Ausgabe 04/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 04

Java aktuell

iJUG Verbund
www.ijug.eu

Java 14
Einblick in die neuen Features

Testing
Neue Impulse für das effektivere Testen von Software

Testautomatisierung
Sind automatisierte Tests eine Illusion?

Grüne Inseln im Schlamm – Mit Side-by-Side Refactoring allzeit lieferbereit, Teil 1

Georg Berky, Valtech Mobility

Dieser Artikel zeigt anhand von Emily Baches „Gilded Rose"-Kata [1] eine fortgeschrittene Art des Refactoring von Legacy Code, bei der der Code jederzeit lieferbar bleibt, auch wenn das Refactoring unterbrochen werden muss. Dadurch wird der Code auch schrittweise wieder wartbar gemacht. Dies ist Teil 1 des vollständigen Artikels.

Legacy Code ist eine große Herausforderung für Entwickler. Das ursprüngliche Team, das ihn entwickelt hat, ist nicht mehr da. Der Code hat nur schlechte oder keine Dokumentation und schon gar keine Tests. Ändert man an der einen Stelle etwas im Code, tritt in einer komplett unerwarteten Klasse ein seltsamer Fehler auf. Man spricht von Brownfield oder Legacy Code – als würde man tief im Schlamm eines Feldes waten, wo jeder Schritt viel zu viel Kraft kostet. Es braucht Zeit, Nerven und Ressourcen, solchen Code wieder in einen wartbaren Zustand zu bringen.

Vor Kurzem bin ich auf Twitter über folgendes Zitat von Michael Feathers gestolpert: „Key lesson is, you can always (read: most of the time) do greenfield in a legacy codebase. [...]"

Dieses Zitat habe ich zum Anlass genommen, nochmal den Dschungel an Literatur und Videos zum Thema Legacy Code zu durchsuchen, und ein paar Techniken mitgebracht, die die IDE nicht automatisch ausführen kann, aber dafür an schwierigen Stellen oft umso wirkungsvoller sind.

Sich um Software kümmern

Software lebt zusammen mit den Menschen, die sie pflegen. Der Code beeinflusst die Menschen. Die Menschen beeinflussen den Code. Obwohl Code komplett substanzlos ist, gehen wir froh nach Hause, wenn wir ihm etwas Gutes getan haben, und haben schlechtere Laune, wenn wir ihn auch nach mehreren Anläufen nicht verstehen konnten oder vergeblich versucht haben, eine schwierige Stelle zu verschönern. Code, der nicht mehr gepflegt wird, wird langsam unbenutzbar. Bibliotheken veralten, man findet Security-Schwachstellen, die Version der Programmiersprache steigt. Code braucht also konstante Pflege und Aufmerksamkeit, nur um benutzbar zu bleiben – eine Aufgabe, die deswegen zur regulären Arbeit der Entwickler gehören sollte. Ein klassischer Werkvertrag, der nur Entwicklung und Abnahme des Codes beinhaltet, greift also oft zu kurz und sorgt langfristig für Probleme beim Kunden. Das sollte bei Auftragsverhandlungen besprochen und berücksichtigt werden.

Idealerweise sollte der Benutzer des Codes auch nicht direkt bemerken, dass wir Änderungen vornehmen. Das System sollte weiterhin wie gehabt funktionieren. Im Folgenden möchte ich einen Ansatz für ein von mir oft benutztes Refactoring zeigen, bei dem der Code jederzeit release- und lieferbar bleibt, auch wenn das Refactoring noch nicht abgeschlossen ist. So pflanzen wir kleine grüne Inseln im Brownfield, die im Laufe der Zeit größer werden und so den Code schrittweise verbessern.

Als Simulation eines Legacy-Systems möchte ich das „Gilded Rose Kata" *[1]* benutzen.

Code, den wir nicht ändern können

Manchmal haben wir Code vor uns liegen, den wir nicht ändern können. Das könnte die Bibliothek eines Drittanbieters sein, die wir nur binär, aber nicht im Quelltext vorliegen haben. Vielleicht ist der Anbieter auch längst vom Markt verschwunden oder passt die Bibliothek aus anderen Gründen nicht mehr an.

Auch in der „Gilded Rose" findet sich ein Beispiel dafür: „However, do not alter the `Item` class or property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn't believe in shared code ownership." Meine Übersetzung davon: „Die Klasse `Item` oder ihre Properties darfst du trotzdem nicht verändern. Sie gehört dem Goblin in der Ecke. Er glaubt nicht an geteilte Verantwortung für den Code und haut dich sonst sofort wie ein Berserker auf einen Schlag um."

Die Klasse `Item` hat ein Design-Problem: Alle Felder sind `public`, jede Klasse kann darin herumschreiben und die Logik, die mit Daten aus `Item` arbeitet, liegt natürlich nicht in `Item`. Der Code Smell dazu heißt „Feature Envy" *[2]* beziehungsweise „Anemic Domain Model" *[3]* – je nachdem, von welcher Seite aus man darauf blickt.

Wenn wir also nicht wollen, dass uns der Goblin in der Ecke umhaut, weil wir es gewagt haben, seinen Code anzufassen, dann können wir `Item` nicht ändern. Wenn wir etwas nicht reparieren können, können wir es vielleicht codetechnisch vergolden.

Im Legacy Code kann man dafür die Technik „Wrap Class" *[4]* aus „Working Effectively with Legacy Code" (WELC) verwenden.

Das Sicherheitsnetz beim Refactoring

Vor jedem Refactoring baue ich jedoch ein Sicherheitsnetz durch Characterization-Tests auf. Diese lasse ich nach jedem kleinen Refactoring-Schritt laufen, besonders, wenn ich unsicher bin, ob ich gerade etwas kaputt gemacht habe. Je unsicherer ich bin, desto kleiner wähle ich meine Schritte. Im Idealfall komme ich durch ein einziges „Undo" wieder zum letzten grünen Stand des Codes zurück. Nach einigen Zwischenschritten stage ich meine Änderungen mit `git add` oder committe lokal mit `git commit`.

Beim Schreiben der Tests hilft mir die Funktion „run with coverage" meiner IDE, die mir zeigt, ob ich beim Schreiben der Tests Stellen im Code übersehen habe. Hundert Prozent Coverage garantieren nicht, dass ich nichts vergessen habe, aber ein offensichtliches Loch ist trotzdem der Beweis, dass noch ein Test für diesen Teil des Codes fehlt.

Der gebotenen Kürze wegen kann ich hier nicht im Detail auf Characterization-Tests eingehen. Im Prinzip funktionieren sie so, dass man, statt direkt Erwartungen an den Code zu formulieren, die wirklich gelieferten Ergebnisse eines Funktionsaufrufs ansieht, verifiziert und zum Schluss in Tests festhält, die das Verhalten dokumentieren.

In echten Systemen muss man oft auch noch Dependency-Breaker einsetzen, um beispielsweise den Zugriff auf Datenbank oder Dateien durch Stubs zu ersetzen.

Details finden sich in Michael Feathers' Blog *[4]* und in WELC *[5]*.

Die Bruchstellen vergolden

Zuerst wende ich mich also der `Item`-Klasse zu. Ich kann sie selbst nicht ändern, aber ich kann ändern, wer sie benutzt. Die Klasse enthält nur Daten, also kann ich sie auch zu einem nicht ganz perfekten Field einer schöner designten Model-Klasse machen, die zum Wrapper um `Item` wird. Nicht ganz perfekt ist ok. Code ist organisch. Organisch ist nicht perfekt.

Also fange ich mit dem Wrapper an *(siehe Listing 1)*. Das ist noch nicht viel, aber wenn wir fertig sind, soll `GildedItem` die einzige Klasse sein, die jemals eine Instanz von `Item` manipuliert.

Wir wissen jetzt, was wir mit `Item` machen wollen, aber noch nicht, wie wir diese Änderung am Code so sicher machen, dass wir jederzeit lieferfähig sind, also so, dass wir jederzeit bei grünen Tests committen, pushen, releasen und liefern können. Die Zeit, in der die Tests dabei rot bleiben, soll so kurz wie möglich sein. Das bedeutet, dass wir kleine, sichere Schritte machen müssen. Damit das möglich wird, müssen wir ein paar Schritte in die Gegenrichtung machen: Der Weg beim „Preparatory Refactoring" *(siehe Abbildung 1)* führt zuerst zurück, um dann an einem Punkt auf die befestigte Straße zu treffen, die uns letztendlich schneller ans Ziel bringt als der *direkte* Weg durch den Wald. Bis zu diesem Punkt macht man manchmal Schritte rückwärts, bevor man wirklich anfangen kann, sauber zu machen.

Mein erster Schritt rückwärts: Ich benutze `Item` und `GildedItem` nebeneinander *(siehe Listing 2)*. `GildedRose` hat jetzt ein zweites Field, ein Array mit Instanzen von `GildedItem`. Es liegt auch noch neben dem alten Array mit `Item`. Dazu habe ich einen zweiten Konstruktor eingeführt, der ein Array von `GildedItem` als Parameter akzeptiert. Den bestehenden Konstruktor habe ich auch modifiziert. Um alles noch schlimmer zu machen, erstellen beide Konstruktoren zusätzliche Instanzen von `Item` oder `GildedItem`, die auch noch aufeinander *aliased* sind. Wenn ich ein `Item` ändere, ändere ich auch das dazugehörige `GildedItem`, das es wrappt. Unter normalen Umständen würden wir solche unsichtbaren Änderungen an gekapselten Objekten um jeden Preis vermeiden wollen. In diesem Fall ist es genau das, was wir wollen: `GildedRose` kann dadurch wie gehabt seine Instanzen von `Item` manipulieren, ohne dass ich sofort alle ändernden Stellen im Code umbauen muss. Gleichzeitig sieht `GildedItem` durch das *Aliasing* diese Änderungen. In der Gegenrichtung manipuliert auch `GildedItem` dieselben Instanzen und `GildedRose` sieht dessen Änderungen. Beide Arrays sind `private`. Die Unordnung hat also wenig Chancen, andere Klassen zu beeinflussen. Nach einer Änderung kann ich lokal committen, sobald die Tests wieder grün sind. So kann ich jederzeit den letzten grünen Stand wiederherstellen. Dank des Schrittes zurück kann ich während des Umbaus jetzt jederzeit liefern, solange meine Tests grün sind.

```
public class GildedItem {

    private final Item item;

    public GildedItem(Item item) {
        this.item = item;
    }

    Item getItem() {
        return item;
    }
}
```

Listing 1: Wrapper für Item

```
class GildedRose {

    private final GildedItem[] gildedItems;
    private final Item[] items;

    public GildedRose(Item[] items) {
        this.items = items;
        this.gildedItems = Arrays.stream(items)
                .map(GildedItem::new)
                .toArray(GildedItem[]::new);
    }

    public GildedRose(GildedItem[] items) {
        this.items = Arrays.stream(items)
                .map(GildedItem::getItem)
                .toArray(Item[]::new);
        this.gildedItems = items;
    }
}
```

Listing 2: Wrapper und Item Side-by-Side

Die Tests umziehen

Wo wir gerade bei Tests sind: Diese haben zu Beginn den Konstruktor benutzt, der ein Array von `Item` annimmt. Das habe ich so gelassen, bis ich mit meiner Änderung am Produktivcode fertig war. Ich versuche zu vermeiden, Produktiv- und Testcode gleichzeitig zu ändern, weil sie sich gegenseitig testen. Ich würde sonst das Sicherheitsnetz verlieren, das ich mir vorher durch die Characterization-Tests erar-

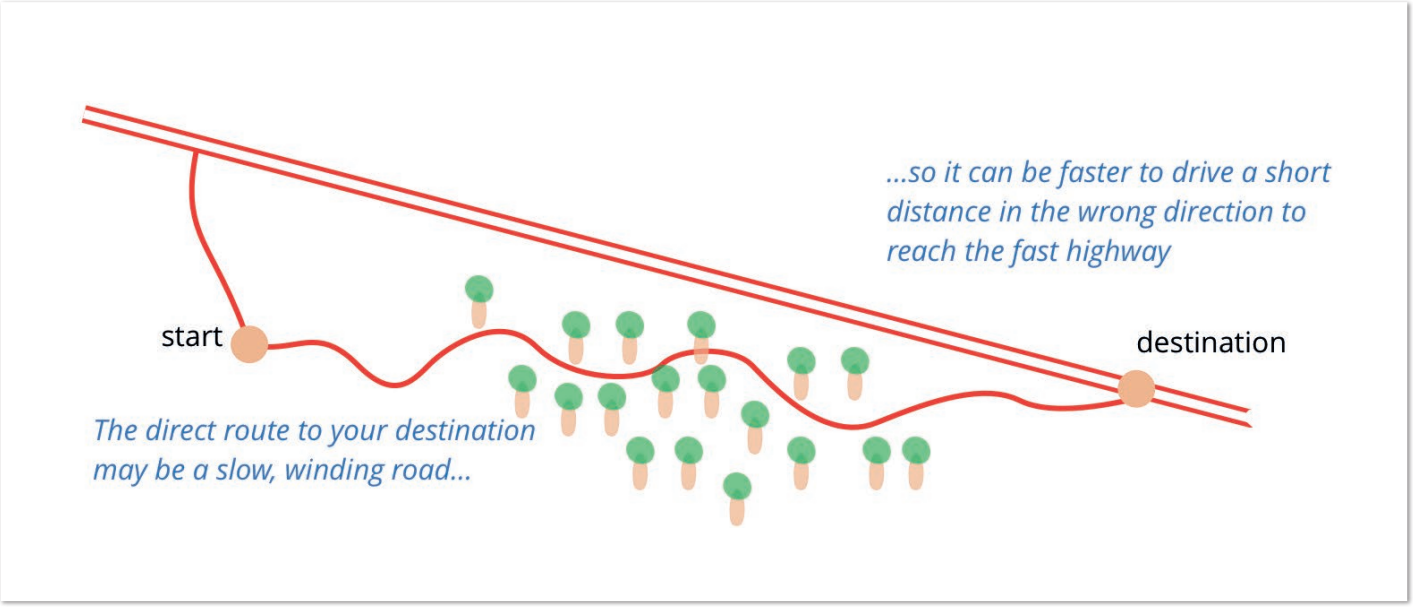


Abbildung 1: Preparatory Refactoring (© Jessica Kerr [6])

beitet habe. Nach dem Hinzufügen des neuen Konstruktors lasse ich meine Tests laufen und sehe, dass noch alles funktioniert. Ich habe durch das neue Array nichts kaputtgemacht.

Im nächsten Schritt kann ich jetzt die Tests ändern, damit sie den neuen Konstruktor benutzen *(siehe Listing 3* und *Listing 4)*. Jetzt verifiziert mein Produktionscode, dass sich die Semantik meiner Tests nicht geändert hat.

Protest?

„Aber das macht doch alles nur noch schlimmer!" – Ja, es ist noch nicht schöner geworden, aber wenn ich auf diese Weise arbeite, kann ich `Item` schrittweise aus `GildedRose` entfernen, ohne jemals zu viel auf einmal machen zu müssen und dabei nicht lieferfähig zu sein. Das ist der Weg, der zurückzuführen scheint, der uns aber zum Highway bringt. Die Alternative wäre, lange nicht lieferfähig zu sein, weil das Refactoring nicht fertig ist.

Auf dem Highway zum Ziel

Jetzt, wo wir `Item` und `GildedItem` an Ort und Stelle haben, können wir Funktionalität von `GildedRose` nach `GildedItem` verlagern. Dabei benutze ich einen leicht modifizierten Ansatz, der immer wieder bei vielen Refactorings funktioniert hat, wenn ich mit dem Smell Anemic Domain Model konfrontiert bin:

Normalerweise würde mein Ansatz so aussehen:

1. Extrahiere eine Methode, die die anämische Instanz als Parameter annimmt
2. Manipuliere das anämische Modell in der Methode
3. Benutze das *Move Method* Refactoring, um die Methode in die Klasse des anämischen Modells zu verschieben

Das anämische Modell wird durch die neue Methode weniger anämisch und bekommt eigene Funktionalität, die vorher außerhalb des Modells verstreut im Code lag.

```
private GildedRose createApp(Item item) {
  Item[] items = new Item[] { item };
  return new GildedRose(items);
}
```

Listing 3: Tests mit altem Konstruktor

```
private GildedRose createApp(Item item) {
GildedItem[] items = new GildedItem[] {
  new GildedItem(item)};
return new GildedRose(items);
}
```

Listing 4: Tests mit neuem Konstruktor

In unserem Beispiel dürfen wir `Item` aber nicht verändern. Darum kann ich nicht direkt Methoden dorthin verschieben und mich auch nicht auf die vollautomatische Version von *Move Method* verlassen. Stattdessen wende ich die händische Version des Refactoring an:

1. Erstelle eine Methode in der **neuen** Model-Klasse.
2. Kopiere den manipulierenden Code, der auf `Item` arbeitet, in die neue Methode (noch nicht kompiliert).
3. Ändere den Code so, dass er stattdessen das gekapselte Item manipuliert (kompiliert).
4. Ersetze den Aufruf des alten Codes durch den neuen.

In der Gilded Rose:

1. Code, den wir verschieben wollen *(siehe Listing 5)*
2. und 3. Methode in `GildedItem` anlegen *(siehe Listing 6)*
4. Aufrufer ändern *(siehe Listing 7)*

Jetzt lassen wir zur Sicherheit nochmal die Tests laufen und sehen, dass immer noch alles funktioniert. Mit `quality` machen wir analog weiter *(Listing 8)*.

```
if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
    items[i].sellIn = items[i].sellIn - 1;
}
```

Listing 5: Zu verschiebender Code im Aufrufer

```
public class GildedItem {

    private final Item item;

    public GildedItem(Item item) {
        this.item = item;
    }

    ...

    public void decreaseSellIn() {
        //nach 2.: items[i].sellIn = items[i].sellIn - 1;
        item.sellIn = item.sellIn - 1;

        //später: refactor to item.sellIn--;
    }
```

Listing 6: Verschobene Methode in GildedItem

```
if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {
    gildedItems[i].decreaseSellIn();
}
```

Listing 7: Aufrufer danach

```
//vorher:
items[i].quality = items[i].quality + 1;
//nachher:
gildedItems[i].increaseQuality();


//vorher
items[i].quality = items[i].quality - 1;
//nachher
gildedItems[i].decreaseQuality();
```

Listing 8: Refactoring von Quality

Immer noch auf dem Weg

Für unsere `Item`-Klasse gibt es natürlich noch mehr zu tun als das, was ich in diesem Artikel zeigen konnte. `GildedItem` könnte ein paar Prädikatsmethoden gut vertragen, mit denen wir die Conditionals ersetzen. Die Methoden `increaseQuality()` und `decreaseQuality()` könnten ihre Invarianten besser forcieren und etwa verhindern, dass `quality` jemals negativ oder größer als 50 wird. Wenn wir mit all dem fertig sind, können wir das Array `Items[]` aus der Klasse `GildedRose` entfernen und `GildedItem` ist der einzige Ort, wo `Item` noch verwendet wird und bleiben kann. Unser Code ist trotz des Umbaus jederzeit lieferbar, weil wir uns für paralleles Refactoring entschieden haben.

Mit der vorgestellten Arbeitsweise sind Refactorings kein Blocker für den Rest der Arbeit mehr. Man kann sie auch während der Arbeit anhalten, etwas Wichtigeres einschieben und später fortsetzen. Zusammen mit den Tests, die wir als Sicherheitsnetz hinzugefügt haben, gewinnen wir an Sicherheit und Flexibilität und reduzieren das Risiko für unser Projekt.

In der nächsten Ausgabe der Java aktuell finden Sie Teil 2 dieses Artikels.

Quellen

[1] Emily Bache (2011): Gilded Rose Kata.
https://github.com/emilybache/GildedRose-Refactoring-Kata

[2] C2 Wiki: Feature Envy Smell
http://wiki.c2.com/?FeatureEnvySmell

[3] Martin Fowler (2003): Anemic Domain Model
https://martinfowler.com/bliki/AnemicDomainModel.html

[4] Michael C. Feathers (2016): Characterization Tests
https://michaelfeathers.silvrback.com/characterization-testing

[5] Michael C. Feathers (2004): Working Effectively with Legact Code. Prentice Hall, Upper Saddle River, NJ, United States

[6] Jessica Kerr (2015): Preparatory Refactoring
https://martinfowler.com/articles/preparatory-refactoring-example.html

Georg Berky
Valtech Mobility
georg.berky@valtch-mobility.com

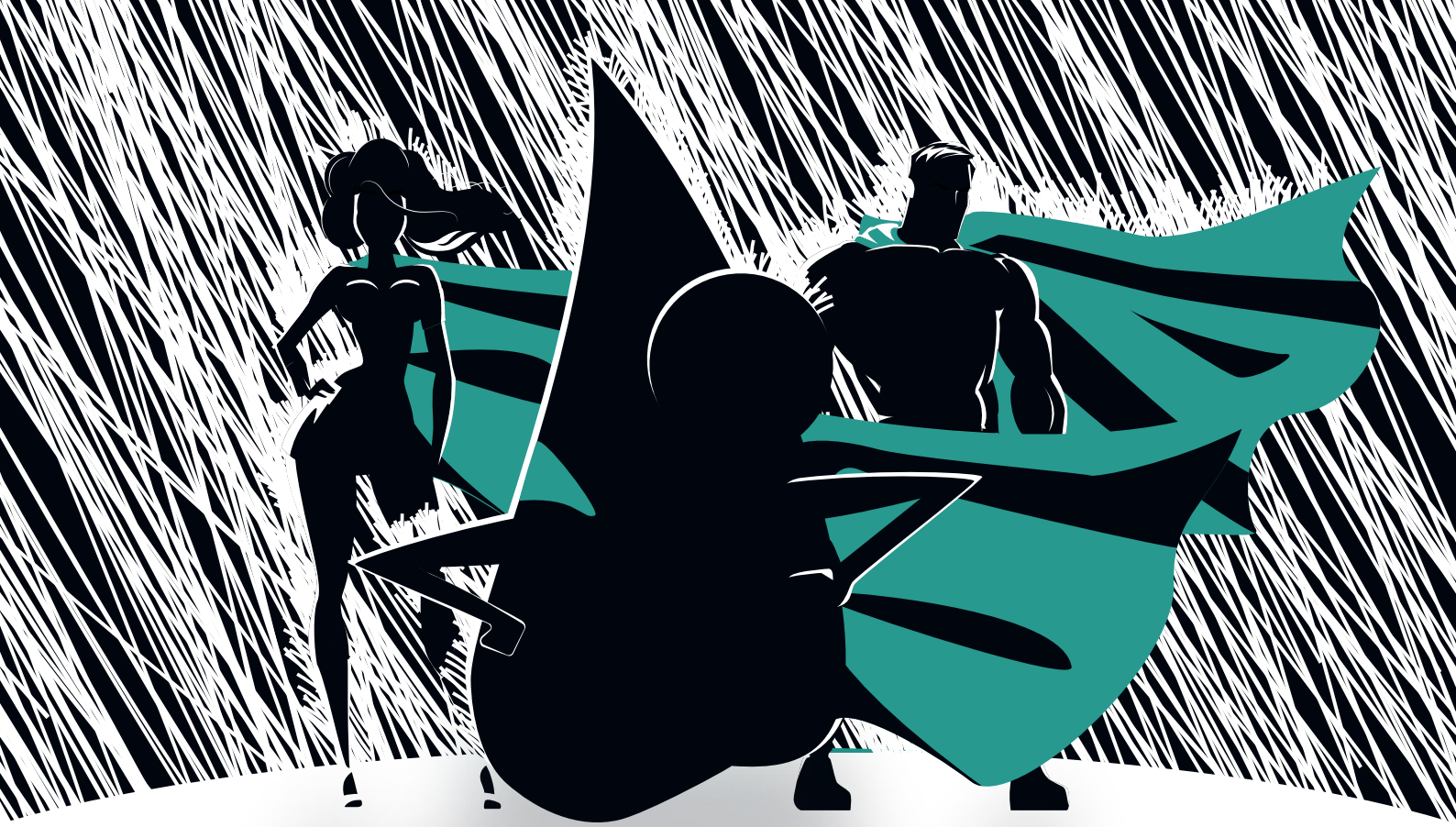
Georg entwickelt hauptberuflich Connected-Car-Dienste. Handwerk und Leidenschaft ist für ihn die Programmierung, meistens in JVM-Sprachen wie Java, Groovy oder Clojure. Zum Handwerk gehören für ihn auch Themen wie die Pflege von Legacy Code, Automatisierung von Builds und Deployments oder Agilität im Team. Seit einigen Jahren ist er Co-Organisator der Software-Craftsmanship-Communities im Ruhrgebiet und in Düsseldorf. Wenn er mal nicht programmiert, spielt er Trompete oder praktiziert Aikido. Er twittert als @georgberky.

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu



IHR SEID UNSERE JavaLand HEROES!

Zahlreiche Teilnehmer, Referenten und Sponsoren haben entschieden, die JavaLand in der Krise mit einem finanziellen Beitrag zu unterstützen.

Wir sind überwältigt von eurer Großzügigkeit.
Schön, dass wir auf eine starke Community zählen können!

Bisher konnten wir dadurch über 46.000 Euro sammeln.
Insgesamt müssen aufgrund des Veranstaltungsausfalls mehr als 200.000 Euro gedeckt werden.

JavaLand

Besucht unsere Hall of Fame: