Einstieg in Java und OOP

Christian Silberbauer

Übungsblatt 10

Entwickeln Sie einen `ManagedThread`, der ereignisbasierte Kommunikation ermöglicht.

Aufgabe 1

Gegeben sind folgende Framework-Klassen:

```
public abstract class Event {
}

public abstract class Action extends Event {
	abstract void execute();
}
```

Zudem sind folgende Testklassen definiert:

```
class HelloWorldAction extends Action {

	public void execute() {
		System.out.println("Hello World!");
	}
}

class GoodByeWorldAction extends Action {

	public void execute() {
		System.out.println("Good Bye World!");
	}
}

public class Main {

	public static void main(String[] args) throws InterruptedException {

		ManagedThread mt = new ManagedThread();
		mt.start();

		mt.add(new HelloWorldAction());
		mt.add(new HelloWorldAction());
		mt.add(new GoodByeWorldAction());
		mt.add(new GoodByeWorldAction());

		Thread.sleep(5000);
		mt.cancel();

		mt.join();
	}
}
```

Implementieren Sie eine Klasse `ManagedThread`, die die Klasse `Thread` erweitert, und folgende Attribute und Methoden besitzt:

- `Queue<Action> queue`: Queue von zu verarbeitenden `Action`s
- `boolean isCancelled`: Flag, das zum Beenden des Threads führt, wenn es gesetzt ist
- `void add (Action action)`: Fügt der Queue eine `Action` hinzu und benachrichtigt die `run()`-Methode, um die `Action` zu verarbeiten. Nutzen Sie zur Benachrichtung der `run()`-Methode `Object.notify()`.
- `void cancel()`: Setzt das Flag `isCancelled`, wodurch die `run()`-Methode beendet werden soll. Nutzen Sie zur Benachrichtung der `run()`-Methode `Object.notify()`.
- `void run()`: Verarbeitet `Action`s in der Queue, indem sie ihre `execute()`-Methode aufruft und sie aus der Queue entfernt. Falls `isCancelled` gesetzt ist, soll der Thread beendet werden. Falls die Queue leer ist, wartet hier der Thread mittels `Object.wait()`.

Implementieren Sie zudem einen Defaultkonstruktor.

Aufgabe 2

Implementieren Sie eine Klasse `PushAction`, die es bei ihrer Ausführung erlaubt, eine andere `Action` dem `ManagedThread` hinzuzufügen. Stellen Sie durch Ausprobieren fest, dass damit die Ausführreihenfolge der Ereignisse nicht mehr deterministisch ist (, weil sowohl im Main-Thread als auch im `ManagedThread` `Action`s erzeugt werden).
Für den Test finden Sie hier eine erweiterte `Main`-Klasse. Ggf. müssen Sie die fünf `add()`-Aufrufe je mehrmals hintereinander ausführen, um die Nichtdeterminiertheit zu erkennen.

```
public class Main {

	public static void main(String[] args) throws InterruptedException {

		ManagedThread mt = new ManagedThread();
		mt.start();

		mt.add(new HelloWorldAction());
		mt.add(new HelloWorldAction());

		mt.add(new PushAction(mt, new HelloWorldAction()));

		mt.add(new GoodByeWorldAction());
		mt.add(new GoodByeWorldAction());

		Thread.sleep(5000);
		mt.cancel();

		mt.join();
	}
}
```

Aufgabe 3

Ändern Sie die Klasse `Event` wie folgt ab:

```
public abstract class Event {

	private long triggerTime;

	public Event() {
	}

	public Event(long triggerTime) {
		this.triggerTime = triggerTime;
	}

	public long getTriggerTime() {
		return triggerTime;
	}

	public void setTriggerTime(long triggerTime) {
		this.triggerTime = triggerTime;
	}
}
```

Ändern Sie die Klasse `ManagedThread` dahingehend ab, dass sie `Action`s erst zur definierten `triggerTime` ausführt, statt unmittelbar, wenn dessen Wert größer der aktuellen Zeit ist, welche Sie mit `System.currentTimeMillis()` abfragen können.

Lösungshinweise:

1. Verwenden Sie hier im `ManagedThread` eine `List` statt einer `Queue`, da nicht immer nur nach FIFO die Elemente entfernt werden müssen, sondern eben auch die `triggerTime` zu berücksichtigen ist.
2. Verwenden Sie zum Durchlaufen der `List` nicht den `Iterator`, da dieser es nicht erlaubt, dass während des Iterierens Elemente durch ihn gelöscht werden und anderweitig weitere Elemente hinzugefügt werden, was durch die `PushAction` aber der Fall wäre. Sie bekämen hier ggf. eine `ConcurrentModificationException`.

Folgende `Main`-Klasse können Sie zum Testen verwenden:

```
public class Main {

	public static void main(String[] args) throws InterruptedException {

		ManagedThread mt = new ManagedThread();
		mt.start();

		mt.add(new HelloWorldAction());
		mt.add(new HelloWorldAction());

		mt.add(new PushAction(mt, new HelloWorldAction()));

		Action a = new HelloWorldAction();
		a.setTriggerTime(System.currentTimeMillis() + 1000);
		mt.add(a);

		mt.add(new GoodByeWorldAction());
		mt.add(new GoodByeWorldAction());
```

```
            Thread.sleep(5000);
            mt.cancel();

            mt.join();
      }
}
```

Aufgabe 4

Ergänzen Sie das Interface `EventListener`:

```
public interface EventListener<T extends Event> {
      void update(T event);
}
```

Ermöglichen Sie der `add()`-Methode der Klasse `ManagedThread`, dass statt `Action`s beliebige `Event`s hinzugefügt werden können.
Fügen Sie der Klasse `ManagedThread` die Methode `public <T extends Event> void addListener(Class<T> eventClass, EventListener<? super T> listener)` hinzu, um über `Event`s eines bestimmten Eventtyps benachrichtigt zu werden.
Berücksichtigen Sie in der `run()`-Methode die Benachrichtigung von `Event`s, indem Sie `notifyListeners()` aufrufen und sie nur noch mittels `execute()` ausführen, wenn es sich bei den `Event`s auch um `Action`s handelt (`if (e instanceof Action)`…).
Implementieren Sie die Methode `private <T extends Event> void notifyListeners(T event)`. Sie soll registrierte `EventListener` benachrichtigen, wenn `Event`s vom entsprechenden Typ ausgelöst wurden. Berücksichtigen Sie hierbei, dass `EventListener` auch durch abstraktere Typen benachrichtigt werden. Beispielsweise soll ein `EventListener<Event>` über alle `Event`s informiert werden.
Ermöglichen Sie zudem, dass eine `PushAction` nicht nur `Action`s, sondern beliebige `Event`s einem `ManagedThread` hinzufügen kann.

Zum Test können Sie folgenden Code verwenden:

```
class HelloWorldActionHandler implements EventListener<HelloWorldAction> {

      public void update(HelloWorldAction event) {
            System.out.println("HelloWorldAction triggered.");
      }
}

class EventHandler implements EventListener<Event> {

      public void update(Event event) {
            System.out.println("Event triggered.");
      }
}


public class Main {
```

```
	public static void main(String[] args) throws InterruptedException {

		ManagedThread mt = new ManagedThread();
		mt.start();

		mt.addListener(HelloWorldAction.class,
			new HelloWorldActionHandler());
		mt.addListener(Event.class, new EventHandler());

		mt.add(new HelloWorldAction());
		mt.add(new HelloWorldAction());

		mt.add(new PushAction(mt, new HelloWorldAction()));

		Action a = new HelloWorldAction();
		a.setTriggerTime(System.currentTimeMillis() + 1000);
		mt.add(a);

		mt.add(new GoodByeWorldAction());
		mt.add(new GoodByeWorldAction());

		Thread.sleep(5000);
		mt.cancel();

		mt.join();
	}
}
```