Implementation of Programming Languages

Feed RSS reader in REScala

Group: Przemyslaw Chrzastowski
Johannes Späth
Markus Hauck

Instructor: Guido Salvaneschi

TECHNISCHE UNIVERSITÄT DARMSTADT

Implementation of Programming Languages - WiSe 2012/13

Contents

1 Introduction

The task given was to write a Feed RSS reader in REScala. Due to problems in the framework our project does not make use of signals and instead uses events provided by EScala.

1.1 Invocation

The project can be started either via eclipse, sbt or by using the supplied jar file. Note that the jar was compiled with scala 2.9.2. The reader takes an optional argument on the command line which represents the path of a text file to read initial feed urls from.

Sample invocations:

- JAR:

```
> scala ReactiveReader.jar
> scala ReactiveReader.jar urls.txt
```

- SBT:

```
> sbt run
> sbt 'run urls.txt'
```

- Eclipse:

```
Run > Run as > Scala Application
```

1.2 Functionality

Once started the reader immediately starts fetching feeds from the urls of the text file if given. Otherwise some default feeds are loaded.

The top field represents the currently used channels, by clicking one of these the bottom left list shows all currently fetched items. Fetched items are immediately displayed after parsing, additionally, you can track the number of feeds and channels in the lower right notification bar.

The lower left bar provides information of the current fetching status. The selected feed and its content is rendered (as html) in the right view. Some more information, depending on the fetched xml, e.g. date and title of a feed, is provided in this view as well.

By selecting the checkbox "auto refresh" the reader automatically checks the channels periodically for any new items. Otherwise the user can manually check for new feeds via a button.

New urls can be added via `Edit -> Add url`. After being validated the url will be added to the reader. If the url was not valid an error dialog will appear.

2 Structure and Architecture

The following diagram shows the **rough** structure of the readers source code in form of class dependencies. Note that some details are omitted or simplified for clarity.

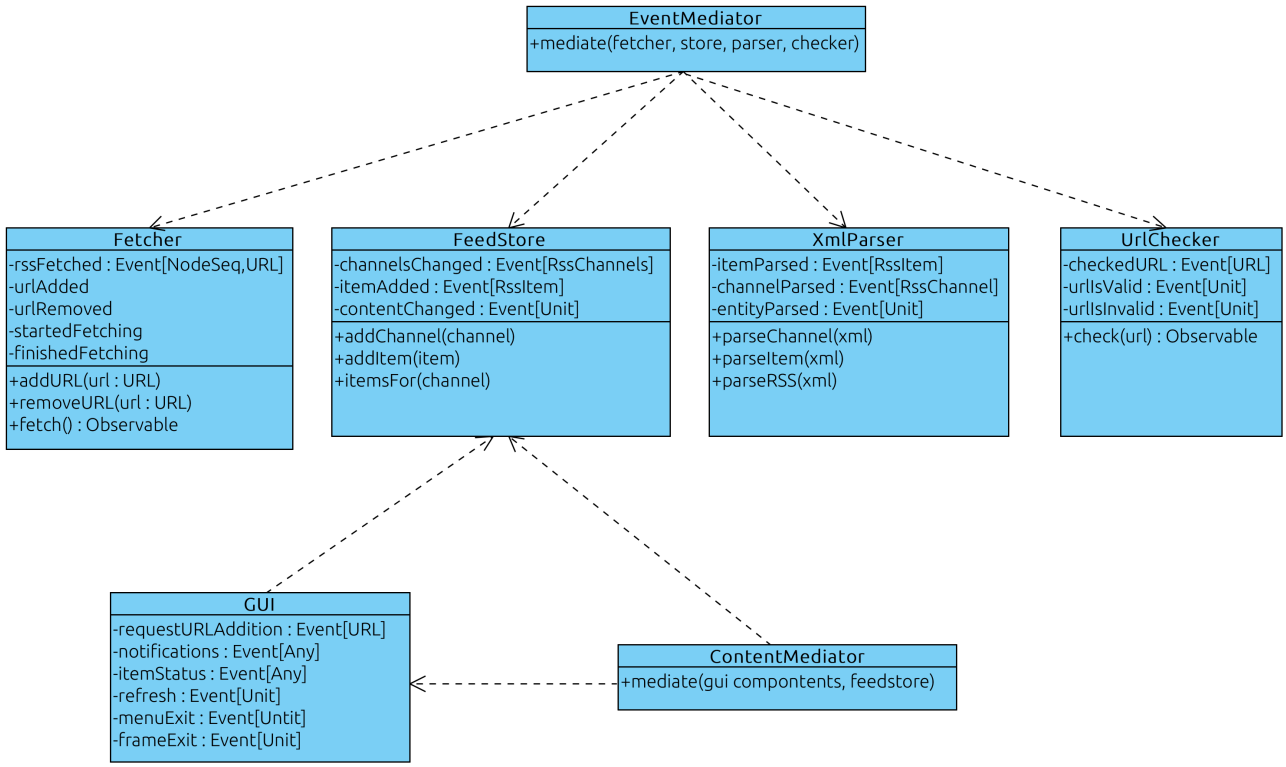

Figure 2.1: Main components

2.1 Components

It follows a short description of each of the main components.

2.1.1 Fetcher

Task:

The Fetcher is responsible to get the xml from the network. To accomplish this, it stores the urls of rss feeds for which it will try to download the feed data when triggered.

Events:

- Several trigger events are used to notify about internal processes e.g. after an url was added or the fetcher started to download the data from the urls.

- The `fetch` and `removeURL` methods are Observable to provide hooks for some of the triggers above.
- Fetching can be triggered by using the `trigger` event from the outside.

2.1.2 FeedStore

Task:
Stores all of the channels with its associated items and provides access to items of each channel. The other components have access to the stores data.

Events:
- Changes to the internal content like the addition of a channel or item are published via events

2.1.3 XmlParser

Task:
Responsible for transforming and mapping the downloaded xml structure to instances of the classes RssItem and RssChannel. Theses instances are then stored in the FeedStore described above.

Events:
- On the outside, there are several trigger events that publish the parsed items and channels
- Internally, there are observable parse methods for channels and items. Their results are filtered via event predicates and only the filtered events are publicly published.

2.1.4 UrlChecker

Task:
Given a String that represent a url, the checker does some basic checking to verify the validity of the url. Due to the scope of the project these checks are very limited.

Events:
- Internally, various events and event-expressions are used to further transform the output of the private check method.
- public events provide information about the result of the observable checkURL method.

2.1.5 GUI

Task:
The GUI displays the channels and feeds to the user. The main user interface consists of:

- a channel list

- a list of items in the selected channel
- a panel rendering the item which is currently selected

A notification bar at the bottom communicates basic information of the fetching status as well as the number of channels and feeds currently retrieved.

Events:
Various events provide communication with outside components. For example the public event notifications can be triggered from outside and the result is displayed in the GUI. Additionally the GUI makes use of some wrapper classes that provide event interfaces for common swing components.

2.1.6 Content- and EventMediator

Task:
The mediators connect various main components by using callbacks to their public events

- The event mediator drives the logic of adding checked urls to the fetcher, fetching the data of the feed urls, parsing the retrieved data via the XmlParser and storing the results in the FeedStore.
- The content mediator synchronizes the display of the stored content in the FeedStore.

Events:
The mediators only uses callbacks to the public events fired by the components.

2.1.7 Main

Task:
- Initialize the gui and main components, wire up some basic events
- Initialize the mediators with the components
- Load urls from a file or provide some defaults
- Periodically trigger the fetcher

Events:
The periodical fetching event is bind to the checkbox in the GUI. If deselected the event can manually be fired via the refresh button.

2.2 Discussion

The use of events makes it very easy to decouple the dependencies of classes.
In this project for example, the Fetcher, FeedStore, XmlParser and UrlChecker are completely separated and only connected via the EventMediator.

The use of this structure allows to centrally organize the logic as well as stack or combine different mediators.

Examples:

- Use SimpleReporter and CentralizedEvents in combination
- Add a Logger as a mediator to log important errors / events
- Express the system logic via hierarchical mediators, single mediators group parts of independent components into logical chunks, which are further grouped by others.

3 Statistics

The following table provides an overview over the use of events in the project.

Class/Object	imperative	implicit	expressions	callbacks
FeedStore	2	2	1	0
XmlParser	1	8	4	0
GUI	7	0	1	2
SyncAll (ContentMediator)	0	0	0	3
CentralizedEvents (EventMediator)	0	0	0	4
SimpleReporter (EventMediator)	0	0	3	9
EventButton	1	0	0	0
EventCheckbox	1	0	2	0
EventListView	1	1	0	1
EventText	0	1	0	1
RSSItemRenderPane	0	0	0	0
Fetcher	2	7	0	1
UrlChecker	1	7	3	1
Main	1	0	3	9
Total	17	26	17	31