Software Architecture

1.1 Version

Version	Date	Change	Author
0.1	8.10.2014	Setup document	JR
0.2	12.10.2014	Class diagrams	JR
0.3	16.10.2014	Text	JR
0.4	20.10.2014	Activity diagram	JR
1.0	21.10.2014	Grammar, Diagram fixes	JR
1.1	22.10.2014	Adding description of all business logic classes, approach and extended introduction	JR
1.2	23.10.2014	Grammar	JR,LR

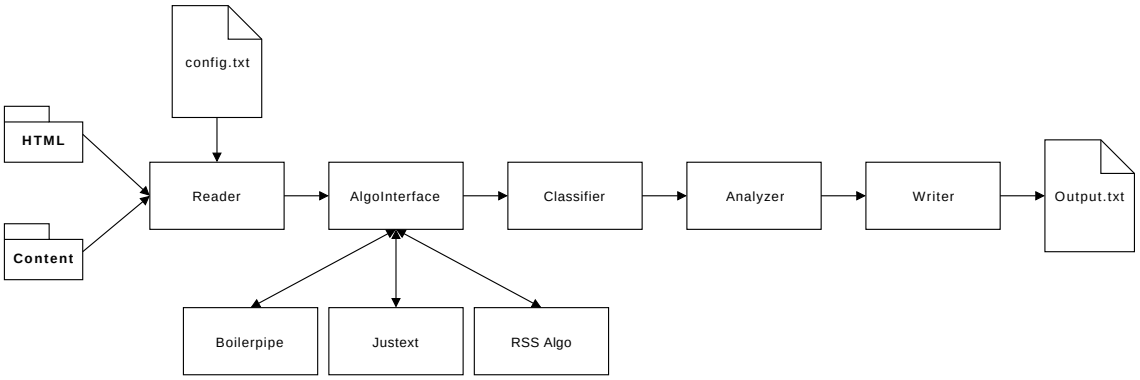
1.2 Introduction

This document describes the software architecture of the context extraction test framework. The context extraction test framework will perform automated text extraction on a set of HTML test data with two to three different text extraction algorithms. After measuring the performance of each algorithm, an output file with the measured results is generated.

1.3 Logical view

1.3.1 Approach

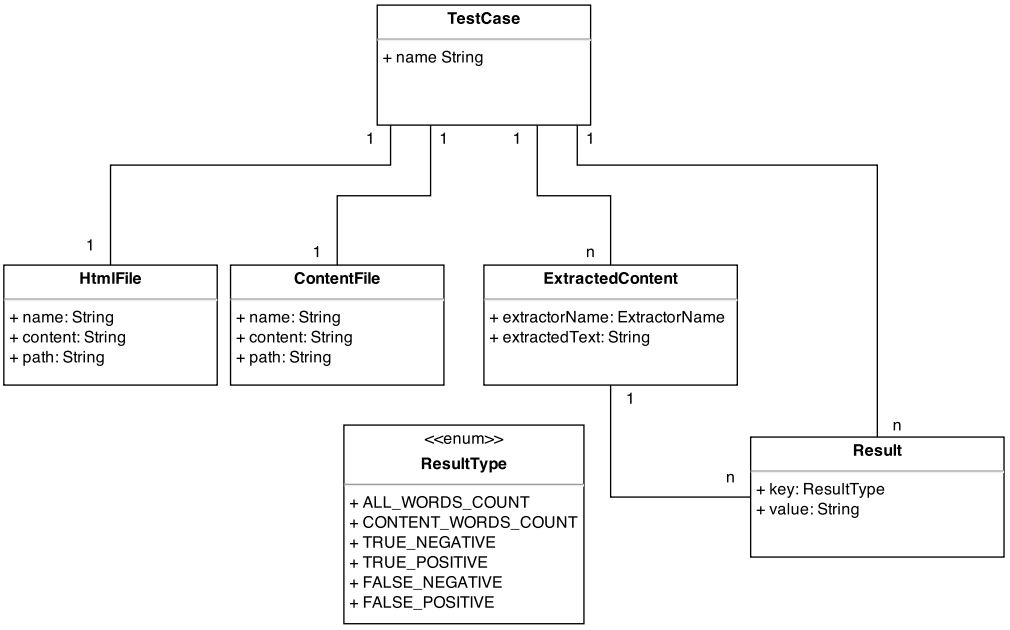
This section describes the approach for elaborating the software architecture. The following figure from the software requirement specification is the basis for elaborating the software architecture.



First all possible entities in the domain are identified and a data model is elaborated. The data model is described in 1.4. Then the business logic is researched based on the figure above. With this knowledge, the five packages main, reader, classifier, analyzer and writer are defined. The functions for each package is then evaluated and split into single classes. During the implementation of the first approach, some classes are adapted due to unforeseen circumstances. The outcome is described in 1.5.

1.4 Data model

The following diagram shows the data model of the application.



1.4.1 TestCase

A TestCase object is generated for each HTML/content file pair in the input folders. A TestCase has a a name which is unique and which matches the name of the content and HTML file.

1.4.2 HtmlFile

Each TestCase has an HtmlFile object. It contains the content of the actual HTML file as String and the file path.

1.4.3 ContentFile

Each TestCase has a ContentFile object. It contains the content of the actual text file as String and the file path.

1.4.4 ExctractedContent

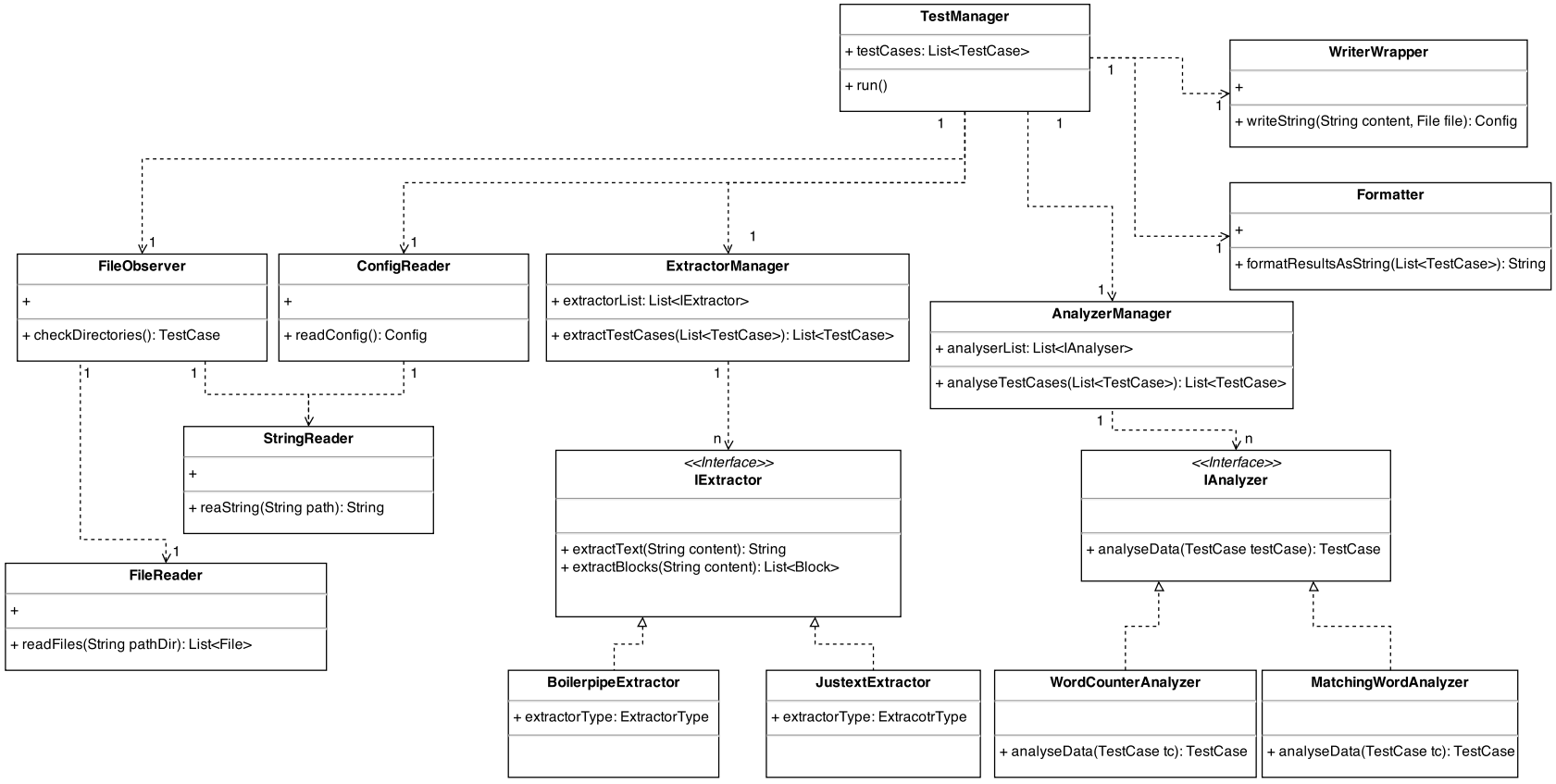
Each TestCase can have multiple ExctractedContent objects. Each of them represents a result of a content extraction from an extractor such as Justext or Boilerpipe.

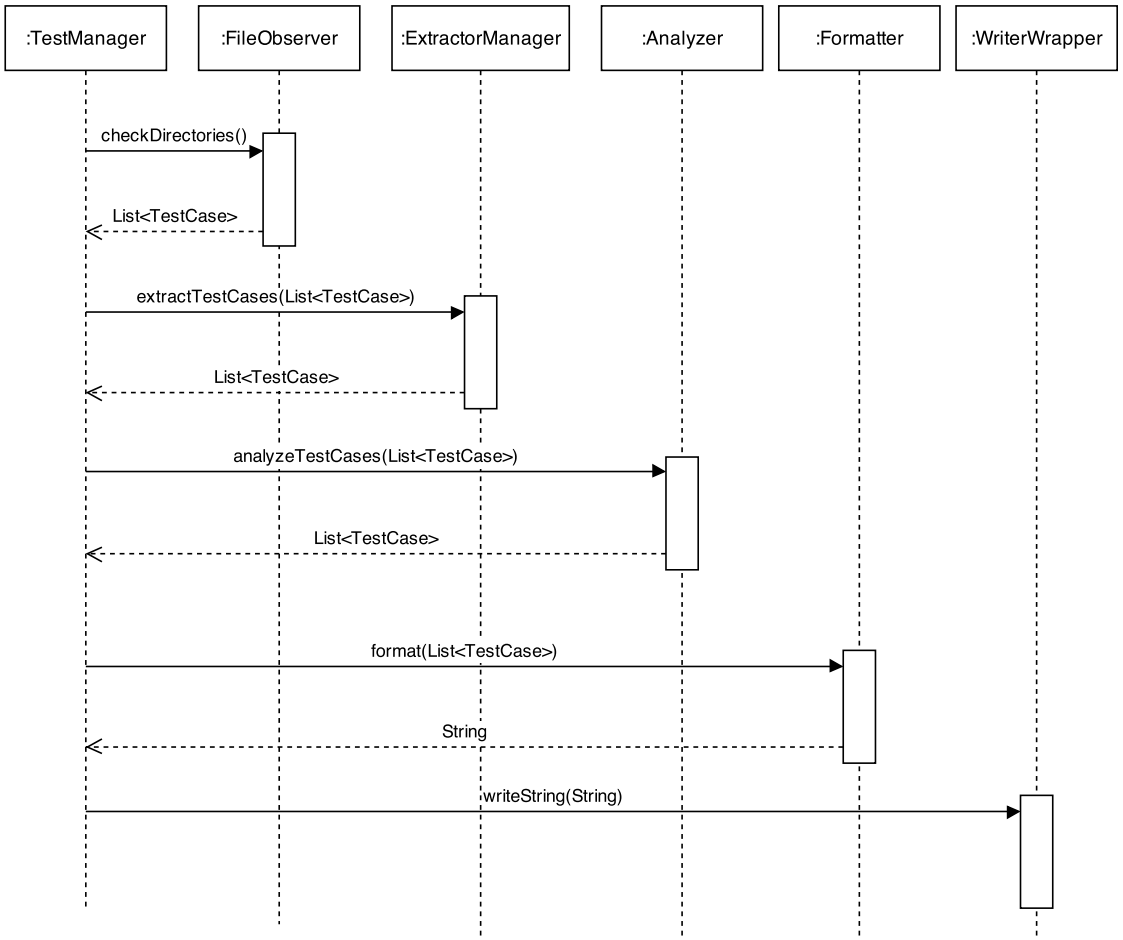
1.4.5 Result

A TestCase or an ExctractedContent object can have Result objects. The results are key value pairs which represent analytical data. An example for a Result related to a TestCase would be the word count of the content file, which is generally valid. An example for a Result related to an ExctractedContent would be the word count of true negative words, which is only valid for one specific ExctractedContent.

1.5 Business Logic

The following diagram shows the business logic of the application. The diagram does not show all of the classes but the most important ones.





The data model is passed through the business logic and is enriched with data during the test procedure. First the input directories are checked for files by the FileObserver and TestCases are generated for each file pair with the same name. Then all the TestCases are then handed over to the ExtractorManager. The ExtractorManager extracts each TestCase with all available implementations of IExtractor and puts the Results into ExtractionResults. After that, all the TestCases are handed over to the Analyzer which runs each implementation of IAnalyzer. Each IAnalyzer produces at least one Result and puts it into the TestCase. To simplify the diagram, only two Analyzers are drawn. After generating some Results, the Formatter serializes the Result Objects into a String as a CSV table and the WriterWrapper persists the CSV data into an output file.

1.5.1 Description of single classes

Class	Package	Description
TestManager	testManager	The TestManager class manages the whole business logic that manages TestCase objects through the whole test process from reading the file content to writing the test results into an output file.
FileObserver	reader	The FileObserver class checks the HTML and content directory for files of the same name and creates TestCases from each found pair. The folders are checked with the FileReader class and the content of the files are read with the StringReader class.
StringReader	reader	The StringReader class reads a text file and returns the content as String. The class is made for easier mocking of the BufferedReader so that testing of other classes which are dependent on external files becomes much easier.
FileReader	reader	The FileReader class returns a File objects for each found file in a directory given by a parameter.
ExtractorManager	classifier	The ExtractorManager manages all available Extractors. Each extractor which is used for the actual test must be initialized in this class and added to the ExtractorList. Each TestCase is then extracted by every IExtractor in the ExtractorList.
IExtractor	classifier	The IExtractor is the interface to the different extractor. The interface is very lightweight. The parameter is the text that should be extracted and the return value is the extracted text.
BoilerpipeExtractor	classifier	The BoilerpipeExtractor implements the IExtractor and is the interface to the Boilerpipe package. It handles all dependencies on the Boilerpipe package and returns the extracted content as a string.

JustextExtractor	classifier	The BoilerpipeExtractor implements the IExtractor and is the interface to the Justext python program. The Java ProcessBuilder is used to create operating processes. One can then perform operating system commands and run the python script. The python script creates a text file with the extracted content which is read by the JustextExtractor class and returned as String.
AnalyzerManager	analyzer	The AnalyzerManager manages all available analyzers. Each analyzer which is used for the actual test must be initialized in this class and added to the AnalyzerList. Each TestCase is then analyzed by every IAnalyzer in the AnalyzerList.
IAnalyzer	analyzer	The IAnalyzer interface is a simple interface to the different analyzers. Each analyzer can generate one or more Result objects.
WordCounterAnalyzer	analyzer	The WordCounterAnalyzer is a simple analyzer which counts all words of the content file, the HTML file and each extracted content.
MatchingWordAnalyzer	analyzer	The MatchingWordAnalyzer compares the content file with each extracted content and calculates the values for true positive, true negative, false positive and false negative.
Formatter	writer	The Formatter class formats a string from all TestCase objects as a CSV file structure. This means that a table with the Result keys as table header for each column is created. For each TestCase a row with the Result value field as values is added to the table. The outcome is a CSV file which can easily be imported into another program such as Excel so that one can work with the data.
WriterWrapper	writer	The writer wrapper writes a string into a text file. It is used to wrap the BufferedWriter so that mocking and testing of dependent classes is easier.
ConfigReader	reader	The ConfigReader class reads the config.txt file and puts the key value pairs into a HashMap.

1.6 Development view

This chapter describes the used frameworks and tools which are used during the development process.

1.6.1 git

Git is a distributed version control system. Unlike other version control systems like Subversion, git is not using a central server but each user has his own copy with the complete history on the local system. It is much easier to work with additional branches or tags. Because of these advantages and because Layzapp is working with it as well, git was chosen to use for this project for the code sources and as well for the documentation. The resources are open source and are available under following links.

- code: https://github.com/heya87/pawiTwo
- documentation: https://github.com/heya87/pawi_doc

1.6.2 Gradle

Gradle is a project automation tool which uses a Groovy-based domain-specific language (DSL) instead of the more traditional XML form of declaring the project configuration. All the dependencies of the project are handled with Gradle and it is very easy to deploy on a new system. The user only needs to download the git repository and can build the project with the Gradle wrapper without installing any new software. All dependencies are then downloaded automatically and the user can start working without caring about missing packages. The build process is defined in the build.gradle file which is located in the source directory of the project.

1.6.3 Travic CI

Travic CI is an open source build server. It is easy to use in combination with git. One only needs to add a config file in the source directory of the project and define the git repository. Afterwards the project is build for every change. If the build does not pass, a mail is sent to the user. The actual status of the build can be found under following link https://travis-ci.org/heya87/pawiTwo.

1.6.4 Justext

An implementation of the Justext algorithm is available in Python. The ressources are available under following link https://code.google.com/p/justext/. There is no implementation in Java so for the first approach the python application is called from Java with a ProcessBuilder. The command to extract an HTML file with justext is very simple:

```
justext -s English /path/page.html > cleaned-page.txt
```

This command extracts the HTMl file page.html into a text file called cleaned-page.txt. The Java ProcessBuilder performs system commands as they are used in a console and can handle the outcome if needed. For the project there is no return values needed. The needed content is the generated text file which is then read and processed for further use.

1.6.5 Boilerpipe

An implementation of the Boilerpipe Algorithm is available in Java. The resources are available under following link https://code.google.com/p/boilerpipe/. This algorithm can be used out of the box with calls against the Java API.

1.7 Process view

1.8 Physical view