Wenn es um die Wurst geht

Horden von Zombies

Phillip Ghadir

Kennen Sie das Computer-Spiel „Plants vs. Zombies"? Das Szenario ist einfach: Zombies stürmen auf unser Haus zu. Als Spieler haben wir eigentlich nur eine Chance, der Übermacht zu begegnen: Wir müssen die Verteidigungsmechanismen vorbereiten, die wir im Eifer des Gefechts brauchen werden. Paprikaschoten, Erbsen und Walnüsse werden uns helfen, die Angriffswut der Zombies zu stoppen. In dieser Ausgabe nutze ich die Kulisse des Spiels, um ein paar Eigenschaften von Reactive Extensions zu vorzustellen.

Bereits in [Gha13] habe ich Reactive Extensions [RX] vorgestellt. Auch in dieser Kolumne setze ich auf das quelloffene Framework RxJava, das mittlerweile umgezogen ist und jetzt mit folgender Dependency-Deklaration in der Maven-POM hinzugefügt werden kann:

```
<dependency>
  <groupId>io.reactivex</groupId>
  <artifactId>rxjava</artifactId>
  <version>1.0.10</version>
</dependency>
```

Durch Einbinden von RxJava können wir nun sogenannte Observables beobachten. Dazu registriert man Implementierungen von `rx.Observer<T>` an geeigneten `rx.Observable<T>`.

`Observable<T>` bietet ähnliche Funktionen, wie die in Java 8 mit dem Streaming-API hinzugekommenen, beispielsweise `map`, `filter` und `reduce`.

Ein `Observable<T>` kann man als Sequenz asynchroner Ereignisse auffassen. Die Daten fließen durch die Observables zu den Observern, indem ein Observable jeden Wert per Aufruf der `Observer.onNext`-Methode an den registrierten Observer übermittelt. Des Weiteren können Observables von Reactive Extensions Fehler an registrierte Observer melden sowie Observer per Aufruf von `onComplete()` informieren, wenn das Observable keine weiteren Ereignisse mehr liefern wird.

Die Signatur für einen Observer sieht wie folgt aus:

```
public interface Observer<T> {
  void onCompleted();
  void onError(Throwable e);
  void onNext(T t);
}
```

Jedes `Observable<T>` muss garantieren, dass `onNext()` nicht aufgerufen wird, wenn ein voriger Aufruf noch nicht beendet wurde. Ein Observable kann diese Methode beliebig häufig – oder auch niemals – aufrufen. Falls ein Observable `onCompleted()` oder `onError()` aufruft, darf es nur eine der beiden Methoden aufrufen.

In [Gha13] sind einige Eigenschaften von RxJava beschrieben. Obwohl die Strukturen syntaktisch so einfach und überschaubar sind, bildet sich manchmal doch ein kleiner Knoten im Kopf. Deshalb möchte ich ein paar Eigenheiten von Datenflüssen am Beispiel von angreifenden Zombie-Horden erläutern. Es geht in diesem Heft schließlich um Sicherheit.

P vs. Z

Das Spiel selbst besteht aus einer Menge von Levels. Ein Level kann nur gespielt werden, wenn der Level zuvor gemeistert wurde – das heißt, wenn die Zombieinvasion zuvor überstanden ist. Es beginnt mit Level 1.

Jeder Level definiert ein Spielfeld, den Ort des Geschehens, an dem die Zombies einfallen und wir sie aufhalten müssen. Das Spielfeld ist im (Vor-)Garten oder auf dem Dach angesiedelt und wird in ein zweidimensionales Feld unterteilt.

In einem Level ist die Angriffssequenz der Zombies – vielleicht mit ein paar Variationen – relativ bekannt. Zombies greifen in einem für den Level typischen Rhythmus an. Dabei greifen die Zombies von rechts an und bewegen sich in einer Zeile (des Spielfelds) mit ihrer eigenen Geschwindigkeit stetig auf die eigene Verteidigungslinie zu. Abbildung 1 zeigt eine Beispieldefinition für einen Angriffsplan.

Das Vorrücken eines Zombies lässt sich durch Hindernisse aufhalten. Je nach deren Größe und Robustheit kann die Zeit variieren, die sich dadurch gewinnen lässt. Aber letztlich lässt sich ein Zombie nicht endgültig aufhalten: Er muss zerstört werden.

Jeder Level besteht aus den drei Phasen:

- Vorbereitung,
- Verteidigung und
- Nachsorge.

Begriff	Erläuterung
Energie	Um Pflanzen zu bauen, wird Energie benötigt. Diese wird von Energielieferanten (insbesondere Energie erzeugenden Pflanzen) produziert
Pflanze	Das Mittel der Verteidigung gegen die untoten Beißerlein (siehe Zombie)
Pflanzenart	Definiert die Eigenschaften einer Pflanze. Lässt sich grob in zwei Gruppen unterteilen: Energie erzeugende Pflanzen (produzieren Energie) und Abwehrpflanzen (können Zombies behindern/verletzen). Es gibt zu jeder Gruppe verschiedenste Pflanzenarten
Samen	Es gibt Samen unterschiedlichster Pflanzenarten. Im Spiel funktionieren Samen (kurz für Pflanzensamen) anders, als in der Realität: Aus einem Samen mit ausreichender Energie kann sofort eine Pflanze gezogen werden. Danach muss der Samen aber erst einmal für eine definierte Zeit regenerieren
Zombie	Sie greifen nach einem vorgegebenen, recht ausgeklügelten Angriffsplan an: Immer geradeaus auf den Gegner zulaufen. Wenn ein Hindernis auftaucht, niedermachen und dann weiter

Glossar zu „Plants vs. Zombies

Zombie-Angriffswelle Dummy-Level

Einsatz bei ms	Zombie-Art	Zeile
1800	Zombie 2	5
1000	Zombie 1	2
3400	Turbo Zombie 1	3

Abb. 1: Auszug der Zombie-Angriffswelle des Dummy-Levels

Rhythmus, Spannung und Spielspaß

Die Spannung in diesem Spiel entsteht dadurch, dass die Zombies kontinuierlich näher rücken. Um Abwehrpflanzen zu setzen, braucht es Zeit. Auch das Niedermachen eines heranrückenden Zombies erfordert Zeit. Man sieht also plastisch, wie die Zeit verrinnt, und fragt sich, ob ein Zombie noch rechtzeitig aufgehalten werden kann. Je schwieriger der Level, desto mehr Stress verursacht die Zombie-Plage.

Das Austarieren von Feuerkraft, der Regenerationszeit nach dem Setzen einer Abwehrpflanze und der Geschwindigkeit, mit der sich die fürs Pflanzen erforderliche Energie aufbaut, hat großen Einfluss auf den Spielspaß. Aber den klammern wir hier im Folgenden aus.

Angriffssequenz mit rx.Subject<T>

Subjects haben wir in [Gha13] bereits kennengelernt. Ein `Subject<T, R>` implementiert `Observer<T>` und spezialisiert `Observable<R>`. Damit lässt sich in RxJava an beliebigen Stellen ein Observable realisieren, das aus anderen Quellen gespeist wird.

In Listing 1 ist die Angriffssequenz mit Hilfe von `java.util.concurrent.ScheduledThreadPoolExecutor` (mit *core thread pool size* = 1) und einem `PublishSubject<T>` implementiert. Die Begrenzung der *core thread pool size* auf 1 sorgt dafür, dass sich ein `ScheduledThreadPoolExecutor` verhält wie ein `java.util.Timer`.

```
public static Observable<PlannedEvent>
    create(final Collection<PlannedEvent> plannedAttacks) {
  Subject<PlannedEvent, PlannedEvent> subject =
    PublishSubject.<PlannedEvent>create();
  long delay = 0;

  for ( PlannedEvent ev : plannedAttacks ) {
    scheduler.schedule(
      () -> subject.onNext(ev),
      ev.delay,
      TimeUnit.MILLISECONDS );
    delay = Math.max( ev.delay, delay );
  }
  scheduler.schedule(
    () -> subject.onCompleted(),
    delay,
    TimeUnit.MILLISECONDS );
  return subject;
}
```

Listing 1: Angriffssequenz wird als Hot Observable implementiert

Wir instanziieren in der `create`-Methode ein `PublishSubject`. `PublishSubject`s schicken ein Ereignis an registrierte Observer, puffern Ereignisse aber nicht.

Im Nachgang registrieren wir für jede Zeile aus Abbildung 1 ein Runnable, welches das Subject über den entsprechenden Zombie-Angriff benachrichtigt, da das Subject ja `rx.Observer` implementiert. Zusätzlich registrieren wir ein nachgelagertes Runnable, das das Subject über das Ende der Angriffswelle informiert. Schlussendlich geben wir das erzeugte Subject zurück.

Ein Hot Observable

Wir haben im vorigen Abschnitt ein sogenanntes Hot Observable gebaut. Zur Erinnerung: Ein Observable kapselt eine Menge beliebig vieler (potenziell asynchroner) Ereignisse, über die es Observer benachrichtigt.

Bei einem Hot Observable passieren auch dann Ereignisse, wenn gar kein Observer registriert ist, sodass hier Ereignisse verloren gehen können.

Im obigen Beispiel (Listing 1) werden die Runnables erzeugt, die, wenn sie aufgerufen werden, das Subject per `onNext()` und am Ende mit `onComplete()` benachrichtigen.

In der Methode wird kein Observer auf dem Subject selbst registriert. Es kann beliebig viel Zeit zwischen dem Aufruf der `create`-Methode und dem Erzeugen der Runnables auf der einen Seite und dem Registrieren eines Observers beim Subject auf der anderen Seite vergehen.

Das gewählte PublishSubject liefert an die registrierten Observer die Ereignisse in dem Moment, in dem sie eintreffen. Ereignisse werden aber weder gepuffert noch gespeichert. Es gäbe zwar das `ReplaySubject`, das Ereignisse auch puffern kann, aber das würde nur die Reihenfolge der Ereignisse – und nicht deren Verteilung auf der Zeitachse – rekonstruieren.

Angriffssequenz als Cold Observable

Wählen wir daher eine Implementierungsstrategie, die für unsere Zwecke nachvollziehbarere Ergebnisse liefert – wie in Listing 2 dargestellt.

```
public static Observable<PlannedEvent>
    create(final Collection<PlannedEvent> plannedAttacks) {
  return Observable.<PlannedEvent>create(subscriber -> {
    long delay = 0;
    for ( PlannedEvent ev : plannedAttacks ) {
      scheduler.schedule(
        () -> subscriber.onNext(ev),
        ev.delay,
        TimeUnit.MILLISECONDS );
      delay = Math.max( ev.delay, delay );
    }
    scheduler.schedule(
      () -> subscriber.onCompleted(),
      delay + 1000l,
      TimeUnit.MILLISECONDS );
  });
}
```

Listing 2: Angriffssequenz als Cold Observable implementiert

Die Registrierung der Runnable-Instanzen erfolgt genauso wie in Listing 1 dargestellt. Allerdings erfolgt sie erst dann, wenn sich ein Subscriber bei dem von der `create`-Methode erzeugten `Observable<PlannedEvent>` registriert, da wir die Funktionalität jetzt in einem Lambda gekapselt und als Parameter für `create` übergeben haben.

Diese Implementierung stellt zumindest sicher, dass die Folge von Ereignissen erst mit der Registrierung eines `Observer<T>` beginnt. Observables mit dieser Eigenschaft nennt man Cold Observables.

Einhaltung der onNext()-Sequenz

Der Kontrakt von Reactive Extensions sichert zu, dass ein Observable auf einem registrierten Observer nicht `onNext()` aufruft, solange ein vorher erfolgter Aufruf dieser Methode noch nicht zurückgekehrt ist.

Genau das gleiche Verhalten sichert ein `ScheduledThreadPoolExecutor` – mit *core thread pool size* = 1 – oder ein `java.util.Timer` zu.

Damit sind wir also nicht davor sicher, dass ein schlecht programmierter Observer die Angriffssequenz aus dem Takt bringen könnte, aber zumindest erfüllt unsere Implementierung die Zusicherungen von Observable.

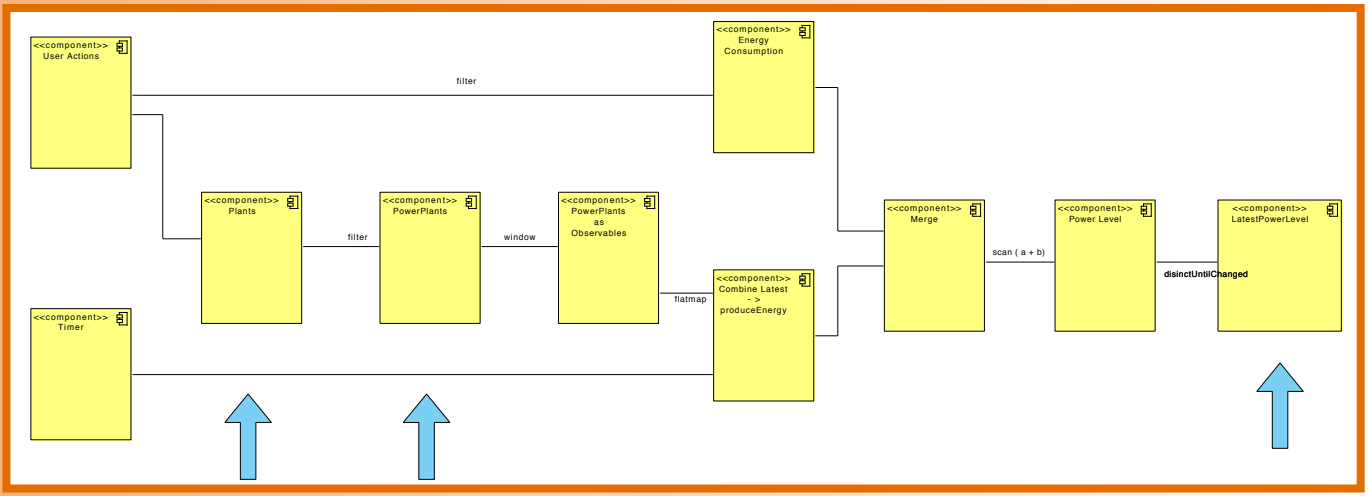

Abb. 2: Durch die Objekte fließen die Daten (Datenfluss kann an den blauen Pfeilen abgegriffen werden)

Status Quo

Die statische `create`-Methode ist in der Klasse `AttackSequence` definiert und verantwortet das sukzessive Auftauchen von neuen Zombies auf der Bildfläche. Bevor wir uns damit beschäftigen, wo diese sind, legen wir die Grundlagen für die Verteidigung gegen die hirnfressenden Untoten.

P vs. Z – Energiebedarf

Um eine Pflanze zu pflanzen, werden einerseits Samen und andererseits ausreichend Energie benötigt. Der Samen nimmt dabei die Rolle einer Fabrik-Methode ein. Der Aufruf einer Fabrik-Methode kostet abhängig von der Pflanzenart Energie. Ist nicht genug Energie vorhanden, passiert nichts. Konnte die Pflanze aber erzeugt werden, so wird der Samen für eine definierte Zeit deaktiviert.

Es gibt Pflanzen, die Energie liefern. Je nach Pflanzenart liefern sie in definierten Intervallen unterschiedlich viel Energie.

Für die Verteidigungsphase eines Levels definiere ich deshalb in einer Klasse `DefenseGame` den in Abbildung 2 skizzierten Datenfluss. In Ermangelung einer besseren Notation sind die Observables als Komponenten dargestellt, durch die die Daten fließen. Komponentennamen und auch Beschriftungen der Beziehungen weisen auf die Daten hin.

Den Takt für zeitgesteuerte Dinge im Spiel übernimmt das Observable time:

```
Observable time = Observable.timer(0l, 500l, TimeUnit.MILLISECONDS );
```

Um die Benutzeroberfläche (oder Tests) einfach anbinden zu können, werden Benutzeraktionen über `PublishSubject`-Instanzen transportiert.

```
PublishSubject userActions = PublishSubject.create();
PublishSubject plants = PublishSubject.create();
```

Für Berechnung der aktuell verfügbaren Energie definieren wir ein BehaviorSubject, über das der Energieverbrauch gemeldet wird. Es liefert bei der Registrierung von Observern den zuletzt gültigen Wert beziehungsweise zu Beginn den Default-Wert:

```
Subject<Integer> energyConsumptions = BehaviorSubject.create( 0 );
```

Für die Energieberechnung werden zwei Observables benötigt: das gerade definierte `energyConsumptions`, das den Energieverbrauch als negative Werte liefert, sowie das Folgende:

```
Observable<PowerPlant> powerPlants =
  plants.filter( o -> o instanceof PowerPlant );
```

Zu guter Letzt folgt die Definition des Observables in Listing 3 zum Beobachten der aktuell verfügbaren Energie (in Abbildung 2 „Latest Power Level“ genannt).

```
Observable power = powerPlants
  .window(1)
  .flatMap(this::toScheduledEnergyProduction)
  .mergeWith(energyConsumptions)
  .scan((a, b) -> add(a, b))
  .distinctUntilChanged();
```

Listing 3: Das Observable power meldet stets den aktuellen Energie-Status

Der in Listing 3 initialisierte Datenfluss baut auf dem Cold Observable `powerPlants` auf. Im Prinzip meldet jede Energie liefernde Pflanze über die Zeit ihre produzierte Energie. Das geschieht in der Methode `toScheduledEnergyProduction()`. Sie erzeugt ein Observable, über das die Energieproduktion der einzelnen Pflanzen beobachtet werden kann. Hierzu wird in der Methode ein Observable erzeugt, das stets für jede `PowerPlant`-Instanz deren `produce()`-Methode aufruft und deren Ergebnis transportiert:

```
private Observable<Integer> toScheduledEnergyProduction(
    Observable<PowerPlant> plant) {
  return Observable.<PowerPlant, Long, Integer>combineLatest(
    plant, time, (p, t) -> p.produce());
}
```

Die verwendete Methode `Observable.combineLatest` funktioniert so, wie eine Formel in einer Tabellenkalkulation. In diesem Falle wird die Formel *(plant,time) -> p.produce()* definiert. Jedes Mal, wenn `plant` oder `time` einen neuen Wert enthalten, berechnet `combineLatest` einen neuen Wert.

In Listing 3 liefert `window(1) Observables<Observable<PowerPlant>>`, die mit dem `combineLatest` im `flatMap` benutzt werden können.

Am Ende wollen wir die aktuell verfügbare Energie. Deshalb werden per `mergeWith()` noch die Energieverbräuche eingebunden, die dann per `scan` aggregiert und ausgegeben werden.

Da wir oben auch das Observable `time` mit `combineLatest` kombinieren, liefert das `scan`-Observable zu jedem Zeitpunkt einen

Energiestand. Um hier nur bei Änderungen informiert zu werden, schließen wir die Definition mit `distinctUntilChanged()` an.

Status Quo

Mit den angelegten Strukturen wird automatisch die verfügbare Energie aufgefrischt, sofern wir Energielieferanten haben.

Die Subject-Instanz `plants` ermöglicht das Hinzufügen von Pflanzen, sodass hier einfach irgendeine Steuerung oder Benutzerschnittstelle Pflanzen anlegen kann.

Was fehlt, sind der Angriff und die Verteidigung. Da wir bereits Cold und Hot Observables vorgestellt haben, beschränke ich mich im kommenden Teil darauf, dynamisch Observables zu verknüpfen und eine Rückkopplung einzubauen.

P vs. Z – Der Angriff

In Abbildung 3 ist die Struktur von `Plant` und `Zombie` dargestellt. Beide Klassen kapseln Observables, die über den aktuellen Zustand Auskunft geben. Es gibt potenziell mehrere Zombies und mehrere Pflanzen auf einem Spielfeld, sodass es auch viele Observables geben kann. Das ist in der Darstellung vereinfacht.

Die Observables der Pflanzen und Zombies werden per `merge` zu einem Observable gemacht, das alle Aktionen und Meldungen der Beteiligten enthält. Das Spielfeld kann darüber erkennen, ob es zum Kampf zwischen einzelnen Pflanzen und Zombies kommt, und dann Kontrahenten miteinander über ein neues Spielfeld-internes Observable verbinden.

Bei einer direkten Rückkopplung würde der Kurzschluss zu einem Abbruch führen. Dies vermeiden wir, indem Pflanzen und Zombies nicht direkt an das Spielfeld gekoppelt werden, sondern dies erfolgt über ein `zip` mit dem `Timer`.

Connectable Observables

Das Spielfeld wird viele Ereignisse von unterschiedlichen Quellen empfangen. Wenn es jetzt Zombies und Pflanzen auf einem neuen Observable für den Feedbackzyklus registriert, sollte die Benachrichtigung erst beginnen, wenn die Beteiligten registriert sind.

Jedes Observable lässt sich in RxJava durch den Aufruf der `publish()`-Methode in ein Connectable Observable umwandeln. Observer können sich wie üblich per `subscribe()` registrieren. Die Benachrichtigungen beginnen erst, wenn auf dem Connectable Observable die Methode `connect()` aufgerufen wird.

Connectable Observables sind Hot Observables. Das heißt, es kann passieren, dass zwischen dem Zeitpunkt des Erzeugens per `publish()` und dem Start der Benachrichtigungen per `connect()` bereits Ereignisse aufgetreten sind, die die Observer nicht mitbekommen.

Fazit

Das Computer-Spiel „Plants vs. Zombies“ zeigt ganz gut, wie sich verschiedene Elemente als Observables repräsentieren lassen.

Das Spiel enthält allerdings auch Wechselwirkungen, die uns zwingen, Feedbackschleifen zu realisieren. Um hier Timing-Probleme zu vermeiden, haben wir Timer verwendet, um Ereignisse zu takten. Das ist sicherlich keine Lösung für alle Fälle.

In unserem kleinen Ausschnitt ist vielleicht recht untypisch, dass wir verhältnismäßig wenige Benachrichtigungen zwischen verhältnismäßig vielen beteiligten Observables und Observern haben.

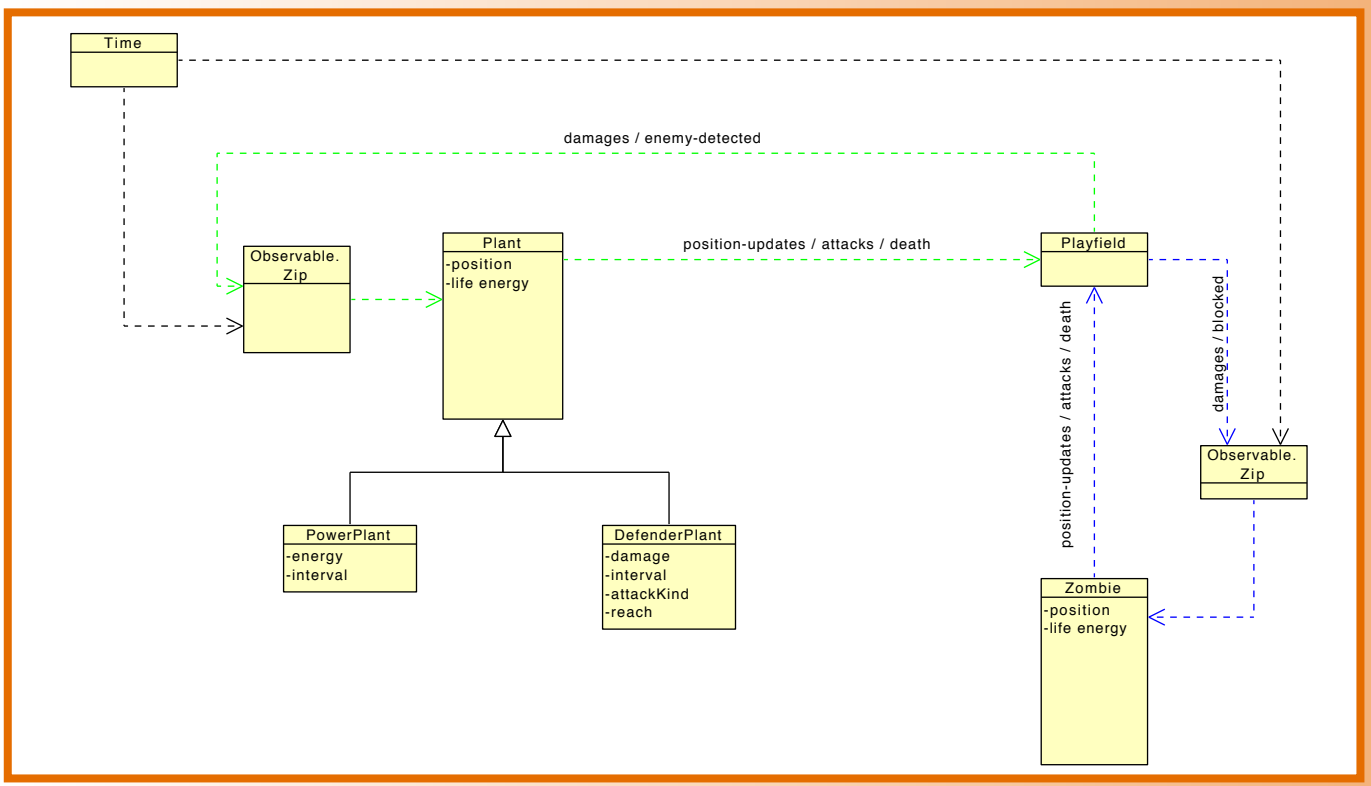

Abb. 3: Plant und Zombie benachrichtigen Playfield über Zustandsänderungen, werden in Intervallen über Änderungen in der Umgebung informiert

Wir haben die Konzepte Hot, Cold und Connectable Observables sowie einige Kombinator-Methoden der Reactive Extensions gesehen.

Literatur und Links

[Gha13] Ph. Ghadir, Reactive Extensions in Java, in: JavaSPEKTRUM, 5/2013, s. a.: http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2013/05/ghadir_JS_05_13_kdje.pdf

[RX] ReactiveX.io Homepage, http://reactivex.io/

[Wiki] https://de.wikipedia.org/wiki/Pflanzen_gegen_Zombies

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com