DbC UND TDD: PARTNER ODER KONTRAHENTEN?

Testgetriebene Entwicklung (TDD) ist Stand der Technik im modernen Software-Engineering und für agile Entwickler unverzichtbar. Design by Contract (DbC), bekannt geworden mit der Programmiersprache Eiffel, verfolgt eine ähnliche Philosophie: die Trennung von Implementierung und Interface. Bis heute fristet DbC ein Schattendasein, doch das könnte sich ändern: Microsoft hat DbC-Unterstützung in Visual Studio integriert und auch für Java gibt es eine DbC-Implementierung. Somit stellt sich die Frage, ob DbC und TDD im Widerspruch zueinander stehen oder ob sich beide Ansätze sogar ergänzen? In diesem Artikel schildern wir zunächst die beiden Ansätze, diskutieren ihre Gemeinsamkeiten und zeigen dann auf, wie sie sich miteinander kombinieren lassen.

Während *Test Driven Design (TDD)* zum Stand der Technik gehört, findet *Design by Contract (DbC)* bislang recht wenig Berücksichtigung. Das könnte sich jedoch in naher Zukunft ändern: Microsoft hat das Framework *Code Contracts* (vgl. [PDC08]) entwickelt und dieses in „Visual Studio" integriert. Und mit „C4J" (vgl. [C4J]) ist ein leistungsfähiges, frei verfügbares Tool für Java entstanden. Da TDD und DbC beide das gleiche Ziel verfolgen, nämlich eine höhere Softwarequalität, stellt sich die Frage, ob die beiden Ansätze Kontrahenten oder Partner sind. Könnte die Synthese beider Ansätze vielleicht sogar eine neue Stufe im Software-Engineering-Prozess bringen?

In diesem Artikel berichten wir zunächst von Erfahrungen mit DbC für Java in der Hochschul-Lehre. Anschließend diskutieren wir kurz den TDD-Ansatz, um darauf basierend die Gemeinsamkeiten und Unterschiede dieser scheinbar so gegensätzlichen Ansätze herauszuarbeiten. Diese Analyse hilft dabei, die im Vorgängerartikel in OBJEKTspektrum (vgl. [Buc11]) aufgeworfenen Fragen zu beantworten und führt abschließend zu einem Fazit und Ausblick.

DbC – Überblick und Einsatz in der Forschung

Am Karlsruher Institut für Technologie (KIT), dem Zusammenschluss der Universität Karlsruhe (TH) mit dem Forschungszentrum, wird seit dem Wintersemester 2008 das objektorientierte Paradigma anhand von DbC gelehrt. Der Entwurf von Klassen auf Basis von Verträgen (also DbC) stellt den Kern der objektorientierten Programmierung dar, die Bertrand Meyer 1988 als „die Entwicklung von Softwaresystemen als strukturierte Sammlungen von Implementierungen Abstrakter Datentypen (ADT)" definiert hat (vgl. [Mey88], siehe Kasten 1).

Die Implementierung eines ADT ist nach Meyer eine Klasse. Das Design solch einer Klasse erfolgt in Java idealerweise über ihr *Interface*, das die öffentliche Schnittstelle der Klasse und damit den *syntaktischen* Teil des ADT (*Typ* und *abstrakte Methoden*) definiert. Der *semantische* Teil des ADT (*Vor- und Nachbedingungen* sowie *Klasseninvarianten*) kann in Java mit Hilfe des DbC-Pakets C4J in einer eigenen Klasse, der so genannten *Vertragsklasse*, formuliert werden. Interface und Vertragsklasse zusammen erlauben so die vollständige Beschreibung des Designs eines ADT ausschließlich mit den Ausdrucksmitteln von Java. Die Umsetzung des ADT erfolgt dann in einer weiteren Klasse, die den produktiven Code enthält. Der Softwareentwurf mit DbC hat viele Vorteile:

- Die Funktionalität einer Klasse wird durch den Vertrag spezifiziert und dokumentiert.
- Die Vor- und Nachbedingen zu den Methoden eines Interface verwenden ausschließlich die Methoden des Interface selbst – inklusive seiner geerbten Methoden. Dadurch gewinnt es schon im Designprozess die für den produktiven Einsatz notwendige Reife und Stabilität.
- DbC setzt die *Command-Query-Separation* voraus: Methoden sind entweder *Kommandos* (Seiteneffekte, kein Rückgabewert) oder *Abfragen* (keine Seiteneffekte, Rückgabewert). In den Vertragsklassen dürfen nur Abfragen verwendet werden, da diese den

> Mit DbC kann man die Funktionalität einer Klasse mittels eines Vertrags spezifizieren, den die Klasse einhalten muss (siehe Abbildung 1). Ein Vertrag besteht aus folgenden Vertragsklauseln:
>
> - *Vorbedingungen:* Welche Zusicherungen muss der Aufrufer (*Kunde*) vor dem Aufruf der Funktionalität erfüllen, um das erwartete Ergebnis zu erhalten (*Pflichten des Kunden* = *Rechte des Anbieters*).
> - *Nachbedingungen:* Welche Zusicherungen muss der Aufgerufene (*Anbieter*) der Funktionalität nach der Ausführung der Methode erfüllen, sofern die Vorbedingungen erfüllt sind (*Pflichten des Anbieters* = *Rechte des Kunden*).
> - *Klasseninvarianten:* Welche Zusicherungen müssen zu jedem sichtbaren Zeitpunkt der Klasse – also direkt vor oder direkt nach jedem Methodenaufruf – erfüllt sein.
>
> Das Vorgehen, das Design einer Klasse vor ihrer eigentlichen Implementierung zu erstellen und exakt zu spezifizieren, entspricht dem Ingenieursansatz. Das Formulieren von Vor- und Nachbedingungen erfordert das Verständnis der fachlichen und technischen Anforderungen an eine Klasse.

Kasten 1: *Design by Contract (DbC).*

die autoren

Dr. Lars Alvincz
(E-Mail: lars.alvincz@andrena.de)
ist Softwareentwickler und Team Lead bei der andrena objects ag und hat in mehreren agilen Projekten umfassende Erfahrungen mit dem Einsatz von TDD in der Praxis gesammelt.

Hagen Buchwald
(E-Mail: hagen.buchwald@kit.edu)
war von 2002 bis 2008 Vorstand für den Bereich Banking bei der entory AG. Von 2008 bis 2010 war er Vorstandsvorsitzender des CyberForums e. V. Zur Zeit hat er einen Lehrauftrag am Karlsruher Institut für Technologie (KIT).

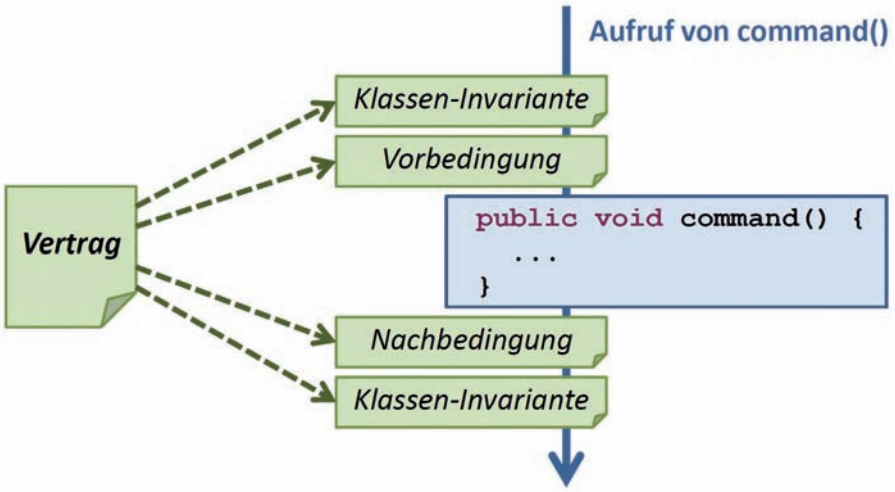

Abb. 1: Design by Contract: Absicherung durch Verträge.

Zustand der zu Grunde liegenden Klasse nicht verändern. Das führt zu einer besseren Architektur.

- Verletzungen der Verträge und deren konkrete Ursachen werden zur Laufzeit sofort erkannt und können schnell und gezielt behoben werden. Das steigert die Produktivität in der Entwicklung – gerade im Verbund mit Tests – enorm.

Darüber hinaus fügt C4J noch weitere Vorteile hinzu:

- Verträge werden in einer eigenen Klasse beschrieben. Für die Vertragsklauseln benötigt man somit keine komplexe Formalismen, wie z. B. die Prädikatenlogik.
- Wenn eine Subklasse von einer Superklasse erbt, dann erbt sie auch die in der Super-Vertragsklasse definierten Verträge. In C4J wird beim Vererben von Verträgen die Einhaltung des *Liskovschen Substitutionsprinzips* (vgl. [Lis93]) überwacht, d. h. Vorbedingungen dürfen höchstens aufgeweicht und Nachbedingungen nur verschärft werden.

Den Softwareentwurf mit DbC und C4J zeigt **Kasten 2** beispielhaft. Die positiven

Wie DbC mit C4J in der Praxis funktioniert, sieht man am besten an einem Beispiel. Nehmen wir einmal an, wir möchten eine Klasse entwickeln, die Zinsen berechnet, aufsummiert und schließlich die Summe liefert. Beginnen wir mit dem Interface:

```
@ContractReference(contractClassName = "ZinsRechnerSpecContract")
public interface ZinsRechnerSpec{
  void berechneZinsen(BigDecimal betrag);
}
```

Nun wechseln wir zur Vertragsklasse und erstellen eine Vor- und eine Nachbedingung für das Kommando berechneZinsen(..): Der übergebene Betrag darf nicht null und sein Wert muss positiv sein. Im Gegenzug sichern wir zu, die Zinsen korrekt zu berechnen:

```
public class ZinsRechnerSpecContract extends ContractBase<ZinsRechnerSpec>{
  public void pre_berechneZinsen(BigDecimal betrag) {
    assert betrag != null;
    assert betrag.compareTo(BigDecimal.ZERO) >= 0;
}
  public void post_berechneZinsen(BigDecimal betrag) {
    BigDecimal zinsSatz = m_target.getZinsSatz();
    BigDecimal result = betrag.multiply(zinsSatz);
    assert result.compareTo(m_target.getErgebnis()) == 0;
 }
}
```

Die von C4J definierte Variable m_target verweist auf das zu schützende Objekt. Für die Nachbedingung müssen wir das Interface um die Abfragen getZinsSatz() und getErgebnis() erweitern. Auch hier geben wir jeweils eine Nachbedingung an:

```
assert getReturnValue() != null;
```

Für getErgebnis() geben wir zusätzlich als Vorbedingung an, dass zuvor berechneZinsen(..) aufgerufen wurde. Das realisieren wir, indem wir ein neues boolean-Feld hinzufügen, das in post_berechneZinsen(..) gesetzt und dann in pre_getErgebnis() geprüft wird. Um schließlich auch die Zinssumme abfragen zu können, erweitern wir das Interface um getZinsSumme() und übernehmen im Vertrag Vor- und Nachbedingung von getErgebnis(). Nun können wir uns berechneZinsen(..) widmen. In pre_berechneZinsen(..) merken wir uns die bislang berechnete Zinssumme:

```
setPreconditionValue("old_summe", m_target.getZinsSumme());
```

In post_berechneZinsen(..) überprüfen wir, ob die Zinssumme korrekt erhöht wird:

```
BigDecimal old_summe = (BigDecimal) getPreconditionValue("old_summe");
BigDecimal summe = m_target.getZinsSumme();
assert summe.compareTo(old_summe.add(result)) == 0;
```

Die Funktionen set-/getPreconditionValue() werden von C4J vorgegeben und setzen *thread*-sichere Eigenschaften um. Nun benötigen wir noch ein Kommando, um den Zinssatz zu setzen. Wir erweitern das Interface um setZinsSatz(..) und fügen entsprechende Vor- und Nachbedingungen hinzu (Parameter darf nicht null sein, Zinssatz wird übernommen). Eine Übersicht über die erstellten Klassen zeigt **Abbildung 2**.

Kasten 2: Design by Contract mit C4J an einem Beispiel.

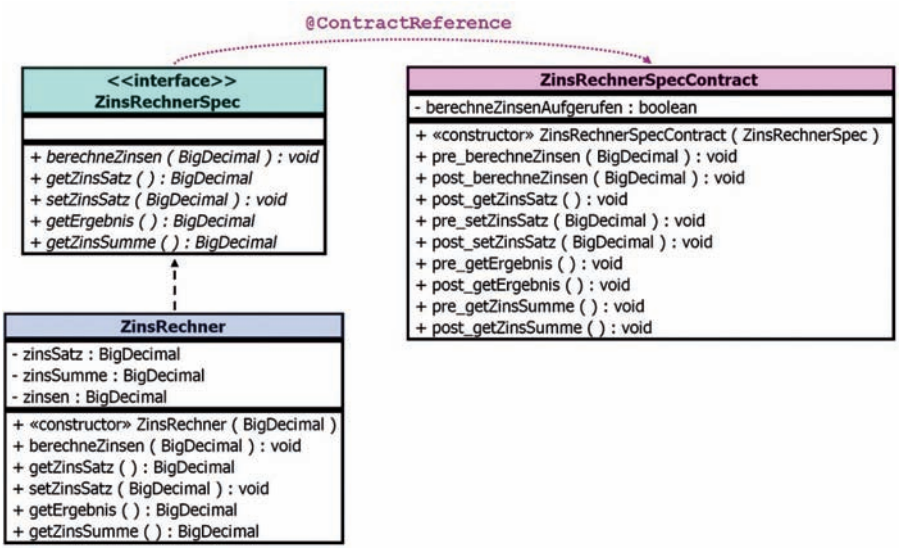

Abb. 2: Design by Contract mit C4J.

Auswirkungen der Entwicklung mit DbC konnten auch am KIT belegt werden: In einem Experiment sollten 250 Teilnehmer, eingeteilt in zwei Gruppen, eine programmiertechnische Aufgabe lösen. Die eine Gruppe erhielt lediglich die Aufgabenstellung. Die andere Gruppe erhielt zusätzlich eine Spezifikation in Form von Test- und Vertragsklassen. Beide Gruppen hatten 60 Minuten zur Lösung der Aufgabe. Das Experiment zeigte nun, dass nach Ablauf der Zeit nur 5 % der Entwickler der ersten Gruppe eine korrekte Lösung umgesetzt hatten, wohingegen es bei der zweiten Gruppe über 30 % waren.

Im Gegensatz zum klassischen DbC-Ansatz, der zunächst die Entwicklung eine umfassenden Spezifikation vorsieht, werden in der Wirtschaft auch pragmatischere Varianten eingesetzt. Zum Beispiel werden bei TbC die Vertragsklassen für bereits bestehende Implementierungen nachträglich erstellt, indem die verallgemeinerbaren Zusicherungen in die Vertragsklassen ausgelagert werden.

Anders als bei Unit-Tests können Verträge auch während der Ausführung von Produktiv-Code überprüft werden. Sollte es zu Performanceproblemen kommen, kann die Überprüfung stufenweise reduziert werden. So können beispielsweise zur Produktivsetzung lediglich die Vorbedingungen bestimmter Klassen überprüft werden.

TDD

TDD wurde stark von Kent Beck geprägt, der auch schon an der Entwicklung von *XP (eXtreme Programming)* beteiligt war. TDD beschreibt einen Entwicklungsprozess, der testgetrieben vorgeht (*Test-First-Programmierung*, siehe Kasten 3). TDD kann auch gut mit anderen XP-Praktiken (Paar-Programmierung, fortlaufende Integration usw.) kombiniert werden und ist fester Bestandteil der „Scrum Development Practices". Durch JUnit und die Integration in Entwicklungsumgebungen wie z. B. Eclipse fand TDD zunehmende Verbreitung. Über die letzten Jahre ist der Anteil der Projekte, die mit Scrum und TDD arbeiten, stetig gewachsen. Der Softwareentwurf mit TDD ist beispielhaft in Kasten 4 beschrieben.

Dass Tests zur Qualitätssicherung von Software nötig sind, ist allgemein anerkannt. TDD gehört im modernen Software-Engineering zum Stand der Technik. Eine Ausnahme stellen höchstens Projekte dar, bei denen ein sehr hoher Zeitdruck, geringe Qualitätsanforderungen und eine geringe Lebensdauer zusammentreffen. Das Schreiben von Tests erhöht zwar zunächst die Entwicklungszeiten. Diesen Mehraufwand bekommt man jedoch im weiteren Software-Lebenszyklus mehr als zurück: TDD führt zu testbarem Code, der einfach strukturiert ist. Die Testfälle geben dem Programmierer Sicherheit bei umfangreichen Refaktorisierungen. Darüber hinaus stellen sie auch eine Dokumentation bzw. Spezifikation des Programms dar.

Gemeinsamkeiten und Unterschiede

TDD und DbC verfolgen dasselbe Ziel: die Verbesserung der Softwarequalität durch modulare Strukturierung und durch Sicherstellung der funktionalen Korrektheit. Beide unterscheiden sich aber deutlich in der Art und Weise, wie sie dieses Ziel erreichen wollen: Bei TDD beschreiben einzelne Testfälle das erwartete Verhalten (*Specification by Example*), während DbC das Verhalten (bzw. eine Teilmenge davon) vollständig beschreibt. TDD geht inkrementell vor: Die Zahl der Testfälle erhöht sich in dem Maße, wie die Software an Funktionalität dazu gewinnt. Der klassische DbC-Ansatz dagegen scheint eher in die Designphase zu passen, wo vor der Implementierung das Verhalten spezifiziert wird.

Wie lassen sich nun beide Ansätze zusammenbringen? Lassen sie sich gewinnbringend miteinander kombinieren, um das

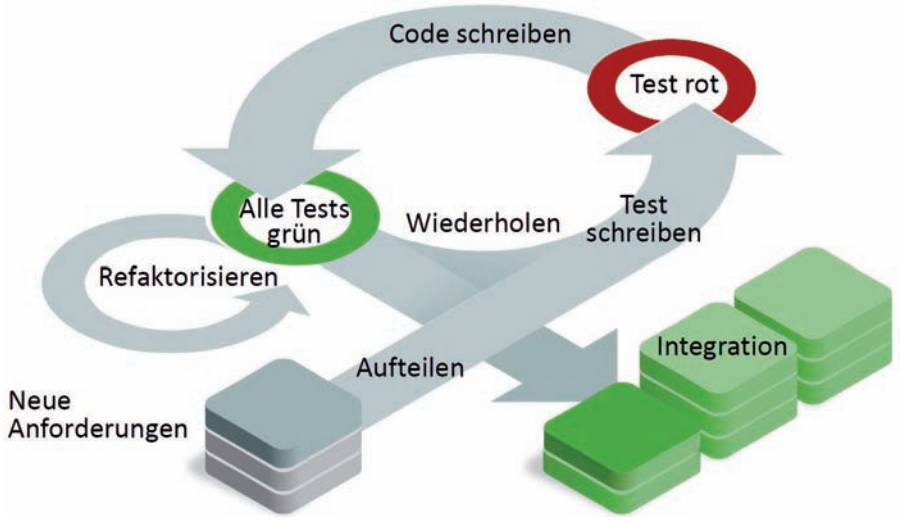

Abb. 3: Der TDD-Kreislauf.

TDD sieht folgenden Ablauf vor, um eine Funktionalität zu implementieren:

- *Test schreiben*, der die zu implementierende Funktionalität überprüft. Dieser Test sollte initial fehlschlagen (*make it red*).
- *Code schreiben*, damit der Test grün wird (*make it green*). Lieber einfach als perfekt, optimiert wird im nächsten Schritt.
- *Code refaktorisieren*, um die Codequalität zu verbessern (*make it better*). Insbesondere ist sicherzustellen, dass der Test grün bleibt

Zur Umsetzung einer Aufgabe wird dieser Ablauf mehrfach für jede Teilfunktionalität durchlaufen. Insbesondere bedeutet TDD, dass neuer Code nur dann geschrieben wird, wenn Tests fehlschlagen (Schritt 2). Das Vorgehen, den Test vor der Implementierung zu schreiben, mag auf den ersten Blick der Intuition widersprechen, hat aber viele Vorteile:

- Der geschriebene Code wird so entwickelt, dass er testbar ist.
- Die Tests sind keine Nachimplementierungen des Produktivcodes.
- TDD erzwingt einfache Schnittstellen, was sich positiv auf die Architektur auswirkt.
- TDD führt zu einer hohen Testabdeckung und ermöglicht somit überhaupt erst ein sicheres Refaktorisieren.
- Der Code wird durch die Tests dokumentiert.

Testen und insbesondere testgetriebenes Entwickeln sind nicht immer ganz einfach. Man benötigt einiges an Programmier- und Testerfahrung, um gute Tests zu schreiben und um abzuschätzen, zu welchem Zeitpunkt genügend Tests vorhanden sind. Man merkt aber schnell, dass Testen nicht nur Zeit kostet, sondern auch einen erheblichen Qualitätszuwachs mit sich bringt und auf lange Sicht wegen des einfacheren Designs sogar Zeit spart. Den vollständigen TDD-Prozess zeigt Abbildung 3. Hier ist auch dargestellt, wie die Anforderungen abgearbeitet werden und wie das Ergebnis schließlich integriert wird (für ausführliche Informationen vlg. z. B. [Bec03] und [Lin05]).

Kasten 3: *Test Driven Design (TDD).*

Ziel von hoch qualitativer Software noch schneller zu erreichen? Oder sind sie zu widersprüchlich und somit nicht zusammenzubringen? Gleich vorweg: An dieser Stelle können und werden wir diese Fragen nicht abschließend beantworten. Wir beschreiben, wie beide Verfahren miteinander kombiniert werden könnten – ob dieser Ansatz so tatsächlich umzusetzen ist und die Entwicklung verbessert, wird sich erst in der Praxis zeigen.

DbC und agil?

Auf den ersten Blick passt DbC nicht in die agile Entwicklung. Eine detaillierte Vorab-Spezifikation des Verhaltens zu erstellen, passt besser zu klassischen Entwicklungsmodellen. Allerdings zeigt schon der pragmatische Ansatz *Test by Contract*, dass DbC auch anders eingesetzt werden kann. Darüber hinaus stellt sich die Frage: Wie hoch sind die Überschneidungen zwischen Verträgen und Tests? Beschreibe ich erst das Verhalten im Vertrag und schreibe dann einen Test dazu, der dieses mit Verhalten abprüft? Wie viel Flexibilität erlaubt DbC, insbesondere in Hinblick auf das Refaktorisieren? Diese Fragen deuten schon an, auf was zu achten ist, wenn man TDD und DbC miteinander kombiniert. Das *DRY*-Prinzip (*Don't Repeat Yourself*) sollte auf keinen Fall verletzt werden; auch sollte das Refaktorisieren nicht deutlich behindert werden.

TDD plus DbC

Betrachten wir das Vorgehen bei TDD: Die Anwendung soll testgetrieben (*test-first*) entwickelt werden, d. h. wir definieren das gewünschte Verhalten in Form von Unit-Tests und entwickeln dann den Programmcode, der dieses Verhalten umsetzt. Nebenbei bemerkt: Somit lassen sich die Unit-Tests auch als (partielle) Spezifikation des Verhaltens ansehen (*Specification by Example*). Der Entwickler entwirft also zunächst Testfälle, die das gewünschte Verhalten beschreiben. Zur Validierung überlegt er sich in der Regel mögliche Fehlerfälle und entwickelt entsprechende Unit-Tests. Dazu gehören beispielsweise die Überprüfung von Eingabeparametern (not-null-Validierung, die Einhaltung eines Wertebereichs) und das Sicherstellen eines bestimmten Ablaufs bei Funktionsaufrufen (beispielsweise dass eine Verbindung nur geschlossen werden darf, wenn diese zuvor besteht). Im Allgemeinen führen solche Validierungen zu vielen trivialen Testfällen und blähen die Tests unnötig auf. Gleichzeitig sind sie aber äußerst hilfreich, um bei einem Fehler die Ursache einzugrenzen.

Insbesondere bei den externen Schnittstellen einer Anwendung oder einer Komponente sind diese Validierungen unabdingbar. Hier bietet es sich an, die Fehlerfälle nicht in den Unit-Tests, sondern in den Verträgen zu behandeln. Mittels Klasseninvarianten sowie Vor- und Nachbedingungen lassen sich die Validierungen leicht umsetzen. Mit DbC kann man darüber hinaus auch für Interfaces Verträge angeben – das ist mit Unit-Tests nicht möglich. Diese Verträge gelten dann für alle Klassen, die die Interfaces implementieren. Eine Möglichkeit, TDD um DbC zu erweitern, wäre es also, zunächst wie gehabt *test-first* zu entwickeln. Sobald eine Funktionalität umgesetzt ist (also nach mehreren TDD-Zyklen), können für die entwickelten Klassen und Interfaces Verträge angegeben werden. Dabei kann man zunächst mit einfachen Validierungen anfangen (*not-null*-Checks, Wertebereiche). Dadurch werden die Tests und auch der Code schlanker, weil entsprechende Überprüfungen zentral im Vertrag durchgeführt werden. Aber mit diesen Validierungen ist nur ein kleiner Teil des Potenzials von DbC ausgeschöpft: Auch umfangreichere Vor- und Nachbedingungen sind möglich. So lässt sich ein gewisses Protokoll bei den Aufrufen von Funktionen einer Klasse überprüfen. Nachbedingungen können die Situation vor und nach der Funktionsausführung vergleichen und somit das Ergebnis validieren. Auch Klasseninvarianten können angegeben werden. Bei alledem ist es jedoch wichtig, darauf zu achten, dass das DRY-Prinzip eingehalten wird.

Derzeit fehlt für DbC noch eine Werkzeugunterstützung. Es gibt zwar ein Eclipse-Plug-In für C4J, das sich aber noch in einem sehr frühen Stadium befindet. Die Verträge sind an die entsprechenden Klassen gekoppelt. Bei C4J geschieht dies durch eine Annotation in der Klasse und durch die Verwendung des entsprechenden Methodennamens in den Vor- und Nachbedingungen im Vertrag. Selbst einfa-

Zur Veranschaulichung von TDD verwenden wir ebenfalls den Zinsrechner, der Zinsen berechnen und aufsummieren kann. Zunächst schreiben wir folgenden Testfall: Bei einem Betrag von 0 Euro sollen keine Zinsen anfallen[1]).

```
public void testKeineZinsen() {
    rechner = new ZinsRechner();
    rechner.berechneZinsen(BigDecimal.ZERO);
    assertEquals(BigDecimal.ZERO, rechner.getErgebnis());
}
```

Jetzt gibt es viele Compiler-Fehler, also legen wir die Klasse ZinsRechner einschließlich des Konstruktors und der zwei Methoden an. Wir machen es uns einfach, lassen alle Methodenrümpfe leer und lassen getErgebnis() konstant den Wert 0 zurückliefern. Mit Erfolg: Der Test ist grün – die Implementierung aber bei Weitem noch nicht vollständig. Wir schreiben den nächsten Testfall: Wenn wir 10 % Zinsen von 50 Euro berechnen, soll das Ergebnis 5 Euro sein. Die Übergabe des Zinssatzes erfolgt über den Konstruktor:

```
public void testMitZinsen() {
    rechner = new ZinsRechner(BigDecimal.valueOf(0.10d));
    rechner.berechneZinsen(BigDecimal.valueOf(50d));
    assertEquals(5d, rechner.getErgebnis().doubleValue());
}
```

Dieser Test ist zunächst rot. Nun fügen wir der Klasse ZinsRechner die Felder zinsSatz und zinsErgebnis hinzu. Der Konstruktor speichert sein Argument in zinsSatz. Die Funktion berechneZinsen(..) erwartet einen Betrag, berechnet die Zinsen und speichert sie in dem Feld zinsErgebnis, das wiederum von getErgebnis() zurückgeliefert wird. Nun ist auch der zweite Test grün. Schließlich schreiben wir noch einen Test für die Summenbildung:

```
public void testMitZinsenUndSumme() {
    rechner = new ZinsRechner(BigDecimal.valueOf(0.10d));
    rechner.berechneZinsen(BigDecimal.valueOf(50d));
    rechner.berechneZinsen(BigDecimal.valueOf(100d));
    assertEquals(15d, rechner.getZinsSumme().doubleValue());
}
```

Auch dieser Test ist zunächst rot. Wir fügen der Klasse ZinsRechner ein neues Feld zinsSumme hinzu, setzen es im Konstruktor auf 0 und erhöhen es bei jedem Aufruf von berechneZinsen(..) um das ermittelte zinsErgebnis. Schließlich fügen wir noch den Getter getZinsSumme() hinzu. Nun sind alle drei Tests grün und wir können zum einen sicher sein, dass die Klasse ZinsRechner wie erwartet funktioniert, zum anderen haben wir – in Form der Testfälle – eine Dokumentation geschaffen, die das Verhalten spezifiziert. Der Einfachheit halber haben wir keine Refaktorisierungen der Testfälle durchgeführt. Dies sollte aber stets geschehen, sobald ein Test grün wird, denn an Testcode gelten die gleichen Ansprüche wie an Produktivcode.

***Kasten 4:** TDD am Beispiel.*

che Refaktorisierungen, wie das Umbenennen von Methoden, werden somit verhindert. Das ist aber kein inhärentes Problem von DbC, sondern vielmehr ein Ausdruck fehlender Werkzeugunterstützung. Hier gibt es noch Handlungsbedarf, bevor sich DbC in der Praxis einsetzen lässt. Der D3-Explorer (siehe Kasten 5) ist ein erster Schritt in diese Richtung.

[1]) *Für Geldbeträge sollte BigDecimal statt Double verwendet werden, da Double die Werte im Binärsystem darstellt und viele Werte wie z. B. 0,1 nicht exakt darstellen kann.*

Fragen und Antworten

Folgende Fragen haben wir im vorausgegangenen Artikel ([Buc11]) gestellt und versuchen diese im Folgenden zu beantworten:

- *Wie groß ist das Risiko, dass Verträge den Produktivcode „zementieren" und Refaktorisierungen verhindern?* Ein typischer Fehler, der jedoch gerne beim Erstellen von Vertragsklassen gemacht wird, ist die Überspezifikation: Die Verträge werden mit einem klaren Bild der Implementierung vor Augen erstellt. Das geschieht vor allem beim nachträglichen Erstellen der Vertragsklassen zu bereits bestehenden Implementierungen. Daher ist es besser, Vertragsklassen für Interfaces zu erstellen, da hier der notwendige Grad der Abstraktion automatisch gewahrt bleibt.
- *Wie sieht es generell mit der Refaktorisierungsunterstützung der Entwicklungsumgebungen aus?* Dadurch, dass Vor- und Nachbedingungen über die Präfixe pre_ bzw. post_ rein textuell an die referenzierte Methode gekoppelt sind, besteht tatsächlich der Bedarf nach erweiterten Refaktorisierungswerkzeugen, die solche Zusammenhänge auch berücksichtigen.
- *Welche Werkzeugunterstützung ist generell nötig, um TDD bzw. DbC effizient anzuwenden?* Für TDD gibt es schon vielfältige Unterstützung durch Werkzeuge. Für DbC benötigt man zum einen eine Implementierung wie C4J. Darüber hinaus ist für eine effizientes Arbeiten eine Unterstützung durch die Entwicklungsumgebung über den kompletten Lebenszyklus eines Vertrags (Erzeugung der Klasse, Erweitern, Refaktorisierung) unabdingbar.
- *Kann man sich als Verfechter von TDD darauf einlassen, Teile der Test-Suite aus definierten Verträgen erzeugen zu lassen, anstatt diese test-first zu entwickeln?* Ja, man kann, muss es jedoch nicht. Der Entwickler kann sich folgende Frage stellen: Kann ich die in den Tests beispielhaft formulierte Spezifikation einer Klasse in allgemein gültige Regeln übersetzen? Falls ja, ist eine Vertragsklasse die ideale Form, um diese Regeln zu hinterlegen. Zudem werden die Regeln dadurch vererbbar – und entfalten somit ihre wahre Mächtigkeit.
- *Ist bei nicht-trivialen Beispielen der Vertrag tatsächlich einfacher als die Implementierung selbst? Oder besteht die Gefahr, die Verträge zu dicht an der Implementierung zu formulieren?* Vertragsklassen sollten sich auf der gleichen Abstraktionsebene befinden

Die Werkzeugunterstützung für C4J befindet sich noch in den Anfängen. Derzeit läuft zu diesem Thema eine Diplomarbeit, die vom KIT und der Firma andrena objects ag gemeinsam betreut wird. In dieser Arbeit wird das Eclipse-Plug-In „D3-Explorer" entwickelt. Mit diesem Plug-In können Vertragsklassen zu einem bestehenden Interface oder einer bestehenden Klasse generiert werden – mitsamt Vorlagen für die pre- und post-Funktionen. Weiterhin erlaubt der D3-Explorer das automatisierte Umbenennen von Klassen und Methoden. Auch Tests für Randfälle können automatisch generiert werden. Der D3-Explorer befindet sich zur Zeit in der Pre-Release-Phase und wird im kommenden Sommersemester am KIT ausgiebig auf seine Einsatztauglichkeit getestet.

***Kasten 5:** D3-Explorer.*

wie Interfaces. Daher lautet unsere Empfehlung, Vertragsklassen für Interfaces zu erstellen. Wer dieser Regel folgt, kann per Definition nicht in die Falle einer Überspezifikation durch ein White-Box-Denken verfallen. Die Gefahr einer solchen Überspezifikation ist jedoch fraglos gegeben und es erfordert Disziplin, ihr nicht zu erliegen. Falls doch: Auch Vertragsklassen können verbessert werden – das ist ganz normal und durchaus im Sinne des Erfinders.

- *Verträge benötigen zu ihrer Überprüfung ebenso Testfälle. Insofern stellt sich die Frage, ob bei der Definition von Verträgen zusätzlich zu den Unit-Tests das DRY-Prinzip verletzt wird.* Die Zusicherungen in den Vertragsklassen werden nur überprüft, wenn die geschützte Klasse auch genutzt wird. Diese Nutzung sollte bereits in der Implementierungsphase durch Unit-Tests erfolgen. Das DRY-Prinzip wird hierbei nicht verletzt, wenn die verallgemeinerbaren Zusicherungen von der Testklasse in die Vertragsklasse ausgelagert werden.
- *Ließe sich das Definieren der Verträge in den TDD-Lebenszyklus integrieren? Oder erfolgt das im Allgemeinen möglichst vollständig im voraus und stellt sich DbC damit konträr zu TDD auf?* Auch das Erstellen von Vertragsklassen kann – wie im DbC-Beispiel gezeigt – einem iterativen und inkrementellen Ansatz folgen. Daher lässt sich das Erstellen der Vertragsklassen auch in den agilen Entwicklungszyklus integrieren. Vielversprechend erscheint der Ansatz, induktiv mittels TDD beispielhaft die allgemeinen Regeln zu finden, nach denen die Klasse funktionieren soll, und diese Regeln dann systematisch in Vertragsklassen auszulagern.

Fazit und Ausblick

Die Verbindung von TDD und DbC erscheint vielversprechend, verfolgen doch beide Ansätze dasselbe Ziel. Wenn man die verallgemeinerbaren Validierungen in die Verträge verschiebt, kann man sich in den Testfällen auf das Wesentliche konzentrieren. Außerdem erlaubt DbC es, auch für Interfaces Verträge anzugeben. Mit C4J lassen sich die Verträge für die gesamte Anwendung aktivieren – gleichzeitig kann die Überprüfung stückweise abgeschaltet werden, sollte es zu Performanceproblemen kommen. Um über die Tauglichkeit der Verbindung von TDD und DbC jedoch urteilen zu können, fehlen bislang Erfahrungen aus der Praxis, da die Werkzeugunterstützung sich noch in den Anfängen befindet. Hier sollte sich die Java-Community an Microsofts Plug-In *Code Contracts* orientieren, das überzeugend demonstriert, wie DbC den Programmierer beim Entwickeln qualitativ hochwertigen Codes unterstützen kann. Einen ersten Schritt in diese Richtung gehen wir mit der Entwicklung des D3-Explorers. Die Evaluierung dieses Tools in realistischen Projekten wird weiteren Aufschluss darüber geben, wie TDD und DbC zusammenpassen. Wir laden den Leser ein, die Diskussion zum Thema im Internet weiterzuführen (vgl. [GoGr]). ■

Literatur & Links

[Bec03] K. Beck, Test Driven Development. By Example, Addison Wesley Longman 2003

[Buc11] H. Buchwald, T. Reinstorf, Design by Contract und TDD: Partner oder Kontrahenten?, in: OBJEKTspektrum 3/2011

[C4J] C4J – Design by Contract for Java, siehe: http://c4j.sourceforge.net

[GoGr] Google-Gruppe TDD und DbC, siehe: http://groups.google.com/group/tdd-und-dbc

[Lin05] J. Link, Softwaretests mit JUnit, dpunkt.verlag 2005

[Lis93] B.H. Liskov, J.M. Wing, Family Values: A Behavioral Notion of Subtyping, Pittsburgh 1993

[Mey88] B. Meyer, Object Oriented Software Construction, Prentice Hall 1988

[PDC08] Microsoft Research, Code Contracts, siehe: http://channel9.msdn.com/blogs/pdc2008/tl51 und http://research.microsoft.com/contracts