|  |  |  |
| --- | --- | --- |
| Aktion | Java | Scala |
| Folie: Domain-Model mit Bank, Identification und Partner / Person |  |  |
| * Step 1 |  |  |
| J: Package ‚eightydays.java8’ erstellen |  |  |
| J: Abstract Class ‚Partner’ erstellen mit Feld ‚Name’  Konstruktor, Getter+Setter |  | Warum machst du Setter? |
|  | Damit ich den Namen anpassen kann. In der Java Beans ist das so üblich. |  |
|  |  | Immutability ist vorzuziehen. Den Namen kannst du beim erstellen der Instanz mit dem Konstruktor definieren. |
|  | Wieso? |  |
| Folie: Warum Immutability wichtig ist |  |  |
|  | Wie verändere ich dann meine Daten? |  |
|  |  | Daten einer Instanz werden nicht geändert. Wir erstellen eine neue Instanz mit den veränderten Daten. Anstelle einen Wert zu verändern, erstellen wir eine Instanz mit dem neuen Wert. |
| J: Feld ‚Name’ wird final, Setter wird gelöscht | (Aha) |  |
| J: Class ‚Person’ extends ‚Partner’ erstellen mit Feld ‚Firstname’  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Class ‚LegalEntity’ extends ‚Partner’ erstellen mit Feld ‚form’  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Class ‚Identifikation’ erstellen mit Feld ‚UUID’, Getter, Equals + Hashcode |  |  |
| J: Class ‚Bank’ mit Felder ‚name’ und HashMap ‚partners’  Konstruktor, addPartner(partner), Optional getPartner(id)  🡪 bracuehn wir Equals + Hashcode? |  |  |
|  | Testfälle dürfen natürlich nicht fehlen, da ich mit Hashcode und Equals Verhalten programmiert habe. |  |
| * Step 2 |  |  |
| J: Tests ausführen und zeigen, dass sie erfolgreich sind |  |  |
|  | Wie erstellt man jetzt solche Klassen in Scala? |  |
| S: Package ‚eightydays.scala’ erstellen |  |  |
| S: Abstract Class ‚Partner’ erstellen mit Feld ‚name’ |  | Fertig! |
|  | Du bist doch noch nicht fertig! Wo ist das Feld ‚name’? Wo ist der Konstruktor? Wo der Getter? |  |
|  |  | Das was bei Java von Hand geschrieben oder mit der IDE generiert wird, macht bei Scala der Compiler für mich.  Ich muss mir keine Sorgen machen, dass ich einmal ein Feld vergesse anzupassen, wenn ich Änderungen mache. |
|  |  | Ein weiterer Vorteil davon ist, dass ich mir ganz nebenbei viel Redundanz erspare. Ein Beispiel ist das der Konstruktor zwingend gleich heissen muss wie die Klasse selbst.  Ebenfalls muss das Feld ‚name’ nur einmal geschrieben werden. Bei Java ist es ein Mehrfaches, was die Wartbarkeit und Fehleranfälligkeit erhöhen kann. |
|  |  | Ich habe in Scala auch die Einschränkung nicht, dass die Datei gleich heissen muss wie die die Klasse. D.h. ich kann in einem File mehrere Elemente definieren und so zusammengehörende Fachlichkeit auch zusammen definieren. |
|  |  | Für einen einfachen Vergleich zwischen Java und Scala sind separate Dateien besser, daher verzichte ich jetzt darauf. |
| S: Class Person mit Feld ‚firstName’ |  |  |
| S: Class LegalEntity mit Feld ‚form’ |  |  |
| S: Class Identification mit Feld ‚number’ |  |  |
|  |  |  |
| S: Class Bank mit Feld ‚name’ und private[this] var ‚partners’ |  |  |
|  | Was ist das [this] bei private? |  |
|  |  | Ein Unterschied zu Java ist, dass in Scala alles per default einen public-Scope hat. Wir haben ja alle Daten Immutable, daher ist das kein Problem. |
|  |  | Das hier ist ein private-Scope nicht nur für die Klasse, sondern für die Instanz einer Klasse. Nur diese hat auf dieses Feld Zugriff. Etwas, was in Java leider nicht möglich ist. |
| S: Class Bank mit partner(id) = partners get id |  |  |
|  | Wo sind die Punkte bei der Implementierung hier? |  |
|  |  | Dass ich die Semikolons weggelassen habe, ist dir sicherlich schon aufgefallen. |
|  |  | Unter bestimmten Umständen können auch die Punkte oder auch die Klammern weggelassen werden.  Dies ermöglicht zum Beispiel die einfache Erstellung von internen DSLs und kann auch die Lesbarkeit und Verständlichkeit erhöhen. |
|  | Nett, aber die Funktion gibt noch nichts zurück, da fehlt noch das Return-Statement. |  |
|  |  | Auch das Return-Statement selbst ist in Scala optional.  Normalerweise ist das Resultat, also der Rückgabewert einer Funktion, die letzte Expression. Der Return-Type wird durch den Compiler nach definierten Regeln bestimmt.  Auch ein If-Statement oder eine for-/ while-Schlaufe liefert einen Wert zurück und haben daher einen Type. |
|  | Das ist etwas gewöhnungsbedürftig.  Da sehe ich nicht mehr auf den ersten Blick, wann was und wie aus einer Methode zurückgegeben wird! |  |
|  |  | Ein wesentlicher Vorteil von Scala gegenüber Java ist, dass ich mit Scala den Code auf das Wesentliche, auf die Essenz, reduzieren kann.  Scala ist sehr minimalistisch. Das beste Beispiel dazu ist, wie wir bereits gesehen haben, die Definition einer Case-Klasse, reduziert auf eine Zeile Code. |
|  | Das leuchtet mir ein.  Und was ist mit Testfällen? |  |
| * Step 3 |  |  |
| S: Test ausführen und Resultat zeigen. |  |  |
|  |  | Da ich mit Scala das ganze Java-Universum verwenden kann, kann ich natürlich auch auf JUnit zurückgreifen.  Es gibt jedoch noch andere Testframeworks, z.B. ScalaTest. |
|  |  | ScalaTest kennt verschiedene Test Stile.  Neben dem „Function Style“, ähnlich dem von jUnit, gibt es auch strukturierte Tests, wie z.B. WordSpec, welcher ein BDD/ Behaviour-Driven Development Stil implementiert. |
|  | Bei JUnit hatte ich immer das Problem, dass ich meine Testfälle nicht strukturieren konnte.  ScalaTest löst das für mich schön elegant. |  |
|  |  | Lass uns das Domain Model weiter implementieren. |
| Folie: Domain-Model mit zusätzlichen Accounts |  |  |
| * Step 4 |  |  |
|  | <Code erklären> |  |
|  | Das Design, resp. die Implementierung der verschiedenen Kontotypen ist etwas schwierig. |  |
| Folie: AccountTypen und Regeln, die orthogonal zur Hierarchie sind |  |  |
|  | Die Regeln und Berechnungen sind Querschnittsfunktionen, die unabhängig von der Vererbungshierarchie benutzt werden.  Um die Anforderungen sauber (d.h. ohne Redundanz und unter Beachtung des open-closed Prinzip) zu implementieren, habe ich mich für den Ansatz mit einem Decorator auf den Methoden withdraw und deposite entschieden. So kann ich bei jeder Klasse bestimmen, welche Regeln angewendet werden und dies unabhängig von der Basis-Klasse. |  |
| * Step 5 |  |  |
|  | <Code erklären>  Zusätzlicher Code für die Implementierung des Pattern.  Saving und dann Current |  |
| J: Tests ausführen |  |  |
|  |  | Das funktioniert, aber das ist technischer Boilerplate-Code, welcher nicht fachlich getrieben ist.  Mit Scala habe ich andere, einfachere Möglichkeiten |
| * Step 6 (Account, Sub-Klassen und Traits) |  |  |
|  |  | <Code erklären>   * Account   + ‚balance’ berechnet Saldo mit Hilfe der Bookings |
|  | Das Import-Statement ist am falschen Ort. |  |
|  |  | Nicht ganz. Ich kann Imports schreiben, wo sie benötigt sind. Diese sind dann auch an den jeweiligen Kontext gebunden. |
|  | Was ist Amount? Diesen Typ kenne ich nicht. In Java ist das BigDecimal. |  |
|  |  | Amount ist ein abstrakter Typ. Eine Art Alias für BigDezimal. Man kann dies für alle möglichen Arten von Typen -Definitionen verwenden.  Dies erhöht wiederum die Les- und Wartbarkeit von Code. |
|  | Moment, da ist noch ein Problem!  Wie kannst Du ein Amount zu einer Booking mit einem + addieren? |  |
|  |  | Das hier ist etwas Scala-Magie.  Hier wird ähnlich den Case-Klassen durch den Compiler Code generiert, welcher diese Konvertierung erst ermöglicht, eine sogenannte implizite Konvertierung.  Java kennt das übrigens auch: Autoboxing und Unboxing von primitiven Typen.  In Scala jedoch kann ich solche Konvertierungen selbst definieren.  Es können auch Methoden, Klassen und Parameter implizit sein, was mir ganz neue Möglichkeiten eröffnet.  Ich als Entwickler muss daher wissen, was der Compiler hier für mich macht. Dadurch steigt aber auch die Anforderung an mich als Entwickler. |
|  | Aber wie weiss ich, welche implizite Methode da aufgerufen wird? Ich sehe das dem Code ja nicht direkt an. |  |
|  |  | Das ist in der Tat so. Da muss ich mich auf die Hilfe von IDEs verlassen.  Sie hilft mir in diesem Fall und zeigt mir eine implizite Konvertierung an <grauer Unterstrich und Ctrl + Q bei booking>.  In der Entwicklung selbst gilt es wie immer einen Mittelweg zwischen den Möglichkeiten der Sprache und der Verständlichkeit des Codes zu finden.  Je mächtiger eine Sprache ist, desto mehr muss ich mir das bewusst sein.  Ein weiteres Beispiel für die Möglichkeiten von Scala ist hier die Addierung von Amount mit Booking.  In Java muss ich immer die add-Methode aufrufen.  Da in Scala Operatoren auch Funktionen sind, kann das ‚+’ dafür verwendet werden. Im übrigen ruft diese Funktion auch einfach das ‚add’ auf, ist aber verständlicher.  Die JVM erlaubt dies und Scala nutzt diese Möglichkeit. In Java jedoch ist die Sprache selbst der limitierende Faktor.  Aber schauen wir uns doch die weitere Implementierung an.  Bei ‚withdraw’ habe ich einen Default Value für den Parameter ‚valuta’ definiert. So muss ich keine Methoden überladen und bleibe trotzdem flexibel.  Aber interessant wird es mit dem Saving-Account:  Wir erweitern die abstrakte Account-Klasse zusätzlich mit sogenannten Traits.  Diese Traits sind Mixins, welche mir zusätzliche Funktionalitäten bereitstellen können.  In diesem Fall mixen sie die bestimmte Logik beim Abheben eines Betrages ein. |
|  | Das ist doch das ähnlich wie die Default Methoden in Interfaces bei Java 8. |  |
|  |  | Die Motivation bei Java ist eine andere. Da werden sie auch ‚Public Defender Methods’ genannt. Es geht dabei um Interface Evolution und nicht um die eigentliche Modularisierung von Fachlichkeit bzw. von Komponenten.  Bei Java wird die default Implementierung eines Interfaces nur aufgerufen, wenn sie in einer implementierenden Klasse nicht überschrieben wird.  In Scala mit Traits ist das anders.  Scala definiert bei Traits eine strikte Reihenfolge, wie gleich benannte Methoden bei mehreren vorhandenen Traits nacheinander aufgerufen werden. Dies löst das bekannte Diamond-Problem bei mehrfach Vererbung.  Im Falle eines Saving-Accounts wird die Logik für eine Limitierung eines Withdraws und zusätzlich eine Verhinderung des Überzugs beigemischt.  Die anderen Account-Typen sind ähnlich aufgebaut.  Current definiert einen Account, welches zusätzlich Limited und LowBalancedPerbooking verwendet.  Saving ist das gleiche, jedoch mit NoOverdraw anstatt LowBalancedPerBooking.  Und natürlich dürfen die Tests dazu nicht fehlen. |
| S: Tests ausführen |  |  |
| * Step 7 | Weiter mit den Methoden des Zusammenspiels Partner/Account mit Bank |  |
| J/S  Deposite/withdraw ueber bank  Asset (collection)  Find (pattern matching)  Filter partner/account |  |  |
|  |  | Ich habe noch einige Funktionen bei der Bank hinzugefügt. Zum einen, dass die Kontos verwaltet werden, openAccount und auch das Buchen withdraw und deposit. Da beim Buchen eine neue Instance des Kontos entsteht, muss diese bei der Bank aktualisiert werden.  Im Weiteren habe ich auch die Funktion asset für die Bestimmung des gesamten Guthabens eines Partner hinzugefügt. Diese filtert alle Kontos auf Grund des Owners und summiert dann den Saldo der Kontos.  Für die Suche nach Partner und Personen sind die Funktionen searchPartner und searchPerson implementiert. Bei searchPerson wird das Filter mit einem pattern match der Partners realisiert. Das case prüft die Klasse und den Wert des Feldes firstName.  Auch für diese Funktionen habe ich einen Test geschrieben. |
| S: Tests ausführen und sie sind grün |  |  |
|  |  | Wie sieht die Implementierung hier in Java aus? |
|  | Die Funktionen lassen sich ähnlich implementieren.  Es ist etwas mehr zu schrieben, da Java BigDecimal nicht als primitiven Type kennt und es auch keine implizite Umwandlung in anderen Typen gibt.  Speziell ist die Funktion searchPartner. Hier musste ich die Tests für die Klasse und den Wert firstName selber programmieren. Es gibt in Java kein Pattern-Matching. |  |
| J: Test ausführen und sie sind grün |  |  |
|  | Die Tests sind aber auch grün. |  |