Vorgehensweise und Entscheidungsfindung

Zunächst haben wir die gewünschten Klassen erstellt und die Interfaces implementiert.
Anschließend haben wir uns die Methodenrümpfe und Konstruktoren erstellen lassen.
Die Methoden haben wir sinnvoll ausimplementiert und ggf. noch die Konstruktoren angepasst.
Nach Bestehen der JUnit Tests haben wir uns an die Objekterstellung und Ausgabe gemacht.

class Bahnhof (main-Methode)

In unserem „Bahnhof“ werden zunächst drei Lokomotiven erstellt, eine von jedem Typ.
Weiterhin bilden wir insgesamt drei Züge, mit je einer der erstellten Lokomotiven.
In dem folgenden try/catch-Block erstellen wir in einer Endlosschleife (for(;;)) in jedem Durchlauf die zwei gegebenen Wagontypen. In dem inneren, zweiten try/catch-Block füllen wir eine Ladung mit zufälligem (aber gültigem) Gewicht hinzu. Anschließend werden die befüllten Wagons den drei Zügen zugeordnet. In unserem erstellen Beispielcode wird die „WagonOverloadException“ nicht geworfen, da nur passende Ladung hinzugefügt wird.
Würde man z.B. in der Schleife den “RungenwagenKBS” mit einer zufälligen Ladung zwischen 0 und 280.001 statt zwischen 0 und 28.001 befüllen, würde bei jeder Zufallsladung, die größer als 28.001 die “WagonOverloadException” ausgelöst. Das letzte Ladungsstück würde also nicht mehr hinzugefügt werden und die Schleife würde wieder von Anfang an starten.
Ähnlich verhält es sich mit der “addWagon” Methode für die Züge. Soll ein Wagon hinzugefügt werden und die Lok könnte den Zug anschließend nicht mehr ziehen, wird eine “OverloadException” geworfen und der Wagon nicht angefügt.
Wir haben uns nun einige Zeilen Code gespart, da wir beide Wagontypen in der Schleife erstellen und allen drei Zügen auf einmal Wagons hinzufügen. Allerdings ist so das Hinzufügen dann bereits zu Ende, wenn einer der drei Züge die OverloadException wirft. Dann sind zwar noch nicht alle Züge bestmöglich mit Wagons beladen, allerdings sind sie weiterhin gültig, können also fahren. Dieser Schritt dient rein der Übersichtlichkeit der Klasse. Genaugenommen würden wir drei Blöcke erstellen und an jeden Zug einzeln Wagons koppeln, bis die OverloadException geworfen wird. So hätte dann jeder Zug eine optimale Auslastung von Wagons.
Weiterhin KÖNNTE man sich den einen try/catch-Block sparen, da wir in unserem Fall nur gültige Ladung mit radom.nextInt() erstellen. Dies ist also nur eine vorrausschauende Lösung.
In unserem Fall gibt es neben der “OverloadException” noch die “WagonOverloadException”, wobei die “WagonOverloadException” nur die “OverloadException” erweitert (WagonOverloadException extends OverloadException).
Diese Unterscheidung haben wir nur eingeführt, um in der for-Schleife prüfen zu können, wo genau die Exception auftritt, beim Beladen oder beim Wagon hinzufügen. Würden beide Blöcke nur eine “OverloadException” werfen, würden wir schnell in eine Endlosschleife laufen, da der zweite catch-Block nie erreicht werden würde …
Für die beiden Exceptions haben wir uns zwei Klassen erstellt, allerdings enthalten diese nocht nicht wirklich viel an Inhalt, dies war aber auch nicht gefordert nach Aufgabenstellung.

class ZugAusgabe

Um unseren Bahnhof klein zu halten, haben wir die Klasse ZugAusgabe erstellt.
In dieser Klasse gibt es nur zwei Methoden. Die Methoden erhalten jeweils ein Zugobjekt übergeben und geben danach die Daten der Lok aus und alle angehängten Wagons mit Netto-, Tara- und Bruttogewicht.

Der einzige Unterschied ist, dass die Methode ausgabeUnsortiert die Wagons in der Reihenfolge, wie sie hinzugefügt wurden, ausgibt. Die Methode ausgabeSortiert erstellt zunächst eine neue ArrayList, in die alle Wagonelemente aus dem Übergabeobjekt eingefügt werden. Danach ordnet sie die Wagons zunächst aufsteigend nach deren Gesamtgewicht und gibt sie anschließend aus.
Die neue ArrayList sorgt nur dafür, dass die originale Reihenfolge nach dem Sortieren nicht verloren geht, das ursprüngliche Objekt also nicht verändert wird.

class WagonComperator
Um unsere Wagons zu sortieren, nutzen wir die Methode sort(List list, Comparator c) aus java.util.Collections.
Die zu sortierende Liste ist ja unsere ArrayList mit IWagon Objekten. Für den Comparator haben wir eine eigene Klasse WagonComparator geschrieben.
Sie enthält nur die überschriebene compare-Methode.
Diese Methode bekommt zwei IWagon Objekte übergeben und gibt das Ergebnis der Subtraktion der beiden Wagongesamtgewichte zurück. Ist das Ergebnis 0, passiert nichts, da das Gewicht gleich ist. Ist das Ergebnis negativ, ist der zweite Wagon schwerer, also kommt der erste Wagon vor den zweiten.
Ist das Ergebnis positiv, ist der erste Wagon schwerer, also kommt der zweite Wagon vor den Ersten.

Erweiterungen der anderen Klassen:

- Die Klasse Lokomotive wurde um eine get-Methode für den Typ erweitert, damit dieser in der Klasse ZugAusgabe verwendet werden kann.
- Die Klasse Zug wurde um get-Methoden für die Lokomotive und die Wagonliste erweitert, so kann man aus dem erstellten Zug über die Lokomotive auf deren get-Methoden zugreifen.
- Die Klasse Wagon wurde um die Methode getTyp() erweitert, diese Methode dient nur zur Vermeidung der Warnung, dass die Variable typ niemals verwendet wird. Sie wird in unserem Programm nicht verwendet.

Martin Slowikowski
Matrikelnummer: 199 91 66

Jan-Tristan Rudat
Matrikelnummer: 200 78 52

Teamname: Bernie und Ert

Quellenangaben

Comparator Interface
Sun Microsystems, Inc.; Interface Comparator
URL: http://java.sun.com/j2se/1.4.2/docs/api/java/util/Comparator.html
(abgerufen am 24.05.2010)

Collections Klasse
Sun Microsystems, Inc.; Class Collections
URL: http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collections.html
(abgerufen am 24.05.2010)