Paradigmen der Programmierung
Praktikum 1 - Entwurfsmuster

In dieser Aufgabe soll ein Thermometer simuliert werden. Das Thermometer erfasst die aktuelle Temperatur über Messdaten von Sensoren. Diese Messdaten sollen erzeugt und später dekoriert bzw. gefiltert werden. Registrierte Beobachter sollen über Temperaturänderungen informiert werden. Einer der Beobachter soll ein Heizsystem sein, das mit unterschiedlichen Strategien auf Temperaturänderungen reagiert .

Empfohlene Literatur:

- Alle Screencasts zu Entwurfsmustern.
- Kapitel 22.1 "Das Strategiemuster" und 22.2 "Das Dekorierermuster" aus dem Buch Programmieren lernen mit Kotlin.
- Kapitel 1 "Welcome to Design Patterns", 2 "Observer Pattern" und 3 "Decorator Pattern" aus dem Buch Entwurfsmuster von Kopf bis Fuß.
- Chapter 2, 4 und 5 von Foundation Design Patterns von Eric Freeman und Elisabeth Robson.
- Die Videos zu Strategy (ep 1), Observer (ep 2) und Decorator (ep 3) von Christopher Okhravi auf YouTube.

In diesem Übungsblatt programmieren Sie mit Kotlin. Erledigen Sie alle Programmieraufgaben in IntelliJ.

Contents

1 Strategien für Temperaturwerte

In der ersten Aufgabe legen Sie zwei unterschiedliche Strategien an, um Temperaturwerte zu erhalten.

a) Definieren Sie eine Schnittstelle `Sensor`, welche die Methode `getTemperature(): Double` besitzt. Über diese Methode liefert ein Sensor eine bestimmte Temperatur zurück.

b) Es soll **drei konkrete Komponenten** geben, die die Schnittstelle `Sensor` implementieren und Temperaturwerte liefern. Die Komponenten *unterscheiden sich in dem Algorithmus*, nach dem die Temperaturwerte geliefert werden. Implementieren Sie folgende konkrete Komponenten:

- `RandomSensor`: liefert zufällige Temperaturwerte innerhalb eines Wertebereichs. Der Wertebereich wird über die beiden Eigenschaften `min` und `max` vom Typ `Double` festgelegt. Die beiden Eigenschaften werden im Konstruktor übergeben.
- `ConstantSensor`: liefert immer eine konstante Temperatur. Hierfür wird der im Konstruktor übergebene Temperaturwert verwendet.
- `IncreasingSensor`: liefert einen linear steigenden Temperaturverlauf. Hierfür wird zunächst eine Starttemperatur im Konstruktor übergeben. Diese Temperatur wird bei jedem Zugriff um `0.5` Grad erhöht.
- **Bonus** `SinusoidalSensor`: liefert einen sinusförmigen Temperaturverlauf. Informieren Sie sich hierfür über harmonische Schwingungen bzw. Sinusschwingungen. Als Parameter benötigen Sie die Amplitude, Frequenz und Phasenverschiebung (Veränderung über Zeit)[1].

Testen Sie alle Sensoren, indem Sie diese instanziieren und die `getTemperature`-Methode in einer Schleife aufrufen:

```
fun main() {
  val randomSensor = RandomSensor(min = 2.0, max = 8.0) // liefert
     zufällige Temperaturen zwischen 2.0 und 8.0 Grad
  println("Random Sensor")
  repeat(4) {
    println(randomSensor.getTemperature())
  }

  val constantSensor = ConstantSensor(temp = 21.5) // liefert jedes
     Mal 21.5 Grad
  println("Constant Sensor")
  repeat(4) {
    println(constantSensor.getTemperature())
```

[1] Die Verwendung vom SinusoidalSensor und eine entsprechende Konsolenausgabe finden Sie hier: `https://www.gm.fh-koeln.de/~dobrynin/me/sinusoidalSensor`

```
  }

  val increasingSensor = IncreasingSensor(startTemp = 15.0) // fängt
     bei 15 Grad an und erhöht jedes mal die Temperatur um 0.5 Grad
  println("Increasing Sensor")
  repeat(4) {
    println(increasingSensor.getTemperature())
  }
}
```

Die Konsolenausgabe kann beispielsweise so aussehen:

```
> Random Sensor
> 6.496897428041999
> 3.5319770622098154
> 6.142194856685258
> 7.782588831959013
> Constant Sensor
> 21.5
> 21.5
> 21.5
> 21.5
> Increasing Sensor
> 15.5
> 16.0
> 16.5
> 17.0
```

2 Strategien verwenden

In dieser Aufgabe sollen die ersten Vorteile der Strategie ersichtlich werden. Hierfür benötigen wir einen Client, der die Strategie verwendet.

a) Schreiben Sie eine Klasse `Thermometer`, die einen `Sensor` im Konstruktor entgegennimmt. Diese Variable sollte *veränderlich* sein, damit Sie die Strategie später austauschen können. Implementieren Sie die Methode `measure(times: Int)`, welche die `repeat`-Funktion verwendet, um `times` Mal die Temperatur vom Sensor abzufragen.

b) Erzeugen Sie ein Thermometer in der `main` Funktion. Übergeben Sie dem Thermometer eine der in Aufgabe 1 definierten Strategien. Rufen Sie jeweils die `measure` Methode auf und schauen Sie sich die Ausgaben in der Konsole an. Überprüfen Sie, ob die ausgegebenen Werte der Implementierung der Strategie entsprechen.

Beispielhafte Verwendung:

```
fun main() {
  // Thermometer mit erster Strategie initialisieren
  val thermometer = Thermometer(sensor = RandomSensor(2.0, 8.0))
  thermometer.measure(10)
}
```

c) Nach dem Aufruf der `measure` Funktion: Ändern Sie die Strategie des Thermometers auf eine *andere Strategie*. Rufen Sie erneut die `measure` Methode auf und schauen Sie sich die Ausgaben in der Konsole an. Nun sollten die ausgegebenen Werte der Implementierung der anderen Strategie entsprechen.

Beispielhafte Verwendung:

```
fun main() {
  // Thermometer mit erster Strategie initialisieren
  val thermometer = Thermometer(sensor = RandomSensor(2.0, 8.0))
  thermometer.measure(10)

  thermometer.sensor = IncreasingSensor(startTemp = 15.0) // Strategie
     wechseln
  thermometer.measure(10)
}
```

d) Welchen Vorteil bringt die Strategie für dieses Beispiel?

e) Welche objektorientierten Design Prinzipien werden vom Strategie Muster erfüllt? Begründen Sie Ihre Antwort.

3 Sensoren dekorieren

In dieser Aufgabe sollen die Strategien aus Aufgabe 1 um weitere Funktionalitäten erweitert werden.

a) Implementieren Sie die folgenden Dekorierer:

- `SensorLogger`: Schreibt bei jeder Temperaturabfrage den aktuellen Wert auf die Konsole.
- `RoundValues`: Rundet die Temperatur auf ganze Zahlen. So wird beispielsweise `19.4` zu `19.0` gerundet.
- `FahrenheitSensor`: Rechnet den Temperaturwert von Celsius in Fahrenheit um.

b) Testen Sie jetzt die Dekorierer, indem Sie folgende Aufgaben erledigen:

- Erzeugen Sie einen Sensor, welcher *zufällige* Temperaturen zwischen `2.0` und `5.0` *rundet* und diese auf der *Konsole ausgibt*.
- Erzeugen Sie einen Sensor, welcher *linear aufsteigende* Temperaturen ab `20.0` Grad Celsius in *Fahrenheit* umrechnet, diese danach *rundet* und anschließend auf der *Konsole ausgibt*.
- Erzeugen Sie einen Sensor, der das gleiche wie in der Aufgabe davor macht, aber zusätzlich die Temperatur in Celsius ausgibt, bevor in Fahrenheit umgerechnet wird.

Verwenden Sie diese dekorierten Sensoren in Ihrer `main` Funktion. Da Sie jeweils den `SensorLogger` verwenden, müssten Sie die dekorierten Ergebnisse auf der Konsole sehen. Überprüfen Sie diese Konsolenausgaben.

c) Ist die Reihenfolge beim Dekorieren relevant? Begründen Sie Ihre Antwort, indem Sie prüfen, ob es einen Unterschied zwischen
`val t1 = Thermometer(SensorLogger(RoundValues(RandomSensor(2.0, 5.0))))` und
`val t2 = Thermometer(RoundValues(SensorLogger(RandomSensor(2.0, 5.0))))` gibt.

d) Was für Vorteile bringt der Dekorierer? Hätte das alles auch mit weiteren Strategien funktioniert? Wenn nein, was wäre das Problem gewesen?

e) Was ist der grundsätzliche Unterschied zwischen einem Dekorierer und einer Strategie? Wann wird was verwendet?

f) Welche objektorientierten Design Prinzipien werden vom Dekorierer Muster erfüllt? Begründen Sie Ihre Antwort.

4 Beobachten des Thermometers

In dieser Aufgabe werden Sie ermöglichen, dass andere Objekte das `Thermometer` beobachten können und über Temperaturänderungen benachrichtigt werden.

a) Definieren Sie dazu eine Schnittstelle `TemperatureObserver` mit einer `update(tmp: Double)` Methode. Diese Methode soll die neue Temperatur als Parameter erhalten.

b) Definieren Sie folgende Beobachter:

- `TemperatureAlert`: Schreibt eine Nachricht auf der Konsole, wenn eine bestimmte Temperatur erreicht wird. Die Klasse nimmt den Schwellwert und die Nachricht im Konstruktor entgegen. So wird z.B. die Nachricht "Ganz schön heiß" bei einer Schwelltemperatur von 30 Grad ausgegeben.
- `HeatingSystemObserver`: Schaltet eine Heizung an oder aus, basierend auf der Durchschnittstemperatur der letzten 5 Temperaturen. Zunächst werden 5 Temperaturwerte in einer Liste gesammelt. Wenn 5 Werte vorhanden sind, wird der Durchschnitt berechnet. Liegt der Durchschnitt über einer bestimmten Grenze, wird "Heizung aus" auf der Konsole ausgegeben. Liegt der Durchschnitt unter einer bestimmten Grenze, wird "Heizung an" ausgegeben. Anschließend wird die Liste für die nächsten 5 Temperaturen geleert. Die beiden Schwellwerte werden im Konstruktor übergeben.

c) Das `Thermometer` ist das zu beobachtende **Subjekt** (Publisher). Daher muss es das folgende Interface implementieren:

```
interface TemperatureSubject {
  val observers: MutableList<TemperatureObserver>
  fun addObserver(o: TemperatureObserver)
  fun removeObserver(o: TemperatureObserver)
}
```

Implementieren Sie das Interface so, dass `TemperatureObserver` hinzugefügt und entfernt werden können. Sorgen Sie auch dafür, dass alle registrierten `TemperatureObserver` **benachrichtigt werden, wenn sich die Temperatur ändert**.

Testen Sie das Thermometer in Zusammenspiel mit den beiden Beobachtern, indem Sie z.B. eine Benachrichtigung auf der Konsole ausgeben, sobald eine Temperatur über 30 Grad gemeldet wird. Zudem soll die Heizung ab beispielsweise 19 Grad eingeschaltet und unter 23 Grad ausgeschaltet werden.
Hier ein Beispiel:

```
fun main() {
  val sensor = SensorLogger(RoundValues(RandomSensor(10.0, 50.0)))
  val thermometer = Thermometer2(sensor)
  val alertObserver = TemperatureAlert(
    alertTmp = 30.0,
    alertMsg = "Ganz schön heiß"
  )
  val heatingSystemObserver = HeatingSystemObserver(
    offThreshold = 23.0,
    onThreshold = 19.0
  )
  thermometer.addObserver(alertObserver)
  thermometer.addObserver(heatingSystemObserver)
  thermometer.measure(20)
}
```

Die Konsolenausgabe kann beispielsweise so aussehen:

```
> 15.0
> 21.0
> 9.0
> 31.0
> Ganz schön heiß
> 32.0
> Ganz schön heiß
> Die Durchschnittstemperatur der letzten 10 Messungen ist 21.6
> 33.0
> Ganz schön heiß
> 24.0
> 7.0
> 1.0
> 1.0
> Die Durchschnittstemperatur der letzten 10 Messungen ist 13.2
> Heizung an!
...
```

d) Welches Problem löst ein Beobachter? Wie wäre die Alternative, wenn man beispielsweise in Teilaufgabe **c)** keinen Beobachter verwenden würde?

e) Welche objektorientierten Design Prinzipien werden vom Beobachter Muster erfüllt? Begründen Sie Ihre Antwort.