Programmieren III (Java)

1. Praktikum: Testen mit JUnit

Wintersemester 2023
Christopher Auer

Lernziele

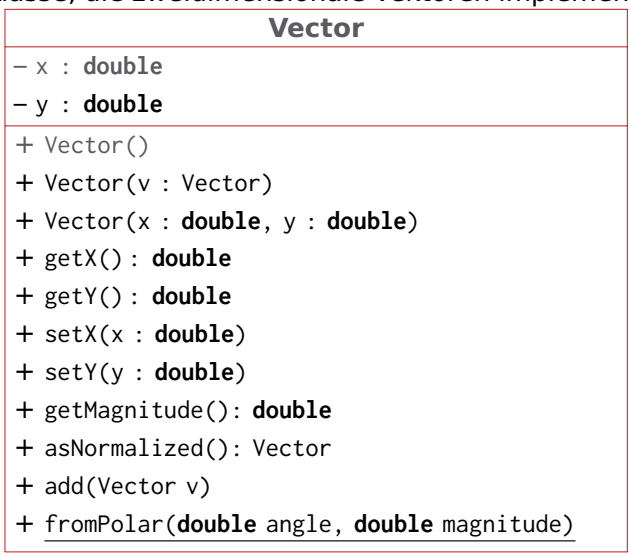
- Testen mit JUnit
- Schreiben von Tests nach einer Spezifikation
- Implementieren nach einer Spezifikation
- Erfahrungen sammeln mit:
  - Implementierung *vor* dem Schreiben der Tests
  - Implementierung *nach* dem Schreiben der Tests

Hinweise

- Die Praktika sind eine *ausgezeichnete Prüfungsvorbereitung*; aber nur, wenn Sie sie *eigenständig* bearbeiten oder es zumindest *versuchen*. Nachvollziehen der Lösungsvorschläge *reicht nicht* aus.
- Bearbeiten Sie die Aufgaben *vor* dem Praktikumstermin.
- Im Praktikum können Sie *Ihre Lösung* zeigen und *Fragen* stellen.
- Je nach *zeitlichem Verlauf*, wird während des Praktikumstermins der *Lösungsvorschlag besprochen*.
- Der *Lösungsvorschlag* wird online gestellt, nachdem *alle Gruppen* das Praktikum durchlaufen haben.

Aufgabe 1: Erst implementieren, dann testen

In dieser Aufgabe entwickeln wir erst eine einfache Klasse und schreiben *dann* die Tests dafür. Gegeben ist folgende Klasse, die zweidimensionale Vektoren implementiert:



Die Methoden sind dabei wie folgt spezifiziert:

- `Vector()` — Default-Konstruktor, erstellt den *Nullvektor*.
- `Vector(x : double, y : double)` — initialisiert den Vektor mit den Koordinaten `x` und `y`.
- `Vector(Vector v)` — Kopier-Konstruktor
- `getX/getY/setX/setY` — Setter und Getter
- `getMagnitude()` — liefert die Länge des Vektors nach $\sqrt{x^2 + y^2}$.
- `asNormalized()` — gibt die *normalisierte Version* des Vektors als *neue Instanz* zurück. Bei einem Vektor der Länge 0 wird eine `IllegalStateException` generiert. *Hinweis*: Die normalisierte Version ergibt sich, indem Sie die Koordinaten durch die Länge des Vektors teilen.
- `add(Vector v)` — *addiert* die Koordinaten von `v` zum Vektor hinzu.
- `fromPolar(double angle, double magnitude)` — erzeugt eine `Vector`-Instanz aus *Polarkoordinaten*, wobei `angle` der Winkel zwischen `0` und `2*Math.PI` und `magnitude >= 0` die Länge ist. Dabei gilt `x = magnitude * Math.cos(angle)` und `y = magnitude * Math.sin(angle)`.

Implementierung der Klasse `Vector`

Implementieren Sie die Klasse `Vector` wie oben spezifiziert. Achten Sie dabei auf eine „*defensive Programmierweise* “, d.h., prüfen Sie Eingabeparameter und generieren Sie entsprechende Ausnahmen im Fehlerfall.

Implementierung der Tests

Implementieren Sie nun JUnit-Tests, die Ihre Implementierung möglichst umfassend testet:

- Testen die die *grundlegenden Funktionen* wie oben beschrieben.
- Testen Sie *Rand- und Fehlerfälle*.
- Achten Sie bei Vergleichen von **`double`**-Werten darauf eine *geeignete Fehlerschranke* anzugeben.
- Welche *Beobachtungen* machen Sie beim Schreiben und Ausführen der Tests?

- Decken Ihre Tests *Fehler* auf? Wenn ja, *wieviele*?
- Sind Ihnen schon bei der Implementierung der Tests (Fehler-)Fälle aufgefallen, die Sie in Ihrer `Vector`-Implementierung nicht oder nicht richtig beachtet haben?
- Wie stark hat sich die *Code-Qualität Ihrer* `Vector`*-Implementierung* bei bzw. nach der Implementierung der Tests verbessert?

Aufgabe 2: Erst Tests schreiben, dann implementieren

In dieser Teilaufgaben machen wir es genau *umgekehrt*: Wir schreiben zuerst die Testfälle und *danach* die Implementierung. Dazu verwenden wir folgende Klasse, die Brüche $\frac{p}{q}$ modelliert:

Fraction
+ `Fraction(nom : int, denom : int)`
+ `Fraction(f : Fraction)`
+ `getNom(): int`
+ `getDenom(): int`
+ `setNom(nom : int)`
+ `setDenom(denom : int)`
+ `normalizeSign()`
+ `asReduced(): Fraction`
+ `value(): double`
+ `equals(Object f): boolean`
+ `valueEquals(Fraction f): boolean`

- `Fraction(nom : int, denom : int)` — Konstruktor erstellt Bruch mit *Zähler* `nom` und *Nenner* `denom`, wobei `denom != 0`.
- `Fraction(f : Fraction)` — Kopier-Konstruktor
- `getNom/getDenom/setNom/setDenom` — Getter/Setter
- `normalizeSign()` — setzt das Vorzeichen von Zähler und Nenner so, dass bei negativen Brüchen nur der *Zähler negativ* ist. Bei positiven Brüchen sind Zähler *und* Nenner *positiv*. Sollte der Zähler den Wert 0 haben, so wird der *Nenner auf 1* gesetzt. Natürlich soll diese Operation den *Wert* des Bruchs nicht verändern.
- `asReduced()` — gibt eine *vollständig gekürzte Version* des Bruchs als *neue* `Fraction`*-Instanz* zurück. Die *ursprünglichen Vorzeichen* von Zähler und Nenner bleiben dabei *erhalten*.
- `value()` — liefert den *Wert des Bruchs* als **`double`**, z.B., `0.5` für $\frac{1}{2}$ oder ungefähr `0.3333` für $\frac{2}{6}$.
- `equals(Fraction f)` — prüft auf *Wertgleichheit*, d.h. Zähler und Nenner sind jeweils *gleich*. Beispielsweise sind $\frac{2}{4}$ und $\frac{1}{2}$ *nicht wertgleich* in diesem Sinne.
- `valueEquals(Fraction f)` — prüft auf *numerische Gleichheit*, d.h. die Brüche haben den gleichen *numerischen* Wert, z.B. $\frac{2}{4}$ und $\frac{1}{2}$.

Implementierung der Tests

Implementieren Sie nun JUnit-Tests, die der obigen Spezifikation entspricht:

- Testen die die *grundlegenden Funktionen* wie oben beschrieben.
- Testen Sie *Rand- und Fehlerfälle*.
- Achten Sie bei Vergleichen von **`double`**-Werten darauf eine *geeignete Fehlerschranke* anzugeben.

- Achten Sie auf eine möglichst „*offensive Programmierweise* “ bei Ihren Tests, d.h. testen Sie die (noch nicht existente) Implementierung von `Fraction` möglichst intensiv durch *Sonderfälle und ungültige Parameter*.
- Um die Implementierung der Tests angenehmer zu gestalten (Autovervollständigung, etc.), können Sie die Klasse `Fraction` *ohne Attribute und Implementierungen der Methoden* deklarieren.
- Um einen Bruch *vollständig zu kürzen* müssen Sie den *größten gemeinsamen Teiler* berechnen. Verwenden Sie dazu eine private Methoden `gcd(int p, int q)`, wobei `p` *und* `q` *positiv sein müssen*:

```
private int gcd(int p, int q){
  return (q != 0 ? gcd(q, p % q) : p);
}
```

Implementierung der Klasse `Fraction`

Implementieren Sie nun die Klasse `Fraction` wie folgt:

- Implementieren Sie *Methode für Methode* und testen Sie nach jedem Schritt Ihre Implementierung mit den JUnit-Tests der entsprechenden Methode.
- Sind die Tests einer Methode *erfolgreich*, fahren Sie mit der *nächsten Methoden* fort.
- Welche *Beobachtungen* machen Sie bei der Implementierung?
  - Decken Ihre Tests *Fehler* auf? Wenn ja, *wieviele*?
  - Haben Sie *Tests* während Ihrer Implementierung *korrigieren* müssen?
  - Wie schätzen Sie die Code-Qualität Ihrer Implementierung von `Fraction` gegenüber der *Implementierung* von `Vector` *vor und nach Ausführung* der `Vector`-JUnit-Tests ein?