Christoph Lüth
Martin Ring

4. Übungsblatt

Ausgabe: 12.06.14
Abgabe: 26.06.14

4.1 *Back Pressure* *10 Punkte*

Auf Grundlage der in der Vorlesung erarbeiteten Implementierung von Observables, sollen Sie in dieser Aufgabe einen einfachen Back-Pressure-Mechanismus entwickeln. Dafür können Sie die Quellen in `Observables.zip` als Grundlage verwenden.

Bisher wird `onNext` blockierend aufgerufen. Das führt dazu, dass `Observable.from(nats)` (wobei `nats` ein unendlich langer `Iterable` ist) immer nur Werte produziert, wenn der Kontrollfluss von `onNext` wieder an den Aufrufer übergeben wurde. In einem ersten Schritt wollen wir hier den produzierenden und den konsumierenden Thread entkoppeln. Das machen wir mit einer Funktion entsprechend der Implementierung in der `RxJava` Bibliothek:

```
trait Observable[T] {
  ...
  def observeOn(scheduler: Scheduler): Observable[T]
}
```

Diese Funktion führt dazu, dass für den produzierenden Observable alle Aufrufe von `onNext` die Kontrolle sofort an den Aufrufer übergeben (und natürlich die tatsächliche Ausführung des unterliegenden `onNext` dem Scheduler übergeben wird). Achten Sie dabei darauf, dass die Subscriptions weiterhin funktionieren und die Ausführungsreihenfolge nicht verletzt wird.

Durch die Entkopplung der Threads haben wir ein neues Problem. Der produzierende Thread kann nun theoretisch viel schneller Werte produzieren, als der Konsument sie verarbeiten kann (fehlende Back Pressure).

Um einen sehr simplen Back-Pressure-Mechanismus zu implementieren, ändern Sie das Interface von Observables, so dass `onNext` nicht mehr `Unit`, sondern `Future[Unit]` zurück gibt, und passen Sie die Implementierung von Observern, Observables und Subscription entsprechend an.

Testen sie Ihre Implementierung. Überprüfen Sie dabei zum Beispiel, dass der Produzent keine Daten mehr produziert, wenn der Konsument ein `Future.never` (wie es auf dem letzten Übungsblatt entwickelt wurde) zurückgibt.

4.2 *MinXnarXeitXX* *10 Punkte*

Nachdem wir letztes mal Ordnung in die Mine gebracht haben, wird sie dieses mal wieder durcheinander gebracht! Ändern Sie die Version von `robo-mine` auf `1.3-SNAPSHOT`. Leider wurde die neue Version unter Zeit- und Kostendruck entwickelt, so dass die Entwickler einen Schraubenzieher auf dem Mainboard einiger Roboter liegen lassen haben, der nun dazu führt, dass die Roboter zeitweise fehlerhafte Daten produzieren.

Wenn Sie fehlerhafte Werte parsen, sollten sie eine Exception werfen, welche durch die `onError` Methode an die Observer weiter gereicht wird. Implementieren sie nun eine Fehlerbehandlungsstrategie wie folgt:

- Für den entsprechenden Sensor existiert ein Unzuverlässigkeitslevel. Dieser liegt zu Beginn bei 0.
- Mit jedem fehlerhaften Wert wird der Unzuverlässigkeitslevel für diesen Sensor erhöht.
- Bis zu einem bestimmten Unzuverlässigkeitslevel sollten sie den Sensor im Fehlerfall neu starten, oberhalb eines bestimmten Levels sollten sie den Sensor als kaputt einstufen und eine Ausweichstrategie aktivieren, die ohne den Sensor auskommt.
- Mit der Länge fehlerfreier Phasen wird der Unzuverlässigkeitslevel wieder gesenkt.

Zum Glück ist so ein Schraubenzieher nicht allzu groß, daher kann nur einer beiden Sensoren **Laser** oder **Gold Detektor** ausfallen. Es ist also insbesondere sinnvoll, bei Robotern, bei denen beispielsweise der Gold Detektor ausgefallen ist, auf ein Verhalten auszuweichen, das nicht davon abhängig ist.

4.3 *Zu Hilfe!* *10 Bonuspunkte*

Brechen sie das Verhalten eines Roboters mit einer `LowBatteryException` ab, wenn ein Liegenbleiben unausweichlich wird und schalten sie den betreffenden Roboter aus. Nutzen Sie andere Roboter (Rettungsroboter), um den verunglückten Roboter zurück in die Basis zu schieben. Roboter, deren Gold-Detektoren nicht mehr funktionieren, können Sie nun als Rettungsroboter nutzen.