und präzise zu beschreiben, und zwar als abstrakten Datentyp mit den Mitteln, die Java bietet. Ausgangspunkt ist das wie folgt beschriebene Interface:

```
package jbf;

public interface AccountSpec {

  int MAX_AMOUNT = 100000;

  void deposit(int amount);

  void withdraw(int amount);

  int getBalance();

  String getOwner();

  int availableAmount();

}
```

Um nun den Vertrag zu formulieren, hält sich unser Beispielprogrammierer an die sechs Prinzipien, die Richard Mitchell und Jim McKim in ihrem Buch „Design by Contract, by Example“ [6] vorstellen.

Das erste dieser Prinzipien enthält die Forderung, Abfragen durch *@Pure* von Kommandos (Command-Query-Separation) zu trennen. Abfragen erheben einfach einen Wert, ohne ihn zu ändern. Kommandos hingegen können das Objekt verändern, liefern aber kein Ergebnis zurück.

Das Beispiel des Kontos illustriert nun sehr schön, warum diese Unterscheidung weitaus mehr ist als von akademischem Interesse. Abfragen, also Methoden, die das Objekt absolut nicht verändern sollen, sind die Frage nach dem Kontostand (*getBalance*), dem Kontoinhaber (*getOwner*) und nach dem im Moment verfügbaren Betrag (*availableAmount*). Wir gehen hier davon aus, dass das Konto nicht unbegrenzt überzogen werden darf, sondern maximal ein Betrag in der Höhe des aktuellen Guthabens abgehoben werden kann (*MAX_AMOUNT*). Es ist unmittelbar einsichtig, warum diese Werte gleich bleiben müssen, unabhängig davon, wie oft nachgefragt wird. Eine bloße Kontostandsabfrage darf schließlich keinesfalls dessen Höhe verändern. Genau das Gegenteil ist bei den Methoden *Einzahlung* (engl.: *deposit*) oder *Abhebung* (engl.: *withdraw*) der Fall, denn hier ist unbedingt erforderlich, dass sich der sichtbare Zustand des Objekts, also der Kontostand, ändert. Alle Beträge werden in Cent ausgedrückt, 1 Euro entsprechen also 100 Cent.

Um nun dem ersten Prinzip gerecht zu werden, importiert unser Entwickler die Anweisung

```
import de.vksi.c4j.Pure;
```

aus dem C4J Package. Methoden, die mit *@Pure* annotiert wurden, dürfen keinen Seiteneffekt bewirken, sie sind also Abfragen. Die strikte Einhaltung dieser *@Pure*-Eigenschaft wird von C4J zur Laufzeit automatisch überprüft. Darüber hinaus kann unser Entwickler Kommentare einfügen, um Kommandos ganz klar von Abfragen zu trennen. Übersetzt in Code sieht das dann aus wie in Listing 1. Damit ist unser Entwickler beim zweiten Prinzip angekommen, das besagt:

Trenne elementare Abfragen von abgeleiteten Abfragen.

Der Unterschied zwischen beiden besteht darin, dass sich abgeleitete Abfragen (engl.: derived queries) aus elementaren Abfragen (engl.: basic queries) zusammensetzen. In unserem Beispiel trifft das für die Ab-

Listing 1

```
package jbf;

import de.vksi.c4j.Pure;

public interface AccountSpec {

  int MAX_AMOUNT = 100000;

  // commands

  void deposit(int amount);

  void withdraw(int amount);

  // queries

  @Pure
  int getBalance();

  @Pure
  String getOwner();

  @Pure
  int availableAmount();

}
```

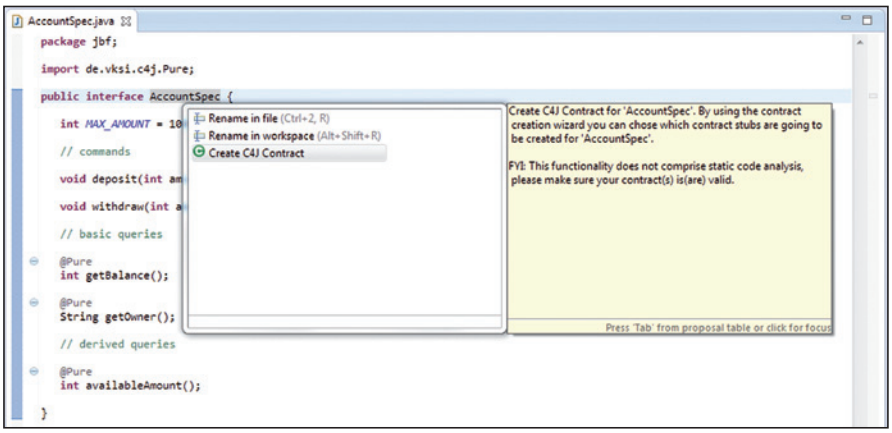


Abb. 1: Erstellen eines Interface-Contracts mithilfe des C4J-Eclipse-Plugins (Aufruf des Wizards)

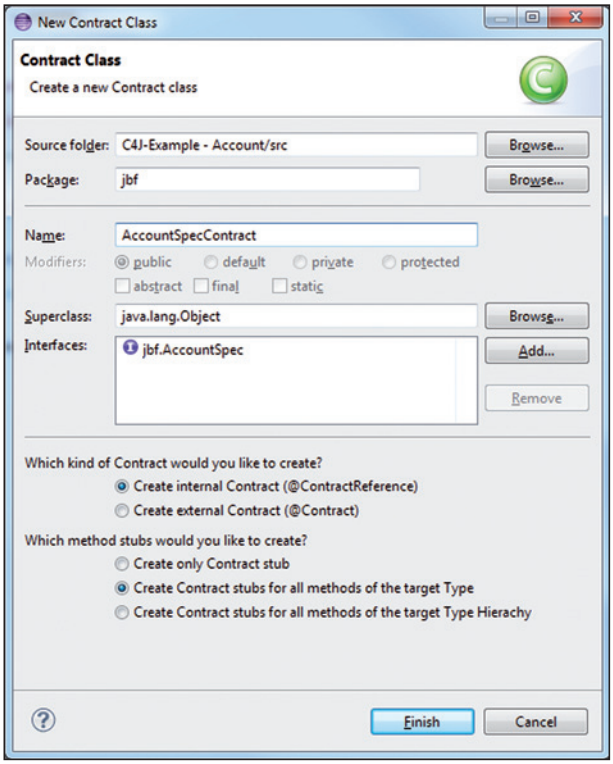


Abb. 2: Erstellen eines Interface-Contracts mithilfe des C4J-Eclipse-Plugins (Wizard)

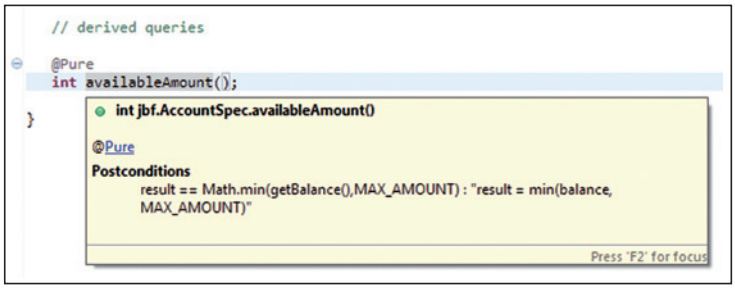
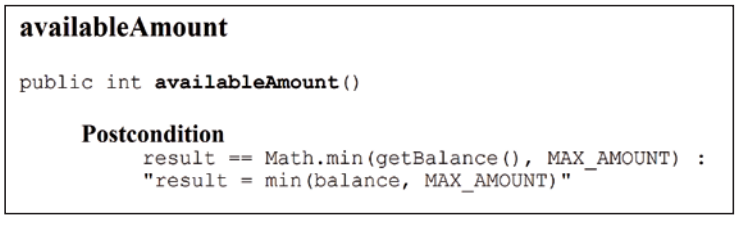


Abb. 3: Text-Hover-Information für geschützte Methoden



availableAmount

```
public int availableAmount()
```

Postcondition

```
result == Math.min(getBalance(), MAX_AMOUNT) :
"result = min(balance, MAX_AMOUNT)"
```

Abb. 4: JavaDoc-Dokumentation der Methode „availableAmount“ des Interfaces „AccountSpec“

frage des verfügbaren Betrags zu. Um zu ermitteln, wie viel Geld maximal abgehoben werden kann, ehe das Konto überzogen ist, muss der aktuelle Kontostand bekannt sein. Die abgeleitete Abfrage *availableAmount* errechnet sich also aus der elementaren Abfrage *getBalance*. Elementare Abfragen sind typischerweise *get*-Methoden, die direkt den Wert eines Attributs einer Klasse abfragen.

In Anwendung des zweiten Prinzips teilen sich die Anfragen in zwei Gruppen:

```
// basic queries

@Pure
int getBalance();

@Pure
String getOwner();

// derived queries

@Pure
int availableAmount();
```

Damit sieht das Interface nunmehr aus wie in Listing 2 dargestellt. So weit, so gut. Das dritte Prinzip fordert nun:

Erstelle für jede abgeleitete Abfrage eine Nachbedingung, die beschreibt, welches Ergebnis zurückgegeben wird, ausgedrückt durch eine oder mehrere elementare Abfragen.

Um eine Nachbedingung in Java formulieren zu können, benötigt unser Entwickler nun eine Vertragsklasse für die Schnittstelle *AccountSpec*. Das „Skelett“ der Vertragsklasse *AccountSpecContract* lässt sich unser Entwickler direkt durch das C4J-Eclipse-Plugin erzeugen (**Abb. 1, 2**).

Da unser Softwareentwickler die Option „Create internal Contract“ gewählt hat, wird dem Interface die C4J-Annotation *@ContractReference* hinzugefügt:

```
package jbf;

import de.vksi.c4j.ContractReference;
import de.vksi.c4j.Pure;

@ContractReference(AccountSpecContract.class)
public interface AccountSpec {
```

Darüber hinaus erzeugt der Wizard die in der *@ContractReference* referenzierte Interfacevertragsklasse *AccountSpecContract* (Listing 3).

Als Nachbedingung für die abgeleitete Abfrage *availableAmount* formuliert unser Entwickler Listing 4.

Die statische C4J-Methode *result()*, die den Rückgabewert der Methode liefert, importiert er über die *import*-Anweisung

```
import static de.vksi.c4j.Condition.result;
```

Jetzt verfügt die abgeleitete Abfrage *availableAmount* über eine Nachbedingung, die auf Basis der elementaren Abfrage *getBalance* formuliert ist. Diese Nachbedingung wird für unseren Entwickler auch direkt für die nunmehr durch einen Vertrag geschützte Methode im Interface sichtbar, wie **Abbildung 3** zeigt. Zudem wird diese Bedingung nun auch beim Generieren der JavaDoc-Dokumentation [7] des Interfaces ebenfalls aufgelistet (**Abb. 4**). Das vierte Prinzip lautet:

Erstelle für jedes Kommando eine Nachbedingung, die den Wert jeder elementaren Abfrage auflistet. Der vollständige sichtbare Effekt eines jeden Befehls ist nun bekannt.

Der Entwickler steht nun vor der Aufgabe, die Wirkung der beiden Kommandos *deposit* und *withdraw* vollständig zu beschreiben, und das mithilfe der elementaren Abfragen *getBalance* und *getOwner*. Hier wird wieder der inhaltliche Unterschied zwischen Kommandos und Abfragen besonders deutlich – wer Geld auf sein Konto einbezahlt, formal also das Kommando *deposit* gibt, möchte definitiv, dass sich der Kontostand erhöht, und zwar exakt um den einbezahlten Betrag. Was sich allerdings keinesfalls verändern sollte, ist der Kontoinhaber (Listing 5).

Umgekehrt sollte beim Abheben eines Geldbetrags der Kontostand um den abgehobenen Betrag sinken. Dessen ungeachtet darf sich auch jetzt der Kontoinhaber nicht verändern (Listing 6).

Die Variable *target* zeigt hier auf das zu schützende Objekt, in unserem Fall auf das Bankkonto. Diese Variable muss unser Entwickler mithilfe der C4J-Annotation *@Target* als privates Attribut der Vertragsklasse deklarieren:

Listing 2

```
package jbf;

import de.vksi.c4j.Pure;

public interface AccountSpec {

  int MAX_AMOUNT = 100000;

  // commands

  void deposit(int amount);

  void withdraw(int amount);

  // basic queries

  @Pure
  int getBalance();

  @Pure
  String getOwner();

  // derived queries

  @Pure
  int availableAmount();

}
```

Listing 3

```
package jbf;

import static de.vksi.c4j.Condition.ignored;
import static de.vksi.c4j.Condition.postCondition;
import static de.vksi.c4j.Condition.preCondition;
import de.vksi.c4j.ClassInvariant;
import de.vksi.c4j.Target;

public class AccountSpecContract implements
                                    AccountSpec {

  @Target
  private AccountSpec target;

  @ClassInvariant
  public void classInvariant() {
    // TODO: write invariants if required
  }

  @Override
  public void deposit(int amount) {
    if (preCondition()) {
      // TODO: write preconditions if required
    }
    if (postCondition()) {
      // TODO: write postconditions if required
    }
  }

  @Override
  public void withdraw(int amount) {
    if (preCondition()) {
      // TODO: write preconditions if required
    }
    if (postCondition()) {
      // TODO: write postconditions if required
    }
  }

  @Override
  public int getBalance() {
    if (preCondition()) {
      // TODO: write preconditions if required
    }
    if (postCondition()) {
      // TODO: write postconditions if required
    }
    return ignored();
  }

  @Override
  public String getOwner() {
    if (preCondition()) {
      // TODO: write preconditions if required
    }
    if (postCondition()) {
      // TODO: write postconditions if required
    }
    return ignored();
  }

  @Override
  public int availableAmount() {
    if (preCondition()) {
      // TODO: write preconditions if required
    }
    if (postCondition()) {
      // TODO: write postconditions if required
    }
    return ignored();
  }

}
```

Listing 4

```
@Override
 public int availableAmount() {
  if (postCondition()) {
   int result = result();
assert result == Math.min(target.
getBalance(),MAX_AMOUNT) : "result =
          min(balance, MAX_AMOUNT)";
  }
  return ignored();
 }
```

Listing 5

```
@Override
  public void deposit(int amount) {
    if (postCondition()) {
      assert target.getBalance() == old(target.getBalance()) + amount :
        "balance = old balance + amount";
      assert unchanged(target.getOwner()) : "owner unchanged";
    }
  }
```

Listing 6

```
  @Override
  public void withdraw(int amount) {
    if (postCondition()) {
      assert target.getBalance() == old(target.getBalance()) - amount :
        "balance = old balance - amount";
      assert unchanged(target.getOwner()) : "owner unchanged";
    }
  }
```

Listing 7

```
@Override
  public void deposit(int amount) {
    if (preCondition()) {
      assert amount > 0 : "amount > 0";
    }
    if (postCondition()) {
      assert target.getBalance() == old(target.getBalance()) + amount :
        "balance = old balance + amount";
      assert unchanged(target.getOwner()) : "owner unchanged";
    }
  }

  @Override
  public void withdraw(int amount) {
    if (preCondition()) {
      assert amount > 0 : "amount > 0 ";
      assert target.getBalance() >= amount : "balance >= amount";
    }
    if (postCondition()) {
      assert target.getBalance() == old(target.getBalance()) - amount :
        "balance = old balance - amount";
      assert unchanged(target.getOwner()) : "owner unchanged";
    }
  }
```

```
public class AccountSpecContract implements AccountSpec {

  @Target
  private AccountSpec target;
```

Diese Arbeit hat das C4J-Plugin unserem Entwickler bereits abgenommen, sodass die Vertragsklasse vom Zeitpunkt ihrer Erzeugung an kompilierbar ist.

Gemäß des fünften Prinzips:

Prüfe für jede Abfrage und jedes Kommando, ob eine Vorbedingung erforderlich ist,

kontrolliert unser Entwickler nun die Abfragen und Kommandos, wobei er mit Letzteren beginnt. Rein nach logischen Kriterien sollte der Betrag, der einbezahlt wird, größer als Null sein. Denn eine Einzahlung von null Cent ist de facto keine Einzahlung, also nutzlos, und ein Betrag kleiner Null wäre keine Einzahlung mehr, sondern im Gegenteil eine Abhebung. Folglich ist für das Kommando *deposit* eine Vorbedingung in der Tat erforderlich. Sie lautet:

```
assert amount > 0 : "amount > 0";
```

Analog gilt beim Abheben, dass der fragliche Betrag größer Null sein soll, hier kommt noch hinzu, dass das Konto trotz der Abbuchung nicht überzogen wird. Demnach kann der abhebbare Betrag maximal dem aktuellen Kontostand entsprechen, niedrigere Werte sind erlaubt, höhere nicht. Daraus ergibt sich folgende Vorbedingung für das Kommando *withdraw*:

```
assert amount > 0 : "amount > 0 ";
assert target.getBalance() >= amount : "balance >= amount";
```

Unnötig sind dagegen Vorbedingungen für die drei Abfragen *getBalance*, *getOwner* und *availableAmount*, da sie jederzeit und ohne Einschränkungen ausgeführt werden können. Anders gesagt ist es für die reine Ermittlung des Kontostands unerheblich, ob das Konto ein Guthaben aufweist oder im Gegenteil ein Defizit. Diese Angabe entscheidet zwar, ob spätere Kommandos möglich sind (z. B. *withdraw*), die Abfrage selbst bleibt davon jedoch unberührt.

Die erkannten Vorbedingungen werden als *if*-Block vor die Nachbedingung geschrieben. Die entsprechenden Methoden der Vertragsklasse sehen – nach Anwendung des fünften Prinzips - nun aus wie in Listing 7 dargestellt.

Das sechste und letzte Prinzip der Vertragserstellung lautet:

Formuliere Invarianten, um die sichtbaren Eigenschaften des Objekts zu beschreiben, die nach jeder Methodenausführung erfüllt sein müssen.

Die objektorientierte Programmierung kennt das so genannte DRY-Prinzip: Don't Repeat Yourself! Wiederhole dich nicht, schaffe keine Redundanzen! Klasseninvarianten sind eine Möglichkeit, das DRY-Prinzip auch für Vertragsklassen einzuhalten. Invarianten aus der Sicht von C4J sind nichts anderes als Zusicherungen, die allen

Nachbedingungen der Kommandos einer Klasse gemeinsam sind. In unserem Beispiel ist dies die Zusicherung, dass sich der Name des Kontoinhabers nicht verändert hat:

```
assert unchanged(target.getOwner()) : "owner unchanged";
```

Diese Zusicherung lagern wir nun in eine Klasseninvariante aus. Sie ist eine Methode, die durch die C4J-Annotation *@ClassInvariant* markiert ist:

```
@ClassInvariant
public void invariant() {
  assert unchanged(target.getOwner()) : "owner unchanged";
}
```

Damit sind alle sechs Prinzipien der Vertragserstellung auf die Vertragsklasse angewendet worden. Den vollständigen Vertrag zeigt Listing 8.

Durch die systematische Anwendung der sechs Prinzipien guter Vertragserstellung kann unser Beispielentwickler sicher sein, einen guten Vertrag formuliert zu haben, der das dynamische Verhalten eines Bankkontos vollständig und präzise beschreibt.

Und mit diesem Vertrag hat er nun mit den sprachlichen Mitteln von Java den Datentyp Account vollständig durch sein Interface *AccountSpec* (Typ und Methoden) und die zugehörige Vertragsklasse *AccountSpecContract* (Vor- und Nachbedingungen und Klasseninvariante) beschrieben. In anderen Worten: Er hat den abstrakten Datentyp Account vollständig mit den Mitteln von Java beschrieben. Und damit erfüllt Java die Definition der objektorientierten Programmierung [3]: die Erstellung von Softwaresystemen als strukturierte Sammlung von Implementierungen abstrakter Datentypen (ADTs).

Der Diplom-Wirtschaftsingenieur **Hagen Buchwald** ist seit 1994 in der IT-Industrie tätig. Seit Oktober 2011 verstärkt er als Vorstand das Expertenteam für agiles Software Engineering der andrena objects ag, Karlsruhe.

Florian Meyerer erwarb seinen B. Sc. in Med. Informatik und wird mit seiner Arbeit zu „Contracts for Java“ den M. Sc. in Software Engineering abschließen. Ab Mai 2013 promoviert er im Bereich selbsttestender und -adaptierender Komponenten. Interessensschwerpunkte sind Software Quality und Requirements Engineering.

Links & Literatur

[1] http://c4j-team.github.com/C4J/

[2] Download des C4J-Eclipse-Plugins unter: https://github.com/C4J-Team/C4J-Eclipse-Plugin/tree/master/update-site

[3] Meyer, Bertrand: „Object-oriented Software Construction“, Prentice Hall International, 1988

[4] http://c4j.sourceforge.net/

[5] http://www.vksi.de/c4j.html

[6] Mitchell, Richard; McKim, Jim: „Design by Contract, by Example“, Addison-Wesley Longman, Amsterdam, 2001

[7] Download des C4J Doclets unter: https://github.com/C4J-Team/C4J-Doclet

Listing 8

```
package jbf;

import static de.vksi.c4j.Condition.ignored;
import static de.vksi.c4j.Condition.old;
import static de.vksi.c4j.Condition.postCondition;
import static de.vksi.c4j.Condition.preCondition;
import static de.vksi.c4j.Condition.result;
import static de.vksi.c4j.Condition.unchanged;
import de.vksi.c4j.ClassInvariant;
import de.vksi.c4j.Target;

public class AccountSpecContract implements
AccountSpec {

  @Target
  private AccountSpec target;

  @ClassInvariant
  public void invariant() {
    assert unchanged(target.getOwner()) :
                                "owner unchanged";
  }

  @Override
  public void deposit(int amount) {
    if (preCondition()) {
      assert amount > 0 : "amount > 0";
    }
    if (postCondition()) {
assert target.getBalance() == old(target.
getBalance()) + amount : "balance = ld balance +
                                          amount";
    }
  }

  @Override
  public void withdraw(int amount) {
    if (preCondition()) {
      assert amount > 0 : "amount > 0 ";
      assert target.getBalance() >= amount :
                                "balance >= amount";
    }
    if (postCondition()) {
assert target.getBalance() == old(target.
getBalance()) - amount : "balance =
                                old balance - amount";
    }
  }

  @Override
  public int getBalance() {
    if (postCondition()) {
      int result = result();
      assert result >= 0 : "result >= 0";
    }
    return ignored();
  }

  @Override
  public String getOwner() {
    if (postCondition()) {
      String result = result();
      assert result != null : "result != null";
    }
    return ignored();
  }

  @Override
  public int availableAmount() {
    if (postCondition()) {
      int result = result();
assert result == Math.min(target.getBalance(), MAX_
    AMOUNT) : "result = min(balance, MAX_AMOUNT)";
    }
    return ignored();
  }
}
```