Computergrafik und Visualisierung II Belegaufgabe

Huy Tao (41652), Sven Pieper (41887)

07.05.2018

1 Einführung

Aufgabe unserer Belegarbeit war die Erstellung verschiedener Klassen: Vektor2D, Vektor3D und LineareAlgebra. Weiterhin sollten sinnvolle Hilfsmethoden angeboten werden. Folgende Funktionen sollten für Vektor2D sowie Vektor3D implementiert werden: setPosition, isNullVector, add, sub, mult, div, isEqual, isNotEqual, length und normalize.

Für die Klasse LineareAlgebra waren folgende Funktionen gefordert: add, sub, mult, div, isEqual, isNotEqual, length, normalize, euklDistance, manhattanDistance, crossProduct, dotProduct, cosEquation, sinEquation, angleRad, angleDegree, radToDegree, degreeToRad, determinante, abs und show.

All diese Funktionen sollten testgetrieben entwickelt werden. Hierfür wurde Test Driven Development benutzt, welches im Folgenden erläutert wird.

2 Test Driven Development (TDD)

Beim Test Driven Development handelt es sich um eine Methode der Softwareentwicklung, die den Test von Anfang an miteinbezieht.

Die gewöhnliche Softwareentwicklung schlägt vor, dass man zuerst phasenweise die Kundenanforderungen umsetzt und dann am Ende den Code unabhängig testet. Hier werden die Tests hauptsächlich am Ende der Entwicklung erstellt und ausgeführt. Test Driven Development zeichnet sich damit aus, diese Herangehensweise umzudrehen. Der Entwickler überlegt sich bereits vor der eigentlichen Programmierung der Funktionen, welche Anforderungen die Software erfüllen muss und welche Tests sinnvoll sind. Anhand der Testergebnissen, erfolgt die schrittweise Programmierung in kurzen Abschnitten. Jeder Zyklus erweitert das Programm um weitere Eigenschaften oder Funktionen. Dabei wird wie folgt vorgegangen:

1. Test erstellen
   Für jede neue Softwareanforderung wird ein neuer Test generiert.

2. Test durchführen
   Es ist noch kein Programmcode geschrieben worden, somit wird dieser Test fehlschlagen. Dieser Test dient lediglich zur Überprüfung der Testumgebung.

3. Code schreiben und anpassen
   Als nächstes wird ein möglichst simpler und minimalistischer Code geschrieben, der den Test bestehen muss. Funktionen weiterer Phasen sollen nicht vorweg genommen werden.

4. Neuen Test durchführen
   Ein weiterer Test wird geschrieben, dieser beinhaltet erweiterte Anforderungen an den Code. Nun werden die Schritte 2 bis 4 so oft wiederholt bis alle Anforderungen erfüllt sind.

5. Refaktorsierung
   Im letzten Schritt wird der Code durch Refaktorisierung vereinfacht und verständlich gemacht. Hier empfiehlt es sich erneut den Test durchzuführen, um sicherzustellen dass durch die Refaktorisierung keine Funktionalitäten beschädigt wurden.

Die Tests sorgen für einen Fokus auf die Ziele der Software und deren Funktionalität. Zudem ist es leichter, weitere Funktionen hinzuzufügen und die Software zu erweitern – selbst für Entwickler, die nicht am Projekt beteiligt waren. Weiterhin wird Redundanz effektiv vermieden.

3 Beispiele zu TDD

Folgend soll das Prinzip von Test Driven Development anhand einiger Funktionen aus den Klassen Vektor2D/3D und LineareÄlgebra verdeutlicht werden.

3.1 Add-Funktion Vektor2D

Diese Funktion soll einen Vektor auf einen gegebenen Vektor addieren. Die X-und Y-Werte sollen paarweise behandelt werden.

Test 1

```
1  @Test
2  void testAdd() {
3          Vektor2D a = new Vektor2D(3.0, 4.0);
4          Vektor2D b = new Vektor2D(1.0, 1.0);
5          Vektor2D erg = new Vektor2D(4.0, 5.0);
6
7          a.add(b);
8
9          assertEquals(a.x, erg.x, 0.000000001);
10         assertEquals(a.y, erg.y, 0.000000001);
11 }
```

Code 1

```
1  public void add() {
2  }
```

Diskussion

Der Test schlägt fehl. Dies ist auch nicht verwunderlich, da der Code noch keinerlei Funktionalität beinhaltet. Im weiteren wurde der Code mit den minimalsten Funktionen bestückt, dass der Test glückt.

Code 2

```
1  public void add(Vektor2D summand) {
2          this.x += summand.x;
3          this.y += summand.y;
4  }
```

Diskussion

Nun ist der Test erfolgreich, alle geforderten Funktionen werden dem Test gerecht. Folgend müssen sich weitere Funktionen für den Code überlegt und etwaige Probleme bedacht werden. Speziell für die Add-Funktion ist zu nennen, dass ein Überlauf des Speichers auftreten kann und somit falsche Ergebnisse hervorkommen können. Ein neuer Test muss geschrieben werden.

Test 2

```
1  @Test
2  void testAdd() {
3          Vektor2D a = new Vektor2D(Double.MAX_VALUE, Double.MAX_VALUE);

4          Vektor2D b = new Vektor2D(2, 2);
5          Vektor2D erg = new Vektor2D(Double.MAX_VALUE+2, Double.MAX_VALUE+2);

6          a.add(b);
7          assertEquals(a.x, erg.x, 0.000000001);
8          assertEquals(a.y, erg.y, 0.000000001);
9  }
```

Diskussion

Der neue Test schlägt fehl, da der Code noch nicht auf die Problematik des Überlaufs angepasst wurde. Der Code wird nun stückweise angepasst.

Code 3

```
1  public void add(Vektor2D summand) {
2          if ((Double.MAX_VALUE - this.x) < summand.x || (Double.MAX_VALUE -
               this.y) < summand.y ) {
3                  System.out.println("Addition Overflow");
4          }
5          else {
6                  this.x += summand.x;
7                  this.y += summand.y;
8                  System.out.println("Addiction Accepted");
9          }
10 }
```

Diskussion

Der angepasste Code beseitigt die Problematik des Überlaufs und besteht den Test. Somit sind alle Anforderungen implementiert und der Code ist fertig.

3.2 Normalize-Funktion Vektor3D

Ein ausgewählter Vektor soll zu einem Einheitsvektor umgewandelt werden. Hierzu werden die Werte des Vektors durch die Länge des Vektors geteilt.

Test 1

```
1  @Test
2  void testNormalize() {
3          Vektor3D a = new Vektor3D(4, 3, 3);
4          Vektor3D a_norm = new Vektor3D(0.6859943405700353,
               0.5144957554275265, 0.5144957554275265);
5
6          a.normalize();
7
8          assertEquals(a.x, a_norm.x);
9          assertEquals(a.y, a_norm.y);
10         assertEquals(a.z, a_norm.z);
11 }
```

Code 1

```
1  public void normalize() {
2  }
```

Diskussion

Der Test schlägt fehl. Dies ist auch nicht verwunderlich, da der Code noch keinerlei Funktionalität beinhaltet. Im weiteren wurde der Code mit den minimalsten Funktionen bestückt, dass der Test glückt.

Code 2

```
1  public void normalize() {
2          double length = this.length();
3
4          this.x = this.x / length;
5          this.y = this.y / length;
6          this.z = this.z / length;
7  }
```

Diskussion

Nun ist der Test erfolgreich, alle geforderten Funktionen werden dem Test gerecht. Folgend müssen sich weitere Funktionen für den Code überlegt und etwaige Probleme bedacht werden. Speziell für die Normalize-Funktion ist zu nennen, dass das teilen durch Null nicht erlaubt ist. Somit darf der Vektor nicht die Länge Null besitzen.

Test 2

```
@Test
void testNormalize() {
        Vektor2D a = new Vektor2D();

        a.normalize();
        if(a.x != a.x) {
                assertTrue(true);
        }
        else if(a.x != 0 || a.y != 0) {
                assertTrue(false);
        }
}
```

Diskussion

Der neue Test schlägt fehl, da der Code noch nicht auf die Problematik der Division durch Null angepasst wurde. Der Code wird nun stückweise angepasst.

Code 3

```
public void normalize() {
        double length = this.length();
        if(length == 0) {
                System.out.println("Normalisierung nicht mögich da Vektor
                    länge Null");
        }
        else {
                this.x = this.x / length;
                this.y = this.y / length;
                this.z = this.z / length;
                System.out.println("Vektor wurde normalisiert");
        }
}
```

Diskussion

Der angepasste Code beseitigt die Problematik und besteht den Test. Somit sind alle Anforderungen implementiert und der Code ist fertig.

3.3 Mult2D-Funktion LineareAlgebra

Die Werte des vorgebenen Vektors sollen paarweise mit einem double-Wert multipliziert werden. Die erzeugten Werte werden in einen neuen Vektor geschrieben.

Test 1

```
1  @Test
2  void testMult2D() {
3          Vektor2D a = new Vektor2D(2,2);
4          double b = 2;
5
6          Vektor2D erg = LineareAlgebra.mult(a, b);
7
8          assertEquals(4, erg.x, 0.0000001);
9          assertEquals(4, erg.y, 0.0000001);
10 }
```

Code 1

```
1  public static Vektor2D mult() {
2  }
```

Diskussion

Der Test schlägt fehl. Dies ist auch nicht verwunderlich, da der Code noch keinerlei Funktionalität beinhaltet. Im weiteren wurde der Code mit den minimalsten Funktionen bestückt, dass der Test glückt.

Code 2

```
1  public static Vektor2D mult(Vektor2D a, double b) {
2          return new Vektor2D(a.x * b, a.y * b);
3  }
```

Diskussion

Nun ist der Test erfolgreich, alle geforderten Funktionen werden dem Test gerecht. Folgend müssen sich weitere Funktionen für den Code überlegt und etwaige Probleme bedacht werden. Auch hier ist zu sagen, dass die Problematik des Überlaufs besteht. Dies muss abgefangen werden, hierzu wird ein neuer Test geschrieben.

Test 2

```
1  @Test
2  void testMult2D() {
3          Vektor2D a = new Vektor2D(Double.MAX_VALUE, Double.MAX_VALUE);
4          double b = 2;
5
6          Vektor2D erg = LinearеAlgebra.mult(a, b);
7
8          assertEquals(0, erg.x, 0.0000001);
9          assertEquals(0, erg.y, 0.0000001);
10 }
```

Diskussion

Der neue Test schlägt fehl, da der Code noch nicht auf die Problematik des Überlaufs angepasst wurde. Der Code wird nun stückweise geändert.

Code 3

```
1  public static Vektor2D mult(Vektor2D a, double b) {
2          if(a.x != 0 && ((a.x*b) / a.x) != b || a.y != 0 && ((a.y*b) / a.y)
               != b) {
3                  System.out.println("2D Mult Overflow, created Nullvektor");
4                  return new Vektor2D();
5          }
6          else {
7                  System.out.println("2D Mult Accepted");
8                  return new Vektor2D(a.x * b, a.y * b);
9          }
10 }
```

Diskussion

Der angepasste Code beseitigt die Problematik und besteht den Test. Somit sind alle Anforderungen implementiert und der Code ist fertig.

4 Fazit und Schlussworte

Die Bewältigung der Belegaufgabe mit Test Driven Development hat uns einen neuen Blick auf die Erstellung von Software gegeben. Die Herangehensweise schafft einen klaren und engen Fokus und mindert somit die Komplexität der Tests sowie der Implementierung und schafft somit eine höhere Qualität. Jedoch kann der Einstieg für Programmieranfänger ungewöhnlich sein, da man sich fragt, wie man etwas testen soll, das noch garnicht vorhanden ist.