Algorithmen und Programmierung 2
Praktikum 2 - Polymorphie, Interfaces und Listen

Dieses Übungsblatt beschäftigt sich mit dem Thema Polymorphie. Zusätzlich werden Sie ein eigenes Interface sowie verschiedene Algorithmen auf Listen implementieren.

Die Vorlesungsmaterialien (Lehrbuch, Screencast, Skript) aus dem ILU-Kurs können Sie als Hilfestellung nutzen.

Dieses Übungsblatt basiert auf dem Code aus dem vorangegangenen Übungsblatt. Die Musterlösung dazu können Sie sich hier anschauen und mit Ihrer Lösung vergleichen / ggf. ergänzen.

Inhaltsverzeichnis

1 Polymorphie

In dieser Aufgabe sollen Unterklassen von `Task` erstellt und spezifiziert werden.

1.1 Unterklassen erstellen

Definieren Sie die Unterklassen `SingleTask` und `RecurringTask` für die Klasse `Task`. Eine `SingleTask` ist eine Aufgabe, die nur ein Mal erledigt werden muss. Sie hat eine zusätzliche berechnete Eigenschaft `reminder`, die eine Erinnerung zwei Tage vor der Deadline darstellt. Sie ist, genauso wie die Deadline selbst, in „Tagen ab heute“ angegeben. Es soll auch überprüft werden, ob die Erinnerung schon in der Vergangenheit liegt (also kleiner als `0` ist) und in diesem Fall auf `0` gesetzt werden. Ein `RecurringTask` ist eine wiederkehrende Aufgabe. Sie hat eine zusätzliche Eigenschaft `frequency`, die die Frequenz der Aufgabenwiederholung in Tagen speichert.
Passen Sie auch die Klasse `Task` so an, dass keine Objekte von dieser mehr erzeugt werden können.

1.2 Polymorphie verwenden

Implementieren Sie für die Klasse `Project` eine zusätzliche Methode `checkTasks`, die über die Liste der Aufgaben `tasks` iteriert. Für jede einmalige Aufgabe `SingleTask` soll hier überprüft werden, ob die Erinnerung heute fällig ist. Wenn ja, soll ein entsprechender Alarm ausgegeben werden. Wenn es sich um eine wiederkehrende Aufgabe `RecurringTask` handelt, soll überprüft werden, ob die Deadline in der Vergangenheit liegt. Ist dies der Fall, kann die Deadline mithilfe der Aufgabenfrequenz `frequency` neu gesetzt werden, sofern sie noch vor der Deadline des darüberliegenden Projekts liegt. Andernfalls kann die bereits definierte `Exception` geworfen werden.

1.3 Testen mit Objekten

Erweitern Sie Ihre `main` Funktion um Code, der die neue Methode testet. Eine beispielhafte Ausgabe könnte so aussehen. In Ihrem Code können Sie natürlich auch andere Objekte verwenden.

```
Wiederkehrende Aufgabe 'Gerichte planen' ist überfällig und wird neu geplant.
Neue Deadline für 'Gerichte planen' in 6 Tagen
Alarm für einmalige Aufgabe 'Einkaufsliste schreiben'!
```

2 Interfaces

In dieser Aufgabe sollen Arbeitseinheiten über ein Interface automatisch priorisiert werden. Sie brauchen dazu das folgende Enum. Den vorgegebenen Code können Sie hier kopieren.

```
enum class Priority {
    HIGH,
```

```
    MEDIUM,
    LOW;

    companion object {
        fun fromFactor(factor: Double): Priority {
            return when(factor.toInt()) {
                1 -> HIGH
                2 -> MEDIUM
                3 -> LOW
                else -> throw IllegalArgumentException("Ungültiger
                    Faktor: $factor")
            }
        }
    }
}
```

In diesem Enum werden drei Prioritäten festgelegt. Außerdem gibt es ein companion object, das die Priorität anhand eines Faktors zurückgibt.

2.1 Interface erstellen

Definieren Sie ein Interface `Prioritizable`, welches eine Methode `prioritize` besitzt. Die Methode hat den Rückgabetyp `Double`.

2.2 Interface implementieren

Die Klassen `WorkUnit`, `Task` und `Project` sollen nun das Interface `Prioritizable` implementieren. Die Funktion `prioritize` soll einen Faktor für die Priorität bestimmen.
Die berechnete Eigenschaft `priority` soll zusätzlich in der Klasse `WorkUnit` mit dem folgenden Code implementiert werden. Den vorgegebenen Code können Sie hier kopieren.

```
var priority: Priority? = null
        get() = Priority.fromFactor(this.prioritize())
```

Hier wird der Rückgabewert der Methode `prioritize` verwendet, um die Priorität festzulegen. Dazu wird das `companion object` aus dem Enum verwendet.

Klasse `WorkUnit`

In der Methode `prioritize` wird der Faktor der Priorität aus den Eigenschaften `deadline` und `status` berechnet.
Eine Arbeitseinheit ist wichtiger, wenn die Deadline früher ist. Ist die Deadline innerhalb einer Woche, ist der Faktor `1.0`. Ist sie innerhalb eines Monats, liegt der Faktor bei `2.0` und ansonsten bei `3.0`.
Was den Status angeht, so sind bereits angefangene Aufgaben am Wichtigsten und abgeschlossene Aufgaben am Unwichtigsten. So hat `DOING` den Faktor `1.0`, `TODO` Faktor `2.0` und `DONE` Faktor `3.0`.

Die Methode soll den Durchschnitt der beiden Faktoren zurück geben.

Klasse `Task`

In der Methode `prioritize` wird der Faktor der Priorität **zusätzlich** aus den Eigenschaften `steps` und `estimatedTime` berechnet.
Aufgaben mit mehr Schritten sind wichtiger als Aufgaben mit wenig Schritten. Gibt es mehr als 11 Schritte, ist der Faktor `1.0`. Zwischen 5 und 10 Schritten ist der Faktor `2.0` und ansonsten `3.0`.
Außerdem sollen Aufgaben, die schneller abgeschlossen sind, höher priorisiert werden als lange Aufgaben. Ist die geschätzte Zeit einer Aufgabe unter einer Stunde (60 Minuten), so beträgt der Faktor `1.0`. Zwischen einer und drei Stunden (180 Minuten) ist der Faktor `2.0` und ansonsten `3.0`.
Die Methode soll den Durchschnitt der beiden Faktoren berechnen und den Durchschnitt aller Faktoren (siehe `WorkUnit`) zurückgeben.

Klasse `Project`

In der Methode `prioritize` wird der Faktor der Priorität basierend auf den Faktoren der beinhalteten Aufgaben berechnet.
Es sollen alle Aufgaben in einem Projekt iteriert werden. Der Faktor für die Priorität des Projekts ergibt sich aus dem Durchschnitt der Faktoren für die einzelnen Aufgaben.

2.3 Testen mit Objekten

Erweitern Sie Ihre `main` Funktion um Code, der die neuen Methoden testet.

3 Typen

Geben Sie an, ob die folgenden Ausdrücke in Ordnung (OK) sind, einen Compilerfehler (CF) oder einen Laufzeitfehler (LF) ergeben. Begründen Sie die Fehler jeweils.

```
val a: Prioritizable = WorkUnit("Grundlegende Arbeitseinheit", "Das
   ist die Beschreibung", 10, Status.TODO)
val b = "Aufgabe"
val c = RecurringTask("Deadline Task", "Mit Deadline", 3,
   listOf("Schritt 1", "Schritt 2"), 120, Status.TODO, -3) //OK
val d: WorkUnit = Prioritizable()
val e: WorkUnit = c
val f: WorkUnit = SingleTask("Spezifischer Task", "Beschreibung", 5,
   null, 45, Status.DOING)
val g: Prioritizable = f as RecurringTask
val h: Prioritizable = c
```

```
val i: RecurringTask = Task("Titel für wiederkehrende Aufgabe",
   "Beschreibung", 4, listOf("Schritt A", "Schritt B"), 60,
   Status.TODO)
val j: WorkUnit = WorkUnit("Arbeitseinheit")
```

4 Listen

Die Klasse `Manager` soll das gesamte Aufgabenmanagement verwalten und beinhaltet eine Liste von Projekten. Den vorgegebenen Code können Sie hier kopieren.

```
class Manager(val projects: MutableList<Project>) {
    val todo = mutableListOf<Task>()
}
```

Implementieren Sie die folgenden Methoden in dieser Klasse:

- `generateToDoList`: Überprüft alle Tasks aus den beinhalteten Projekten auf ihren Status und fügt sie zu der To Do Liste hinzu, wenn sie **nicht** abgeschlossen sind. Gibt die To Do Liste am Ende zurück.
- `getPriorityToDo`: Gibt eine Liste mit allen offenen To Dos zurück, die eine hohe Priorität haben.
- `getAvgTime`: Gibt die durchschnittlich benötigte Zeit für alle hochpriorisierten Aufgaben zurück.

4.1 Testen mit Objekten

Erweitern Sie Ihre `main` Funktion um Code, der die neue Methode testet.