The fact extraction system of 101companies

Jan Rüther

June 22, 2014

Abstract

The 101project is a wiki based learning platform for software languages and technologies. The core are about 100 small software projects in different languages. Fact extraction is used to generate fragment trees for all source code artifacts. These fragments help to visualize the structure of the code and therefore improve program comprehension. This paper describes our fact extraction approach in all aspects from requirements to testing to support evolution and maintenance.

Contents

1 Introduction

The 101companies project[6, 13] is a software chrestomathy that enables software engineers to learn and compare languages and technologies. The essential part of 101companies are the more than 100 contributions, which are small software projects all solving an equal problem. By understanding one contribution the software engineer is able to understand other contributions faster. The wiki based platform supports learning of new languages, concepts and idioms.

Fact extraction is a technique used to generate metadata on given source code artifacts. In the 101project fact extraction is used to generate a fragment tree for each source code artifact from all contributions. These fragments are used to enhance the reader's understanding of the artifact's structure.

This paper aims to explain our fact extraction system in detail. Thus all stages of software development will be described. Starting with the requirements engineering in section 2 which features an illustration, a requirements list, use cases for motivation and a glossary to explain used terms.

Section 3 pictures the architecture and the code. It contains a detailed look at the existing fact extractors in 2.3 explaining their scope, extracted facts and implementation characteristics. The following section (3.2.1) is needed to understand the architecture of the fact extraction system and its integration into the whole architecture of 101companies. If designing a new fact extractor this section is important in particular. A step by step tutorial to speed up this process is given in section 3.2.2.

To complete the paper there is a chapter about the testing of fact extractors (4), there is a conclusion (5) and a section about related work (5.1).

2 Requirements engineering

2.1 Illustration

Listing 1 shows a typical example of the 101project. This example written in Java is an excerpt of a class file with fields and methods. The facts extracted from this class are in Figure 2. The structure of the Java class is reduced on the fact that there is a class with the name *Company* which has one fragments. This fragment is classified as *method* and identified through its name. Each fragment is attributed with its line numbers. They are used to display the fact without the surplus as seen in Section 2.3. As additional metadata the package and the imports of the file are extracted. Overloading in Java is handled by indexing the affected methods.

```
1 public class Company implements Serializable {
2
```

```
 3          private String name;
 4          private List<Department> depts = new
                LinkedList<Department>();
 5
 6          public void cut() {
 7                  for (Department d : getDepts())
 8                          d.cut();
 9          }
10  }
```

Listing 1: "Java Class Example (excerpt)"

```
1  {
2    "package": "org.softlang.service.company",
3    "imports": [
4      "java.util",
5      "java.io"
6    ],
7    "fragments": [
8      {
9        "classifier": "class",
10       "name": "Company",
11       "fragments": [
12         {
13           "classifier": "method",
14           "name": "cut",
15           "annotations": [],
16           "startLine": 17,
17           "endLine": 20
18         }
19       ],
20       "annotations": [],
21       "startLine": 7,
22       "endLine": 21
23     }
24   ]
25 }
```

Listing 2: "Facts - Java Class Example"

2.2 Requirements

- The fact extraction approach has to be language independent.
- A fact extractor gets a source file as input.
- A fact extractor outputs a json file with fragments.

- Fragments can be nested.
- Fragments must have the following attributes:
  - name
  - classifier - the type of the fragment
  - startLine
  - endLine
  - fragments - the fragments that are part of this fragment or if none an empty list
  - index - if there are at least two adjacent fragments with the same name. Indexing starts with 1.
- The startLine attribute is the lowest line number of the fragment.
- The endLine attribute is the highest line number of the fragment.
- If the fragment has a comment at the beginning, the startLine attribute is the lowest line number of that comment.
  - This comment can be a single-line or a multi-line comment
  - If there is a free line between the comment and the fragment, the startLine attribute is the lowest line number of the fragment.
- The top level element of the json file is an object with the following attributes:
  - fragments
  - imports - a list of imports if applicable
  - package - name of the package if applicable

2.3 Use Cases

Our motivation for fact extraction is to enhance the understandability of code. The main goal to achieve this is addressability of fragments.

The 101project is a learning platform dedicated to stakeholders like developers, teachers and learners. Thus one goal of the fact extraction is to support the understandability of the code by exposing its structure to the viewer. The tool responsible for this task is the 101explorer[1] (Figure 1). The 101explorer combines the features of an operating system's explorer and a document viewer. The viewer allows to explore the project structure and view the contained files with its metadata. The explorer is not limited on folders and files, but is able to explore fragments alike.

The second goal of our fact extraction approach is addressability. While working with the 101project the need arose to reference not the whole source code artifact, but a fragment of it. Doing so, it is possible to connect documentation or tutorials[2](Figure 2) with source code files. On one hand the viewer can navigate quickly from the tutorial to the code. On the other hand changes in the source code propagate into the tutorial keeping it up to date.

[1] http://101companies.org/resources?format=html
[2] http://101companies.org/wiki/Contribution:haskellStarter

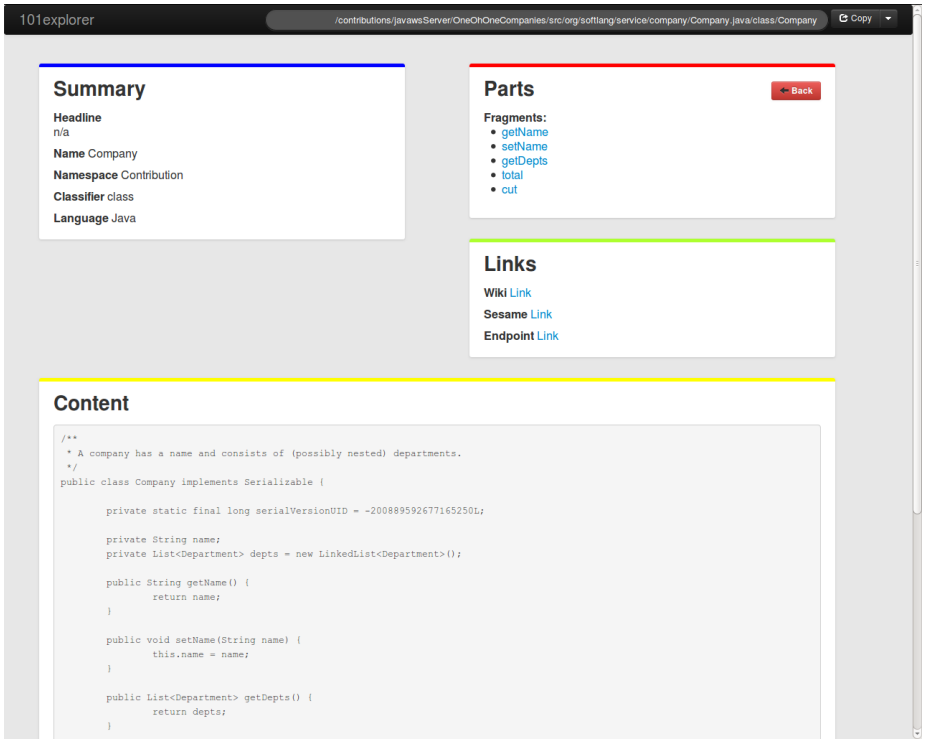

Figure 1: 101explorer

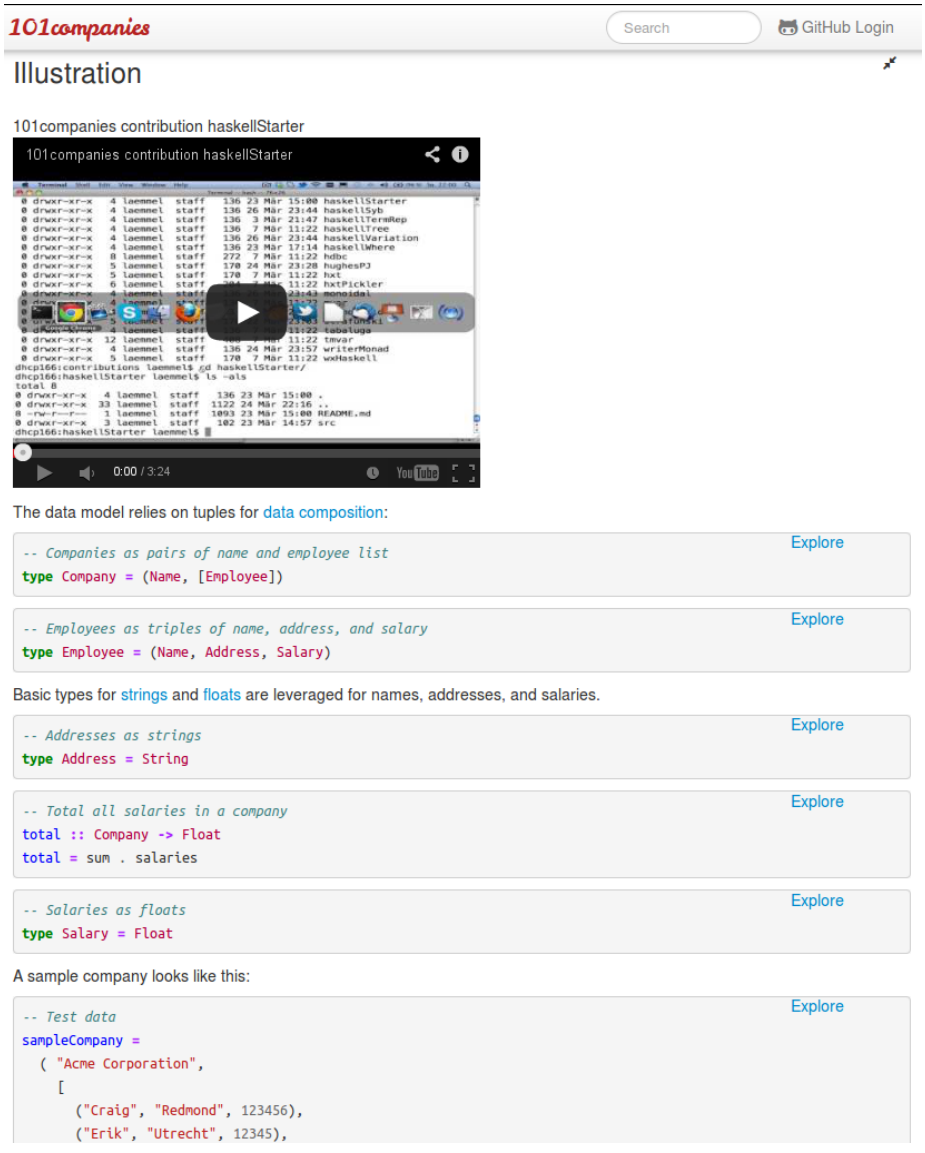

Figure 2: Tutorial: HaskellStarter

A second reason why fragments have to be addressable (usable as scope) is presented in the following. In the 101project metadata is assigned to software artifacts (including fragments) using a rule-based system [5]. The system is supported by the domain-specific language 101meta notated in JSON. Each 101meta rule consists of a condition and a set of metadata units as conclusion. An optional scope for fragment level metadata can be added.

2.4 Glossary

classifier The classifiers are the types of the fragments. When writing a new fact extractor it is necessary to design classifiers for the intended language. Classifiers refer to high level, non-terminal symbols in the syntax of the language. A good point to start are use cases to motivate potential classifiers. An overview about the existing classifiers is given in section 3.1.

contribution The 101companies project consists of more than 100 contributions[3]. They are located in *101repo/contributions*.

fact Metadata about a source file.

fact extractor Shell-executable, standalone program that generates facts, in our case a fragment tree, from a given source file.

fragment A named entity corresponding to a part of a source file. Each fragment has a type which is called classifier, a name and a line range. If the name is not unique on this hierarchy level, these fragments get an index attribute starting with 1. Fragments occur in a tree structure to represent the *is part of* relation.

line range A line range is a part of a document specified through the line numbers that belong to this part. A line range has a two line numbers, one for the first line (*startLine*) and one for the last line (*endLine*) It is not intended to have gaps (excluded lines) inside the line range.

module See worker

repo The 101repo is the location where concepts, contributions, features and languages are located. Physically the 101repo is a gibhub project. A lot of contributions are administrated externally and integrated via the build process of 101worker.

source file A source file or sourcecode file is any file that is processed by the worker and on this note by fact extraction. This embodies not only all files of the contributions, but also files of the architecture like all modules, services and everything from the technology-folder (including the fact extractors themselves).

[3] http://101companies.org/resources/contributions?format=html

language	classifiers	indexed classifiers	comments supported	packages	imports
Java	class, method	method	yes	yes	yes
C#	class, method	method	yes	yes	yes
Python	class, function, var	-	-	-	yes
SQL	create_statement, alter_statement, column	alter_statement	-	-	-
Haskell	function, data, type, newtype, pattern	-	-	-	yes
XML	element, text, attribute	element, text	-	-	-
HTML	tag, attribute, text	tag, text	yes	-	-
JSON	object, list	object	-	-	-
CSS	declaration, rule	-	-	-	-

Table 1: Fact extractor overview

worker The 101project has a production cycle which among other things creates metadata for source files (e.g fragments). This production cycle is part of *101worker* and is executed every night. *101worker* consists of independent[4] modules which are called in the sequence specified in *101worker/configs/production.json*. The *extractFacts* module iterates through all source files and assigns them to the respective fact extractor. More about the module system is accessible via github[5].

3 Implementation and Architecture

3.1 Showcase

This section discusses a lot of languages, currently supported or not supported, regarding fact extraction. Its not possible to define rules to generate the best classifiers given a language, thus a good way is to look at existing classifiers.

Table 1 summarizes existing fact extractors.

3.1.1 Java

The Java fact extractor supports the classifiers *class* and *method* as seen in Figure 3. The *method* classifier is for instance used to build the 101meta

[4]They are independent in the way that they are not calling each other. However a lot of them are dependent on the results of another module. Hence we need an execution sequence file.

[5]https://github.com/101companies/101worker

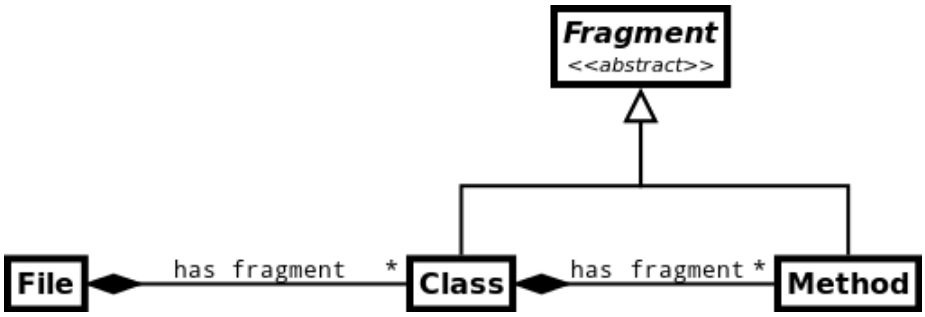

Figure 3: Metamodel of the facts extracted from source files with object-oriented language

rule that links the feature[6] *cut* to the method cut. Additional classifiers could have been *enum* and *variable* or control structures like *if*, *while*, *for* and *try-catch*. They have not been included, because there has not been a use case supporting them yet.

3.1.2 C#

The C# fact extractor uses the visitor pattern going over the abstract syntax tree to generate fragments. It uses the same classifiers as Java (*class* and *method*) and thus the same metamodel (Figure 3)

3.1.3 SQL

The motivation to write a fact extractor for SQL was the megamodel-project[4]. The project introduces modeling of the linguistic architecture of software. The technology hibernate[7] maps Java classes and fields to database tables and columns. The SQL fact extractor was developed to describe these connections via the *CREATE TABLE* statement.

Currently the extractor is supporting the *create_statement*, the *column* and the top level *alter_statement* classifiers (see Figure 4). The extractor could be extended by a lot of other SQL statements like *SELECT*, *UPDATE* or *DELETE*.

The big problem developing this fact extractor was to get a working parser (see tutorial in 3.2.2). One general property leading to difficult to parse languages are different dialects. Each database has a different SQL dialect, which makes it very difficult or up to impossible to find a parser supporting all of them. Indeed there is the norm *ISO/IEC 9075*, which is supported more or less by the different databases, but every database adds its own extensions. A parser and thus an extractor is hard to create without an accepted language standard. As a result of this misery this

[6] http://101companies.org/wiki/Namespace:Feature
[7] http://hibernate.org/

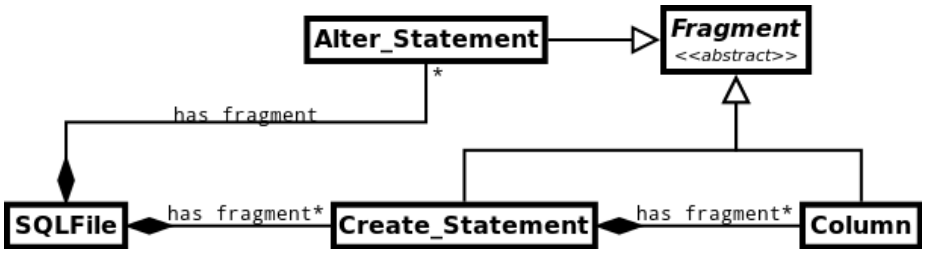

Figure 4: Metamodel of the facts extracted from SQL files

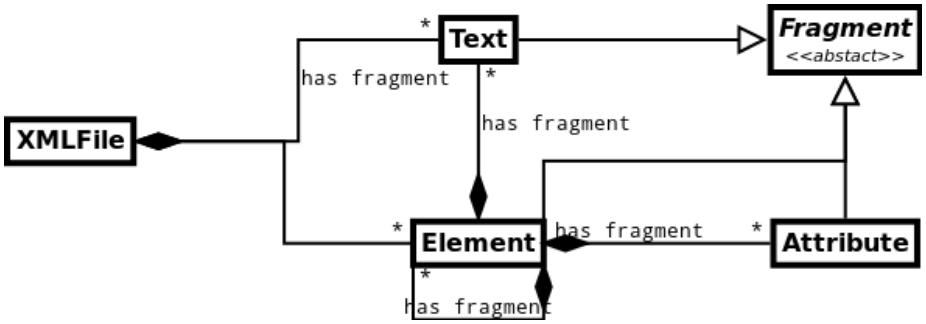

Figure 5: Metamodel of the facts extracted from XML files

extractor implements its own parser and is thus a lot more complicated compared to other extractors.

3.1.4 XML

The XML fact extractor supports the classifiers *element*, *attribute* and *text* (see Figure 5).

The implementation uses a SAX-Parser and is not supporting comments because of parser limitation. Using Python 3 this should be possible. Another problem is that the parser does not provide the line numbers of the *attribute* fragments. When tags are spread over several lines the line numbers of the start tag cannot be used. As a workaround the line numbers of the closing tags are used for attributes as well.

3.1.5 HTML

The HTML fact extractor supports the classifiers *tag*, *attribute* and *text* and is almost similar to the XML fact extractor. Substituting *element* with *tag* they have the same metamodel (see Figure 5). The extractor could be extended with a classifier for doctype declarations.

Unfortunately the used SAX-Parser is not supporting single-tags correctly. A **
** is leading to a wrong fragment file whereas **
** is

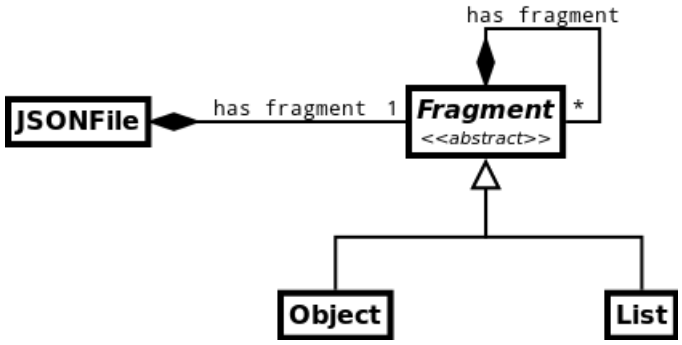

Figure 6: Metamodel of the facts extracted from JSON files

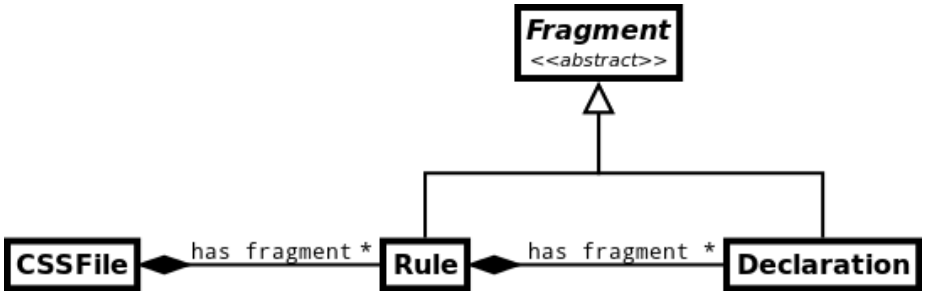

Figure 7: Metamodel of the facts extracted from CSS files

working.

3.1.6 JSON

The classifiers for JSON are *object* and *list* (see Figure 6). The *object* fragments are always indexed, because they have no identifier. The fact extractor could be extended by the text values of the lists or the key-value pairs of the objects. Currently the only properties of the objects that are included are lists and objects.

3.1.7 CSS

The CSS fact extractor is supporting the classifiers *rule* and *declaration* as seen in Figure 7. Both fragments have unique names, thus there is no need to index anything. The *declaration* fragments are key-value pairs. The key becomes the name and the value attribute is copied into the fragments for further use.

3.1.8 JavaScript

There is no fact extractor for JavaScript yet. One of the reasons why not is the possibility of JavaScript to express the same language construct in

Files	Location
fact extractor	*101repo/technologies/LANGUAGEFactExtractor*
extract facts module	*101worker/modules/extractFacts*
101meta files for languages	*101repo/languages/LANGUAGE/.101meta*
module output	*101web/data/resources*
module output summaries	*101web/data/dumps*
module execution sequence	*101worker/configs/production.json*

Table 2: Important locations

different syntactical ways. There exist a lot of ways to create and define objects in JavaScript and to make matters worse the community is discordant which way is the best [8]. This hinders the program understanding as well as the parser and extractor generation.

A fact extractor for JavaScript should support a *function* classifier. An *object* classifier would be nice, but is difficult to implement regarding the above described problems and the fact that objects can be written with the *function* keyword.

3.2 Coding

3.2.1 Convention

This section explains in detail how the fact extraction system is embodied into the 101worker. An overview about important locations is given in table 2.

The fact extractors are located in *101repo/technologies*. Each extractor has a folder (e.g. XmlFactExtractor). There are two essential files, the *extractor.py* and the optional Makefile. If the fact extractor needs to be build this is specified in the Makefile with the *build:* and *clean:* (e.g. JFactExtractor).

The *extractor.py* is the interface to every extractor, even if the extractor is not written in Python (e.g. JFactExtractor). It starts with *#! /usr/bin/env python* to make it executable from the console via *./extractor.py*. The fact extractors get a source code file as input (via *sys.stdin.read()*) and print the JSON file with the facts to the console as output. The command-line execution looks like this:

```
extractor.py < Path_Input_File > Path_Output_File
```

If the fact extractor needs to be build, use the *build* target (see JFactExtractor).

The fact extractors are executed from the module *extractFacts* (*101worker/modules/extractFacts*). The module is not executing the extractor.py

[8] http://stackoverflow.com/questions/383402/is-javascript-s-new-keyword-considered-harmful

from the same folder (using *cd* before). Thus the fact extractors must not depend on the local file addresses.

The *extractFacts* module needs a rule to know which extractor to use. Listing 3 displays the rule for the language JSON (located in *101repo/languages/JSON/.101meta*).

The facts are written into the folder *101web/data/resources*. The facts are in a file with the same name as the origin with the extension of *.extractor.json* to the filename.

```
1  [
2          {
3                  "suffix" : ".json",
4                  "metadata" : [
5                          {"geshi":"javascript"},
6                          {"language":"JSON"},
7                          {"extractor": "technologies/
                               JSONFactExtractor/
                               extractor.py"}
8                  ]
9          }
10 ]
```

Listing 3: "101meta rule for JSON"

3.2.2 Tutorial

This section is a step by step tutorial to build a new fact extractor.

- Design classifiers for your language and communicate them to the team. Ideally there are use cases to motivate these classifiers. An overview about existing classifiers is in section 3.1.
- Find a parser for your language[9].
  - If there is no parser try to use ANTLR. In this case search for a grammar.
  - Write a demo with your parser to check if it has the following properties:
    - It can extract the classifiers.
    - You are able to get the line numbers (startLine, endLine) that belong to the classifiers.
    - It is stable.
- Create a folder *101repo/technologies/LANGUAGEFactExtractor*
- Create a folder *test* and design source files in that language for testing as described in section 4

[9]Normally this is the most complicated task of the whole process.

- Copy the *Makefile* from another fact extractor and modify it to use your test cases.
- Create an *extractor.py*
  - It needs *#! /usr/bin/env python* at the beginning.
  - If you do not write the extractor in python, copy the *extractor.py* from *JFactExtractor* and delegate your extractor.
- Write your fact extractor and test it against the test cases (command line: make test)
- Edit *101repo/languages/LANGUAGE/.101meta* so that the extractor attribute links to your fact extractor. (Listing 3)
- Commit your changes to git.
- The production cycle is executed at night. You can also access the server via ssh and execute it manually[10].
- The simplest way to test if everything is working is to open the 101explorer[11], navigate to a source file of your language and check if the fragments work.
- If not[12] look via ssh into *101web/data/dumps/extractor.json* or the for your language created *.extractor.json* files in *101web/data/resources*.

4 Testing

When writing test cases for a new fact extractor the following test cases are necessary:

- at least one test for each classifier
- if two fragments with the same name on the same hierarchy level exist, they get indexed to be distinguishable. Each classifier that needs indices needs a test case.
- if a fragment has an attached comment it is included into the fragment (the startLine attribute is lesser)
  1. a one-line comment before the fragment is included
  2. a multi-line comment before the fragment is included
  3. a comment before the fragment is not included, because there is one free line after the comment
- the package information is included properly
- imports information are included properly

The test cases are shell based (using *diff*) and executed via the *Makefile*. Each extractor has a *test* folder with the tests on the top level and the desired results in the *baseline* folder. The *output* folder is used as temporary storage for the test execution. The tests are executed via *make test*.

Table 3 is an overview of existing test cases.

[10] https://github.com/101companies/101worker
[11] http://101companies.org/resources?format=html
[12] We often needed to install python libraries.

language	number of test cases
Java	7
C#	1
Python	0
SQL	4
Haskell	1
XML	7
HTML	9
JSON	3
CSS	2

Table 3: Number of test cases per language

5 Conclusion

Our system supports languages with a lot of different paradigms. The supported languages are functional or imperative, object-oriented or declarative, executable or data formats. To handle all these languages our system has to be language-independent. The idea of creating metadata about the structure of the language files meets this criteria. Every language's structure can be broken down to its parts and subparts on a high level of abstraction. Although the approach is language-independent, meaning it can be used for all languages, it cannot be automated. Manual operation for every language is needed to design the desired classifiers of fragments and to discover at which granularity to stop the fragmentation. Representing the full abstract syntax tree would miss the goals and overburden the engine.

5.1 Related Work

Fact Extraction is used in frameworks or reverse-engineering environments (IRE) to support software maintenance, to compute metrics and to enable reverse engineering. Fact Extraction can be ordered by the used lower, middle or higher level metamodel. Lower level metamodels (heavyweight) generate the full abstract syntax tree (AST). These include for C/C++ SolidFX [12] and Columbus [7]. Work has been done to enable validation regarding completeness of fact extraction[11]. A higher level (lightweight) approach is the srcML Toolkit [2] for C/C++. Another approach is AO-Vis [9] working on AspectJ and Java.

Our approach is lightweight and language independent. Other language independent approaches of fact extraction are DeFacto annotating the context-free grammar with fact annotations[1], the middle level Dagstuhl Middle Metamodel[10], JGraLab/GReQL [3], the Meta-Environment (generalization of ASF+SDF)[14], analysis of framework based systems [8] and behavioral analysis for object-oriented source code [15].

A Known issues

- Bug: C# fact extractor indexing is wrong (see the Overloading test case). Currently the first method does not get indexed; the indexing starts with the second method (with the same name).
- Haskell and Python fact extractors need to include line numbers (startLine and endLine) in the fragment files generated by the extractors.
- HTML fact extractor does not support not marked single tags like **
**.
- XML fact extractor does not provide the line numbers of attributes. As workaround the line numbers of the parent element are used.
- Python needs test cases.

References

[1] Hendrikus J. S. Basten and Paul Klint. Defacto: Language-parametric fact extraction from source code. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2008.

[2] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In *SCAM*, pages 173–184. IEEE, 2011.

[3] Jürgen Ebert and Daniel Bildhauer. Reverse engineering using graph queries. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 335–362. Springer, 2010.

[4] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the linguistic architecture of software products. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 151–167. Springer, 2012.

[5] Jean-Marie Favre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In *WCRE*, pages 335–344. IEEE Computer Society, 2012.

[6] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 101companies: A community project on software technologies and software languages. In CarloA. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 58–74. Springer Berlin Heidelberg, 2012.

[7] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from open source software. In *ICSM*, pages 60–69. IEEE Computer Society, 2004.

[8] Jens Knodel and Martin Pinzger. Improving fact extraction of framework-based software systems. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *WCRE*, pages 186–195. IEEE Computer Society, 2003.

[9] Jeffrey W. Koch. *Aovis: Reverse Engineering and Visualization of Aspectj/Java Source Code*. PhD thesis, Richardson, TX, USA, 2011. AAI3494543.

[10] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94(0):7 – 18, 2004. Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003).

[11] Yuan Lin, Richard C. Holt, and Andrew J. Malton. Completeness of a fact extractor. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *WCRE*, pages 196–205. IEEE Computer Society, 2003.

[12] Dennie Reniers, Lucian Voinea, Ozan Ersoy, and Alexandru Telea. The solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Science of Computer Programming*, 79(0):224 – 240, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[13] Software Languages Team. The 101companies project. University of Koblenz-Landau, Germany, 2014. http://101companies.org/.

[14] M. G. J. van den Brand, Magiel Bruntink, G. R. Economopoulos, H. A. de Jong, Paul Klint, A. Taeke Kooiker, Tijs van der Storm, and Jurgen J. Vinju. Using the meta-environment for maintenance and renovation. In René L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca, editors, *CSMR*, pages 331–332. IEEE Computer Society, 2007.

[15] Neil Walkinshaw, Marc Roper, and Murray Wood. Understanding object-oriented source code from the behavioural perspective. In *IWPC*, pages 215–224. IEEE Computer Society, 2005.