|  |  |  |
| --- | --- | --- |
| Aktion | Java | Scala |
| Step 1 |  |  |
| J: Package ‚eightydays.java’ erstellen |  |  |
| J: Java Class ‚Partner’ erstellen mit Feld ‚Name’  Konstruktor, Getter+Setter |  | Warum machst du Setter? |
|  | Damit ich den Namen setzten kann |  |
|  |  | Wir möchten Immutability. Namen setzt du beim erstellen der Instanz mit Konstruktor. |
|  | Wieso? |  |
| Folie: Warum Immutability wichtig ist |  |  |
|  | Wie verändere ich dann meine Daten? |  |
|  |  | Das ist eine neue Instanz mit den veränderten Daten. Anstelle den Wert zu aendern, erstellen/kopieren wir die Instance mit dem neuen Wert gesetzt  🡪 Kein Slide mehr? |
| J: Feld ‚Name’ wird final, Setter wird gelöscht | (Aha) |  |
| J: Java Class ‚Person’ extends ‚Partner’ erstellen mit Feld ‚Firstname’  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Java Class ‚LegalEntity’ extends ‚Partner’ erstellen mit Feld ‚form  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Java Class ‚Identifikation’ erstellen mit Feld ‚UUID’  Konstruktor, Getter, Equals + Hashcode |  |  |
| J: Java Class ‚Bank’ mit Felder ‚name’ und HashMap ‚partners’  Konstruktor, Optional getPartner(id), addPartner, Equals + Hashcode |  |  |
|  | Natürlich schreibe ich dazu Testfälle |  |
| * Step 2 |  |  |
| Tests ausführen | Tests zeigen und starten. Sind grün |  |
|  | Und wie sieht das in Scala aus? |  |
| S: Package ‚eightydays.scala’ erstellen |  |  |
| S: Scala Class ‚Partner’ erstellen mit Feld ‚Name’ |  |  |
|  | Du bist doch da noch nicht fertig! Wo ist das Feld name? Wo ist der Construcotr, Wo ist der getter, hashCode und Equals |  |
|  |  | Das was bei Java die IDE gemacht hat, macht bei Scala der Compiler. Bei einer case classe werden getter, hashCode und equals unter anderem erstellt.  Ein weiterer Vorteil davon ist, dass ich mir viel Redundanz erspare, wie z.B das der Construcotr gleich heissen muss wie die Klasse und ich muss das Feld name nur einmal schreiben. Bei Java ist es ein Mehrfaches. |
| S: Scala Class Person mit firstName, LegalEntity mit form und Identification mit number |  | Ich habe die Einschränkung, dass das File gleich heissen muss die die Klasse nicht. D.h. ich kann in einem File mehrere Elemente definieren. |
|  | Du hast da den Type von Number vergessen | Nein, der Compiler leitet den Typ aus dem Expression ab. In Scala ist fast alles eine Expression. D.h. auch ein If, for oder while liefert einen Wert. |
| S: Scala Bank mit partners (private[this]) und def partner/id:Identifciation) | Was ist das [this] bei private? | Das ist private nicht nur für die Klasse, sondern nur die Instance hat hierauf zugriff. Etwas, was Java nicht kann.  Ein Unterschied zu Java ist, dass in Scala alles per default public ist, Bei Java ist es ja package. |
|  | Wo sind die . bei der Implementierung von partner? | In Scala können unter bestimmten Umständen die Punkte oder auch die ; oder auch die () weggelassen werden. Dies ermöglicht die einfache Erstellung von internen DSLs. |
|  | Die Funktion gibt aber nichts zurück, da fehlt doch das Return? | In Scala ist das Return optional, normalerweise ist das Resultat der letzten Expression das Resultat der Funktion |
|  | Das ist doch etwas gewöhnungsbdürftig. Da sehe ich nicht mehr, wann ich was wie aus einer Methode liefere! | Return ist nichts Anderes als ein goto  Return ist boilerplate code  Mit Scala kann ich den Code auf das Wesentliche, auf die Essenz reduzieren.  Scala ist minimalistisch. Das beste Beispiel ist die Definition einer case class. |
|  | Und Testfälle |  |
| * Step 3 |  |  |
| Test ausführen und Test Resultat zeigen |  | Bei Scala gibt es nebst JUnit, auch andere Testframeworks, z.B. ScalaTest. ScalaTest kennt verschiedene Test Style.  Neben Function style (ähnlich xUnit) gibt es auch strukturierte Tests, wie z.B. WordSpec, welcher ein BDD (behaviour driven) style implementiert. |
|  | Bei JUnit hatte ich immer das Problem, dass ich meine Testfälle nicht strukturieren konnte. ScalaTest löst das für mich. | Stimmt. Lass uns das Domain Model weiter implementieren. |
|  |  |  |
|  | <Code erklären> | Das Design hat ein schwerwiegendes Problem. Was wenn die Regeln für die Berechnung der Gebühr oder die Regeln für den Bezug bei verschiedenen Account-Typen verwendet werden? |
| Folie: Klassendiagramm und AccountTypen und Regeln, die orthogonal zur Hierarchie sind | In Java mache ich für solche Anforderungen jeweils ein Strategie-Pattern. Das sieht so aus. |  |
| * Step 5 | <erklären>  Zusätzlicher Code für die Implementierung des Pattern | Das funktioniert, aber das ist technischer Boilerplate-Code, welcher nicht fachlich ist.  Das geht einfacher |
| * Step 6 (Account, Sub-Klassen und Traits) |  | <erklären>   * Account   + mit type ist alias auf BigDecimal   + balance berechnet Saldo mit Hilfe der Booking |
|  | Moment, da ist ein Fehler! Wie kannst Du ein Amount zu einer Booking mit einem + addieren? | Das ist etwas Scala-Magie. Hier wird durch den Compiler Code hinzugefügt welcher diese Konvertierung ermöglicht. In Java ist das übrigens das selbe wie mit Autoboxing. In Scala kann ich solche Konvertierungen selbst definieren.  Diese implizite Konvertierung ist eine sehr mächtige Möglichkeit in Scala. Daneben können auch Methoden, Klassen und Parameter implizit sein.  Ich als Entwickler muss wissen, was der Compiler hier für mich macht. Dadurch steigt die Anforderung an mich als Entwickler. |
|  | Aber wie weiss ich, welche implizite Methode da aufgerufen wird. Ich sehe das dem Code nicht an. | Das ist in der Tat so. Die DIE hilft mir in diesem Fall und zeigt mir das an <grauer Unterstrich und Ctrl + Q bei booking>.  Hier gilt es auch, einen Mittelweg zwischen Möglichkeiten und Verständlichkeit zu finden. |
|  |  | * Account   + Bei withdraw habe ich einen Default Value für den Parameter valua definiert. Ist ein Equivalent zu einer überladenen Methode in Java. * Package fee mit trait LowBalancePerBooking ist ein MixIn, welches die Methode post überschreibt und dadurch die Funktion erweitert. Ich könnte hier auch neue Funktionalität hinzuefuegen.   (geht auf val anstelle von def beim trait?) |
|  | Das ist doch das gleiche wie die default methoden in Interfaces bei Java 8.  https://kerflyn.wordpress.com/2012/07/09/java-8-now-you-have-mixins/ | Die Motivation bei Java ist eine andere. Hier geht es um Interface Evolution, nicht um Modularisierung von Fachlichkeit/Komponenten. Auch definiert Scala eine Reihenfolge, wie die gleichen Methoden bei mehreren Traits nacheinander aufgerufen werden. Dies löst das Diamond-Problem.  <http://stackoverflow.com/questions/16410298/what-are-the-differences-and-similarties-between-scala-traits-vs-java-8-interfa> |
|  |  | * Current: hier wird ein Account definiert, welches zusätzlich Limited und LowBalancedPerbook beinhaltet. Dadurch wird beim Post zusätzliche Fachlichkeit ausgeführt. Fee und Limiten werden als zusätzliche Constructor-Parameter übergeben. * Saving ist das gleiche, aber mit NoOverdraw anstatt LowBalancedPerBooking |
| Tests ausführen |  |  |
|  | Weiter mit den Methoden des Zusammenspiels Partner/Account mit Bank |  |