Grüne Inseln im Schlamm – Mit Side-by-Side Refactoring allzeit lieferbereit, Teil 2

Georg Berky, Valtech Mobility

Dieser Artikel zeigt anhand von Emily Baches „Gilded Rose"-Kata eine fortgeschrittene Art des Refactoring von Legacy Code, mit der auch unter schwierigen Bedingungen neuer Code testgetrieben in komplizierte Legacy-Anwendungen integriert werden kann. Wir bleiben wie im vorigen Teil 1 dabei allzeit lieferbereit.

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 einer 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.

Im vorigen Teil habe ich das Refactoring „Wrap Class" gezeigt, mit dem man gut anämische Datenklassen, die man nicht verändern darf, verbessern und besser handhabbar machen kann. Auch dieses Mal habe ich eine weitere Technik mitgebracht, die die IDE nicht automatisch ausführen kann, die aber dafür an schwierigen Stellen oft umso wirkungsvoller ist.

Die Techniken sind so einsetzbar, dass man sie kleinschrittig umsetzen kann und der Code dabei ständig lieferbar bleibt. Ich benutze als Beispiel wieder die „Gilded Rose Legacy Code Kata" von Emily Bache aus dem letzten Teil *[1]*.

Wir fangen wieder am Ausgangspunkt der Kata an. Lediglich die Tests habe ich aus dem letzten Kapitel übernommen. Der Code dazu findet sich im Branch „sprouting" meines Git Repository. Im Branch „master" findet sich der Ausgangszustand nur mit den Tests *[2]*.

Sprouting

Sprouting kann man verwenden, wenn man in wenig Zeit ein neues Feature hinzufügen muss, ohne dabei auf testgetriebene Entwicklung zu verzichten. Sprouts kommen in mehreren Formen, „Sprout Method" und „Sprout Class", vor. Beide Varianten werden an einem Seam in den vorhandenen Code eingesetzt.

Seams

Das Seam-Modell kommt aus Michael Feathers Buch „Working Effectively with Legacy Code" (WELC): „A place where you can alter behavior (...) without editing in that place" (WELC, p.31, *[3]*).

An Seams kann man das Verhalten des vorhandenen Codes verändern, ohne an diesen Stellen editieren zu müssen. Je nach Programmiersprache und deren Sprachmitteln gibt es mehrere Typen von Seam. Da unser Beispiel in Java gehalten ist, zähle ich hier ein paar Seams auf, die man in Java gut benutzen kann. Jeder Seam hat einen Enabling Point, an dem entschieden wird, welches Verhalten aktiviert wird.

Override Seam

Durch Überschreiben der Methode einer Klasse in einer Unterklasse kann man Verhalten der abgeleiteten Klasse ersetzen. Enabling Point ist die Instanziierung der Klasse und ihre Zuweisung zu einer Variablen.

Ausgangssituation sei die Klasse `Service`. Stellen wir uns vor, sie sei unheimlich unhandlich, was ihre Geschäftslogik angeht, aber das Laden aus der Datenbank konnten wir wenigstens in eine eigene Methode extrahieren *(siehe Listing 1)*. Wir wollen die Daten jetzt aus einem anderen Datenbanksystem laden. Dafür machen wir die Methode zunächst abstrakt und erstellen dann zwei Unterklassen, eine mit der Legacy-Datenbank und eine mit der neuen Datenbanktechnologie *(siehe Listing 2)*.

Man könnte in Versuchung kommen, die Klasse direkt abzuleiten und nur die eine Methode zu überschreiben. Das ist jedoch nicht ratsam. Jeder, der schon einmal Klassenhierarchien mit mehr als zwei Vererbungsstufen bearbeiten musste, wird bestätigen, dass es besser ist, direkt zu sehen, wo die Unterklasse verwendet wird.

GeePaw Hill beschreibt die Gründe genauer in „Avoid Implementation Inheritance" *[4]* oder in seinem Tweet dazu *[5]*.

Methodenparameter Seam

Durch Verwendung eines anderen Methodenparameters kann neues Verhalten in die benutzende Methode injiziert werden, wenn auf dem Parameter Methoden aufgerufen werden. Enabling Point ist der Methodenaufruf.

Ausgangssituation ist eine Klasse ähnlich wie oben, nur lädt sie die Datenbankwerte, indem sie eine Methode auf dem Parameter `loader` aufruft *(siehe Listing 3)*. Hier erhalten wir einen Seam, indem wir ein Interface aus dem Loader extrahieren und eine neue Implementierung schreiben, die mit der neuen Datenbank arbeitet *(siehe Listing 4)*.

```
class Service {
    public Result largeUseCase(UseCaseParameters input) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    loadFromDatabase(Keys keys) {
        //loads from legacy database
    }
}
```

Listing 1: Override Seam – davor

```
abstract class Service {
    public Result largeUseCase(UseCaseParameters input) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    abstract loadFromDatabase(Keys keys);
}

class ServiceWithLegacyDatabase extends Service {
    @Override
    loadFromDatabase(Keys keys) {
        //loads from legacy database
    }
}

class ServiceWithNewDatabase extends Service {
    @Override
    loadFromDatabase(Keys keys) {
        //loads from new database
    }
}
```

Listing 2: Override Seam – danach

```
class DatabaseLoader { ... }

class Service {
    public Result largeUseCase(UseCaseParameters input, DatabaseLoader loader) {
        var values = loader.loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }
}
```

Listing 3: Methodenparameter Seam – davor

```
interface DatabaseLoader {
    Values loadFromDatabase(Keys keys);
}
class LegacyDatabaseLoader implements DatabaseLoader {
    public Values loadFromDatabase(Keys keys) {
        //legacy database access
    }
}
class ShinyNewDatabaseLoader implements DatabaseLoader {
    public Values loadFromDatabase(Keys keys) {
        //new database access
    }
}
```

Listing 4: Methodenparameter Seam – danach

Konstruktorparameter Seam

Analog zu den Methodenparametern können wir Kollaborateure auch im Konstruktor übergeben. Auch hier können wir ein Interface extrahieren und die neue Funktionalität durch eine weitere Implementierung des Interface in die Anwendung bringen. Enabling Point ist der Konstruktor-Aufruf.

Make non-static and override

Eine statische Methode kann nichtstatisch gemacht und überschrieben werden. Enabling Point ist die Instanziierung der Klasse *(siehe Listing 5)*. Auch hier verzichten wir darauf, die Methode direkt zu überschreiben, sondern behalten eine abstrakte Oberklasse und zwei konkrete Unterklassen *(siehe Listing 6)*.

Unklarheiten in den Anforderungen

Unser Auftrag für die „Gilded Rose" Kata lautete, dass wir verzauberte „conjured" Gegenstände unterstützen sollen, die doppelt so schnell wie normale Gegenstände ihre Qualität verlieren sollen. In guter Tradition vieler Online-Rollenspiele interpretieren wir das auf diese Weise:

Jeder Gegenstand kann verzaubert werden. Sein Name bekommt dann „Conjured" als Präfix vorangesetzt. Trotzdem bleiben ein paar Unklarheiten: Aus den Anforderungen geht etwa nicht hervor, ob man einen „Aged Brie" auch verzaubern kann und ein „Conjured Aged Brie" dann doppelt so schnell an Qualität gewinnt oder die Qualität stattdessen konstant bleibt. Gibt es „Conjured Backstage Passes"? „Conjured Sulfuras" sollte keine Auswirkungen auf den Qualitätsverlust haben. Verlieren verzauberte reguläre Gegenstände viermal so schnell Qualität, wenn ihr Ablaufdatum erreicht ist?

Fragen über Fragen, die wir am besten Allison stellen, wenn wir sie in der Mittagspause sehen. Bis dahin fangen wir mit verzauberten regulären Gegenständen an, die keine weiteren Eigenschaften haben.

```
class Service {
    public Result largeUseCase(UseCaseParameters input, DatabaseLoader loader) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    static Values loadFromDatabase(Keys keys) {
        //load from legacy database
    }
}
```

Listing 5: Make non-static and override – davor

```
abstract class Service {
    public Result largeUseCase(UseCaseParameters input, DatabaseLoader loader) {
        var values = loadFromDatabase(input.keys);
        //other complex legacy computations on values
        return result;
    }

    abstract Values loadFromDatabase(Keys keys);
}

class ServiceWithLegacyDatabase extends Service { ... }
class ServiceWithNewDatabase extends Service { ... }
```

Listing 6: Make non-static and override – danach

Sprout Method

Wir wollen den vorhandenen Code nicht noch schlimmer machen. Deswegen verwenden wir eine Sprout Method, um die neue Funktionalität vom vorhandenen Chaos zu trennen. Ein Sprout ist ein Code-Sprössling, den wir testgetrieben neben den vorhandenen Code „pflanzen", um das Risiko zu vermeiden, dort etwas kaputt zu machen.

Wir beginnen mit einigen vorbereitenden Schritten und führen nach jedem Refactoring unsere Tests aus dem vorigen Teil aus, damit wir sicher sind, nichts kaputt gemacht zu haben.

Vorbereitungen: Den Seam finden und erzeugen

Die Funktionalität der Methode `updateQuality()` besteht darin, über alle vorhandenen Instanzen von `Item` zu iterieren und ihre Attribute zu modifizieren. Um die alte Funktionalität zu isolieren, extrahieren wir zunächst das aktuelle `Item` in eine Variable: `Item itemToUpdate = items[i];`.

Danach können wir das aktuelle Verhalten in eine Methode `private void updateOtherItem(Item item)` extrahieren. Unsere Methode `updateQuality()` sieht jetzt so aus, wie in *Listing 7* gezeigt.

Jetzt könnten wir mit dem Sprouting anfangen. Um die Signatur der Sprout Method zu definieren, schreiben wir sie in den vorhandenen Code, als ob wir uns wünschen können, dass sie da wäre. Als Enabling Point fügen wir noch eine Unterscheidung in verzauberte und andere Gegenstände ein *(siehe Listing 8)*.

Unsere IDE kann uns die gewünschte Methode `void updateConjured(Item item)` jetzt anlegen. Damit wollen wir es aber mit den Modifikationen des Produktionscodes belassen. Wir haben zwar aus dem vorigen Teil noch ein Sicherheitsnetz für die bestehende Funktionalität, für verzauberte Gegenstände haben wir allerdings noch keine Tests. Generell ist es auch bei Legacy Code ein guter Ansatz, keinen Produktionscode ohne Tests dafür zu schreiben. Manchmal müssen wir jedoch – wie beim Sprouting – ein paar Modifikationen vornehmen, um überhaupt die Möglichkeit zu haben, neuen Code testgetrieben einzuführen.

```
public void updateQuality() {
    for (int i = 0; i < items.length; i++) {
        Item itemToUpdate = items[i];
        updateOtherItem(itemToUpdate);
    }
}
```

Listing 7: Vorbereitung zum Sprouting

```
public void updateQuality() {
    for (int i = 0; i < items.length; i++) {
        Item itemToUpdate = items[i];
        if (itemToUpdate.name.startsWith("Conjured ")) {
            updateConjured(itemToUpdate);
        } else {
            //here be dragons
            updateOtherItem(itemToUpdate);
        }
    }
}
```

Listing 8: Frisch gepflanzter Sprössling: Sprout Method

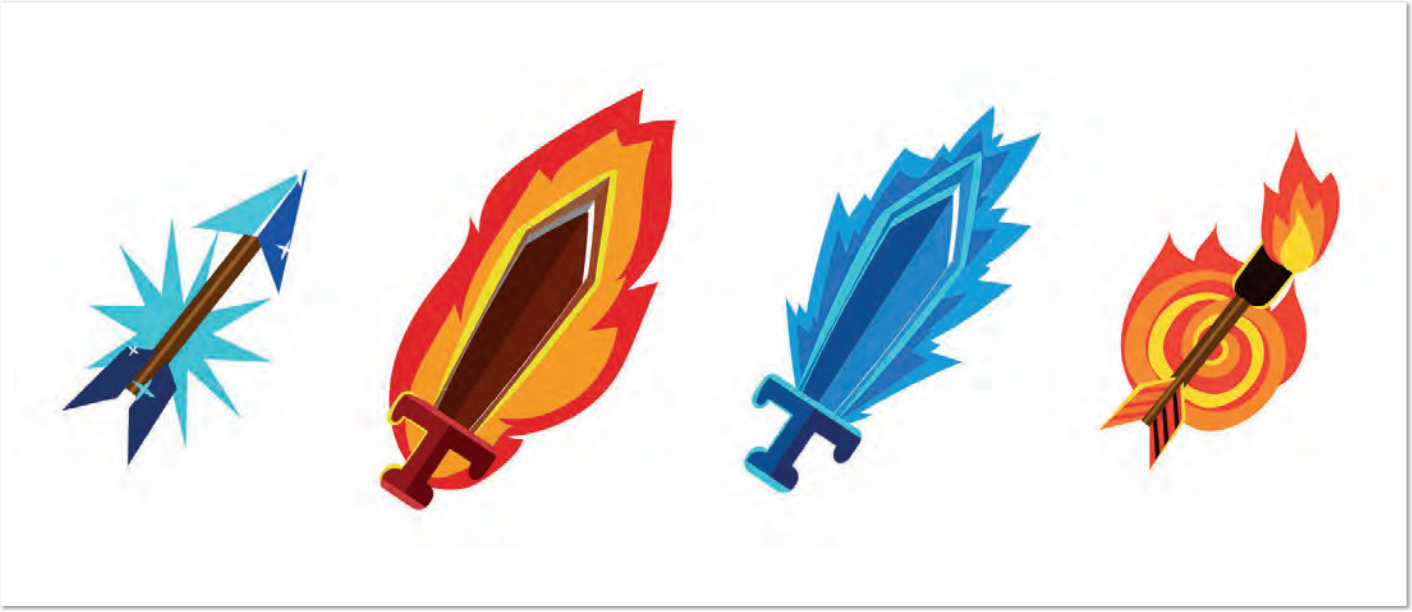
```
void updateConjured(Item item) {
    //TODO: implementieren
}
```

Listing 9: Noch leere Sprout Method

Um den Code lieferbar zu halten, kommentieren wir den neuen Methodenaufruf zunächst aus. Wenn wir jetzt liefern müssen, können wir ohne Einschränkungen auf die bestehende Funktionalität zurückgreifen.

Das Pflänzchen einsetzen

Jetzt haben wir ein Methodenskelett, das wir testgetrieben mit Leben füllen können *(siehe Listing 9)*. Beginnen wir mit den Tests für einen regulären verzauberten Gegenstand. Wir nehmen, wie oben angesprochen, an, dass er pro Tag zwei Qualitätspunkte verliert,



```
@ParameterizedTest
@ValueSource(ints = {10, 9, 8})
public void conjuredItem_sellInDateNotPassed_degradesQualityByTwo(int initialQuality) {
    Item item = new Item("Conjured Regular Item", notPastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isEqualTo(initialQuality - 2);
}
```

Listing 10: Testgetriebener Sprössling – erste Tests

```
void updateConjured(Item item) {
    item.quality -= 2;
}
```

Listing 11: Produktionscode zu den ersten Tests (aus Listing 10)

```
@ParameterizedTest
@ValueSource(ints = {10, 9, 8})
public void conjuredItem_sellInDatePassed_degradesQualityByFour(int initialQuality) {
    Item item = new Item("Conjured Regular Item", pastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isEqualTo(initialQuality - 4);
}
```

Listing 12: Zweites Szenario – Tests

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(item.sellIn < 0) {
        degeneration *= 2;
    }

    item.quality -= degeneration;
}
```

Listing 13: Zweites Szenario – Produktionscode

```
private boolean pastSellInDate(Item item) {
    return item.sellIn < 0;
}
```

Listing 14: Extrahierte Methode: Ablaufdatum erreicht

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(pastSellInDate(item)) {
        degeneration *= 2;
    }

    item.quality -= degeneration;
}
```

Listing 15: Sprout Method nach Extraktion der Methode

wenn er sein Ablaufdatum noch nicht erreicht hat und danach doppelt so viele, also vier pro Tag. Wir testen zunächst direkt die Sprout Method. Im Folgenden zeige ich der gebotenen Kürze wegen direkt parametrisierte Tests. Normalerweise extrahiere ich diese aus den zuerst von Hand geschriebenen Tests für die aktuelle Anforderung *(siehe Listing 10)*.

Wir starten mit einem Test, der nur den Anfangswert `10` für die Qualität betrachtet, und fügen dann schrittweise Beispiele für den aktuellen Testfall hinzu. Zuerst geben wir eine Konstante zurück und kommen zum Schluss durch Refactoring auf eine generalisierte Lösung (Triangulation, *siehe Listing 11*).

Zeit für das nächste Szenario, dessen Tests wir in *Listing 12* sehen. Wir brauchen jetzt eine Prüfung, ob der Gegenstand sein Haltbarkeitsdatum schon erreicht hat. In diesem Fall verdoppeln wir den Qualitätsverlust *(siehe Listing 13)*.

Jetzt sehen wir, dass der Prüfung eigentlich ein fachlicher Name fehlt und extrahieren die Methode *(siehe Listing 14)*. Unsere IDE hilft uns, indem sie bemerkt, dass die gleiche Prüfung auch im alten Code verwendet wird, und ersetzt sie auch dort durch den Methodenaufruf. Unsere Sprout Method sieht jetzt so aus, wie in *Listing 15* gezeigt.

Jetzt müssen wir sicherstellen, dass die Qualität nicht negativ wird, sowohl bei Gegenständen vor dem Ablaufdatum als auch danach.

Die Tests dafür sehen wir in *Listing 16*. Das treibt unseren Produktionscode in den Zustand in *Listing 17*.

Zum Schluss müssen wir noch implementieren, dass sich bei verzauberten Gegenständen das Ablaufdatum verringert – und zwar auch bis hin zu negativen Werten. Um dies zu erreichen, schreiben wir die in *Listing 18* gezeigten Tests. Wir brauchen lediglich eine zusätzliche letzte Zeile im Produktionscode, um diese Tests zu bestehen. Die gesamte Methode ist in *Listing 19* zu sehen.

```
@ParameterizedTest
@ValueSource(ints = {1, 0})
public void conjuredItem_sellInDateNotPassed_doesNotDegradeBelowZero(int initialQuality) {
    Item item = new Item("Conjured Regular Item", notPastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isZero();
}

@ParameterizedTest
@ValueSource(ints = {3, 2, 1, 0})
public void conjuredItem_pastSellInDate_doesNotDegradeBelowZero(int initialQuality) {
    Item item = new Item("Conjured Regular Item", pastSellInDate(), initialQuality);

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.quality).isZero();
}
```

Listing 16: Szenario „Qualität wird nicht negativ" – Tests

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(pastSellInDate(item)) {
        degeneration *= 2;
    }

    item.quality = Math.max(0, item.quality - degeneration);
}
```

Listing 17: Szenario „Qualität wird nicht negativ" – Produktionscode

```
@ParameterizedTest
@ValueSource(ints = {2, 1, 0, -1, -2})
public void conjuredItem_decreasesSellInDateByOne_perDay(int initialSellInDays) {
    Item item = new Item("Conjured Regular Item", initialSellInDays, anyQuality());

    GildedRose app = createApp(item);
    app.updateConjured(item);

    assertThat(item.sellIn).isEqualTo(initialSellInDays - 1);
}
```

Listing 18: Szenario „Ablaufdatum verringert sich" – Tests

```
void updateConjured(Item item) {
    int degeneration = 2;

    if(pastSellInDate(item)) {
        degeneration *= 2;
    }

    item.quality = Math.max(0, item.quality - degeneration);

    item.sellIn--;
}
```

Listing 19: Szenario „Ablaufdatum verringert sich" – Produktionscode

Sprout Class

Das Vorgehen bei Sprout Class wäre ähnlich gewesen. Wir hätten denselben Seam und Enabling Point verwenden können, aber anstatt eine neue Methode aufzurufen, hätten wir eine Klasse instanziiert, die intern die Daten des Gegenstands manipuliert – ähnlich wie wir es im vorigen Kapitel mit Wrap Class gemacht haben.

Testing, Staging, Livegang

Die neue Funktionalität können wir liveschalten, indem wir den vorher noch auskommentierten Methodenaufruf wieder aktivieren. Das ist auch möglich, solange wir noch nicht alle Anforderungen an die verzauberten Gegenstände erhalten und umgesetzt haben, etwa jetzt, wo wir mit regulären verzauberten Gegenständen umgehen können. Gibt es ein Staging-Konzept, können wir den Aufruf etwa nur durchführen, wenn ein Feature Switch aktiviert wurde. Dadurch können je nach Projektsituation Tester schon anfangen, einen Teil der neuen Funktionalität zu prüfen, und wir erhalten für die nächste Iteration zusätzlichen Regressionsschutz durch bereits vorhandene Tests. Haben wir die Tests im Voraus als Akzeptanztest formuliert, sollte ein Teil davon jetzt grün sein, nämlich der für die regulären verzauberten Gegenstände.

Auf lange Sicht: Das Strangler Pattern

Über mehrere Iterationen verteilt und auf lange Sicht lassen sich mit Sprouts schrittweise geordnetere Strukturen in den Code bringen. Im vorigen Beispiel können wir weitere Sprout Methods für bestehende Gegenstände erstellen, sobald wir an deren Funktionalität etwas ändern müssen. Neue Gegenstände bekommen sofort ihre eigene Methode. Das bestehende Brownfield, das wir in die Methode `updateOtherItem()` verbannt haben, wird dadurch immer kleiner und verschwindet zum Schluss komplett.

Sobald wir Ähnlichkeiten unter den Sprouts entdecken, können wir anfangen, gemeinsame Funktionalität zu extrahieren. Wenn wir mit Sprout Classes arbeiten, könnte sich beispielsweise auch eine Klassenhierarchie bilden und die Anwendung arbeitet nur noch mit einem Interface. Das ist jedoch alles Spekulation, in der wir uns nicht verfangen sollten.

An dieser Stelle sei deswegen davor gewarnt, sich vorschnell ein neues Design auszudenken, das alle unsere Probleme lösen wird, und dann in einem einzigen großen Umbau die gesamte Applikation „ordentlich zu machen". Das Risiko, dass das Vorhaben fehlschlägt, ist viel größer als bei kleinen, schrittweisen Umbauten. Große Umbauten blockieren auch die Weiterentwicklung zu lange, sodass dringend nötige Features nicht rechtzeitig eingesetzt werden können. Kleine Schritte geben uns die Möglichkeit zu reflektieren, ob das langfristige Ziel noch zu den aktuellen Anforderungen passt.

Martin Fowler beschreibt das Strangler Pattern in seinem Artikel „Strangler Fig Application" *[6]*.

Fazit

Mit Sprouts lässt sich auch in sehr komplexe Anwendungen testgetrieben neue Funktionalität einbauen. Oft liegt die Kunst darin, den richtigen Ort zu finden, an dem sich die neue oder geänderte Funktionalität am besten einsetzen lässt. Hierbei hat uns das Seam Model geholfen. Durch diese Techniken bleiben wir auch bei längeren Umbauten allzeit lieferbereit.

Quellen

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

[2] *https://bitbucket.org/georgberky/gilded-rose-kintsugi/*

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

[4] GeePaw Hill (2018): Avoid Implementation Inheritance. *https://www.geepawhill.org/2018/03/03/avoid-implementation-inheritance-geepaw-goes-all-geek-y/*

[5] GeePawHill (2020): A common mid-sized refactoring: „Replace Implementation Inheritance" *https://twitter.com/GeePawHill/status/1258005102167838723*

[6] Martin Fowler (2004): Strangler Fig Application *https://martinfowler.com/bliki/StranglerFigApplication.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.