Skyline

Um diese Aufgabe zu lösen, verwende ich Java-Funktionen (die auf Grund des Satzes des Pythagoras arbeiten) um die Entfernung aller vier Eckpunkte der Gebäude zum Achsenursprung (0;0) zu errechnen. Dann vergleiche ich diese Entfernungen, wähle die Geringste aus und teile sie durch 100 („Rastereinheit“). Java-Programme geben bei Ganzzahldivisionen immer den abgerundeten Wert aus, zum Beispiel ergeben 5 / 100 = 0, 99 / 100 = 0 und 250 / 100 = 1. Dies multipliziere ich mit einem „Erhöhungsfaktor“, der in der Aufgabe allerdings 1 ist und deshalb vernachlässigt werden kann. Dazu addiere ich dann die Grundhöhe von der ausgegangen wird, in der Aufgabe also 100.

Im Beispiel:

1. Ausgangskoordinaten (-80;100), (-20;100), (-20;160) und (-80;160)
2. Entfernung zum Achsenursprung 128, 101, 161 und 178
3. Kleinster Abstand 101
4. Ganzzahldivision durch 100 = 1
5. (Multiplikation mit 1 = 1)
6. Addition mit 100 = 101

Maximale Höhe im ersten Beispiel ist 101 Meter.

```
// Koordinatenursprung
final Point mitte = new Point();

// kleinste Entfernung der Eckpunkte zu Koordinatenursprung
double minimumEntfernung = Double.MAX_VALUE;
for(final Point koordinate : grundriss)
      minimumEntfernung = Math.min(minimumEntfernung, ↵
mitte.distance(koordinate));

// Entfernung in Rasterschritten
final int minimumEntfernungRaster = (int)minimumEntfernung / entfernungRaster;

// Gesamthöhe
return minimumEntfernungRaster * rasterErhoehung + ausgangsHoehe;
```

Als erstes wird der Achsenursprung `mitte` auf (0;0) gesetzt (`new Point()`). Dann wird eine lokale Variable definiert, die die maximale Entfernung enthalten wird und auf den größtmöglichen Wert gesetzt, damit sie später sicher ersetzt wird. Darauf folgend wird durch ein Array aus den vier Eckkoordinaten des Grundrisses iteriert und deren jeweilige Entfernung der Variable `minimumEntfernung` zugewiesen, sollte sie kleiner als der voherige Wert sein. Schließlich wird die kleinste Entfernung in „Rastereinheiten“ umgerechnet und die Maximalhöhe errechnet.

Verben

Für die Aufgabe zur Konjugation von Verben habe ich mich auf ein objektorientierteres Modell als auf den prozeduralen Ansatz der „Skyline“-Aufgabe gestützt. Ich benutze dazu ein Interface das eine Methode zur Erkennung und eine Methode zum Umwandeln in den Stamm bietet. Dieses Interface wird von allen Regeln die ich implementiert habe erweitert. Zur Umwandlung in den Stamm werden nacheinander alle Regeln in einer festen Reihenfolge überprüft und sobald eine Regel zutrifft wird sie verwendet und der Stamm zurückgegeben.

Beispiel „schweigend“:

1. Es existiert keine Sonderform für schweigend
2. Es ist kein Verb im Präsens der zweiten Person Singular, da es nicht auf „st“ endet
3. Es ist kein Verb im Er-Präteritum, da es nicht auf „te“ endet
4. Es ist kein Verb im Er-Perfekt, da es nicht mit „ge“ anfängt und mit „t“ aufhört (Regulärer Ausdruck `ge.+t`)
5. Es ist ein Verb im Partizip Präsens Aktiv, da es auf „end“ endet. Es wird schweigend ohne die letzten drei Buchstaben (schweig) zurückgegeben, wozu „en“ hinzugefügt wird, um aus dem Stamm die Grundform zu machen (schweigen)

Dies funktioniert bei Verben wie sagst (sagen), leitete (leiten), geforscht (forschen), schweigend (schweigen), trag (tragen), Gebet (beten), arbeite (arbeiten), genehmigt (genehmigen), gehört (hören), mitgehalten (mithalten) und mitgestalten (mitgestalten), scheitert aber an starken Verben wie buk (buken statt backen).

Wichtige Stellen im Quellcode:

- Das Regelinterface:

```
public interface Regel {
	public boolean kannVerwendetWerden(String verbForm);

	public String getStamm(String verbForm);
}
```

- Eine Regel (Perfekt dritte Person Singular):

```
public class PerfektEr implements Regel {
	@Override
	public boolean kannVerwendetWerden(String verbForm) {
		return verbForm.matches("ge.+t");
	}

	@Override
	public String getStamm(String verbForm) {
		return verbForm.substring(2, verbForm.length() - 1);
	}
}
```

Es wird mit einem regulären Ausdruck überprüft, ob die Regel angewandt werden kann, und falls das möglich ist, der String auf die nötige Länge geschnitten.

- Die Prüfschleife:

```
for(final Regel r : regeln) {
      if(r.kannVerwendetWerden(form)) {
            return r.getStamm(form) + "en";
      }
}
return form;
```

SVG-Kochkurve

Für die Kochkurve habe ich einen objektorientierten Ansatz gewählt, der aus zwei Teilen besteht. Der erste Teil ist eine einfache SVG-Implementierung, die aus einer Bildklasse, einem Elementinterface und einer das Elementinterface erweiternden Linienklasse besteht. Zu einem SVG-Bild können unterschiedliche Elemente (in meinem Programm nur Linien) hinzugefügt werden und schließlich das gesamte Bild in eine Datei ausgegeben werden. Darauf werde ich hier nicht weiter eingehen. Der zweite Teil besteht aus der Klasse `KochLinie`, die eine einzelne Linie des Bildes darstellt. Diese besitzt eine Methode um sich selbst und alle Kind-Linien als SVG-Linien zurückzugeben. Diese Methode überprüft erst, ob die gewünschte Linientiefe erreicht ist, wenn ja, gibt sie nur sich selbst zurück, wenn nein berechnet sie vier Kindlinien (auch Objekte der Klasse `KochLinie`) und ruft die selbe Funktion für diese Linien auf.

```
Collection<SVGLine> getChildLines(int linesToGo, double scale) {
	if(linesToGo == 0) {
		return Collections.singleton(getLine(scale));
	} else {
		final Collection<SVGLine> lines = new ArrayList<>();
		for(final KochLine kl : generateChildLines()) {
			// rufe diese Funktion für alle Kind-Funktionen auf
			lines.addAll(kl. getChildLines(linesToGo - 1, scale));
		}
		return lines;
	}
}
```

Die Funktion `generateChildLines`:

```
private Collection<KochLine> generateChildLines() {
	// teile diese Linie in drei Teile, füge den ersten und den letzten Teil
	// sowie die beiden abstehenden Linien zu einem Array zusammen und gib
	// dieses zurück
	return Arrays.asList(new KochLine(facingDirection, beginPoint, length / ↵
3D, reverse), new KochLine(facingDirection, new ↵
Point2D.Double(beginPoint.getX() + Math.cos(facingDirection * Math.PI / 3) * ↵
length / 3 * 2, beginPoint.getY() + Math.sin(facingDirection * Math.PI / 3) * ↵
length / 3 * 2), length / 3D, reverse), getSecondLine(), getThirdLine());
}
```

Diese Funktion verwendet etwas Trigonometrie um die Kindlinien zu berechnen.

Um nun aus einem bestehenden Bild die Informationen wie Anfangs- und Endpunkt hinauszulesen, gehe ich einfach mit der von Haus aus vorhandenen Java-XML-Bibliothek durch alle Linien des Bildes, speichere die größte X-Koordinate, die kleinste X-Koordinate, die größte Y-Koordinate sowie die Anzahl der Linien, berechne mit dem Logarithmus die Stufe der Kochkurve und berechne aus diesen Informationen schließlich ein neues Bild.

Ausgangsbild und fünf generierte Bilder: