Design – jetzt auch für Tests

Verständliche und wartbare Tests schreiben

David Völkel, Stefan Rottegger

Wer kennt sie nicht? Ellenlange, schwer verständliche und nur mit viel Aufwand wartbare „Testskripte". Verantwortlich dafür sind Designprobleme im Testcode – als „technische Schulden" bremsen sie die Umsetzung neuer Anforderungen. Der Artikel diskutiert sinnvolle Designprinzipien für Testcode und illustriert anhand konkreter Beispiele, wie sich Les- und Wartbarkeit von Tests verbessern lassen.

Viele Teams stellen für Testcode geringere Anforderungen an das Design als für Produktionscode. Doch das rächt sich: Probleme im Testdesign zeigen sich in unverständlichen und schwer wartbaren Tests. Auch für Testcode gilt das Prinzip der „Technischen Schulden" (*Technical Debt*, [C2-Tec]). Neue Anforderungen umzusetzen, dauert immer länger. Bei Veränderungen agil bleiben? Von wegen!

Doch warum ist dabei die Lesbarkeit von Code so wichtig? Untersuchungen zeigen, dass ein Entwickler bis zu zehnmal mehr Zeit damit verbringt, bestehenden Code zu lesen als neuen zu schreiben [Mar08]. Beim Schreiben ist die Lesbarkeit noch zweitrangig. Soll der Entwickler aber später eine Änderung umsetzen, muss er mindestens die bestehenden Codeteile verstehen, an die der neue Code andocken soll. Entscheidend wird die Lesbarkeit also erst zum Zeitpunkt der Wartung. Im Testcode gilt das für folgende Fälle:

- bei Änderungen an einem Test, die eine geänderte Anforderung reflektieren,
- beim Erstellen eines neuen Testfalls, der einen im Betrieb entdeckten Fehler reproduzieren soll, und
- bei Änderungen am Produktivcode, die einen bestehenden Test fehlschlagen lassen.

In vielen Punkten decken sich die Anforderungen an das Testcodedesign mit denen des Produktionscodes. Testcode weist aber auch einige Besonderheiten auf, die der Artikel unter die Lupe nimmt. Insbesondere untersucht er, was Semantik, Abstraktion und Redundanz für das Testcodedesign bedeuten, und bietet Verbesserungsvorschläge in Form konkreter Refaktorierungen (*Refactorings*, [Fow99]), die zu les- und wartbarerem Testcode führen.

Testphasen

Ein Test gliedert sich in vier Phasen (*Four Phase Test*, [Mes07]):

- *Setup:* Erzeugung des Testlings (zu testendes Objekt, *Fixture*) und optional auch der Eingabedaten.
- *Execute:* Zu testender Aufruf auf dem Testling.
- *Verify:* Prüfung des Ergebnisses (*Asserts*), ggf. Fehlermeldung bei Abweichung vom erwarteten Ergebnis.
- *Tear Down:* Wiederherstellung des Zustands vor der Testausführung.

Einmal Semantik bitte!

Wer schon einmal versucht hat, das Ergebnis eines *Disassemblers* nachzuvollziehen, wird bestätigen können: Der Grad an Semantik im Code bestimmt entscheidend seine Verständlichkeit. Ein beliebtes Mittel, diesen Anteil zu erhöhen, besteht darin, den Code durch Kommentare mit Semantik anzureichern:

```
// counter of processed products
int cnt = 0;
```

Semantik im Kommentar ist zwar besser als gar keine Semantik, sollte aber immer kritisch hinterfragt werden [Mar08]. Meist ist es sinnvoller, einen Kommentar zu streichen und seine Bedeutung direkt in verbesserte Namen zu investieren:

```
int processedProductsCount = 0;
```

Was sind also gute Namen für Variablen, Konstanten, Methoden und Klassen? Code dokumentiert das aktuelle Verhalten des Systems und dient so immer auch als Kommunikationsmittel zwischen dem Codeautor und dem Entwickler, der später Änderungen daran durchführen soll. Höchste Priorität hat daher die Minimierung potenzieller Missverständnisse.

Dabei hilft es, eine projektweit einheitliche Sprache (*Ubiquitous Language*, [Eva03]) zu etablieren, die nicht nur die Entwickler sprechen, sondern alle Projektbeteiligten inklusive Kunde und Tester. Sie besteht aus den Begriffen der Problemdomäne. Werden exakt diese Begriffe auch im Code verwendet, entfällt der sonst notwendige gedankliche Schritt, die Sprache des Problems auf die des Codes abzubilden.

Doch nun zu den Besonderheiten von Testcode: Im Produktivcode überwiegt eine algorithmische Sprache, ein Test hingegen arbeitet eher narrativ. Anhand eines Beispiels erzählt er die „Geschichte" einer typischen Verwendung des Testlings. Der beste Name für eine Testmethode besteht daher in einer passenden Überschrift, die diese Geschichte auf den Punkt bringt. Sie reduziert sie auf ihren wesentlichen Kern und abstrahiert von irrelevanten Details. So drückt z. B. `testOrderWithoutPersonsLastName()` im Gegensatz zu einem unspezifischeren `testOrder()` oder dem beliebten `test1()` aus, dass die Testmethode genau den Testfall enthält, bei dem bei einer Bestellung der Nachname weggelassen wurde. Auf der nächsthöheren Ebene drückt ein guter Name für eine Testklasse die konzeptuelle Gemeinsamkeit der enthaltenen Testmethoden aus, und analog spiegelt der Name einer Testsuite möglichst den übergreifenden

Aspekt der enthaltenen Testklassen wider. Gute Namen helfen somit

- bei der Frage, wo im bestehenden Code ein neuer Testfall eingefügt werden soll,
- beim schnelleren Verständnis einer Testmethode und
- beim Lesen des Testreports zum Zwecke der Ursachenforschung nach einem fehlgeschlagenen *Build* auf dem *Continuous Integration System*.

Diagnose negativ

Im Idealfall reicht dem Entwickler bereits der Testreport aus, um die Fehlerursache zu diagnostizieren. Dazu muss allerdings die Kombination aus Testname (Klasse und Methode) und Fehlermeldung aussagekräftig genug sein. Besonders wenn gleich mehrere Tests fehlgeschlagen sind, möchte niemand sämtliche rot markierten Testmethoden durchstöbern müssen.

Da die Fehlermeldungen ganz besonders zum Verständnis beitragen [Fre09], hilft es, die Prüfmeldungen in der Phase *Verify* mit Semantik anzureichern. Quasi im Vorbeigehen verbessert das auch noch die Lesbarkeit des prüfenden *Assertion*-Codes. Als „Brot und Butter"-Variante bietet *JUnit* [JUni] an, mit dem ersten optionalen String-Parameter der `assert*()`-Methoden zu beschreiben, was die Prüfung bedeutet, z. B.:

```
assertFalse("Button should be enabled", button.isDisabled());
```

Ein deaktivierter Button liefert somit:

```
"Button should be enabled"
```

Seit Version 4.4 wartet *JUnit* mit einem noch leistungsstärkeren Hilfsmittel auf: die sogenannten *„Hamcrest Matcher"* [Ham]. Kombiniert mit der `assertThat()`-Methode ermöglichen sie komplexere, komponierbare Bedingungen und ein hohes Maß an Ausdrucksstärke nicht nur im Code, sondern auch in den Fehlermeldungen. So liefert

```
assertThat("final grades", gradeList, everyItem(lessThanOrEqualTo(2)));
```

als Fehlermeldung für die Liste `[1, 2, 3]`:

```
final grades
Expected: every item is a value less than or equal to <2>
  got: <[1, 2, 3]>
```

Teile und fokussiere

Das *Single Responsibility Principle* (SRP, [Mar08]) fordert, eine Klasse auf eine einzige Verantwortlichkeit zu beschränken. Ziel ist eine klar fokussierte Klasse. Für eine Testklasse bedeutet das, dass ihre Testmethoden konzeptuell zusammen gehören sollten. Dieses Prinzip spiegelt sich meist auch in einer starken Kohäsion der Testmethoden wider: Sie verwenden z. B. gemeinsame Hilfsmethoden oder die gleiche *Test Fixture* (Testling, Testdaten), die in einer `setUp()`-Methode initialisiert und in Member-Variablen gespeichert wird.

Auf Ebene der Testmethode ähnelt dem SRP das Prinzip *Single Concept per Test* [Mar08]. Demnach sollte eine Testmethode nur ein Konzept auf einmal testen und damit nur eine *Execute*-Phase enthalten. Dagegen verstößt die gängige Praxis, innerhalb einer Testmethode die Kette *Setup-Execute-Verify* mehrfach leicht variiert aneinander zu hängen. Hier eine – verglichen mit der Realität – recht harmlose Ausprägung:

```
@Test public void testAddGrade() {
  assertFalse(gradeList.contains(2));

  gradeList.addGrade(2);
  assertTrue(gradeList.contains(2));
}
```

Der Entwickler kann stattdessen eine solche Methode in mehrere unabhängige Testmethoden aufsplitten:

```
@Test public void testEmptyListContainsNoTwo() {
  assertFalse(gradeList.contains(2));
}

@Test public void testListWithOneTwoContainsTwo() {
  gradeList.addGrade(2);
  assertTrue(gradeList.contains(2));
}
```

Das erfordert etwas mehr Aufwand, da die einzelnen Testmethoden nun treffende Namen benötigen und die zuvor gemeinsam genutzten Objekte eventuell durch Hilfsmethoden geteilt werden müssen. Diese gerne gescheute Mühe lohnt sich aber. Zunächst einmal sinkt die Abhängigkeit der einzelnen Tests untereinander: Schlägt z. B. ein *Assert* schon im ersten *Verify*-Block fehl, so werden ohne das Splitting die eigentlich noch folgenden *Verify*-Abschnitte gar nicht mehr ausgeführt, da die Testmethode abbricht. Die getrennten Tests liefern also mehr Informationen. Zudem lassen sie sich unabhängiger voneinander bearbeiten. Eine Änderung an einem Testblock beeinflusst nicht mehr ungewollt einen darauf folgenden Testblock innerhalb derselben Methode.

Ein Aufsplitten wie im genannten Beispiel schärft außerdem den Fokus der Tests und erleichtert damit die Wahl eines spezifischeren und dadurch semantisch reicheren Methodennamens. Zur Diagnose eines fehlgeschlagenen Tests muss keine ellenlange Geschichte der Testmethode mehr nachvollzogen werden. Stattdessen ist die Geschichte kurz und prägnant mit einem Testnamen, der sie auf den Punkt bringt.

Man kann aber auch auf eine ganz andere Weise gegen das Prinzip *Single Concept per Test* verstoßen: wenn die Testmethode zwar nur einen einzigen *Execute*-Abschnitt umfasst, der *Verify*-Block aber mittels mehrerer *Assertions* unterschiedliche Konzepte testet:

```
@Test public void testTwoGrades() {
  GradeList gradeList = new GradeList();
  gradeList.addGrade(1);
  gradeList.addGrade(2);
  assertEquals(new BigDecimal("1.5"), gradeList.average());
  assertTrue(gradeList.contains(1));
}
```

`testTwoGrades()` prüft in dem Beispiel sowohl die Notendurchschnittsbildung als auch die `contains()`-Methode in einer einzigen Testmethode. Dieser mangelnde Fokus spiegelt sich auch in einem schwammigen Methodennamen wider, der einzig die gemeinsam benötigte Notenliste benennt. Die Verschärfung *One Assert per Test* [Mar08] fordert daher, den *Verify*-Teil einer Methode auf ein einziges *Assert* zu beschränken. Doch auch wenn es taktisch gut ist, die Anzahl der *Asserts* gering zu halten, so ist ein blindes Befolgen der Regel umstritten. Die Regel lässt sich z. B. dahin gehend aufweichen, nur ein „logisches *Assert*" zu fordern. Konkret wären dann formal mehr als ein *Assert* pro Testmethode möglich, solange sie einen einzigen konzeptuellen Aspekt überprüfen. Auch kann ein Befolgen der Regel leicht dazu führen, dass sich die *Setup-* oder *Verify*-Codeblöcke verdoppeln. Apropos Verdoppelung ...

Redundanzen ade

Redundanzen im Code sind eine der klassischen Ursachen für Wartbarkeitsprobleme. Daher erklärt das Prinzip *Don't repeat yourself* (DRY, [Wiki]) den Kampf gegen die Codedublette zum Ziel. Doch was bedeutet DRY für Tests? Wenn man Tests als ausführbare Spezifikationen (*Executable Specification*, [Info]) der Anforderungen betrachtet, folgt daraus, dass in verschiedenen Tests nicht die gleichen Dinge geprüft werden sollten, da sonst der Aufwand für Wartung und Fehlerdiagnose steigt. Auf der nächst niedrigeren Ebene innerhalb von Testmethoden lässt sich zudem oft doppelter Code vermeiden, indem man mehrfach verwendeten Code in Hilfsmethoden extrahiert.

Doch leider konkurriert die Redundanzfreiheit auch mit anderen Zielen. Geteilte Hilfsmethoden erhöhen z. B. die Gefahr einer ungewollten Abhängigkeit der einzelnen Testmethoden untereinander. Auch darf die Extraktion nicht dazu führen, dass die Verständlichkeit der durch den Test erzählten Geschichte darunter leidet. Es bleibt dem Testautor also nicht erspart, im Einzelfall zu entscheiden, wo genau die Grenze zwischen den gegenläufigen Zielen Redundanzfreiheit und Verständlichkeit bzw. Unabhängigkeit verläuft.

Böser Status

Leider kann im Testling gekapselter Status die Testbarkeit so weit verschlechtern, dass sich das Prinzip *Single Concept per Test* nicht mehr befolgen lässt. Es steigt die Zahl der notwendigen Schritte, um den Testling erst einmal in den Zustand zu bringen, in dem die zu testende Funktionalität überhaupt ausgeführt werden kann. Dadurch muss der Test jeden einzelnen dieser Schritte hintereinander testen und verstößt somit genau gegen die oben beanstandete mehrfache Aneinanderreihung von Einzeltests. Hier muss sich der Entwickler kritisch fragen, ob es wirklich notwendig ist, den Status im Testling zu kapseln. Lässt sich das Design vielleicht so anpassen, dass ein Test den Status direkt setzen kann?

Ein Beispiel: Ein funktionaler Webtest soll in einem Webshop auf der Produktseite den Button *„In den Einkaufswagen"* testen. Speichert die Webapplikation die aktuelle Produkt-ID als internen Status in der Session auf dem Server, benötigt der Test zunächst einen ganzen Pfad an Klicks und Eingaben, noch ohne die eigentlich zu testende Funktionalität ausgeführt zu haben: Klick auf *„Suche Produkt"*-Link im Menü, Eingabe des Produktnamens, Klick auf *„Suche"*-Button und Klick auf das Produkt im Suchergebnis. Das alles nur, um die Produkt-ID in der Session zu setzen, damit die Produktseite (`/showProduct.jsp`) das zugehörige Produkt auch anzeigt. Sollen weitere Tests auf der Produktseite noch andere Funktionalitäten wie z. B. den Button *„Auf meinen Wunschzettel"* prüfen, müssen sie alle den gleichen Pfad dorthin wiederholen und multiplizieren damit das Problem.

Eine Auslagerung in eine Hilfsmethode kann zwar eine Codeduplizierung verhindern, schluckt aber trotzdem bei den ohnehin langsamen Webtests einiges an *Build*-Zeit. Zudem erschwert sie bei einem Fehler auf diesem Pfad die Diagnose, da in diesem Fall gleich sämtliche Tests fehlschlagen. Vorausgesetzt, das verwendete Webframework spielt mit, besteht eine Lösung für das Problem darin, den Zustand von der Session auf den Client zu verlagern und die URL entsprechend zu parametrisieren: `/showProduct.jsp?productId=123`. Damit können die Tests selbst direkt den benötigten Initialzustand herstellen und ersparen sich so den langen Pfad.

Abstraktion

Gerard Meszaros [Mes07] weist auf eine weitere Gefahr für den Fokus einer Testmethode hin: *"When it is not important for something to be seen in the test method, it is important that it not be seen in the test method!"*

Der Testautor soll sich also in einer Testmethode auf das Wesentliche beschränken und zur Ausführung notwendige aber für die Geschichte des Tests irrelevante Details aus der Testmethode herausziehen, um sie wegzuabstrahieren. Die durch Tastenkürzel sehr effektiven Refaktorierungen der Entwicklungsumgebungen unterstützen den Testautor seit Längerem dabei, Methoden, Variablen, Konstanten und sogar Klassen toolgestützt zu extrahieren. In einer Testklasse findet sich viel lohnende Beute für die Extrahiersafari: Werte primitiver Typen mit unklarer Bedeutung (*Magic Numbers*, [C2-M]) etwa lassen sich durch ausdrucksstark benannte Konstanten ersetzen. Im Abschnitt *Setup* warten vor allem umfangreichere Blöcke der Testdatenerzeugung darauf, in eigene `create*()`-Hilfsmethoden ausgelagert zu werden.

Ähnliches Potenzial bieten in der *Verify*-Phase analog die *Asserts*. Doch Vorsicht! Dies kann darauf hindeuten, dass *One Assert per Test* verletzt wurde und damit der Fokus zu schwach ist. Ein Beispiel für eine komplexere Erzeugung eines Testlings:

```
// setup: create order
Customer customer = new Customer("John", "Doe", new Date(1960, 1, 1));
Product product = new Product("6,40", "Java Spektrum", "Magazin für ...");
Item item = new Item(2, product);
Order order = new Order(customer, asList(item));

// execute and verify
assertEquals("12,80", order.sum());
```

Es fällt auf, dass die Konstruktoren einige Parameter wie das `Customer`-Objekt und die Produktbezeichnungen zwingend vorschreiben, die für die Geschichte des Tests aber völlig irrelevant sind. Das Extrahieren und damit Verbergen dieser Details in einer Hilfsmethode kann hier den Fokus des Tests schärfen. Eine Steilvorlage für die Namensgebung bietet dem geneigten Refaktorierer der Kommentar, der ohnehin bereits eine Überschrift für den *Setup*-Codeblock bildet:

```
Order order = createOrder();
assertEquals("12,80", order.sum());
```

Jetzt ist der Test leider nicht mehr verständlich, da der Geschichte ein entscheidender Teil fehlt: Die Bestellung enthält zweimal ein Produkt zum Preis von „6,40". Meszaros' Regel scheint also auch umgekehrt zu gelten: Das Weglassen wesentlicher Informationen verschlechtert die Verständlichkeit des Tests genauso wie ein Zuviel an Irrelevanz. Die goldene Mitte lautet:

```
Order order = createOrderWith2ProductsOfPrice6_40();
assertEquals("12,80", order.sum());
```

Eine Umbenennung hat hier schon gereicht. Durch den neuen Namen wird die Geschichte des Tests um das entscheidende Puzzlestück ergänzt und damit plötzlich schlüssig: „2 x 6,40 = 12,80". Der Name enthält nur noch das für den Test Wesentliche der in der Methode enthaltenen Anweisungen und abstrahiert die irrelevanten Details weg. Auf diese Weise kann nach und nach eine domänenspezifische Sprache für den Testcode entstehen (*Test DSL*, [Mar08]). Tests gewinnen an Verständlichkeit, wenn sie knapper und auf einem höheren Abstraktionsniveau formuliert werden als die geschwätzigen „low-level Testskripte", die verschiedene Abstraktionsebenen mischen. Wem hier durch eine hohe Variabilität in den benötigten Testdaten

die Anzahl der `create*()`-Hilfsmethoden explodiert, der sei auf das Muster *Test Data Builder* [C2-Tes] verwiesen.

Refaktorieren, aber richtig!

Bei Testcode besteht immer das Risiko einer falschen Gewissheit durch *False Positives*: Der Test läuft zwar erfolgreich durch, hat aber nichts oder das Falsche geprüft. Um das zu verhindern, schlägt die testgetriebene Entwicklung mit dem *„Test First"*-Ansatz vor [Fre09], den Test erst einmal fehlschlagen zu lassen. Wenn der Test nach korrekter Implementierung dann von Rot auf Grün wechselt, kann man sicher sein, dass dies tatsächlich der Implementierung zu verdanken ist.

Des Weiteren erhöhen bedingte Anweisungen die Wahrscheinlichkeit für *False Positives*. `if`/`else`- und `try`/`catch`-Blöcke, aber auch Schleifen gilt es daher im Testcode so weit wie irgendwie möglich zu vermeiden. Sie führen bei Fehlern leicht dazu, dass die entscheidende Prüfung gar nicht erst ausgeführt wird:

```
for (int i = 0; i > gradeList.size(); i++)
  assertTrue(gradeList.get(i) <= 2);
```

Da die Schleifenbedingung versehentlich statt einem `<` ein `>` verwendet, führt der Test die *Assertion* nie aus und läuft fälschlicherweise stets erfolgreich durch.

Nach einer Refaktorierung von Produktionscode stellen die Tests sicher, dass sich der Code funktional nicht verändert hat. Dieser doppelte Boden fehlt beim Refaktorieren von Testcode zum Teil, da auch *False Positives* möglich sind. Ganz sicher geht nur, wer auch hier etwas umständlich das *„Test First"*-Spiel wiederholt: kurz den getesteten Produktivcode so modifizieren, dass der Test fehlschlägt. Dann die Änderung wieder zurücksetzen und sicherstellen, dass der Test wieder auf Grün springt. Auch wenn die Gefahr von *False Positives* im Testcode glücklicherweise gerade nicht von den besonders relevanten, IDE-gestützten Refaktorierungen wie Extrahieren oder Umbenennen ausgeht, sollte sie der Entwickler immer im Hinterkopf behalten.

Viele Erfahrungen aus der Refaktorierung von Produktionscode lassen sich aber auch auf Tests übertragen. So bietet es sich bei einer bestehenden, maroden Testcodebasis an, Refaktorierungen nur *lazy* (also bei tatsächlich vorhandenem Bedarf) durchzuführen; beispielsweise wenn der Wartungsentwickler ohnehin gerade Stellen anfassen muss, die nach Überarbeitung schreien. Damit entfällt unnötiger Aufwand für Tests, die sich vielleicht ohnehin nie wieder ändern. Zudem fällt die Arbeit kontinuierlich parallel zur Wartung an und nicht in einem einzigen, praktisch nicht zu bewältigenden Stück.

Um ein *over-engineering* zu vermeiden, schlägt der *Test First*-Ansatz außerdem vor, mit dem Schreiben eines Tests zu beginnen und nicht mit dem Design. So wird dieses so einfach wie möglich, aber auch so komplex wie nötig. Das Vorgehen lohnt sich auch für den Testcode: zuerst den Test herunterschreiben, bis er läuft, und erst hinterher durch Refaktorierung den Fokus des Tests verbessern.

Fazit

Der Artikel hat einige Lösungen für Probleme im Testcode vorgestellt, die den Designzielen Lesbarkeit und Wartbarkeit im Wege stehen, sowie Unterschiede zum Refaktorieren von Produktivcode herausgearbeitet. Um die Ziele zu erreichen, hilft es, Codedubletten zu eliminieren, den Code mit Semantik anzureichern, den Fokus zu schärfen und Abstraktionen zu bilden im Sinne einer *Test DSL*.

Je höher der technische Schuldenberg einer Software, desto mehr Zeit verschlingt jede weitere Änderung. Dasselbe gilt auch für Tests. Wer agil bleiben will, darf mit gutem Design nicht beim Produktionscode haltmachen, sondern muss dem Testcode die gleiche Sorgfalt entgegen bringen! Gewappnet mit den Tipps aus diesem Artikel kann der Leser bei den nächsten Wartungsarbeiten am Testcode diese Devise in die Tat umsetzen. Viel Spaß dabei!

Literatur und Links

[C2-M] Magic Number, `http://c2.com/cgi/wiki?MagicNumber`
[C2-Tec] Technical Debt, `http://c2.com/cgi/wiki?TechnicalDebt`
[C2-Tes] Test Data Builder, `http://c2.com/cgi/wiki?TestDataBuilder`
[Eva03] E. J. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003
[Fow99] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999
[Fre09] St. Freeman, N. Pryce, Growing Object-Oriented Software, Addison-Wesley, 2009
[Ham] Hamcrest, `http://code.google.com/p/hamcrest`
[Info] Virtual Panel: Specification by Example, Executable Specifications, Scenarios and Feature Injection, `http://www.infoq.com/articles/virtual-panel-bdd`
[JUni] JUnit, `http://www.junit.org`
[Mar08] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall International, 2008
[Mes07] G. Meszaros, xUnit Test Patterns, Addison-Wesley, 2007
[Wiki] Don't repeat yourself, `http://de.wikipedia.org/wiki/Don't_repeat_yourself`

David Völkel unterstützt seit neun Jahren als Berater für Software-Engineering-Themen die Kunden der 4A Solutions GmbH bei der Umsetzung von Java Enterprise Solutions. Sein besonderes Interesse gilt dabei dem Softwaredesign im Kontext Agiler und Testgetriebener Softwareentwicklung.
E-Mail: david.voelkel@4a-solutions.de

Stefan Rottegger ist seit zwanzig Jahren in der Softwareentwicklung tätig und seit über zehn Jahren Geschäftsführender Gesellschafter der 4A Solutions GmbH. Er hat somit bereits eine Vielzahl von Technologien kennengelernt und eingesetzt. Ob C, C++, Java, ... – der Schlüssel für den Projekterfolg liegt seiner Meinung nach in strukturierten Anforderungen, einem Mindestmaß an Agilität und dem Testen von Anfang an.
E-Mail: stefan.rottegger@4a-solutions.de