Programming assignment 2

Aljaž Koželj

Fakulteta za računalništvo in informatiko
Univerza v Ljubljani
ak6689@student.uni-lj.si

1 Introduction

For the second programming assignment, we were tasked with implementing three different approaches of structured data extraction from the web. The methods were: using regular expressions, using xpath expressions and implementing a Roadrunner-like algorithm.

This report covers the implementation and explanation for the first two methods (regular expressions and xpath expressions). Due to time constraints, we were unable to implement a Roadrunner-like algorithm. We are, however, confident in the implementation of the other two methods.

2 Technologies used

2.1 Java

We used Java as our programming language of choice. Java is a robust and mature language, that is familiar to the authors of this report. It offers a rich collection of libraries, which are provided by a large community. Both methods were implemented using this language.

2.2 Apache Maven

Apache Maven is a software project management and build automation tool. We use it to compile and build our implementations of methods into a single runnable jar. This tool is used also for including extra dependencies in our build. For this we use the maven-assembly-plugin plugin, which allows us to build a fat jar. A fat jar is a jar which includes all extra dependencies, so we don't have to provide them via classpath.

2.3 HtmlCleaner library

HtmlCleaner is a Java library, that is used for parsing and cleaning of HTML content. We use it to clean html, which results in valid markup. It is included in the fat jar via Maven.

2.4 Gson library

Gson is a Java library, that is used for serializing and deserializing of JSON content from and to Java objects. We use it to build the output of the methods. It is included in the fat jar via Maven.

3 Project structure

We used the standard Maven Java project structure as a base.

- **/:** In the root of the project we can find the Maven pom.xml, where the maven project is defined, together with the dependencies and plugins. We can also find a README.md file and the source folder.
- **/src/main/java/si/fri/kozelj/:** This package (directory) is the root folder of our Java code. It contains the main java class (App.java), an utility class (Utility.java) and further packages.
- **/src/main/java/si/fri/kozelj/models/:** This package contains all models used for serialization. There's at least one model per type of web page. Web pages with lists of items have an additional model class, that wraps a list of data items. All fields in the objects are annotated with Gson annotations, which allow for correct JSON object structure.
- **/src/main/java/si/fri/kozelj/parsers/:** Contains a parser interface (Parser.java), a parser factory (ParserFactory.java) and packages with method (and page type) specific implementations of the parser interface.
- **/src/main/java/si/fri/kozelj/parsers/regex/:** This package contains the abstract implementation of a regex parser and it's further implementations, that are specific for a page type.
- **/src/main/java/si/fri/kozelj/parsers/xpath/:** This package contains the abstract implementation of a xpath parser and it's further implementations, that are specific for a page type.
- **/src/main/resources/pages/:** This directory contains are web pages to be tested on. It includes both the provided pages and the pages we had to selected ourselves.

4 Components

4.1 App.java (main class)

This is the main class, that is run, when the fat jar is run. The main class expects two arguments to be included, else it exits the program. This program supports web page selection in two ways. The user can parse the web pages, that are included in the resources, or he can provide a path to his own local files. Do note that the local files have to be of a page type, that is supported by the program. This includes rtv, overstock and our own chosen web pages.

The first argument defines the method that should be used to extract structure data from a web page. The possible values are:

- **regex**
- **regex-via-path**
- **xpath**
- **xpath-via-path**

By providing a method name, that has a suffix of "-via-path", the program expects a path to a local html file to be provided in the second argument. Without this suffix, the program will attempt to access the web pages that are included in the resources folder.

The main class then loads the file content and asks the parser factory for a parser instance. The parser returns a serialized JSON object, which the main class then prints to the standard output.

4.2 Parser.java (parser interface)

This is the interface, that has to be implemented by all parsers. It contains a one default and one non-default method:

```
String parseJson();

default String cleanMatch(String match) {
    return match.replaceAll("\\n", " ")
                .replaceAll("[\\t]", "")
                .trim();
}
```

Method parseJson() is the one that provides the serialized JSON content, while the cleanMatch(String match) method provides very basic string cleaning, that couldn't be achieved with only regex or xpath expressions.

4.3 ParserFactory.java (parser factory)

This class works as a factory for the parsers. During construction of the factory, we provide the method and file name. Using those, it constructs the correct parser implementation.

The class also includes a private enum class, that is used to determine the page type:

```
private enum PageType {
    OVERSTOCK("^.*jewelry\\d+.html$"),
    RTV("^.*rtv\\d+.html$"),
    BOOKS("^.*books\\d+.html$"),
    NOT_FOUND("$a");

    private final String filePrefix;

    PageType(String filePrefix) {
```

```
            this.filePrefix = filePrefix;
        }

        public String getFilePattern() {
            return filePrefix;
        }

        public static PageType getType(final String
            fileName) {
            return Arrays.stream(PageType.values())
                .filter(o -> fileName.matches(o.
                    getFilePattern()))
                .findAny()
                .orElse(NOT_FOUND);
        }
    }
```

As seen above, the page types are defined by regex patterns. The program expects the html page types to respect the previously web page names, while allowing for a slight (number) variation. The regex patterns are useful here, because they work for both direct web page names or full paths to web page files.

4.4 AbstractRegexParser.java (abstract class for regex parser implementations)

This is the class, that has to be implemented by all regex parser implementations. It provides a method, that extracts all regex matches by using the given regex pattern object.

```
    List<String> getMatches(Pattern pattern, boolean
        cleanString) {
        Matcher matcher = pattern.matcher(pageContent);
        List<String> foundMatches = new ArrayList<>();
        while (matcher.find()) {
            String rawSubstring = pageContent.substring(
                matcher.start(1), matcher.end(1));
            foundMatches.add(cleanString ? cleanMatch(
                rawSubstring) : rawSubstring);
        }

        return foundMatches;
    }
```

Classes, that implement this abstract class, use this method to extract a specific item.

4.5 AbstractXPathParser.java (abstract class for xpath parser implementations)

This is the class, that has to be implemented by all xpath parser implementations.

During the parser's construction, it builds a org.w3c.dom.Document object, that can later be used to extract data via xpath expressions. The Document object is meant to represent XML objects. Here we encountered problems during the development, since the provided web pages were not valid XML syntax. There were many unclosed tags and those had to be corrected. In order to fix that, we used the HtmlCleaner library, as can be seen below:

```
public AbstractXPathParser(String pageContent) {
    this.gson = new Gson();
    this.xPath = XPathFactory.newInstance().newXPath
        ();

    TagNode tagNode = new HtmlCleaner().clean(
        pageContent);
    try {
        /**
         * Clean markup and save into string. We do
             this step, in order to preserve UTF-8
             encoding, while
         * building the Document object.
         */
        CleanerProperties cleanerProperties = new
            CleanerProperties();
        cleanerProperties.setCharset("UTF-8");
        String cleanedHtmlContent = new
            PrettyXmlSerializer(cleanerProperties).
            getAsString(tagNode);

        DocumentBuilderFactory dbFactory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.
            newDocumentBuilder();
        pageDocument = dBuilder.parse(new
            ByteArrayInputStream(cleanedHtmlContent.
            getBytes(Charset.forName("UTF-8"))));
    } catch (ParserConfigurationException |
        SAXException | IOException e) {
        throw new RuntimeException("Error while
            cleaning markup");
    }
}
```

From the initial pageContent string, we build another string object (after cleaning it) and that one is used in the creation of the Document object.

Data is extracted with the following code:

```
private NodeList getNodeList(String expression) {
    NodeList nodeList;
    try {
        nodeList = (NodeList) xPath.compile(
            expression).evaluate(pageDocument,
            XPathConstants.NODESET);
    } catch (XPathExpressionException e) {
        throw new RuntimeException("Error while
            evaluation expression: " + expression);
    }

    return nodeList;
}

List<String> getMatches(String expression) {
    NodeList nodeList = getNodeList(expression);

    List<String> foundMatches = new ArrayList<>();
    for (int i = 0; i < nodeList.getLength(); i++) {
        String nodeValue = nodeList.item(i).
            getNodeValue();
        foundMatches.add(cleanMatch(nodeValue));
    }

    return foundMatches;
}
```

We provide a valid xpath expression and get returned an iterable NodeList object, that contains all matches. These are then used in methods that iterate over the object. In some cases we use an additional regex pattern object together with the xpath expression, in order to extract the correct part of the xpath expression result:

```
List<String> getMatches(String expression, Pattern
    pattern) {
    List<String> foundMatches = getMatches(expression
        );
    List<String> transformedMatches = new ArrayList
        <>();

    for (String match : foundMatches) {
        Matcher matcher = pattern.matcher(match);
```

```
            if (matcher.find()) {
                transformedMatches.add(match.substring(
                    matcher.start(1), matcher.end(1)));
            } else {
                throw new RuntimeException("Error while
                    trying to apply pattern: " + pattern +
                     " to match: " + match);
            }
        }

        return transformedMatches;
    }
```

5 Web page selection

For this assignment, we had to select our own two similar web pages. We picked the following ones:

- https://www.goodreads.com/list/show/3.Best_Science_Fiction_Fantasy_Books
- https://www.goodreads.com/list/show/425.Weirdest_Books_Ever

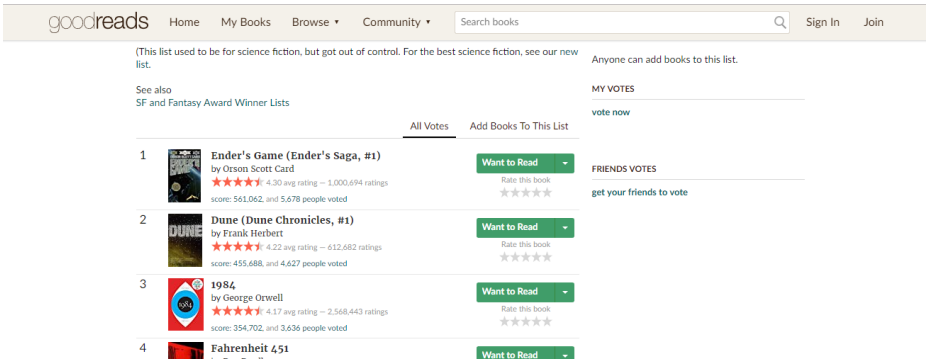


Fig. 1. Web page screenshot

We named them as books01.html and books02.html. Both are included in the resource folder. This pair of web pages include a list of 100 book items on each page. Here we picked the 6 data items, we were to extract:

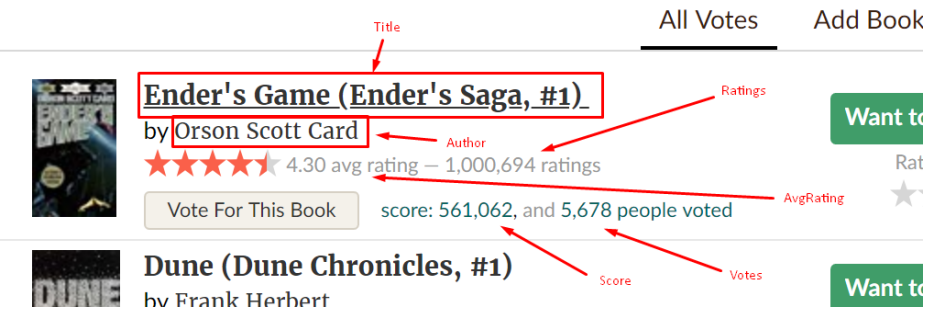


Fig. 2. Data items

6 Web page target data

6.1 Overstock

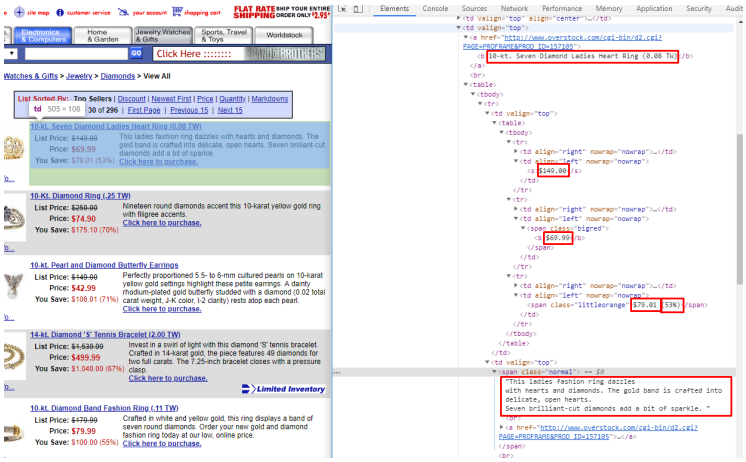


Fig. 3. Data item with data objects marked

We can see from the image above, that there was no distinct class or identifier, that could be used in the regex expressions.

6.2 Rtv

Fig. 4. Page with data objects marked

With this page, we could extensively make use of very specific class-es, that we used to define the regex expressions.

6.3 Books

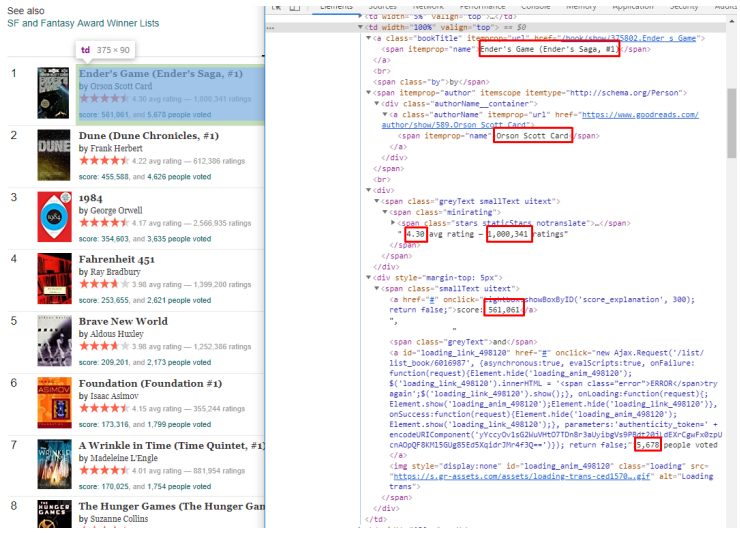


Fig. 5. Data item with data objects marked

Here too, we make use of very specific class-es. We use constant strings as well.

7 Extraction outputs

In this section we display a part of the outputs. Full and all outputs are available in the /outputs folder in the project.

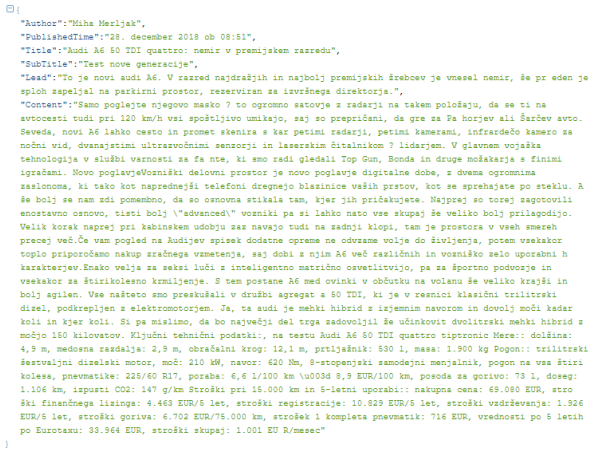
7.1 Overstock


```
{
  "data":[
    {
      "Title":"10-kt. Seven Diamond Ladies Heart Ring (0.08 TW)",
      "ListPrice":"$149.00",
      "Price":"$69.99",
      "Saving":"$79.01",
      "SavingPercent":"(53%)",
      "Content":"This ladies fashion ring dazzles with hearts and diamonds. The gold band is crafted into
      delicate, open hearts. Seven brilliant-cut diamonds add a bit of sparkle."
    },
    {
      "Title":"10-Kt. Diamond Ring (.25 TW)",
      "ListPrice":"$250.00",
      "Price":"$74.90",
      "Saving":"$175.10",
      "SavingPercent":"(70%)",
      "Content":"Nineteen round diamonds accent this 10-karat yellow gold ring with fili gree accents."
    },
    {
      "Title":"10-kt. Pearl and Diamond Butterfly Earrings",
```


Fig. 6. Overstock output

Here is the output of the overstock web page being parsed by regex. Note that around the JSON array, we have a JSON object, with a data property. In this property, we save the JSON array.

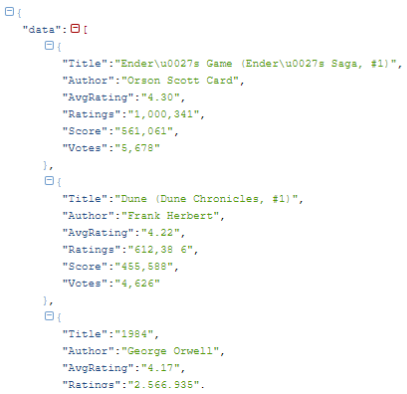
7.2 Rtv



{
"Author":"Miha Merljak",
"PublishedTime":"28. december 2018 ob 08:51",
"Title":"Audi A6 50 TDI quattro: nemir v premijskem razredu",
"SubTitle":"Test nove generacije",
"Lead":"To je novi audi A6. V razred najdražjih in najbolj premijskih žrebcev je vnesel nemir, še pr eden je sploh zapeljal na parkirni prostor, rezerviran za izvršnega direktorja.",
"Content":"Samo poglejte njegovo masko ? to ogromno satovje z radarji na takem položaju, da se ti na avtocesti tudi pri 120 km/h vsi spoštljivo umikajo, saj so prepričani, da gre za Pa horjev ali Šarčev avto. Seveda, novi A6 lahko cesto in promet skenira s kar petimi radarji, petimi kamerami, infrardečo kamero za nočni vid, dvanajstimi ultrazvočnimi senzorji in laserskim čitalnikom ? lidarjem. V glavnem vojaška tehnologija v službi varnosti za fa nte, ki smo radi gledali Top Gun, Bonda in druge možakarja s finimi igračami. Novo poglavjeVozniški delovni prostor je novo poglavje digitalne dobe, z dvema ogromnima zaslonoma, ki tako kot naprednejši telefoni dregnejo blazinice vaših prstov, kot se sprehajate po steklu. A še bolj se nam zdi pomembno, da so osnovna stikala tam, kjer jih pričakujete. Najprej so torej zagotovili enostavno osnovo, tisti bolj \"advanced\" vozniki pa si lahko nato vse skupaj še veliko bolj prilagodijo. Velik korak naprej pri kabinskem udobju zaz navajo tudi na zadnji klopi, tam je prostora v vseh smereh precej več.Če vam pogled na ludijev spisek dodatne opreme ne odvzame volje do življenja, potem vsekakor toplo priporočamo nakup zračnega vzmetenja, saj dobi z njim A6 več različnih in vozniško zelo uporabni h karakterjev.Enako velja za seksi luči z inteligentno matrično osvetlitvijo, pa za športno podvozje in vsekakor za štirikolesno krmiljenje. S tem postane A6 med ovinki v občutku na volanu še veliko krajši in bolj agilen. Vse našteto smo preskušali v družbi agregat a 50 TDI, ki je v resnici klasični trilitrski dizel, podkrepljen z elektromotorjem. Ja, ta audi je mehki hibrid z izjemnim navorom in dovolj močI kadar koli in kjer koli. Si pa mislimo, da bo največji del trga zadovoljil že učinkovit dvolitrski mehki hibrid z močjo 150 kilovatov. Ključni tehnični podatki:, na testu Audi A6 50 TDI quattro tiptronic Mere:: dolžina: 4,9 m, medosna razdalja: 2,9 m, obračalni krog: 12,1 m, prtljažnik: 530 l, masa: 1.900 kg Pogon:: trilitrski šestvaljni dizelski motor, moč: 210 kW, navor: 620 Nm, 8-stopenjski samodejni menjalnik, pogon na vsa štiri kolesa, pnevmatike: 225/60 R17, poraba: 6,6 l/100 km \u003d 8,9 EUR/100 km, posoda za gorivo: 73 l, doseg: 1.106 km, izpusti CO2: 147 g/km Stroški pri 15.000 km in 5-letni uporabi:: nakupna cena: 69.080 EUR, stro ški finančnega lizinga: 4.463 EUR/5 let, stroški registracije: 10.829 EUR/5 let, stroški vzdrževanja: 1.926 EUR/5 let, stroški goriva: 6.702 EUR/75.000 km, strošek 1 kompleta pnevmatik: 716 EUR, vrednosti po 5 letih po Eurotaxu: 33.964 EUR, stroški skupaj: 1.001 EU R/mesec"
}

Fig. 7. Rtv output

7.3 Books



{
"data":[
{
"Title":"Ender\u0027s Game (Ender\u0027s Saga, #1)",
"Author":"Orson Scott Card",
"AvgRating":"4.30",
"Ratings":"1,000,341",
"Score":"561,061",
"Votes":"5,678"
},
{
"Title":"Dune (Dune Chronicles, #1)",
"Author":"Frank Herbert",
"AvgRating":"4.22",
"Ratings":"612,38 6",
"Score":"455,588",
"Votes":"4,626"
},
{
"Title":"1984",
"Author":"George Orwell",
"AvgRating":"4.17",
"Ratings":"2,566,935",

Fig. 8. Books output

Same as the overstock items we wrap the array with a json object, who has a data property (that evaluates to the array of extracted data items).

8 Regex data extraction

In this section we show the regex expressions we used for data extraction. These are provided by images of expressions, since the latex formatting is a nightmare for those regex expressions. The source regex expressions can be accessed in the java classes, that are contained in the si.fri.kozelj.parsers.regex package.

8.1 Overstock (OverstockRegexParser.java)

8.1.1 Title

```
<a .*?><b>(.*)</b><\/a><br>\s[\r\n]*<table>
```

Fig. 9. Title pattern

Here we take advantage of the fact that the title is inside a link tag and the further wrapped between bold tags. To further filter the selection, the expression expects a break tag and a table tag in the next few lines.

8.1.2 ListPrice

```
List Price:<\/b><\/td><td .*?><s>(.*)</s><\/td>
```

Fig. 10. ListPrice pattern

This expression was rather simple, since we can use the string "List Price" to pinpoint the content easily. We capture the text between the strikethrough tags.

8.1.3 Price

```
[^\s]Price:<\/b><\/td><td .*?><span .*?><b>(.*)</b><\/span><\/td>
```

Fig. 11. Price pattern

This expression is quite similar to the one above. Only difference is, that the text is wrapped in an extra span tag. We also have to be careful here, if we want to extract the span tag together with the text, so we don't match the "List Price" data as well.

8.1.4 Saving

```
You Save:<\/b><\/td><td .*?><span .*?>(.*) \(\d+%\)<\/span><\/td>
```

Fig. 12. Saving pattern

Again, we use the common string "You Save", in order to find the correct location. Here there are two items we want, so we need to be careful to select only the first one.

8.1.5 Saving Percent

```
You Save:<\/b><\/td><td .*?><span .*?>.+? (\(\d+%\))<\/span><\/td>
```

Fig. 13. SavingPercent pattern

Nearly identical regex expression. We just change the selection to the second item in the span. We had to be careful and not forget to escape the brackets with a backslash.

8.1.6 Content

```
You Save:.*[\r\n]+.*[\r\n]+.*<td .*?><span .*?>((.|\r|\n)*?)<br>
```

Fig. 14. Content pattern

Here we reused the previous regex expression to move further down the string to where the content is located. This includes matching multiple new line and return characters and multiple rows, until we reach a html row item and a span inside of it. We select the content of the preceding span tag up until the break tag. Thus we don't have to filter out the link tag later on.

8.2 Rtv (RtvRegexParser.java)

8.2.1 Author

```
<div class="author-name">(.*?)<\/div>
```

Fig. 15. Author pattern

We use a class that uniquely defines the data we are looking for.

8.2.2 PublishedTime

```
<div class="publish-meta">(?:.|\r|\n)*?([^\t]*?)<br>
```

Fig. 16. PublishedTime pattern

Here we had to modify the regex expression, because there's some extra, unwanted, data in the second part of the div with the target class. We capture from a new line to when the break tag appears.

8.2.3 Title

```
<h1>(.*)<\/h1>
```

Fig. 17. Title pattern

Title is uniquely defined with a h1 tag.

8.2.4 SubTitle

```
<div class="subtitle">(.*)<\/div>
```

Fig. 18. SubTitle pattern

Again, we use a class that uniquely defines the data we are looking for.

8.2.5 Lead

```
<p class="lead">(.*)<\/p>
```

Fig. 19. Lead pattern

Again, we use a class that uniquely defines the data we are looking for.

8.2.6 Content

```
<article class="article">([\s\S\r]*)<\/article>
```

Fig. 20. Content pattern

Here we capture all inside the article tags. We had to do some extra processing, in order to make the output prettier. The method is the following one: si.fri.kozelj.Utility#cleanRtvContent . We used this to extract only content from p tags, while also trying to do some basic handling of list items in the article.

8.3 Books (BookRegexParser.java)

8.3.1 Title

```
<a class="bookTitle" .*?>[\s\S\r]*?<span .*?>(.*?)<\/span>
```

Fig. 21. Title pattern

This data was well defined by the "bookTitle" class.

8.3.2 Author

```
<a class="authorName" .*?><span .*?>(.*?)<\/span>
```

Fig. 22. Author pattern

This data was well defined by the "authorName" class.

8.3.3 AvgRating

```
([\d\.]+)\savg rating
```

Fig. 23. AvgRating pattern

Here we match to the string "avg rating". Our target is located left of this string.

8.3.4 Ratings

```
avg rating [^\d]*(.*?) ratings
```

Fig. 24. Ratings pattern

Ratings data is nested between ”avg rating” and ”ratings” strings.

8.3.5 Score

```
<a .*?>score: (.*?)<\/a>
```

Fig. 25. Score pattern

Score data is defined by the ”score” string.

8.3.6 Votes

```
<a .*?>(.*?) people voted<\/a>
```

Fig. 26. Votes pattern

Votes data is defined by the ”people voted” string.

9 XPath data extraction

In this section we show the xpath expressions we used for data extraction. Along with those, there are sometimes additional regex expressions, that were necessary to extrac data. Both are provided by images of expressions, since the latex formatting is a nightmare for those regex expressions. The source regex expressions can be accessed in the java classes, that are contained in the si.fri.kozelj.parsers.xpath package.

9.1 Overstock (OverstockXPathParser.java)

Here we made all xpath expressions by traversing a DOM tree of multiple table elements. We won't go into detail at each one, because there is nothing concrete to be said. All expressions share the same relative prefix, since they are all located near each other. After we found this root, it was just a matter of choosing the right path and extracting text data at the end.

A good decision to note, is at the first tr tag in all expressions. Since we selected the second tr child there, we automatically removed all occurrences, where there is only a single tr child there. Similar decisions were made in some expressions towards the end.

9.1.1 Title

```
//table/tbody/tr[2]/td/table/tbody/tr/td/table/tbody/tr/td/a/b/text()
```

Fig. 27. Title pattern

9.1.2 ListPrice

```
//table/tbody/tr[2]/td/table/tbody/tr/td/table/tbody/tr/td[2]/table/tbody/tr/td[1]/table/tbody/tr[1]/td[2]/s/text()
```

Fig. 28. ListPrice pattern

9.1.3 Price

```
//table/tbody/tr[2]/td/table/tbody/tr/td/table/tbody/tr/td[2]/table/tbody/tr/td[1]/table/tbody/tr[2]/td[2]/span/b/text()
```

Fig. 29. Price pattern

9.1.4 Saving

```
//table/tbody/tr[2]/td/table/tbody/tr/td/table/tbody/tr/td[2]/table/tbody/tr/td[1]/table/tbody/tr[3]/td[2]/span/text()
```

Fig. 30. Saving pattern

9.1.5 Saving Percent

```
//table/tbody/tr[2]/td/table/tbody/tr/td/table/tbody/tr/td[2]/table/tbody/tr/td[1]/table/tbody/tr[3]/td[2]/span/text()
```

Fig. 31. SavingPercent pattern

9.1.6 Content

```
//table/tbody/tr[2]/td/table/tbody/tr/td/table/tbody/tr/td[2]/table/tbody/tr/td[2]/span/text()[1]
```

Fig. 32. Content pattern

9.2 Rtv (RtvXPathParser.java)

Similarly to the regex Rtv parser, we use specific / unique classes extensively

9.2.1 Author

```
//div[@class='author-name']/text()
```

Fig. 33. Author pattern

We use a class that uniquely defines the data we are looking for and collect it's text content.

9.2.2 PublishedTime

```
//div[@class='publish-meta']/text()[1]
```

Fig. 34. PublishedTime pattern

Again, we access the node via a specific class and then choose the first text chunk.

9.2.3 Title

```
//h1/text()
```

Fig. 35. Title pattern

Title is uniquely defined with a h1 tag, we take the text value from it.

9.2.4 SubTitle

```
//div[@class='subtitle']/text()
```

Fig. 36. SubTitle pattern

Again, we use a class that uniquely defines the data we are looking for and retrieve its text value.

9.2.5 Lead

```
//p[@class='lead']/text()
```

Fig. 37. Lead pattern

Again, we use a class that uniquely defines the data we are looking for and retrieve its text value.

9.2.6 Content

```
//article
```

Fig. 38. Content pattern

Here we retrieve the whole article node with all it's contents (with the method si.fri.kozelj.parsers.xpath.AbstractXPathParsergetRawMatches). We don't extract the text value, unlike other expressions. Instead we take the node and fully transform it into a string of html nodes (with the method si.fri.kozelj.Utility#getNodeString). After that we filter the content with the same method as the regex parser (si.fri.kozelj.Utility#cleanRtvContent) and the results should be nearly the same. We did notice a few encoding differences between the regex and xpath ways.

9.3 Books (BookXPathParser.java)

With this parser, we encountered some interesting situations, where a xpath expression was not enough to extract the correct content. In some cases, we needed to use extra regex expressions to extract the correct data.

Similarly

9.3.1 Title

```
//a[@class='bookTitle']/span/text()
```

Fig. 39. Title pattern

This data was well defined by the "bookTitle" class. We extract the text from the inner span.

9.3.2 Author

```
//a[@class='authorName']/span/text()
```

Fig. 40. Author pattern

This data was well defined by the "authorName" class. We extract the text from the inner span.

9.3.3 AvgRating

```
//span[@class='minirating']/text()[2]
```

Fig. 41. AvgRating pattern

```
(.*?) avg rating
```

Fig. 42. AvgRating extra regex pattern

The xpath returns a string that contains both avg rating and ratings data information. The location of this string is specified by the "minirating" class. With this data type we match to the string "avg rating" and extract the left side information.

9.3.4 Ratings

```
//span[@class='minirating']/text()[2]
```

Fig. 43. Ratings pattern

```
avg rating.*?(\d+.*?) ratings
```

Fig. 44. Ratings extra regex pattern

Here we reuse the same xpath expressions that we used above. Ratings data is nested between "avg rating" and "ratings" strings, so we use an extra regex expression to extract the necessary data.

9.3.5 Score

```
//a[@class='bookTitle']/following-sibling::div[2]/span/a[1]/text()
```

Fig. 45. Score pattern

```
score: (.*)$
```

Fig. 46. Score extra regex pattern

Here we use xpath axes to select the second div tag sibling of the link tag with class "bookTitle". AFter that we follow the structure to the text value. Again, the text value isn't just the data we need, so we extract the necessary

9.3.6 Votes

```
//a[@class='bookTitle']/following-sibling::div[2]/span/a[2]/text()
```

Fig. 47. Votes pattern

```
^(.*?) people voted
```

Fig. 48. Votes extra regex pattern

Votes data is extracted with a similar xpath expression as the score data. Again we have to apply a regex pattern, to extract the necessary data.

10 Conclusion

Despite missing one of the three methods, we would consider this assignment as a success. We learned a lot about regex and xpath patterns and both were implemented in an orderly fashion. There were some starter issues in how to approach a new pattern, but over time it came about naturally.