252-0027-00: Einführung in die Programmierung

Übungsblatt 11

Abgabe: 10. Dezember 2019, 10:00

Checken Sie die neue Übungs-Vorlage aus. Vergessen Sie nicht, Tests zu schreiben!

Aufgabe 1: Notenauswertung (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin.

Die Klasse `Service` stellt verschiedene Analysen für Prüfungsergebnisse von S Studierenden zur Verfügung. Die Liste von Ergebnissen besteht aus S Einträgen, also jeweils ein Eintrag pro Student/in. Jeder Eintrag besteht aus einer Zeile und enthält (in dieser Reihenfolge):

1. die Immatrikulationsnummer des Studierenden (ein identifizierender positiver `int`-Wert)
2. drei Noten (drei reelle Zahlen im Bereich von 1.0 bis 6.0, getrennt durch Leerzeichen)

Die drei Noten gehören zu den Fächern *Fach 1*, *Fach 2* und *Fach 3*. Zusätzliche Leerzeilen und -zeichen sollen ignoriert werden. Eine Beispiel für eine Liste für 3 Studierende ist:

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Ihre Aufgabe ist es nun, die `Service`-Klasse und ihre Analysen zu implementieren. Die `Service`-Klasse hat einen Konstruktor, welcher alle Prüfungsergebnisse aus einem `Scanner` auslesen und damit das `Service`-Objekt initialisieren soll. Das Objekt soll so initialisiert werden, dass die vorgegebenen Methoden ihre Analysen durchführen können. Sie dürfen dabei Attribute und zusätzliche Methoden frei bestimmen.

a) Implementieren Sie nun die Methode `critical()`, welche die zwei Argumente `bound1` und `bound2` erwartet. Die Methode sucht alle "kritischen" Fälle und gibt eine Liste dieser Studierenden zurück. Ein Student darf maximal einmal in der Liste vorkommen. Die zurückgegebene Liste besteht aus den Immatrikulationsnummern dieser Studierenden (in beliebiger Reihenfolge).

Ein/e Student/in gilt als kritisch, wenn die Note in *Fach 1* $\leq$ `bound1` und die Summe der Noten für *Fach 2* und *Fach 3* kleiner als `bound2` ist.

Für das obige Beispiel gäbe `critical(4, 8)` eine Liste mit dem Element `111111005` zurück.

b) Implementieren Sie nun die Methode `top()`, welche die Studierenden mit den besten Ergebnissen zurückgeben soll. Der Parameter `limit` bestimmt die maximale Anzahl der zurückzugebenden Studierenden. Falls weniger Ergebnisse als `limit` existieren, sollen einfach alle gefundenen zurückgegeben werden.

Der Rückgabewert der Methode ist wieder eine Liste der Immatrikulationsnummern. Ein Student darf maximal einmal in der Liste vorkommen. Diese Liste soll absteigend nach der Leistung sortiert sein (der/die Student/in mit dem besten Ergebnis zuerst). Dabei gilt, dass ein Ergebnis A besser ist als ein Ergebnis B, wenn die Summe aller Noten von A grösser ist als die Summe der Noten von B. Sind die Summen gleich, sind die Ergebnisse gleich gut (und die Reihenfolge in der Liste somit egal).

Für das obige Beispiel gäbe `top(2)` entweder die Liste `[111111004, 111111006]` oder die Liste `[111111004, 111111005]` zurück (beide wären richtig).

In der Klasse `ServiceTest` finden Sie einen ersten kleinen JUnit-Test als Starthilfe. Ausserdem dürfen Sie folgende Annahmen machen: Der Parameter `limit` ist immer grösser als 0 und die beiden Parameter für `critical()` sind immer im Bereich von 0.0 bis 100.0.

Tipp: Verwenden Sie die `Collections.sort(...)` Funktion einer Kollektion, welche mit `import java.util.Collections;` importiert werden kann. Beachten Sie, dass dafür die Klasse, welche Sie für die Elemente der Kollektion verwenden, das Interface `Comparable<T>` (`T` sollte die Klasse selber sein), und damit auch eine Funktion `compareTo` implementieren muss. Diese Funktion nimmt eine Instanz der selben Klasse und gibt 0 zurück, wenn `this` und das Argument gleich sind, gibt 1 zurück, wenn this grösser als das Argument ist, und gibt -1 zurück, wenn `this` kleiner als das Argument ist.

Aufgabe 2: Expression Evaluator

In dieser und in folgenden Übungen werden Sie eine Reihe von Programmen schreiben, welche andere Programme interpretieren, kompilieren oder (in kompilierter Form) ausführen. Die Programmiersprachen definieren wir selber.

Als Einstieg schreiben Sie ein Programm, welches mathematische Ausdrücke (*expressions*) auswertet. Die Ausdrücke bestehen aus Zahlen, Variablen, Operatoren wie $+$ oder $-$ und einfachen Funktionen wie $\sin()$ oder $\cos()$. Die genaue Syntax für diese Ausdrücke finden Sie als EBNF-Beschreibung in Abbildung 1.

Ein Programm, das Ausdrücke auswertet, muss natürlich entscheiden, ob eine gegebene Zeichenkette überhaupt ein gültiger Ausdruck ist[1]. Das nennt man *parsen* und ein solches Programm heisst *Parser*. Aus einer EBNF-Beschreibung wie dieser kann man einfach einen Parser erstellen[2]:

[1] Ähnlich wie Sie, wenn Sie mit einer Tabelle überprüfen, ob ein Symbol einer EBNF-Beschreibung entspricht.
[2] Im Allgemeinen, d.h. für gewisse andere EBNF-Beschreibungen, ist das leider nicht möglich.

digit ⇐ `0` | `1` | ... | `9`
char ⇐ `A` | `B` | ... | `Z` | `a` | `b` | ... | `z`
num ⇐ *digit* { *digit* } [`.` *digit* { *digit* }]
var ⇐ *char* { *char* }
func ⇐ *char* { *char* } `(`
op ⇐ `+` | `-` | `*` | `/` | `^`
open ⇐ `(`
close ⇐ `)`

atom ⇐ *num* | *var*
term ⇐ *open expr close* | *func expr close* | *atom*
expr ⇐ *term* [*op term*]

Abbildung 1: **EBNF-Beschreibung** von *expr*

```
/* checks if the next tokens form a valid term */
void parseTerm(...) {
    if(next token is a "open") {
        consume "open" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else if(next token is a "func") {
        consume "func" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else {
        // check if the tokens are a valid atom:
        parseAtom(...);
    }
}
```

Abbildung 2: Parser-Methode für *term*

```
/* evaluate the next tokens as a term */
double evalTerm(...) {
    if(next token is a "open") {
        consume "open" token
        double val = evalExpr(...);
        check whether next token is a "close" & consume
        return val;
    }
    else if(next token is a "func") {
        consume "func" token
        double arg = evalExpr(...);
        check whether next token is a "close" & consume
        double result = apply function to arg
        return result;
    }
    else {
        return evalAtom(...);
    }
}
```

Abbildung 3: Evaluator-Methode für *term*

- Regeln werden zu Methoden.
- Alternativen werden zu `if`-Anweisungen.
- Regeln auf der RHS werden zu Methodenaufrufen.

Man unterscheidet dabei zwischen zwei Arten von Regeln: *Parser-Regeln* und *Tokenizer-Regeln*. Zuerst teilt ein *Tokenizer* die Zeichenkette aufgrund der Tokenizer-Regeln in eine Reihe von Tokens auf. In unserer EBNF-Beschreibung sind die Tokenizer-Regeln rot dargestellt. Die grauen Regeln werden zwar intern vom Tokenizer verwendet, aber erzeugen keine eigenen Tokens. Zum Beispiel erzeugt die Zeichenkette "`sin(1 + x) * 3.14`" die folgende Reihe von Tokens:

func : `sin(` *num* : `1` *op* : `+` *var* : `x` *close* : `)` *op* : `*` *num* : `3.14`

Danach entscheidet der Parser aufgrund der Parser-Regeln (oben in Schwarz dargestellt), ob eine solche Reihe von Tokens einen gültigen Ausdruck darstellt. Abbildung 2 zeigt, wie die Parser-Methode für *term* aussehen könnte.

a) In der Übungsvorlage finden Sie eine `Tokenizer`-Implementation, eine Vorlage für den `ExprParser` und eine `EvaluatorApp` mit einer `main()`-Methode. Diese parst die vom Benutzer eingegebenen Zeichenketten und gibt an, ob sie gültige Ausdrücke sind. Wenn der Benutzer "exit" eingibt, terminiert das Programm. Ihre Aufgabe ist es, den `ExprParser` zu schreiben.

   Erstellen Sie in der schon vorgegebenen `parse(String)`-Methode eine `Tokenizer`-Instanz. Die Methoden des `Tokenizers` sind denen der `Scanner`-Klasse nachempfunden. Sie können also die `hasNext*()`-Methoden verwenden, um zu prüfen, welche Art von Token als nächstes kommt, und die `next*()`-Methoden, um Tokens zu "konsumieren". Schreiben Sie die nötigen `parse*(...)`-Methoden, eine für jede Parser-Regel. Die erste Ihrer `parse*(...)`-Methoden rufen Sie von `parse(String)` aus auf. Diese Methoden sollen eine `EvaluationException` mit einer sinnvollen Fehlermeldung werfen, falls die Zeichenkette kein gültiger Ausdruck ist. Falls z.B. nach "`(`" und einer *expr* das Token "`10`" statt "`)`" folgt, könnte die Fehlermeldung lauten:

   ```
   Syntax error: unexpected token '10', expected ')'
   ```

b) Um aus dem `ExprParser` einen `ExprEvaluator` zu machen, kann man die Methoden so ändern, dass sie im selben Zug das Resultat berechnen. Jede Methode überprüft dann nicht nur, ob die nächsten Tokens der Regel entsprechen, sondern gibt auch gleich den Wert des entsprechenden Ausdruck-Teils zurück. Dies sehen Sie in Abbildung 3.

   Benennen Sie die Klasse und die Methoden um[3], so dass sie die neue Funktionalität widerspiegeln. Nun können Sie entscheiden: Erstens, welche Funktionen sind erlaubt? Für Aufgabe 3 sollten Sie mindestens sin(), cos() und tan() unterstützen, aber auch andere Funktionen wie abs() oder log() könnten später Spass machen[4]. Zweitens können Sie entscheiden, wie Sie mit Variablen umgehen. Sie sollten mindestens eine "x"-Variable unterstützen, und wir empfehlen, dass Sie den Wert dafür dem `ExprEvaluator`-Konstruktor übergeben. Sie sollten eine Exception werfen, falls unbekannte Funktionen oder Variablen in einem Ausdruck vorkommen.

   Am Schluss sollte die `EvaluatorApp` das Resultat der eingegebenen Ausdrücke ausgegeben, statt nur zu sagen, ob sie gültig sind. Wenn Sie wollen, können Sie dem Benutzer auch die Möglichkeit geben, Werte für Variablen zu definieren.

xkcd: (by Randall Munroe (CC BY-NC 2.5)

Aufgabe 3: Funktions-Plotter

Mit dem `ExprEvaluator` können wir ein praktisches Programm schreiben: einen Funktions-Plotter. Dieser interpretiert einen eingegebenen Ausdruck als Funktion $f(x)$, wertet $y = f(x)$ für verschiedene x aus und zeichnet die resultierenden (x, y)-Punkte.

In der Übungsvorlage finden Sie eine neue `PlotterWindow`-Klasse, welche eine Erweiterung der bekannten `Window`-Klasse ist. Der Unterschied besteht darin, dass das Fenster ein Eingabefeld enthält, wo der Benutzer eine Funktion eingeben kann. Diese kann mit der `getFunction()`-Methode abgerufen werden. Vervollständigen Sie das Programm `PlotterApp`.

a) Im ersten Schritt sollen Sie eine Koordinaten-Transformation von den Ur-Koordinaten (x, y) (in welchen $f(x)$ definiert ist) zu den GUI-Koordinaten (X, Y) implementieren.

   Dem `PlotterApp`-Konstruktor werden Werte für x_{min}, x_{max}, y_{min} und y_{max} übergeben. Diese Werte geben an, welcher Teil des Ur-Koordinatensystems im Fenster sichtbar ist. Hier sind einige Beispiele für die Transformation (w und h stehen für die Breite und Höhe des Fensters):

Ur-Koordinaten (x, y)	$\rightarrow$	(X, Y) GUI-Koordinaten	
$(0, 0)$		$(\frac{w}{2}, \frac{h}{2})$	falls $x_{\text{min}} = -x_{\text{max}}$ und $y_{\text{min}} = -y_{\text{max}}$
$(x_{\text{min}}, y_{\text{min}})$		$(0, h)$	
$(x_{\text{max}}, y_{\text{max}})$		$(w, 0)$	

[3] Verwenden Sie dafür die *Rename*-Funktion von Eclipse, welche Sie in der Übungsstunde gesehen haben.
[4] Schauen Sie sich die `Math`-Klasse für weitere Kandidaten an (oder implementieren Sie selber welche).

Implementieren Sie zwei Methoden `toGuiX()` und `toGuiY()`, welche die Transformation berechnen. Erweitern Sie dann die `PlotterApp.run()`-Methode so, dass sie die x- und y-Achse des Ur-Koordinatensystems zeichnet. Plotten Sie ein paar Punkte im Ur-Koordinatensystem um zu sehen, ob Ihre Berechnung korrekt ist (oder besser, schreiben Sie Tests).

Tipp: Die aktuelle Grösse des Fensters (bzw. des Teils, auf dem Sie zeichnen können) bekommen Sie wie gewohnt mit `window.getWidth()` und `window.getHeight()`.

b) Erweitern Sie `PlotterApp` so, dass die Funktion geplottet wird. Iterieren Sie dazu über alle Werte der X-Achse im GUI-Koordinatensystem ($0 \leq X < w$), finden Sie für jedes X das dazugehörige x im Ur-Koordinatensystem und berechnen Sie $y = f(x)$. Transformieren Sie diese y-Werte zurück ins GUI-Koordinatensystem und verbinden Sie die (X, Y)-Punkte mit Linien. Zusätzlich zur `toGuiY()`-Methode brauchen Sie dafür auch eine `fromGuiX()`-Methode.

Um $y = f(x)$ zu berechnen, brauchen Sie natürlich Ihren `ExprEvaluator`. Übergeben Sie ihm in jeder Iteration den aktuellen x-Wert. Falls bei der Evaluation ein Fehler auftritt, sollen Sie die Fehlermeldung auf dem Fenster ausgeben.

c) **Optional:** Erweitern Sie Ihren Plotter um zusätzliche Features, zum Beispiel:

- Achsen-Striche und -Beschriftung: Zeichnen Sie zusätzlich zu den Achsen Haupt- und Nebenstriche und fügen Sie die dazugehörigen Werte hinzu.
- Automatische Skalierung der y-Achse: Berechnen Sie y_{min} und y_{max} basierend auf den erhaltenen y-Werten in jeder Iteration der `while(window.isOpen())`-Schleife neu.
- Machen Sie Ihre `PlotterApp` interaktiv. Erlauben Sie z.B. Rein- und Raus-zoomen oder Verschieben, oder zeigen Sie für den x-Wert, auf den die Maus zeigt, den y-Wert an.

Aufgabe 4: Interfaces

In dieser Aufgabe üben Sie den Umgang mit Java-Interfaces. Das Besondere an Interfaces ist, dass eine Klasse nicht nur eines, sondern beliebig viele davon implementieren kann.

Die Bibliothek, welche die `Window`-Klasse enthält, enthält auch einige Interfaces. Diese erlauben es, GUI-Programme modularer zu schreiben: Während Sie bisher alle Zeichenbefehle und Interaktionen in der `while(window.isOpen())`-Schleife durchführen mussten, können Sie mit diesen Interfaces verschiedene *Komponenten* erstellen, welche sich selbstständig zeichnen und auf Benutzereingaben reagieren. Ihre Aufgabe ist es, die interaktive Karte vom letzten Übungsblatt mit solchen Komponenten zu implementieren und zusätzlich eine Taste, welche in den "Nachtmodus" wechselt, hinzuzufügen:

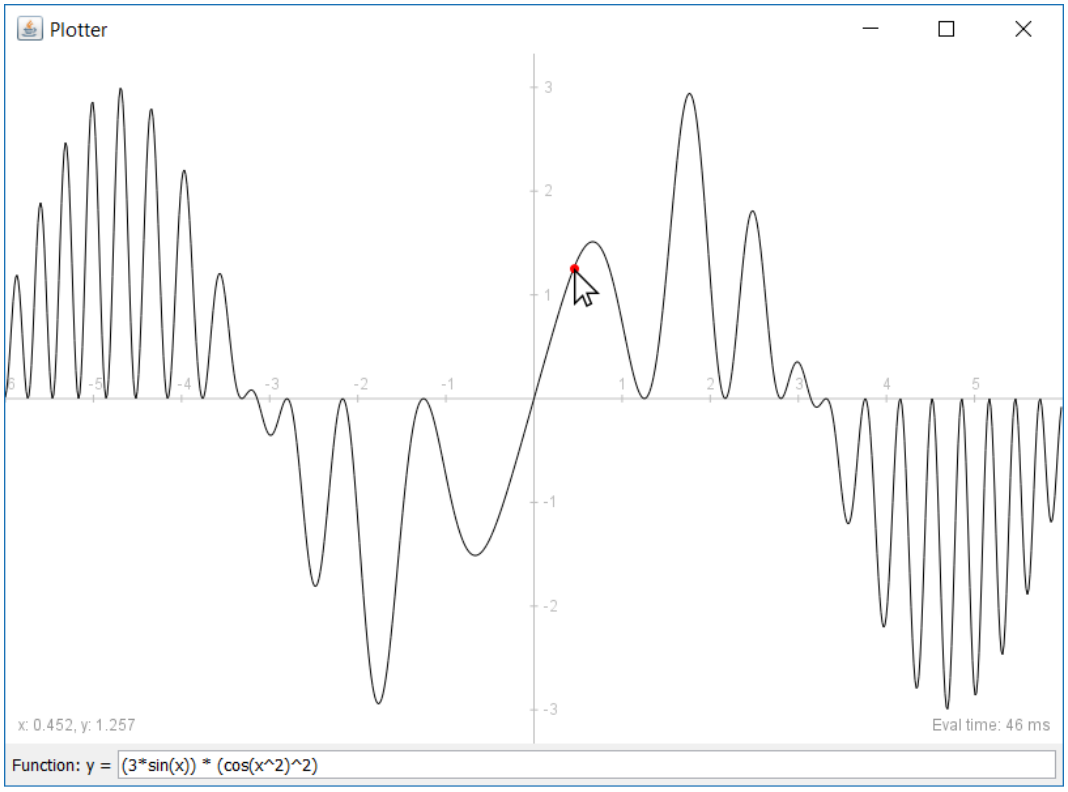
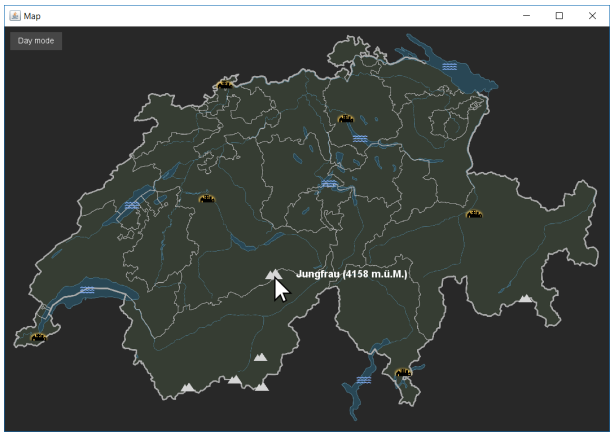


Abbildung 4: `PlotterApp` mit Zusatz-Features



In der Vorlagen finden Sie weiter die `SwissMap`- und die verschiedenen POI-Klassen. Wie in der letzten Übung wird in der Vorlage nur der Kartenhintergrund gezeichnet. Allerdings geschieht dies nun mithilfe des `Drawable`-Interfaces, welches von `SwissMap` implementiert wird. Dies bedeutet, dass die `SwissMap`-Deklaration den Teil "`implements Drawable`" enthält und dass `SwissMap` die `Drawable.draw()`-Methode implementiert. Beachten Sie, dass sich das `Drawable`-Interface, sowie weitere Interfaces und Klassen sich im `gui.component`-Package befinden und wie die `Window`-Klasse importiert werden müssen. Sie können dafür z.B. `import gui.component.*;` verwenden.

Um das `SwissMap`-Objekt im Fenster anzuzeigen, wird es in der `show()`-Methode als Komponente dem Fenster hinzugefügt (`window.addComponent(this);`). Für alle so hinzugefügten `Drawable`-Komponenten wird in `Window.refresh...()` die `draw()`-Methode aufgerufen. Ihre Aufgabe ist es, das ganze Programm mit solchen Komponenten zu implementieren, so dass in der `while(window.isOpen())`-Schleife nur `window.refresh...()` aufgerufen werden muss.

a) Ändern Sie die `PointOfInterest`-Klasse so ab, dass sie ebenfalls `Drawable` implementiert. Sie können die `draw()`-Methode direkt in `PointOfInterest` implementieren, oder in jeder der Subklassen `City`, `Mountain` und `Lake` separat[5]. Im zweiten Fall müssen Sie aber eine (leere) `draw()`-Implementierung in `PointOfInterest` erstellen, um den Compiler zufriedenzustellen[6].

   Die `draw()`-Methode soll den entsprechenden POI auf das übergebene `window` zeichnen. Wenn Sie wollen, können Sie dafür `drawImageCentered()` und die PNG-Bilder in Ihrem Projektordner verwenden. Um den POI an der richtigen Position darzustellen, brauchen Sie die `toGuiX()`- und `toGuiY()`-Methoden von `SwissMap`. Erstellen Sie deshalb in der `PointOfInterest`-Klasse ein Feld, wo eine Referenz zur `SwissMap`-Instanz gespeichert werden kann, und ändern Sie alle nötigen Konstruktoren und Konstruktoren-Aufrufe so ab, dass die Instanz übergeben wird. In der `SwissMap.show()`-Methode könnten die Instanzierungen der POI-Klassen z.B. so aussehen:

```
new City(this, "Zürich", 683354, 247353, 396030, 91.88)
```

   Damit die POIs gezeichnet werden, müssen sie (gleich wie die `SwissMap`) als Komponenten zur `Window`-Instanz hinzugefügt werden. Starten Sie das Programm und stellen Sie sicher, dass alles richtig angezeigt wird, bevor Sie weiterfahren.

[5] Sie können auch Teile davon in `PointOfInterest` und den Rest in den Subklassen implementieren, was vor allem später, wenn Sie auch die Beschreibung anzeigen, Sinn macht.

[6] Hier könnte man wiederum `abstract`-Klassen verwenden.

b) Erweitern Sie `PointOfInterest` jetzt so, dass der Benutzer (wie in der letzten Übung) mit der Maus auf ein POI zeigen kann um dessen Beschreibung anzuzeigen. Dazu muss die Klasse zusätzlich das `Hoverable`-Interface implementieren, welches die beiden Methoden `onMouseEnter()` und `onMouseExit()` und zusätzlich `getBoundingBox()` deklariert. Mit letzterer Methode kann eine Komponente ihren interaktiven Bereich mithilfe eines `Rectangle`-Objekts angeben[7]. Wenn der Benutzer dann seine Maus in diesen Bereich hinein oder aus dem Bereich hinaus bewegt, wird `onMouseEnter()` bzw. `onMouseExit()` aufgerufen.

   Implementieren Sie `onMouseEnter()` und `-Exit()` also so, dass sich der POI merkt, ob im Moment gerade auf ihn gezeigt wird, und verwenden Sie diese Information dann beim Zeichnen in `draw()`, um die Beschreibung entweder anzuzeigen oder nicht.

c) Erstellen Sie als letztes eine neue Klasse `NightModeButton`, welche nicht nur `Drawable` und `Hoverable`, sondern auch `Clickable` implementiert und als Taste auf der Karte angezeigt wird. Wenn der Benutzer auf diese Taste klickt, soll die Karte in den "Nachtmodus" wechseln, welcher alle Komponenten in einer dunkleren Version anzeigt (und bei erneutem Klick zurück).

   `Clickable` deklariert die Methode `onLeftClick()`. Implementieren Sie sie so, dass sie ein Feld `nightMode` in der `SwissMap`-Instanz verändert (welche Sie wieder via Konstruktor dem `NightModeButton` übergeben sollten). Ändern Sie danach alle `draw()`-Methoden so ab, dass die Komponenten hell oder dunkel gezeichnet werden, abhängig vom `nightMode`-Feld der `SwissMap`. Erstellen Sie schliesslich in `SwissMap.show()` eine `NightModeButton`-Instanz und fügen Sie sie als Komponente dem Fenster hinzu.

   Beachten Sie, dass `Clickable` auch `onRightClick()` deklariert. Der Compiler zwingt Sie, diese Methode ebenfalls zu implementieren, aber Sie können sie leer lassen.

[7] Die `Rectangle`-Klasse befindet sich ebenfalls in `gui.component`.