Mit Netz und doppelten Methoden

Wie die Kombination von TDD und Contracts das Refactoring erleichtert

Lars Alvincz, Hagen Buchwald

Test-Driven Development (TDD) gilt als eine der Voraussetzungen für Agile Software Engineering. Als validierte Methode, um eine hohe Rate an automatisierten Tests zu erreichen, knüpft TDD ein Sicherheitsnetz, das die Entwickler beim Refactorisieren schützt. Doch wie umfangreich müssen diese Komponententests sein, um die möglichen Aufrufkombinationen der Methoden der zu testenden Klasse auch wirklich vollständig abzudecken? Fokussiert man sich auf elementare Tests, ist man zwar sehr effizient, kann jedoch die Gefahr nicht ausschließen, dass subtile Bugs einfach „durchrutschen" oder doch Funktionalität gefährdet wird. Mit der Kombination von TDD und Contracts hingegen wird das Netz wesentlich feinmaschiger und ermöglicht, fehlerhafte Zustände direkt beim Testen zu erkennen. Der springende Punkt bei dieser Kombination: TDD macht bislang implizite Annahmen über die Nutzung einer Klasse aus Kundensicht sichtbar, diese Annahmen werden in den Contracts explizit formuliert. Das unterstützt die agilen Prinzipien und erlaubt eine ganz neue Effizienz in der Entwicklung.

Test First

TDD lohnt sich. An dieser Aussage werden wenige agile Entwicklerinnen und Entwickler Zweifel anmelden. Wie kaum eine andere Methode markiert TDD, dass das finstere Mittelalter der Softwareentwicklung lange vorbei ist, jene Zeit, in der die Softwareentwicklung in die vier langwierigen Phasen Design, Analyse, Implementierung und Test unterteilt war. In der agilen Entwicklung sind an ihre Stelle drei Aktionen getreten, die nur Minuten in Anspruch nehmen: Tests formulieren, Code schreiben, refactorisieren. Ausgedehnte Analyse und Design? Relikte aus der düsteren Vorgeschichte. Softwareverträge? Überbleibsel, die der Evolution der objektorientierten Programmiersprachen zum Opfer fielen.

Vielleicht auch nicht. Vielleicht ist das Ganze auch nur eine Frage der Perspektive. Denn, wie bei TDD, einen Test zu schreiben, ehe der dazugehörige Code existiert, bedeutet auch nichts anderes, als die Anforderungen an die Software aus Kundenperspektive zu betrachten. Das macht die „Test first"-Methode zu einem sehr eleganten Weg, die Stiefkinder Analyse und Design wieder in die Familie aufzunehmen, sprich, direkt in die Entwicklung zu integrieren. Jedoch nicht mehr als langwierige, alles vorausdenkende Phase eines *Forward Software Engineering*-Teams, bestehend aus Analysten und Architekten. Sondern als minutenschnelle, auf einen konkreten Testfall fokussierte Aktion eines einzelnen Softwareentwicklers.

Softwareverträge

Wie aber steht es mit den Softwareverträgen? Bei diesem Begriff zucken viele Entwicklerinnen und Entwickler zusammen, weil sie Verträge sofort mit dem Forward Software Engineering assoziieren. Doch diese automatische Koppelung greift zu kurz. Tatsächlich ist es sogar im Gegenteil so, dass die Kombination von TDD und Verträgen ein großer Vorteil für das Refactoring sein kann – und damit für das Agile Software Engineering.

Das Eiffel-Erbgut „Design by Contracts" (kurz: DbC) soll helfen, agil zu entwickeln? Diese Aussage bedarf zugegebenermaßen der Erklärung. Dazu werfen wir zunächst nochmals einen Blick auf die Vorgehensweise bei TDD, die sich in drei Schritte gliedert:

- Schreibe einen Test, der fehlschlägt (und fehlschlagen muss, weil der entsprechende Code noch gar nicht existiert).
- Schreibe exakt so viel Code, dass der Test läuft.
- Beseitige duplizierten Code und andere Code Smells.

Jeder nach dieser Vorgehensweise erstellte Test enthält mindestens ein Assert-Statement, also eine Zusicherung über den Zustand der „class under test".

Beispiel

Unser konkretes Beispiel sei ein *Bankkonto*, dessen dynamisches Verhalten vollständig und präzise beschrieben werden soll. Dabei handele es sich um ein spezielles Konto für Studierende, das per Grundvoraussetzung keinesfalls überzogen werden darf. Zulässige Operationen seien die Abfragen nach dem Kontostand, dem Kontoinhaber und dem maximal verfügbaren Betrag, außerdem die Kommandos zur Einzahlung bzw. zum Abheben.

Die öffentliche Schnittstelle der Klasse `Account` kann wie folgt durch ihr Interface `AccountSpec` spezifiziert werden

```
public interface AccountSpec {
  void deposit(int amount);
  void withdraw(int amount);
  int getBalance();
}
```

Hinweis: Aus Gründen der Verständlichkeit wird in diesem Beispiel der Datentyp `int` verwendet. In einer professionellen Umsetzung würde hier ein dedizierter Datentyp `Money` sicherlich angebracht sein.

Der Testfall `AccountTest` erstellt vor der Ausführung jedes Tests ein neues Bankkonto

```
private AccountSpec account;
```

durch die Anweisung

```
account = new Account();
```

und überprüft das Verhalten des Bankkontos durch vier positive Tests:

- Der Test `newAccount_shouldHaveBalanceZero()` fordert, dass für ein neu angelegtes Bankkonto gilt, dass dieses den Kontostand 0 hat: `assertEquals(0, account.getBalance());`
- Der Test `deposit_shouldUpdateBalance()` fordert, dass nach einer Einzahlung auf ein leeres Bankkonto dieses die Einzahlung als neuen Kontostand ausweist:
  `account.deposit(100); assertEquals(100, account.getBalance());`
- Der Test `repeatedDeposit_shouldUpdateBalance()` fordert, dass sich auch eine wiederholte Einzahlung korrekt im neuen Kontostand widerspiegelt: `account.deposit(100); account.deposit(200); assertEquals(300, account.getBalance());`

- Der Test `depositAndWithdraw_shouldUpdateBalance()` fordert, dass sich auch Ein- und Auszahlungen korrekt im Kontostand widerspiegeln: `account.deposit(500); account.withdraw(300); assertEquals(200, account.getBalance());`

Ebenso enthält er jedoch auch drei negative Tests, die durch die Annotation

```
@Test(expected = AssertionError.class)
```

ausdrücken, dass von diesen Tests erwartet wird, dass sie mit einem `AssertionError` scheitern:

- Der Test `depositNegativeAmount_shouldFail()` fordert, dass der Versuch einer negativen Einzahlung als unzulässig abgewiesen wird: `account.deposit(-100);`
- Der Test `withdrawNegativeAmount_shouldFail()` fordert, dass der Versuch einer negativen Auszahlung als unzulässig abgewiesen wird: `account.withdraw(-200);`
- Der Test `withdrawAmountGreaterThanBalance_shouldFail()` fordert, dass der Versuch einer Auszahlung, die größer als der Kontostand ist, als unzulässig abgewiesen wird: `account.deposit(300); account.withdraw(500);`

Mit diesen sieben JUnit-Assert-Statements haben wir nun das erste Sicherheitsnetz, das den Code testbar und damit flexibel hält:

- `assertEquals(0, account.getBalance());`
- `assertEquals(100, account.getBalance());`
- `assertEquals(300, account.getBalance());`
- `assertEquals(200, account.getBalance());`
- `@Test(expected = AssertionError.class)`
- `@Test(expected = AssertionError.class)`
- `@Test(expected = AssertionError.class)`

Eine ähnliche Spezifikation lässt sich realisieren, wenn anstelle von JUnit-Assertions passende Vor- und Nachbedingungen als Java-Assertions formuliert und in Verträgen festgehalten werden. In Java lassen sich mit Hilfe des leichtgewichtigen DbC-Frameworks C4J [C4J] über sogenannte Vertragsklassen solche Verträge formulieren. Für das oben vorgestellte Interface `AccountSpec` lautet die zugehörige Vertragsklasse `AccountSpecContract` wie folgt

```
public class AccountSpecContract implements AccountSpec {
  @Target
  private AccountSpec target;

  @ClassInvariant
  public void classInvariant() {
    assert target.getBalance() >= 0 : "balance >= 0";
  }
  @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";
    }
  }
  @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() {
    return ignored();
  }
}
```

Betrachtet man nun die Nachbedingungen (Post-Conditions) im Einzelnen, so weisen sie eine große Ähnlichkeit mit den Forderungen der positiven Testfälle auf:

- Das Java-Assert-Statement `assert target.getBalance() >= 0;` in der Klasseninvariante `classInvariant()` fordert, dass zu jedem sichtbaren Zeitpunkt (und damit auch direkt nach der Erzeugung des Objekts Bankkonto) der Kontostand größer oder gleich 0 ist. Hinweis: Für den Konstruktor könnte auch eine dedizierte Nachbedingung erstellt werden, sodass explizit ausgedrückt würde, dass direkt nach der Erzeugung des Kontos der Kontostand 0 beträgt. Aus Einfachheitsgründen wurde hier jedoch bewusst darauf verzichtet und stattdessen die Klasseninvariante genutzt, die für alle Methodenaufrufe der geschützten Klasse `Account` gilt und zusichert, dass der Kontostand stets größer oder gleich 0 ist.
- Das Java-Assert-Statement `assert target.getBalance() == old(target.getBalance()) + amount;` in der Nachbedingung der Methode `deposit(int amount)` fordert, dass der alte Kontostand, also der Kontostand vor der Ausführung der Methode, nach dem Ausführen der Methode um den Einzahlungsbetrag `amount` erhöht wurde.
- Das Java-Assert-Statement `assert target.getBalance() == old(target.getBalance()) - amount;` in der Nachbedingung der Methode `withdraw(int amount)` fordert, dass der alte Kontostand, also der Kontostand vor der Ausführung der Methode, nach dem Ausführen der Methode um den Auszahlungsbetrag `amount` vermindert wurde.

Das gleiche gilt für die Vorbedingungen (Pre-Conditions). Sie weisen eine hohe Ähnlichkeit mit den Forderungen der negativen Testfälle auf:

- Das Java-Assert-Statement `assert amount > 0;` in der Vorbedingung der Methode `deposit(int amount)` fordert, dass der Einzahlungsbetrag `amount` größer 0 sein muss. Ist er kleiner oder gleich 0, wird der Methodenaufruf mit einem `AssertionError` abgewiesen.
- Das Java-Assert-Statement `assert amount > 0;` in der Vorbedingung der Methode `withdraw(int amount)` fordert, dass der Auszahlungsbetrag `amount` größer 0 sein muss. Ist er kleiner oder gleich 0, wird der Methodenaufruf mit einem `AssertionError` abgewiesen.
- Das Java-Assert-Statement `assert target.getBalance() >= amount;` in der Vorbedingung der Methode `withdraw(int amount)` fordert, dass der aktuelle Kontostand größer oder gleich dem Auszahlungsbetrag `amount` sein muss. Ist er kleiner, wird der Methodenaufruf mit einem `AssertionError` abgewiesen.

Diese Ähnlichkeit ist kein Zufall. Softwareverträge kann man sich vorstellen als einen Vertrag zwischen Client und Server, der sich am Vorbild der Verträge zwischen Kunde und Lieferant orientiert. Nehmen wir das Beispiel eines Bäckers, der ein Brötchen für 50 Cent verkauft. Der Kunde muss, um ein Brötchen zu erhalten, 50 Cent bezahlen. Dafür hat der Bäcker die Obligation, als Gegenleistung für diese Zahlung ein Brötchen zu liefern. Vorbedingungen legen bei Softwareverträgen die Verpflichtung des Clients („Zahle 50 Cent“) fest, Post-Conditions regeln, was der Server dafür liefert (ein Brötchen).

Die in Form von Java-Assert-Statements formulierten Vertragsklauseln sind sowohl im JavaDoc als auch im Eclipse Java Editor beim Nutzen der durch Verträge geschützten Methoden in der Hoverbox sichtbar (s. Abb. 1).

```
1  package bank;
2
3  public class AccountDemo {
4
5     public static void main(String[] args) {
6        AccountSpec account = new Account();
7
8        account.deposit(5000);
9        account.withdraw(2000);
10       System.o
11       System.o
12
13    }
14 }
15
```

void bank.AccountSpec.withdraw(int amount)

Preconditions

amount > 0 : "amount > 0"
getBalance() >= amount : "balance >= amount"

Postconditions

getBalance() == old(getBalance()) - amount : "balance = old balance - amount"

Abb. 1: Anzeige der Pre- und Post-Conditions in Eclipse bei installiertem C4J-Plug-in

Diese Verträge spezifizieren die Schnittstellen vollständig, außerdem zeigen sie die Annahmen an, die bei der Implementierung dieser Schnittstelle in Form einer Klasse beziehungsweise Nutzung dieser Klasse beachtet werden müssen. Das vereinfacht die Implementierung deutlich, weil Vorbedingungen als erfüllt angenommen werden können.

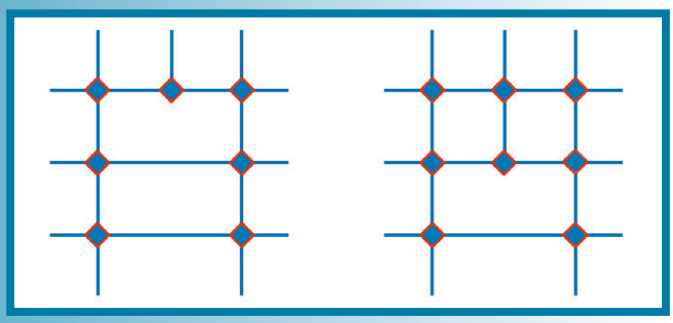

Abb. 2: Netz aus sieben Assert-Statements bei TDD und aus acht Assert-Statements mit Contracts

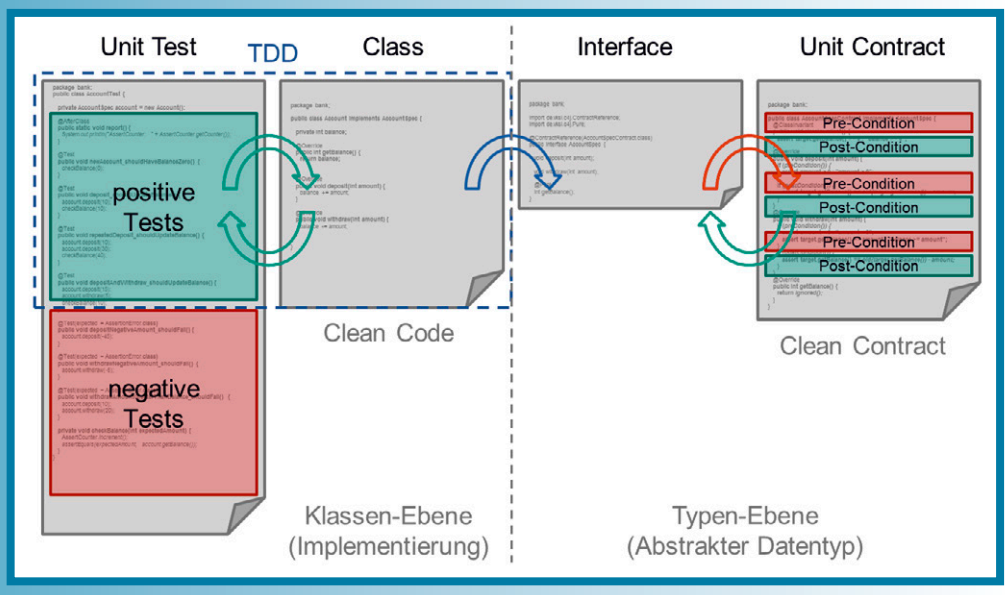

Abb. 3: Kombination von TDD auf der Klassen-Ebene und Contracts auf der Typen-Ebene

Sicherheitsnetz

Ein aus diesen Verträgen aufgebautes Sicherheitsnetz (s. Abb. 2) besteht zur Laufzeit aus acht Java-Assert-Statements, da bei jedem Aufruf einer Methode zuerst die Vorbedingungen überprüft werden, dann die Methode ausgeführt wird und danach die Nachbedingungen und die Klasseninvariante überprüft wird:

- `assert target.getBalance() >= 0;`
- `assert amount > 0;`
- `assert target.getBalance() == old(target.getBalance()) + amount;`
- `assert target.getBalance() >= 0;`
- `assert amount > 0;`
- `assert target.getBalance() >= amount;`
- `assert target.getBalance() == old(target.getBalance()) - amount;`
- `assert target.getBalance() >= 0;`

Der direkte Vergleich zeigt, dass sich die Qualität der durch die Statements erzeugten Sicherheitsnetze kaum unterscheidet, solange nur einer der beiden Ansätze zur Anwendung kommt. Was passiert jedoch, wenn beide Methoden miteinander kombiniert werden?

Um TDD mit Verträgen zu kombinieren (s. Abb. 3), geht der Entwickler in einer Weise vor, die sich in vier Schritte gliedern lässt:

- *Schritt 1 – TDD für alle positiven Tests durchführen:* Er oder sie schreibt einen positiven Test für die Zielklasse, der scheitert. Dann wird die Zielklasse so überarbeitet, dass der neue Test grün wird. Nun folgt das Refactoring der Ziel- und der Testklasse. Dieses Vorgehen wird so lange wiederholt, bis keine weiteren positiven Tests mehr hinzugefügt werden können, ohne Redundanzen zu erzeugen. Waren das bisher alles Arbeiten auf der Klassenebene, dann folgen jetzt die Aufgaben auf Typen-Ebene.
- *Schritt 2 – Interface für die „class under test" ableiten:* Im zweiten Schritt wird der Contracts-Zyklus vorbereitet: Dazu extrahiert der Entwickler aus der Zielklasse das Interface, um in den folgenden Schritten die zugehörige Vertragsklasse – der Unit-Contract – für das Interface zu erstellen.
- *Schritt 3 – Pre-Conditions aus negativen Tests schlussfolgern:* Er oder sie beginnt mit der Erstellung eines negativen Tests für die Zielklasse, der – keine Überraschung – scheitert. Dementsprechend überarbeitet der Entwickler die Pre-Conditions der Vertragsklasse des Ziel-Interfaces so, dass der neue Test grün wird. Jetzt steht das Refactoring von Vertrags- und Testklasse an. Alle drei Schritte werden so lange wiederholt, bis es keine weiteren, redundanzfreien negativen Tests mehr gibt. Im Grunde ist also das Vorgehen in Schritt 3 der exakte Spiegel dessen, was in Schritt 1 getan wurde – nur dass es jetzt die Pre-Conditions sind, die angepasst werden, um den Test grün laufen zu lassen.

- *Schritt 4 – Post-Conditions aus positiven Tests herleiten:* Jetzt bleibt nur noch übrig, ein abschließendes Refactoring gemäß der Clean Contract Prinzipien auf der Vertragsklasse auszuführen. Dazu gehört unter anderem zu beschreiben, wie sich der sichtbare Zustand der Zielklasse durch die Ausführung einer Methode verändert. Das entspricht oftmals einer Verallgemeinerung der durch die positiven Tests formulierten Forderungen. Für diese Beschreibung wird das Format der Post-Conditions gewählt.

Was bringt nun diese Kombination? Zunächst diverse Vorteile im Hinblick auf das Design:

- Die Schnittstellen sind aus Client-Sicht entwickelt und vollständig spezifiziert.
- Code ist mit einem feinmaschigen Sicherheitsnetz aus Unit-Tests und Unit-Contracts testbar.
- Als Nebeneffekt erhalten wir lose gekoppelte Module, die isoliert voneinander automatisiert getestet werden können.

Die Effizienz steigt, da

- unnötiger Code vermieden wird und
- Fehler an der Quelle erkannt werden, was das Debugging vereinfacht.

Für die Dokumentation wirkt sich positiv aus, dass

- die Spezifikation zur Laufzeit überprüfbar ist (Verträge als Teil des Produktivcodes) und
- die Vor- und Nachbedingungen (und Klassen-Invarianten) in der JavaDoc-Dokumentation automatisch festgehalten werden.

Gemeinsam bilden TDD und Contracts ein so feinmaschiges Sicherheitsnetz, dass fehlerhafte Zustände bereits beim automatisierten Testen entdeckt werden (s. Abb. 4).

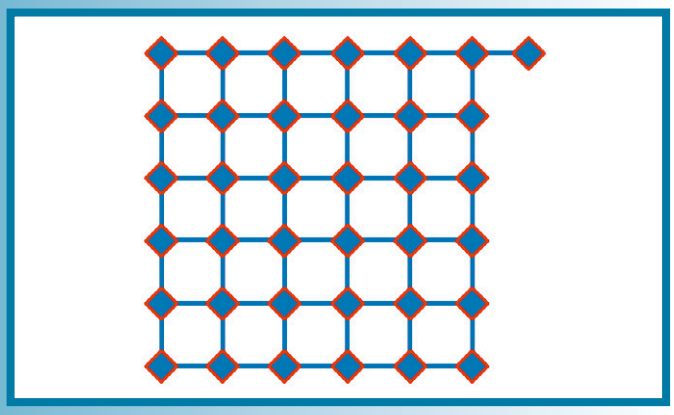

Abb. 4: Netz aus 34 Assert-Statements aus der Kombination von TDD und Contracts

Nett oder notwendig?

Damit bleibt eine andere Frage: Profitieren die Entwicklerinnen und Entwickler überhaupt signifikant, wenn das Sicherheitsnetz feinmaschiger wird, oder fällt dieser zusätzliche Schutz eher in die Kategorie „nett, aber nicht nötig“?

Natürlich kann man darüber diskutieren, was nötig ist und was nicht. Allerdings zeigen die Erfahrungen des Contract-Entwicklers Jonas Bergström [Berg13], dass falsche Annahmen die Hauptfehlerquelle im Projekt „PokerEngine“ bei bwin in Stockholm bildeten. 80 Prozent der bei der PokerEngine durch die Unit-Contracts aufgedeckten Fehler waren auf Verletzungen von Vorbedingungen (und Klasseninvarianten) zurückzuführen. Der Grund waren falsche beziehungsweise vergessene Annahmen darüber, wie eine Klasse genutzt werden sollte. Interessanterweise unterliefen diese Fehler sogar den Autoren der jeweiligen Klassen. Nach einigen Wochen Arbeit an anderen Modulen war den Entwicklern selbst nicht mehr präsent, auf Basis welcher Annahmen sie die Klassen erstellt hatten.

Gleichzeitig stellte Jonas Bergström fest, dass rund 20 Prozent der Klassen etwa 80 Prozent der benötigten Kernfunktionalität zur Verfügung stellten. Es reicht daher aus, jene 20 Prozent mit Softwareverträgen zu schützen.

TDD mit Contracts verwirklicht auch die Forderung der Definition der Objektorientierten Programmierung (OOP), die von Bertrand Meyer 1988 in seinem wegweisenden Werk „Objektorientierte Softwareentwicklung“ [Mey90] als

„Entwicklung von Softwaresystemen als strukturierte Sammlung von Implementierungen Abstrakter Datentypen“

definiert wurde. Der Abstrakte Datentyp `AccountSpec` wurde in obigem Beispiel mit den Mitteln von Java als Kombination von Interface und Unit-Contract formuliert und durch die Klasse `Account` implementiert (s. Abb. 3). Das ist objektorientierte Programmierung in Reinform.

Und diese Reinform empfiehlt sich nachdrücklich. Denn das Fazit aus dem Projekt „PokerEngine“ ist klar: Falsche beziehungsweise fehlende Annahmen darüber, wie eine Klasse genutzt werden sollte, sind eine der häufigsten Fehlerquellen.

Und einfach zu vermeiden. Wenn man TDD mit Contracts kombiniert.

Literatur und Links

[C4J] Contracts for Java,
`http://www.vksi.de/vksi-magazin-nr-7-design-by-contract/`
[Berg13] J. Bergström, Testing by Contract,
Vortrag auf der OOP 2013,
`http://www.sigs-datacom.de/oop2013/konferenz/conference-detail/testing-by-contract-eine-fallstudie-aus-schweden.html`
[Mey90] B. Meyer, Objektorientierte Softwareentwicklung,
Hanser Fachbuchverlag, 1990

Dr. Lars Alvincz ist seit 2010 Softwareentwickler bei der andrena objects ag und hat seitdem verschiedene agile Projekte als Entwickler und Coach unterstützt. Ein Kernthema seiner Arbeit ist die Etablierung agiler Testpraktiken, z. B. TDD und automatisierte Akzeptanztests. Auf der OOP 2015 wird er als Vortragender zu hören sein.
E-Mail: lars.alvincz@andrena.de

Hagen Buchwald ist studierter Wirtschaftsingenieur. Seit Oktober 2011 verstärkt er das Vorstandsteam der andrena objects ag. Seine Überzeugung ist, dass „Software Made in Germany“ zum Qualitätsbegriff werden kann, wenn es gelingt, die Stärken des deutschen Ingenieurdenkens auf das Software-Engineering zu übertragen.
E-Mail: hagen.buchwald@andrena.de