Project report

Cloudcomputing in bioinformatics

CeBiTec, University of Bielefeld

BBXYamlJava

Mark Ugarov

Lecturers: Dr. Alexander Sczyrba
Dipl.-Inform. Jan Krüger
Peter Belmann

Bielefeld, Decempber 2015

Contents

1 Abstract and Basics **1**

2 Implementation **3**

2.1 Basics of parsing 3

2.2 Enhance and simplify parsing for bioboxes 4

3 Usage **9**

3.1 Parsing an abstract data structure into a file 10

3.2 Parsing a file into an abstract data structure 10

Bibliography **13**

1 Abstract and Basics

BBXYamlJava is an extendible project trying to provide easy methods for parsing data used for bioboxes between an abstract data format and files (e.g. biobox.yaml - files) for maven projects. Thus it's not only necessary to implement an abstract data structure and the parsers but also to feature to go in depth of those structures with simple methods after parsing in from a file (what will be called Flatteners) or before parsing out into a file (which I described as Generators).

In addition the implementation contains interfaces so the core (means the parsers themselves) can be left unchanged if the project is extended by new types of data.
The first attempt was to create all methods and classes to parse a list of assemblers out of a file, updating from a server. The second was to create a biobox.yaml - file for bioboxes with simple commands. To test the flexibility I also implemented a data structure for evaluation of assemblers.

This project report will include some figures giving you an overview of the implementation. Only public methods will be shown and the return type is not shown as long it is described by the name of the method or a equivalent set-method exists. Also some names of classes and methods will be bold: A user will not need to invoke any other classes or methods to successfully parse from or into a file with this project (excluding specific cases).

This project [4] was implemented as a maven project using netbeans [5] including an ObjectMapper [6] as a dependency. Originally this was part of a bigger project named BOCCS [3] which was an attempt to make a GUI to run bioboxes [2] on a network.
All graphical elements where generated by using UMLet [7].

2 Implementation

This chapter describes crucial elements of the implementation and contains hints about potential further expansions. You may consider to have a look at figure 2 or at fullStructure.png in directory images while reading. Some important constants are stored in an abstract class `yamlparse.Constants` not shown in the figure. Elements involved in parsing from a file into a data structure are coloured green while elements involved in parsing from a data structure into a file are coloured cyan. Classes belonging to the data structure to parse to or from aregrey. The grey backlayers could be considered *levels* or *column*. Every further attempts to integrate new types of data to parse from or to a file should implement at least one entry in every column.

2.1 Basics of parsing

The project uses an `ObjectMapper` (see more by exploring `com.fasterxml.jackson.databind.ObjectMapper`) with a `YamlFactory` (see `com.fasterxml.jackson.dataformat.yaml.YAMLFactory`) to parse implementation of the interface `ParseableType` (`yamlparse.datatypes.ParseableType`) into files or the other way round. This requires implementations of `ParseableType` as a data structure representing several contents of a file. Currently there are two implementations of `ParseableType`: `BioboxTopType` (for biobox.yaml files, presently only to parse from a data structure into a file) and `Applications` (presently only to get a list of available bioboxes from a string into an abstract data format).

The actual parsing of instances of implementations of the `ParseableType` happens in extensions of the abstract data type `AbstractParser` which is extended into `AbstractInParser` and `AbstractOutParser` (see package `yamlparse.parser.abstracts`). Originally all these classes where also ment to be

interfaces, but JDK 1.7 seems to have a problem with interfaces of interfaces. Also I tried to implement only one `InParser` and one `OutParser` for all implementations of `ParseableType`, but this the `YamlFactory` is not able to process interfaces, so I had to create a `ApplicationsInparser` (to parse in instances of `Applications` from a string) and a `BioboxfileOutparser` (to parse out instances of `BioboxTopType` into a file). For further implementations: Every newly featured data type needs a similar parser itself unless there is a way to simplify the structure as described to only one `InParser` and one `OutParser` for all implementations of the interface `ParseableType` I don't know.

These two components are all you need to parse: the parser and the underlying data structure. But the usability of this without features would be very bad so further classes are implemented to enhance it.

2.2 Enhance and simplify parsing for bioboxes

To use these instances of `AbstractParser` some parameters are necessary: the path of the ouptut file or the content of the input file. To manage them I implemented an abstract class `AbstractParseManager` (which could have been an interface but interfaces do not allow constructors). Every extension of this class provides several methods to make parsing easier (see sections 3.2 and 3.1).

Trying to use these extensions still revealed some weaknesses: standard parameters had to be created every time and especially the construction of the `ParseableType` `BioboxTopType` didn't seem to satisfy the needs of a simple usage. Thus I created group of classes I named *Generators* which invoke an extension of `AbstractParseManager` with standard parameters (without making it difficult to change them if necessary) and provide more adequate methods to create instances of `BioboxTopType` and write them into a file. See more about `AssemblerGenerator` and `AssemblyEvaluationGenerator` in section 3.1 or `ParserGenerator` in section 3.1 or in Figure 2.2.

On the other hand I implemented a class `ApplicationFlattener` which creates a list of `FlatAssembler` to make it easy to get specific parameters out of an entry of `Applications`. Since instances of `Applications` have to be parsed before usage

this class is not used in the `ParserGenerator` but in the `Applications` itself. Read more about `ApplicationFlattener` and `FlatAssembler` in section 3.2 or have a look at Figure 2.2.

Before jumping into the next chapter I want to mention two deviations: As you can see the methods `getNewApplicationInparser()` and `getNewBioboxFileOutparser()` do not return instance of `AbstractParser` but extensions of `AbstractParseManager` since I decided to name those methods in view of readability and intuition instead of clear structure. This seems to be confusing at first, but be sure that the parsing itself is much more comfortable as you can see in 3.1. Feel free to use instances of `AbstractInParser` (e.g. `ApplicationInParser`) and `AbstractOutParser` (e.g. `BioboxfileOutparser`) for special needs. Also I named generators based on their purposed type of *biobox* if they are implemented for specific usage, e.g. `AssemblerGenerator` provides methods to write a *yaml* file for bioboxes containing an assembler. In contrary the `ParserGenerator` has the more general purpose to invoke instances of `AbstractParseManager` without further additions. We will discuss differences in chapter 3.

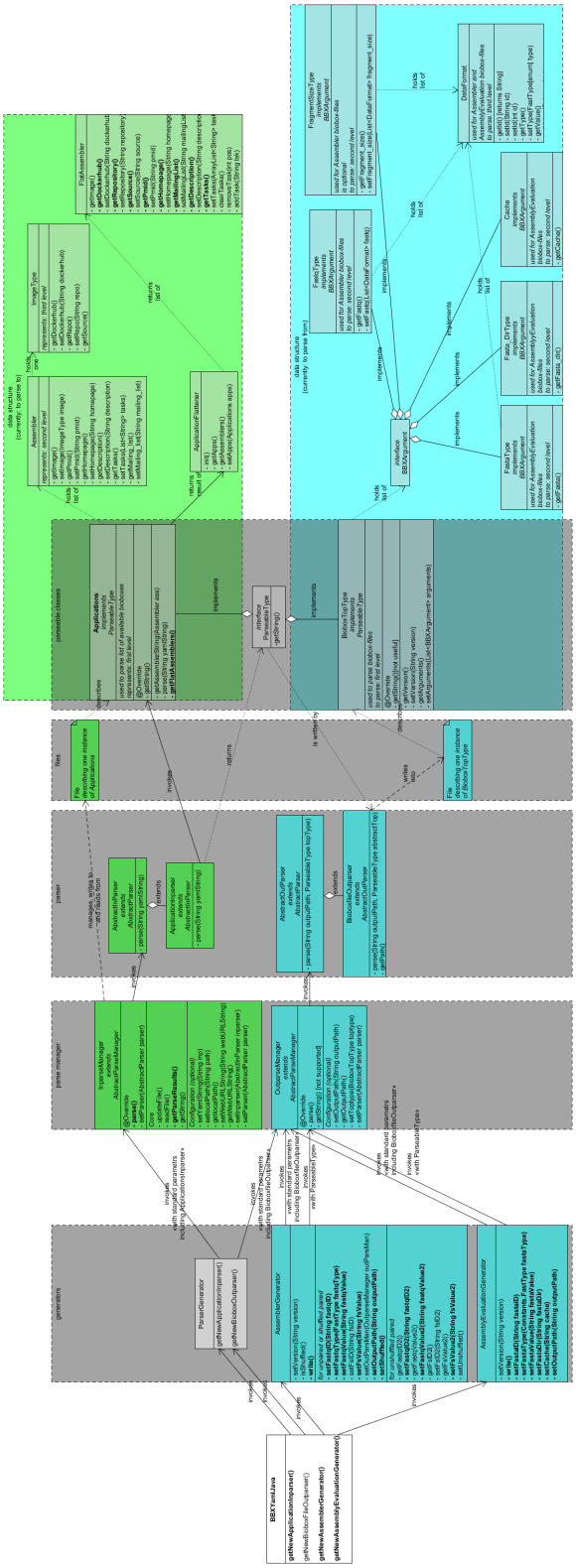

Figure 2.1: An overview of the full structure of the BBXYamlJava-Project. Since this is very tiny you may choose to have a look at the image in images/fullStructure.png

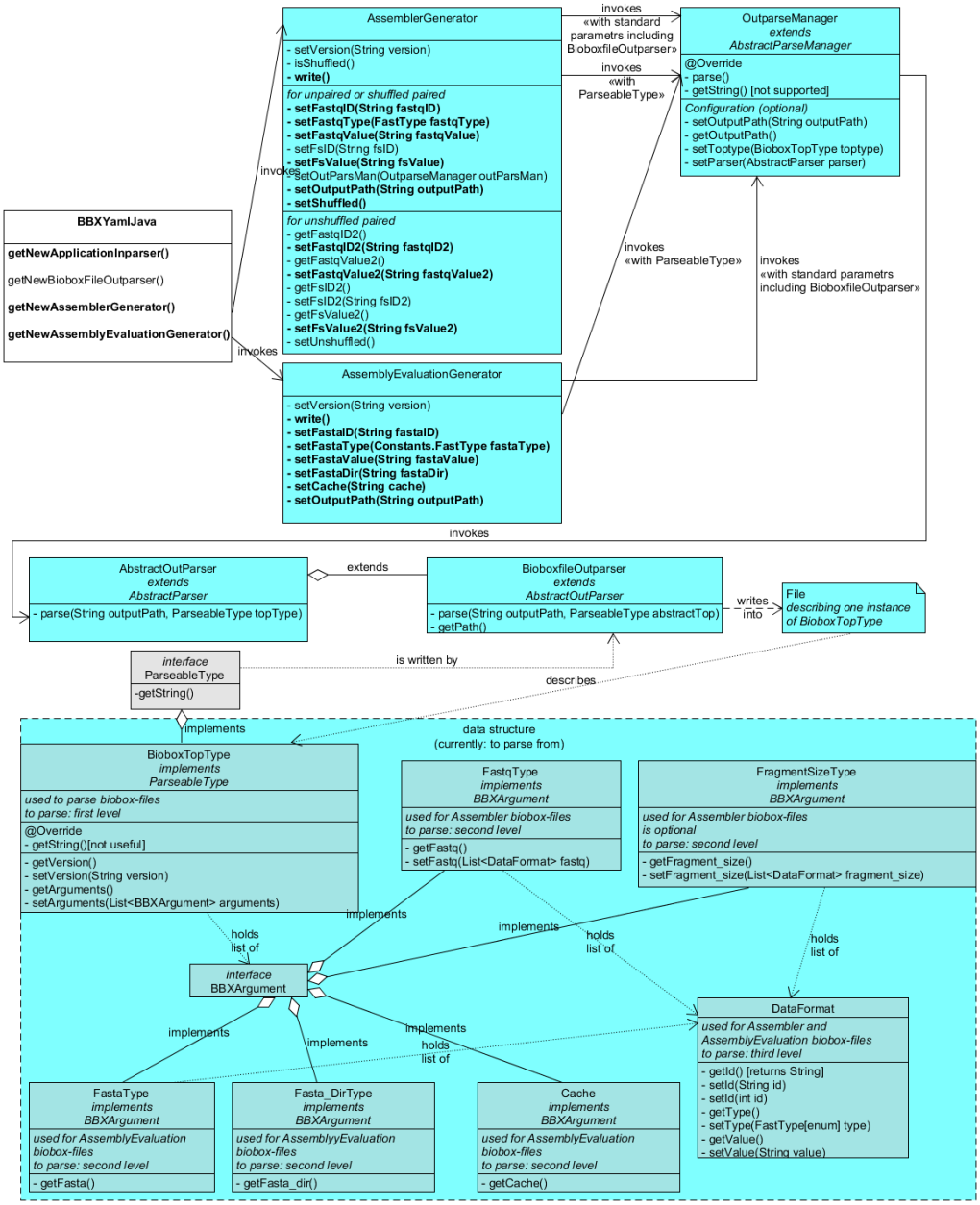

Figure 2.2: An overview of the part of the project providing elements and methods to parse out yaml files for bioboxes in an easy way.

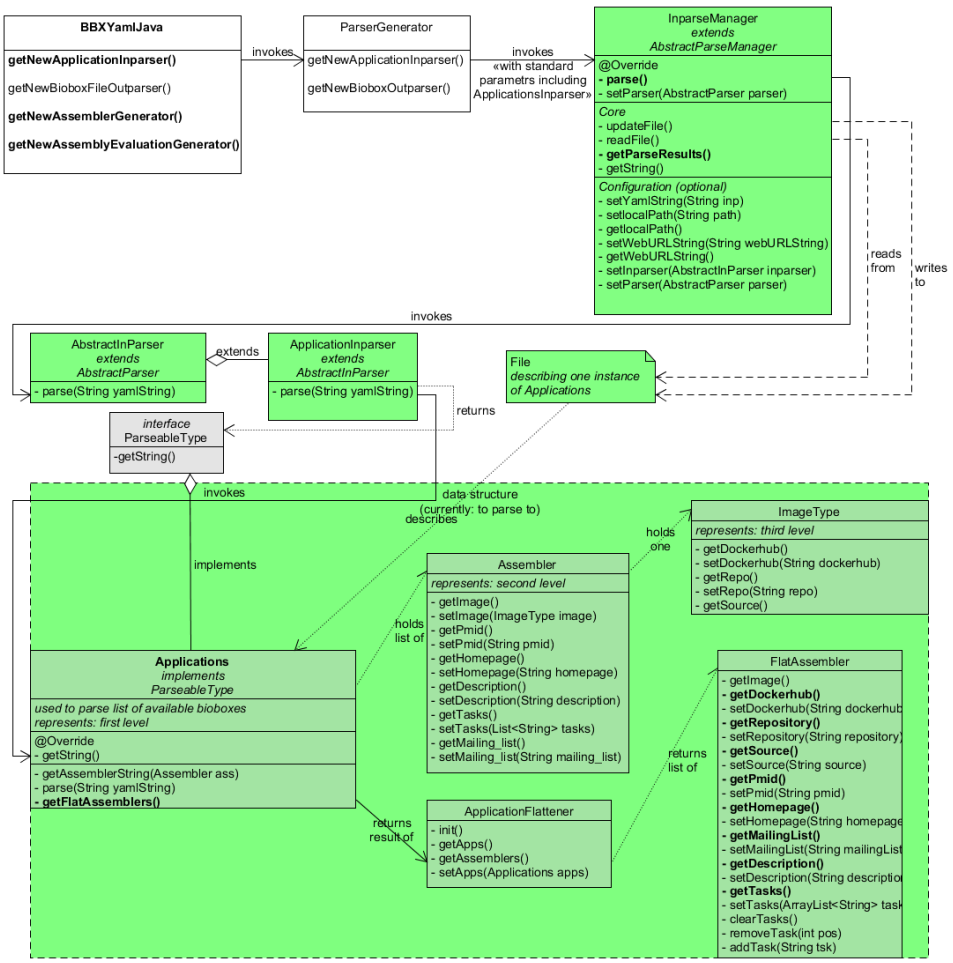

Figure 2.3: An overview of the part of the project providing elements and methods to parse a data structure out of a file in an easy way.

3 Usage

For a good usability take a look at the main class `BBXYamlJava` in package `yamlparse`. Every method uses (or returns) an instance of a group of classes I called *Generators*. The attempt is to invoke crucial classes (or methods) with standard parameters on one hand and make it easy to set crucial parameters on the other hand.

Please note that there are two levels of Generators in this project:

- The `ParserGenerator` only provides methods to invoke extensions of `AbstractParseManager` (`InparseManager` or `OutparseManager`) with standard parameters (including an implementation of `AbstractParser`). Those are implemented for file management like to set paths (to read from or to write to), to download and so on. They are sufficient to parse from a file into an abstract data structure.

- Other *Generators* for usage of specific implementations of `ParseableType`. Those are currently implemented for instances of `BioboxTopType` with differentiations: *bioboxes* containing assemblers do need other parameters than *bioboxes* for assembly evaluation. Their purpose is to ease the process of setting parameters of `BioboxTopType` in a significant amount. There is no need for the user to call other methods than provided by these *Generators* to generate a data type and write it into a file.

Thus don't confuse about the classes AssemblerGenerator and Assembler: An instance of `Assembler` holds values each describing a *biobox* containing an assembler while an instance of `AssemblerGenerator` can write a *yaml* file for the usage of the described assemblers.

3.1 Parsing an abstract data structure into a file

The current implementation only allows to parse out BioboxTopType, so this section will focus on this. If you want to apply your own data type please have a look at figure 2 and section 2.1.

Before parsing a new instance of BioboxTopType is needed. The project BBXYamlJava provides two ways to do so: Manually by the user or automatically by *Generators*. In case you did not read the beginning of this chapter: There is no need to use other methods other than provided by the `AssemblerGenerator` to write a *yaml* file for use with a *biobox* containing an assembler and the same principle applies to `AssemblyEvaluationGenerator`.

Those generators are a more comfortable way to create *yaml* files than invoke a `ParseableType` and parse them with an extension of `AbstractOutParser` (like `BioboxfileOutparser`): They hold the all parameters as a `ParseableType` (e.g `BioboxTopType`) with an usual number of entries would hold, but you can set parameters directly even if they have a higher level. E.g. you can set the `fastq` value by `#AssemblerGenerator().setFastqValue(String fastqValue)` instead of `((FastqType) #BioboxTopType.getArguments().get(0)).getFastq() .setValue(String value)`.

After setting those parameters, the user can use the `write()` method to write it to the `outputPath`.

By usage of `AssemblerGenerator` you will find an important field of type `boolean` named `shuffled` and another of type `Constants.FastType` called `fastqType`: You may know that there are two kind of reads which are *paired* and *unpaired*. The *fastq* files of *paired* reads may occur in two separate files (which means the sequences are *unshuffled*) or alternating in one file (which means the sequences aren't *shuffled* or might say *unshuffled*). Make sure you choose the right setting for your needs.

3.2 Parsing a file into an abstract data structure

To parse an existing file (or string) into an abstract data structure this project includes a class named `InparseManager`. If you choose to use this class with your own

extension of `AbstractParser`, you have to set it as well as at least one of the following parameters:

- the string to parse from
- a path to a file containing the string
- an url directing to a file and the path where to save it

If you choose the first one, consider to use extensions of `AbstractInparser` like `ApplicationInparser` instead.

Since the only current usage is to parse an instance of `Applications` (which contains a list of `Assembler` the following will describe how to use the instance you get by calling `#BBXYamlJava.getNewApplicationInparser()`.

This instance has a default url where to download a file containing all bioboxes currently available [1] at default. Of course you can set your own url in case you want to download any other file.

The following is optional before invoking `parse()`: By invoking the method `updateFile()` the file will be downloaded to a local path (which is the working directory at default) and the yaml string will be set to its content. If you have your own file instead, you may also set the local path and invoke `readFile()`. You can also set the yaml string manually as mentioned.

If at least one parameter is set as described above, you can invoke the method `parse()`. This will read from the local file if the yaml string is not set yet and download the file if the local file is empty.

After parsing you can get the result by `getParseResults()` which returns a `ParseableType` you have to cast to `Applications` which again is a tree like data structure. Thus it seems very unhandy use that class. A more comfortable solution might be to invoke `#Applications.getFlatAssemblers()` which returns a `ArrayList` of `FlatAssembler`. An instance of class `FlatAssembler` contains the same paramters as parsed instance of `Assembler`, but instead of tree like its method only return strings and `ArrayLists` of strings.

As you can see some results have to be cast. This is not very satisfying, but I did not find another solution supporting the structure of the project which I chose to be easy to extend.

Bibliography

[1] *Assembler List Git Raw*. 2015. URL: https://raw.githubusercontent.com/bioboxes/data/master/images.yml (visited on 10/08/2015).

[2] *Bioboxes homepage*. 2015. URL: bioboxes.org (visited on 11/08/2015).

[3] *BOCCS - The Biological Online Computational Cluster Service Git Repository*. 2015. URL: https://github.com/staynerzone/BioboxSITBSources (visited on 10/08/2015).

[4] MarkUgarov. *BBXYamlJava Git Repository*. 2015. URL: https://github.com/MarkUgarov/BBXYamlJava (visited on 10/08/2015).

[5] *Netbeans homepage*. URL: https://raw.githubusercontent.com/bioboxes/data/master/images.yml (visited on 01/05/2016).

[6] *Object Mapper description*. 2015. URL: http://fasterxml.github.io/jackson-databind/javadoc/2.2.0/com/fasterxml/jackson/databind/ObjectMapper.htmll (visited on 10/08/2015).

[7] *UMLet homepage*. 2015. URL: bioboxes.org (visited on 01/05/2015).