DESIGN BY CONTRACT UND TDD: PARTNER ODER KONTRAHENTEN?

Dieser Artikel soll eine Diskussion über die Frage, ob sich die explorative, induktive Herangehensweise von TDD mit dem analytischen, deduktiven Ansatz von DbC verbinden lässt, anschieben. Behindern sich diese Methoden gegenseitig oder können sie sich wechselseitig unterstützen? Die Erkenntnisse aus dieser Diskussion stellen wir in einem vertiefenden Artikel in einer der nächsten Ausgaben von OBJEKTspektrum vor.

Die *testgetriebene Entwicklung (TDD)* (vgl. [Bec02]) ist im Bereich der agilen Software-Engineering-Techniken eine wichtige Kernpraktik. Im Zuge der immer weiter fortschreitenden Etablierung agiler Prozessmodelle bekommt TDD in letzter Zeit vermehrt Aufmerksamkeit bei den Softwareentwicklern. Ähnlich verhält es sich mit *Design by Contract (DbC)*: Von Bertrand Meyer im Rahmen der Programmiersprache Eiffel (vgl. [Mey97]) bekannt gemacht, blieb der Methodik auf Grund von technischen Beschränkungen lange der Durchbruch verwehrt. In der jüngeren Vergangenheit rückt DbC durch leitungsfähigere Hardware und Werkzeugunterstützung wieder stärker in das Blickfeld der Softwareentwickler (vgl. „Microsoft CodeContracts", [PDC08] und [C4J]).

Die generelle Frage, die sich daraus für die Softwareentwicklung ergibt, lautet: Können sich die beiden Ansätze TDD und DbC ergänzen oder sind sie – zumindest teilweise – redundant oder stehen sie konträr zueinander? Die Antwort auf diese Frage wollen (und können) wir an dieser Stelle gar nicht geben, sondern möchten stattdessen Gemeinsamkeiten nennen und anschließend einige Fragen stellen, die uns in diesem Zusammenhang beschäftigen.

Beide Ansätze haben eine weitestgehend identische Zielsetzung: Beide Methoden reklamieren für sich, die Erstellung von qualitativ hochwertiger Software zu fördern. Sie forcieren, dass sich der Entwickler zuerst Gedanken über die Schnittstelle aus Verwendersicht macht und unterstützen – konsequent angewendet – den Entwickler, die Funktionalität in kleine, gut verständliche Einheiten mit jeweils nur genau einer Verantwortung aufzuteilen. Allgemein nehmen beide Ansätze für sich in Anspruch, zu einem besseren, einfacheren Softwaredesign zu führen.

Beide Methoden spezifizieren das Verhalten von Objekten bzw. Methoden. TDD tut dies größtenteils über einzelne Testfälle (Stichwort *Specification by Example*). DbC strebt hingegen die formale Spezifikation einer Schnittstelle an. Folgende Fragen stellen sich:

- Wie groß ist das Risiko, dass Verträge (ebenso wie fragile Tests) den Produktivcode so fest zementieren, dass Refaktorisierungen auf Grund des nötigen Anpassungsaufwands der Verträge unterbleiben?
- Wie sieht es generell mit der Refaktorisierungs-Unterstützung der Entwicklungsumgebungen aus? Werden die Vertragsdefinitionen bei der Durchführung (halb-)automatischer Refaktorisierungen durch die Entwicklungsumgebung berücksichtigt?
- Welche Werkzeugunterstützung ist generell nötig, um TDD bzw. DbC effizient anzuwenden?
- Kann man sich als Verfechter der testgetriebenen Entwicklung darauf einlassen, Teile der Test-Suite aus definierten Verträgen erzeugen zu lassen, anstatt diese *test-first* zu entwickeln?
- Ist bei nicht-trivialen Beispielen der Vertrag tatsächlich einfacher als die Implementierung selbst? Oder läuft man leicht Gefahr, die Verträge zu dicht an der Implementierung zu formulieren?
- Verträge benötigen zu ihrer Überprüfung ebenso Testfälle, können also beispielsweise Unit-Tests nicht ersetzen. Insofern stellt sich die Frage, ob bei der Definition von Verträgen zusätzlich zu den Unit-Tests das *DRY*-Prinzip (*Don't Repeat Yourself*) verletzt ist?
- Eine weitere Frage betrifft den Entwicklungszyklus: TDD definiert den TDD-Zyklus, der in kleinen, beherrschbaren Schritten, inkrementell die Definition von Tests, die Implementierung des Produktivcodes und die Refaktorisierung des Quellcodes umfasst. Ließe sich das Definieren der Verträge in diesen Zyklus integrieren? Oder erfolgt die Definition der Verträge möglichst vollständig im vorhinein und stellt sich DbC damit konträr zu TDD auf?

Eine der zentralen Fragen zum möglichen Zusammenwirken von TDD und DBC ist also: Lässt sich die explorative, induktive Herangehensweise von TDD mit dem analytischen, deduktiven Ansatz von DbC verbinden? Behindern sich die Methoden gegenseitig oder können sie sich wechselseitig unterstützen?

Wir halten das Thema für sehr spannend und relevant. Aus diesem Grund möchten wir einen intensiven Meinungsaustausch zu dem Verhältnis von DbC und TDD anregen. Auf dem Entwicklertag am 27. Mai 2011 in Karlsruhe (vgl. [ENTW]) wird es eine gute Gelegenheit geben, über dieses Thema zu diskutieren: Ein Track der Konferenz mit abschließender Podiumsdiskussion wird sich mit diesem Thema beschäftigen. Außerdem haben wir eine Google-Gruppe zum Meinungsaustausch eingerichtet (vgl. [GOGR]). Zusätzlich ist ein vertiefender Artikel für eine der nächsten Ausgaben von OBJEKTspektrum geplant. ■

Literatur & Links

[Bec02] K. Beck, Test Driven Development. By Example, Addison Wesley Longman, 2002

[Mey97] B. Meyer, Object-oriented Software Construction, Prentice Hall International, 1997

[C4J] C4J (Contracts for Java) Projektseite, siehe: http://c4j.sourceforge.net/

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

[ENTW] Karlsruher Entwicklertag 2011, siehe: http://www.entwicklertag.de/

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

die autoren

Hagen Buchwald
(E-Mail: hagen.buchwald@kit.edu)
hat einen Lehrauftrag am KIT, dem Karlsruher Institut für Technologie, das im Zuge der Exzellenz-Initiative als Zusammenschluss der Universität Karlsruhe und des Forschungszentrums Karlsruhe entstanden ist.

Timm Reinstorf
(E-Mail: timm.reinstorf@andrena.de)
arbeitet seit acht Jahren als Softwareentwickler und Coach für agile Prozess- und Engineering-Methoden bei andrena.