BACHELOR THESIS

Adaptive Issue Mining Through Data Scraping

Author:	Eric Gliemmo
Faculty:	Computer Science
Superadvisor:	Prof. Dr. Andreas Zeller
Advisor:	Sascha Just
Submitted On:	06. Mai 2015

UNIVERSITÄT DES SAARLANDES

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbruecken, den 06.05.2015

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbruecken, den 06.05.2015

Acknowledgement

I would like to thank my advisor Sascha Just for many hours of discussion and encouraging me to just follow my ideas. His feedback and criticism has helped me profoundly to come up with the results of this thesis.
Thanks also go to my supervisor Prof. Andreas Zeller for providing me with this challenging and interesting topic and his continuing feedback and support throughout my work on this thesis.

Abstract

These days the analysis of issues of different bug trackers is a remarkable field of application in software mining. Given a dataset of issues of a bug tracking system like Bugzilla, researchers mine these issues to analyze them and connect them to the source code. One extracts information from the bug reports which are used to predict failures in the concerning program for example. We choose a similar approach, but we want to pursue the mining and don't analyze or evaluate bug tracking systems or its issues itself. To be more accurately, we try to automate the mining process. In fact, we want to create methods, whereby we are able to generate a miner or a mining plan for an arbitrary bug tracker automatically.

Contents

Chapter 1

Overview

1.1 Introduction

Issues are an essential part of computer science and the development of software programs. For every software project there is a data set of issues, containing for example bugs, which disclose the problems of the current version of the program or features, which clarify new requirements for the software.
This is the reason for researchers to deal with issues or issue reports. Particularly with the analysis of these reports. The subject of software mining is concerned with the analysis of issues. Generally data mining means the systematic application of methods to a data set with the goal of identifying new patterns. Obviously, in the case of issue mining, this data set consists of issue reports.
Because of the importance of bugs and features there are several bug tracking systems to manage issue reports. The critical problem for bug tracker miners is the different structure of the tracking systems. To mine issues, it is necessary to extract all the information of an issue. Therefore, bug tracker miners have a mining plan, which clarifies, what data can be found where. Now, the problem is, if you want to mine issues of different tracking systems, you have to create a mining plan for each system.
Furthermore, bug tracking systems are always developing and change their structure as part of a new version for example. Than it is possible, that the origin mining plan doesn't work anymore or the extracted data are not reliable anymore. Given these problems, a mining process could be very uncomfortable for issue miners and researchers.
So our aim is to improve the mining process in terms of an adaptive mining approach. We will try to automate the mining process. Given a bug tracking system, we want to create a method, that generates a mining plan that is specially geared to the structure of the respective tracking system. Finally we will be able to mine

issues independent from a certain bug tracking system or a certain version. Our method will analyze the structure of the issue report and will submit a plan how to extract the data of the respective issue.
This mining process can be called report mining, which means to extract data from human readable computer reports. In our case, these computer reports are the bug reports of the concerning tracking system. And the generic term of report mining is data scraping which can be retrieved in the title of this work.

In detail this means, we will create a data set of bug reports of a certain bug tracking system. When creating the issues we will assign the fields of the issue with predefined markers, so that we are able to recognize the data of the fields in later runs. After that, we will choose one report and access its HTML structure. Next, we will convert the HTML code to XML. With a depth-first search algorithm we will retrieve our markers and remember the respective path to it. We will frame this path as a XPATH expression that we can use to obtain our desired data by applying this expression to the XML code of an other bug report. Ideally we will be able to retrieve all our markers and to extract information of any bug report of this version of the bug tracking system. So the XPATH expression will be our mining plan respectively it denotes where we can find the information reports contain.
Given this expression, we can apply it to the other issues of our created data set for evaluation case. Furthermore our method will be able to generate an expression for an other version of the bug tracking system. In the best case we can extract all data the reports contain. For evaluation purposes we will test our method with three different bug tracking system and variable version of them.

In summary, we try to simplify and automate the mining process concerning issue mining by developing an adaptive issue miner which uses data tracking and which is independent of the tracking system or its current version.

1.2 Related Work

There are similar works concerning mining software repositories. A common method of data mining is the selection of information of bug reports to reference them to the original code. A.Schroeder et al. demonstrated two steps in their work 'If Your Bug Database Could Talk...' [1]. First they identified corrections with the message of the fix, which references to the concerning bug report (Fixed # 3223 for example).
Second is, that they mapped bug reports to releases. The version field of the bug

database lists the release for which the bug was reported. However they noted that the reliability respectively the trust level is very low. This work leads to the following question: In which parts of the code are the most bugs because of the complexity of the code? Therefore it is necessary to define a metric, that measures the complexity of the code. Such kind of metrics already exist ('Mining metrics to predict component failures.'[2]), but this research doesn't give a definite answer. The conclusion is, that new metrics or a combination of existing metrics are required to measure the complexity better and more reliably. Consequently this approach of mining bug reports shows the potential of future research based on such bug data.
A similar approach chose J.Sliwerski, T.Zimmermann and A.Zeller in their paper 'When Do Changes Induce Fixes?' [3]. They analyzed CVS (Concurrent Versions System) archives for changes that lead to problems, indicated by fixes. They showed how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as Bugzilla).
For the standardization of bug reports there already exists a software called MOZKITO, that realizes these tasks. MOZKITO (developed by Sascha Just and Kim Herzig) models bug reports in a way, that we can abstract all these trackers. So MOZKITO mines reports of different trackers, however not adaptive. Our goal is to develop methods, that are able to mine bug reports adaptive. Concerning 'related work' there is nothing more to say, because, as seen, there are some earlier approaches in issue mining, but not in the way we are pursuing.

Chapter 2

Concepts

2.1 Problem

The main problem of mining issues of different bug tracking systems is that they have distinct structure. One tracker uses components which are neglected by another or the tracking system uses various scales of classification. For example one bug tracker has five levels to classify the priority of an issue, the other uses only three levels to grade the priority.
Up to now the common procedure was to mine manually with the help of a spider or a crawler (a computer program that gathers and categorizes information on a website).
The next problem of manual issue mining is the changing API (application programming interface) of the issue tracker. Software programs like Redmine or Bugzilla are always evolving and changing their interface. This fact is very unfavourable for non-adaptive issue miners, because such modifications often remain undetected and the mining software isn't working anymore or the data is not reliable anymore.
Our proposed solution is an adaptive miner. If we would be able to generate our miner automatically, we would create a robustness and independence against changes of the interface. Effectively themes like 'Tackling Interface Evolution in Issue Tracker for Software Mining' are close to our work.
Generally there are various possibilities to access data of bug trackers. Table 2.1 shows a table with the currently most used bug tracking systems and the interface they provide. The HTML surface, the XML part or the REST (Representational state transfer) API for example are possibilities to extract information. The latter is a method of communication between the user and the server of the bug tracking system. Similar possibilities are XML-RPC (Extensible Markup Language Remote Procedure Call), JSON-RPC (JavaScript Object Notation Remote Procedure Call)

Bugtracker	**provided interface or access to data**
Bugzilla	can be accessed and modified by an XML-RPC or a JSON-RPC Webservice interface
Mantis	with MantisConnect it can be accessed via SOAP
Roundup (SourceForge)	can only be accessed via the user interface, i.e. template files of HTML
Redmine	provides a REST API that supports XML and JSON format
Trac	provides a RSS feed, otherwise only the look can be accessed via HTML template and CSS files
JIRA	provides the Atlassian REST API

Table 2.1: The most used bug tracking systems and the interface they provide

and SOAP (Simple Object Access Protocol). RPC means, by sending a HTTP request to a server, that calls a single method of a remote system, one return value is returned. In the case of XML-RPC the input and output format is XML, in the case of JSON-RPC the format is JSON. SOAP is the successor of XML-RPC.
For our case of an adaptive miner, we will decide for the first-mentioned, the HTML surface. All the mentioned capabilities change their structure in consequence of the changing interface of the bug trackers. Basically it doesn't matter which opportunity we will choose, because our conceptual approach is adaptive and independent. Nevertheless by using HTML front-end, we can guarantee that all relevant data are available.
Figure A shows the mentioned changing interface of bug tracking systems as Bugzilla. In version 4.4 in contrast to version 3.3.4 there are some small, but for bug miners very crucial modifications. For example in 3.3.4 the login fields are visible and the user is able to login directly. In 4.4 the user has to click the login button first, before being able to enter his username and password. These modifications of the HTML structure also influence the information miners want to extract. The location of data could change or, as in our example, an additional link could be necessary for the crawler to access the respective information.

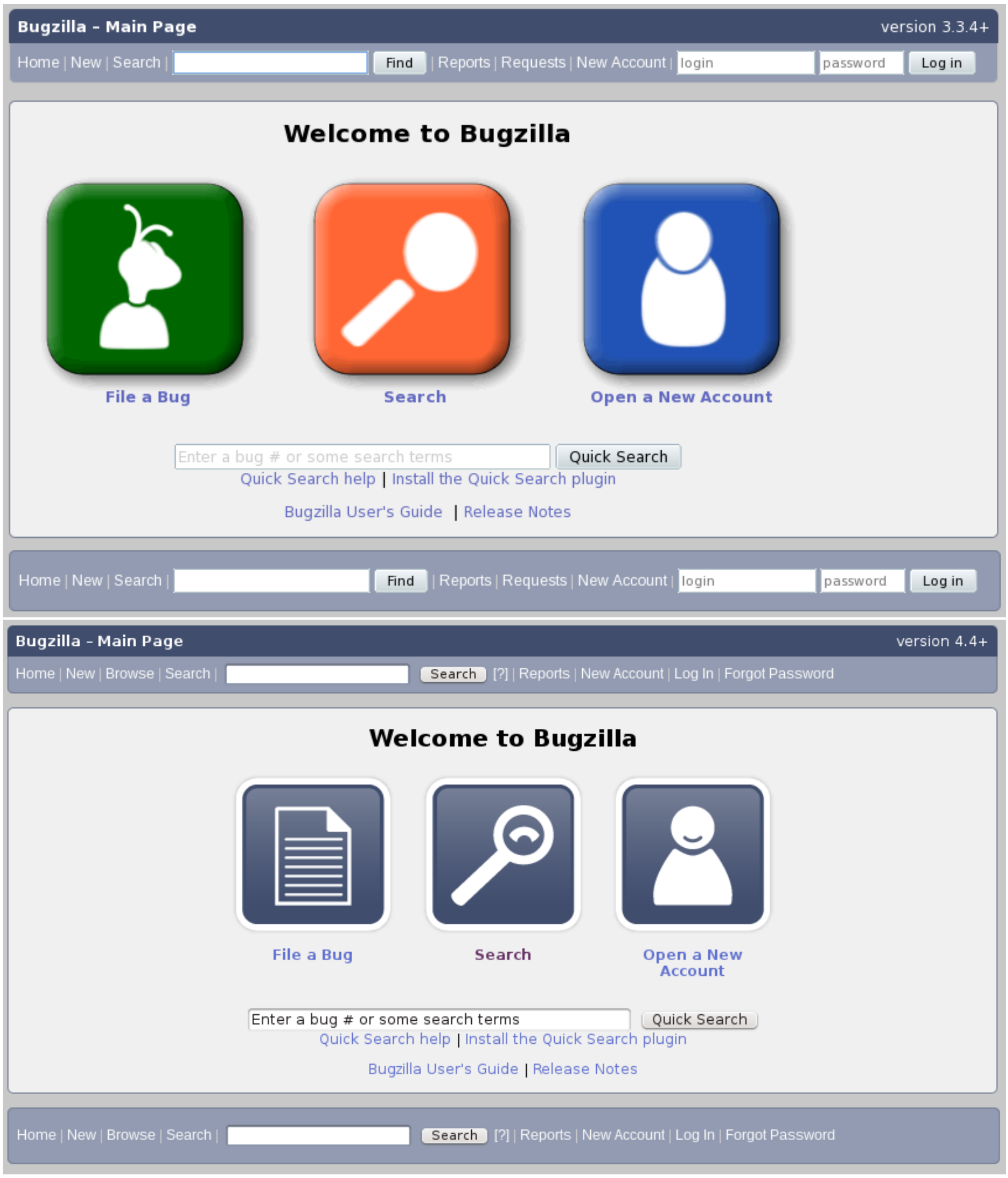

Bugzilla – Main Page
version 3.3.4+
Home | New | Search | Find | Reports | Requests | New Account | login password Log in
Welcome to Bugzilla
File a Bug
Search
Open a New Account
Enter a bug # or some search terms
Quick Search
Quick Search help | Install the Quick Search plugin
Bugzilla User's Guide | Release Notes
Home | New | Search | Find | Reports | Requests | New Account | login password Log in
Bugzilla – Main Page
version 4.4+
Home | New | Browse | Search | Search [?] | Reports | New Account | Log In | Forgot Password
Welcome to Bugzilla
File a Bug
Search
Open a New Account
Enter a bug # or some search terms
Quick Search
Quick Search help | Install the Quick Search plugin
Bugzilla User's Guide | Release Notes
Home | New | Browse | Search | Search [?] | Reports | New Account | Log In | Forgot Password

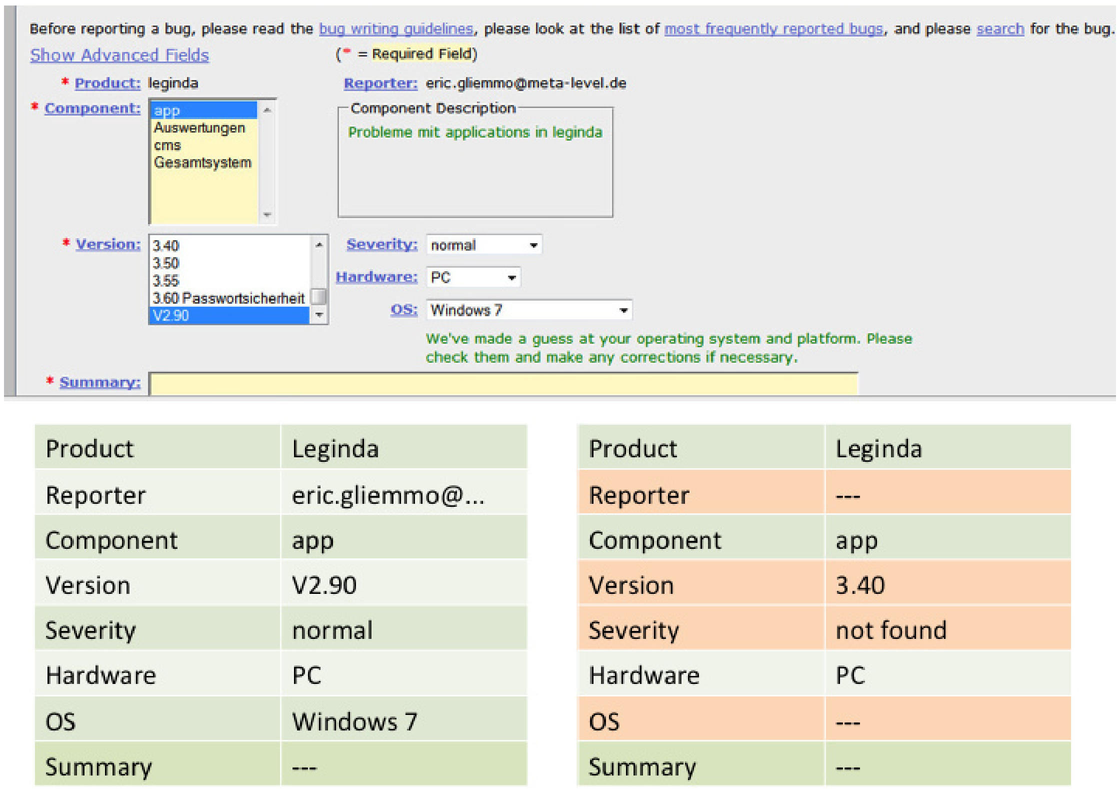
Figure A: *As an example of the changing interface of a bug tracking system, the Bugzilla version 3.3.4 on top, at the bottom the version 4.4*

2.2 Contribution

First, we need a standardized view of bug reports. In the paper 'What Makes a Good Bug Report' 2007 [4], Nicolas Bettenburg et al. deal with the question, which information a good bug report would have to contain as well as in their paper 'Quality of Bug Reports in Eclipse' 2007 [5].
Both papers conclude that information on steps to reproduce and stack traces are very important fields of a bug to be a bug report of high quality. The main problem, which resulted in the survey, that was conducted by Bettenburg et al., is the inaccuracy and incompleteness of bug reports.
Concerning our approach, this means that we need to define some standards to guarantee that we can compare issues of different bug tracking systems. We have to determine which fields of a bug report are important or significant. In other words, a feature-complete representation of issue reports is desirable.

The second point is the reception of a benchmark. To develop our methods, we need to create a set of bug data, which represents the predefined standards in all facets. Partly we will construct this set manually, partly we will extract some existing bug reports from the MOZKITO database. Additionally, this is a basis on which other tools could test their mining results.

The third point is our approach of adaptive mining. We will develop methods, which analyze the changing structure of reports in different bug tracking systems and generates a corresponding issue miner. In other words, on the basis of a set of bug reports, our methods yield a mining plan, which states how to extract information of these reports to write them in a database.
Given such methods, we could automatically create issue miner independent from the respective bug tracking systems. In other words, an on-the-fly production of bug tracker miners would be possible, given a way to deploy our created data set. The advantage of our methods, if the deployment would be defective, would be the recognition of this failure during the generating of the miner. The justification therefore is that we have our original data set. If we couldn't mine it with our methods, we would know, that something goes wrong. We can always compare the mining results with our original data and determine if any data was mined wrong or if any data was lost. Furthermore modifications of the interface of bug tracking system would interest no longer. Such failures as shown in Figure B could also originate from changing interface of a bug tracker that non-adaptive miners don't recognize.



Before reporting a bug, please read the bug writing guidelines, please look at the list of most frequently reported bugs, and please search for the bug.

Show Advanced Fields (* = Required Field)

* Product:	leginda	Reporter:	eric.gliemmo@meta-level.de
* Component:	app Auswertungen cms Gesamtsystem	Component Description	Probleme mit applications in leginda
* Version:	3.40 3.50 3.55 3.60 Passwortsicherheit V2.90	Severity:	normal
		Hardware:	PC
		OS:	Windows 7

We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.

* Summary:

Product	Leginda
Reporter	eric.gliemmo@...
Component	app
Version	V2.90
Severity	normal
Hardware	PC
OS	Windows 7
Summary	---

Product	Leginda
Reporter	---
Component	app
Version	3.40
Severity	not found
Hardware	PC
OS	---
Summary	---

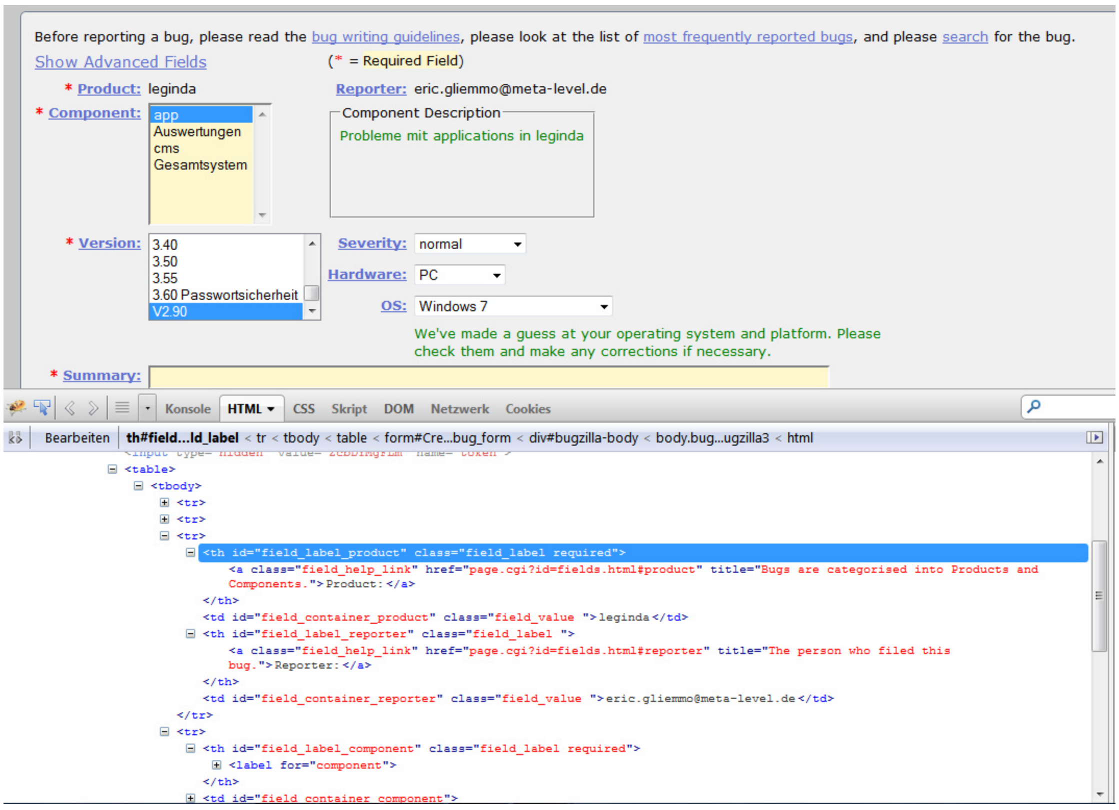
Figure B: *On top, a bug report of Bugzilla , on the left the information of our original report and on the right the obtained data after a defective mining process*

Chapter 3

Method and Implementation

3.1 Methodology

At first we need a data set in the form of an issue report collection of different bug tracking systems. We will construct a large base of issues that we are able to test our methods on the basis of these data. Our data set has to satisfy several characteristics. For the test purposes at later, it is useful to create those data sets in different bug tracking systems (Bugzilla, JIRA) and different versions of them to ensure that our methods don't work only in a particular case. As mentioned before, this collection will be feature-complete and represent our defined standards. Next we access the HTML surface of our bug reports. HTML (HyperText Markup Language) is the main markup language for creating web pages and other information that can be displayed in a web browser. Tim Berners-Lee, the inventor of the World Wide Web, wrote a memo proposing an Internet-based hypertext system.[6] Berners-Lee specified HTML and wrote the browser and server software in late 1990. The first publicly available description of HTML was a document called "HTML Tags", first mentioned on the Internet by Berners-Lee in late 1991.[7] These days the standard version is HTML 4.01. Tools like Firebug (Figure C) submit the HTML structure of a website. This structure can be easily converted to XML. Although the HTML and the XML appear very similar, this conversion is necessary. XML (Extensible Markup Language) is also a markup language that defines a set of rules for encoding documents in a format that is human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C (World Wide Web Consortium), that was also engaged in the development of HTML 4. In contrast to HTML, whose primary purpose is to display data with focus on how the data look, XML is a dynamic language whose primary purpose is the transport and storage of data. That is the reason of our conversion, because we are interested in the containing data of the webpage and not in the design.



Before reporting a bug, please read the bug writing guidelines, please look at the list of most frequently reported bugs, and please search for the bug.

Show Advanced Fields (* = Required Field)

* Product:	leginda	Reporter:	eric.gliemmo@meta-level.de
* Component:	app Auswertungen cms Gesamtsystem	Component Description	Probleme mit applications in leginda
* Version:	3.40 3.50 3.55 3.60 Passwortsicherheit V2.90	Severity:	normal
		Hardware:	PC
		OS:	Windows 7

We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.

* Summary:

Konsole
HTML
CSS
Skript
DOM
Netzwerk
Cookies
Bearbeiten
th#field...ld_label < tr < tbody < table < form#Cre...bug_form < div#bugzilla-body < body.bug...ugzilla3 < html

```
<table>
  <tbody>
    <tr>
    <tr>
    <tr>
      <th id="field_label_product" class="field_label required">
          <a class="field_help_link" href="page.cgi?id=fields.html#product" title="Bugs are categorised into Products and
          Components.">Product:</a>
      </th>
      <td id="field_container_product" class="field_value ">leginda</td>
      <th id="field_label_reporter" class="field_label ">
          <a class="field_help_link" href="page.cgi?id=fields.html#reporter" title="The person who filed this
          bug.">Reporter:</a>
      </th>
      <td id="field_container_reporter" class="field_value ">eric.gliemmo@meta-level.de</td>
    </tr>
    <tr>
      <th id="field_label_component" class="field_label required">
        <label for="component">
      </th>
      <td id="field_container_component">
```

Figure C: *On the top, a standard Bugzilla bug report in creation, below the HTML code of this report, extracted by Firebug*

However, at first glance, the syntax of both languages seems similar. With the use of HTML/XML, we can guarantee the completeness of the data of the website, in contrast to REST, XML-RPC or SOAP, in which this completeness is not secured. A reason therefore is, that HTML is the interface of the biggest target group respectively of the end-user. REST API, XML-RPC or SOAP aren't used by the ordinary user, so they aren't very well maintained and at the most not complete. They are often used only for tools with special assignments. For example the bug tracker MantisBT provides, apart from the HTML surface, SOAP as interface. Therefore the user has to use MantisConnect, a PHP web service that is built on top of Mantis API, that provides an easy way to connect to MantisBT via SOAP. The next step will be the use of a marker to retrieve our required information. In the XML code of the bug reports we will assign the important fields with markers. In this context the expression 'field' means an information field in a bug report or an issue. For example the priority or the description are such 'fields'. Every field gets a unique sequence of letters, by which we will be able to identify the field and will have a reference to the associated bug report. This marker is important concerning the crawler or spider in later runs.
After that we put these adapted bug reports in the bug tracking systems mentioned before. Then the crawler or the spider comes to the use. A Web crawler starts

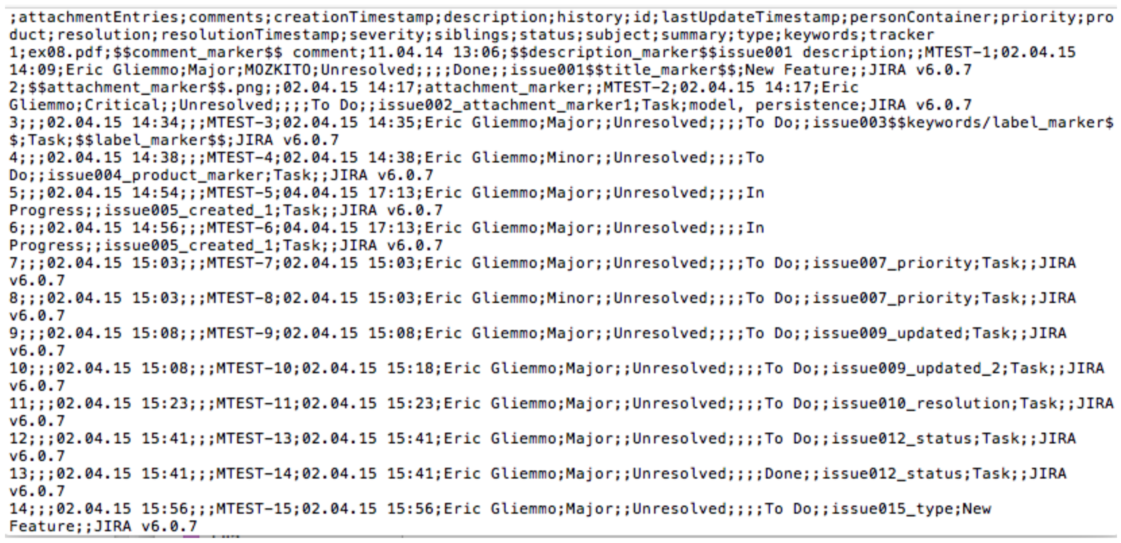
with a list of URLs to visit. In our case, this is the URL of the concerning bug report in the bug tracking system. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit next. In a bug report, these are additional information fields or attachments for example that are located on additional sites. Crawlers can validate hyperlinks and HTML code. So if we encounter one of our markers, we have to memorize where we found this marker respectively the associated field. That means that we have to create a mapping that clarifies which field or marker can be found where. So our abortive goal is to obtain a mining plan, that gives an instruction where we can retrieve our desired information.
Finally we have got methods that submit, given some bug reports of a certain bug tracking system, a mining plan to find and select the information of the original bug report and yields these information back for example in the form of database records.

3.2 Preconditions

3.2.1 Adaption of a benchmark

First we have to model issue reports with MOZKITO. We have to distinguish fields with free text and fields with a predefined selection. In the first case, we can use markers to retrieve the information with a search algorithm. The second case is more problematic. We don't have the possibility to indicate any data, so we have to create two similar bug reports for each field, which only differ in the concerning field.
In concrete terms this means, given a field like priority of a bug report, we have to create a report with, let's say, priority of 3 and it is necessary to create also a report with exactly the same information, but priority of 4. In later runs, we can compare these two reports and work out the position, where they differ. The path of this spot we will use to retrieve the priority of other bugs of the concerning bug tracking system.
In this example, the algorithm, that compares the two bug reports will analyze the first report as a XML document and will search for a '3'. Probably it will return several spots and pathes, because there might be a lot of 3's in this document. So we have to apply these pathes as XPath queries to the second bug report, that we have also converted to a XML document. In the best case, we can note, that only one XPath query does not yield a '3', but a '4'. So the algorithm will return this query, that corresponds to the path to the information of priority of this bug report.

Modelling reports with MOZKITO exactly means to create a CSV file with the concerning data. We use all the fields, MOZKITO uses too as records in this file. A comma-separated values (CSV) file describes the structure of a text file to store data in plain-text form. Plain text means that the file is a sequence of characters, with no data that has to be interpreted as binary numbers. [11] Every record in CSV file is separated by a semicolon.



```
;attachmentEntries;comments;creationTimestamp;description;history;id;lastUpdateTimestamp;personContainer;priority;pro
duct;resolution;resolutionTimestamp;severity;siblings;status;subject;summary;type;keywords;tracker
1;ex08.pdf;$$comment_marker$$ comment;11.04.14 13:06;$$description_marker$$issue001 description;;MTEST-1;02.04.15
14:09;Eric Gliemmo;Major;MOZKITO;Unresolved;;;;Done;;issue001$$title_marker$$;New Feature;;JIRA v6.0.7
2;$$attachment_marker$$.png;;02.04.15 14:17;attachment_marker;;MTEST-2;02.04.15 14:17;Eric
Gliemmo;Critical;;Unresolved;;;;To Do;;issue002_attachment_marker1;Task;model, persistence;JIRA v6.0.7
3;;;02.04.15 14:34;;;MTEST-3;02.04.15 14:35;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue003$$keywords/label_marker$
$;Task;$$label_marker$$;JIRA v6.0.7
4;;;02.04.15 14:38;;;MTEST-4;02.04.15 14:38;Eric Gliemmo;Minor;;Unresolved;;;;To
Do;;issue004_product_marker;Task;;JIRA v6.0.7
5;;;02.04.15 14:54;;;MTEST-5;04.04.15 17:13;Eric Gliemmo;Major;;Unresolved;;;;In
Progress;;issue005_created_1;Task;;JIRA v6.0.7
6;;;02.04.15 14:56;;;MTEST-6;04.04.15 17:13;Eric Gliemmo;Major;;Unresolved;;;;In
Progress;;issue005_created_1;Task;;JIRA v6.0.7
7;;;02.04.15 15:03;;;MTEST-7;02.04.15 15:03;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue007_priority;Task;;JIRA
v6.0.7
8;;;02.04.15 15:03;;;MTEST-8;02.04.15 15:03;Eric Gliemmo;Minor;;Unresolved;;;;To Do;;issue007_priority;Task;;JIRA
v6.0.7
9;;;02.04.15 15:08;;;MTEST-9;02.04.15 15:08;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue009_updated;Task;;JIRA
v6.0.7
10;;;02.04.15 15:08;;;MTEST-10;02.04.15 15:18;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue009_updated_2;Task;;JIRA
v6.0.7
11;;;02.04.15 15:23;;;MTEST-11;02.04.15 15:23;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue010_resolution;Task;;JIRA
v6.0.7
12;;;02.04.15 15:41;;;MTEST-13;02.04.15 15:41;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue012_status;Task;;JIRA
v6.0.7
13;;;02.04.15 15:41;;;MTEST-14;02.04.15 15:41;Eric Gliemmo;Major;;Unresolved;;;;Done;;issue012_status;Task;;JIRA
v6.0.7
14;;;02.04.15 15:56;;;MTEST-15;02.04.15 15:56;Eric Gliemmo;Major;;Unresolved;;;;To Do;;issue015_type;New
Feature;;JIRA v6.0.7
```

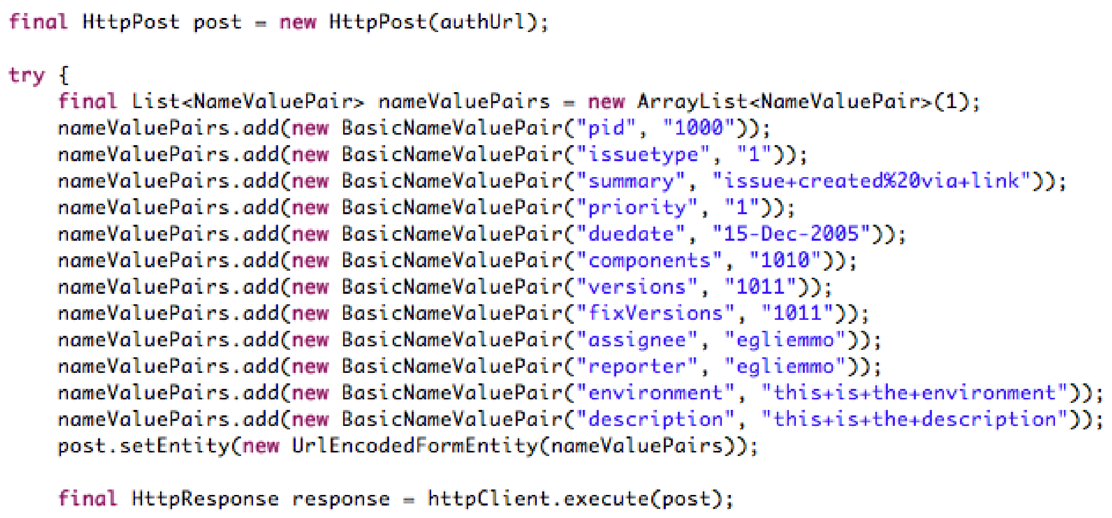
Figure D: *An example of a CSV file. In this case, two lines represent one bug report. The fields of the bug report are consecutively numbered and separated by a semicolon. So the record of the file can be assigned to a field of a bug report obviously*

The next tread is to feed the file into our program. Therefore we create a Java InputStream with our CSV file. Using opencsv (a CSV parser library for Java), we read this CSV file and store each row as a String Array. In later runs, we can use this array to deploy the reports to different bug tracking systems.

3.2.2 Deployment of reports

To deploy our data set automatically to different bug tracking systems, we need a method that realizes this deploy. Similar to the login-procedure in Chapter 3.3.1 to access the HTML structure of a bug report, we will create a HTTP client to connect with the concerning website of the bug tracking system. After this, we can build a HTTP post. In this post we can occupy the fields we want to mine with our data from the read CSV file. The Apache library provides some possibilities for this problem. With this library we are able to create so called NameValuePairs, that consist of the name of the field we want to mine and the corresponding value of each report to deploy of our CSV file. We will post the list of all NameValue-

Pairs by executing the HTTP post. Both, JIRA and Bugzilla, provide instructions how to create reports with HTTP posts.



```
final HttpPost post = new HttpPost(authUrl);

try {
    final List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(1);
    nameValuePairs.add(new BasicNameValuePair("pid", "1000"));
    nameValuePairs.add(new BasicNameValuePair("issuetype", "1"));
    nameValuePairs.add(new BasicNameValuePair("summary", "issue+created%20via+link"));
    nameValuePairs.add(new BasicNameValuePair("priority", "1"));
    nameValuePairs.add(new BasicNameValuePair("duedate", "15-Dec-2005"));
    nameValuePairs.add(new BasicNameValuePair("components", "1010"));
    nameValuePairs.add(new BasicNameValuePair("versions", "1011"));
    nameValuePairs.add(new BasicNameValuePair("fixVersions", "1011"));
    nameValuePairs.add(new BasicNameValuePair("assignee", "egliemmo"));
    nameValuePairs.add(new BasicNameValuePair("reporter", "egliemmo"));
    nameValuePairs.add(new BasicNameValuePair("environment", "this+is+the+environment"));
    nameValuePairs.add(new BasicNameValuePair("description", "this+is+the+description"));
    post.setEntity(new UrlEncodedFormEntity(nameValuePairs));

    final HttpResponse response = httpClient.execute(post);
```

Figure E: *An example of a HTTP post with NameValuePairs for a report in JIRA.*

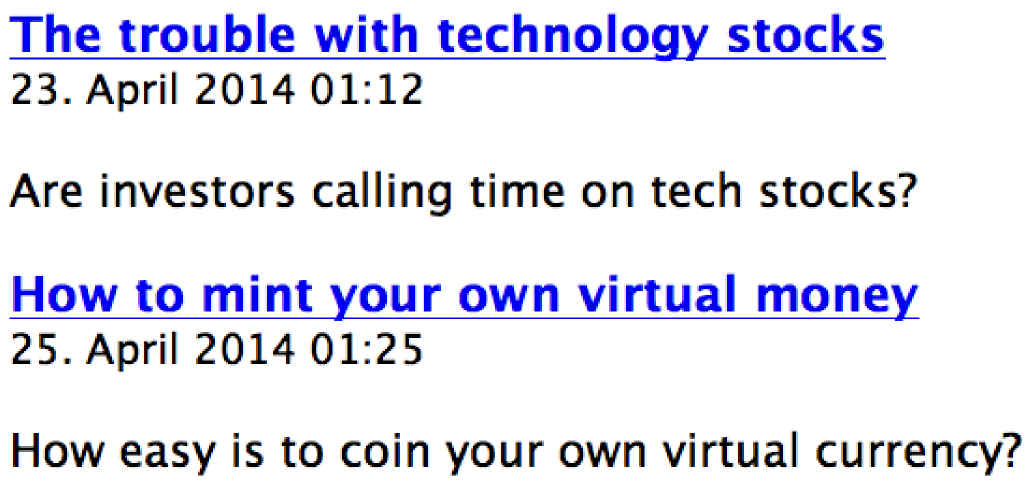
The first bug tracking system, we use to create a data set is JIRA. JIRA is a proprietary issue tracking product that is developed by Atlassian since 2002. The current version is JIRA v6.2, released 25 February 2014. [8] Our data set is created with version v6.0.7.

3.3 Retrieving markers

3.3.1 Access the HTML structure

The first obstacle of the plan is to access the HTML structure of a representative issue report. We need a possibility to handle the HTML code of the website and to work with it. For this purpose we will mainly deal with JDOM2. JDOM2 is the successor of JDOM and generally it is an open source Java-based document object model for XML and it is useful to manipulate XML code. It is a very common library to deal with XML in Java. Developed by Jason Hunter and Brett McLaughlin starting in March 2000, JDOM integrates ”Simple API for XML” (SAX) and supports XPath, that is necessary for our mining plan in later runs.
So we have to find a way to convert the HTML code of the website to XML. This part TagSoup will realize. TagSoup is a SAX-compliant parser written in Java that, instead of parsing well-formed or valid XML, parses HTML as it is found in the web. So TagSoup supplies the parsed XML code of our issue report. By providing a SAX interface, it allows standard XML tools to be applied, such as we will do with JDOM2.

In fact this means, given the URL of an issue report we will apply TagSoup and Saxbuilder (part of the JDOM2 Sax library) to receive a JDOM2 Document with XML structure.
The JDOM API represents a XML document with tree structure and grants access to the single components of the tree structure. Apart from the reading of the document with JDOM2 one is able to manipulate parts of the tree structure and the write back of the structure in the document.
A point, which must be taken account, is the login procedure. Using bug tracking systems like Bugzilla or JIRA, it is necessary to login before we are able to access the page of a bug report. We have to create a HTTP client to connect with the website of the bug tracking system. With this client and the URL of the login page of the tracking system, we can build a HTTP post with respective fields, such as username and password. After we have executed this post, we are able to access the website of the bug report.



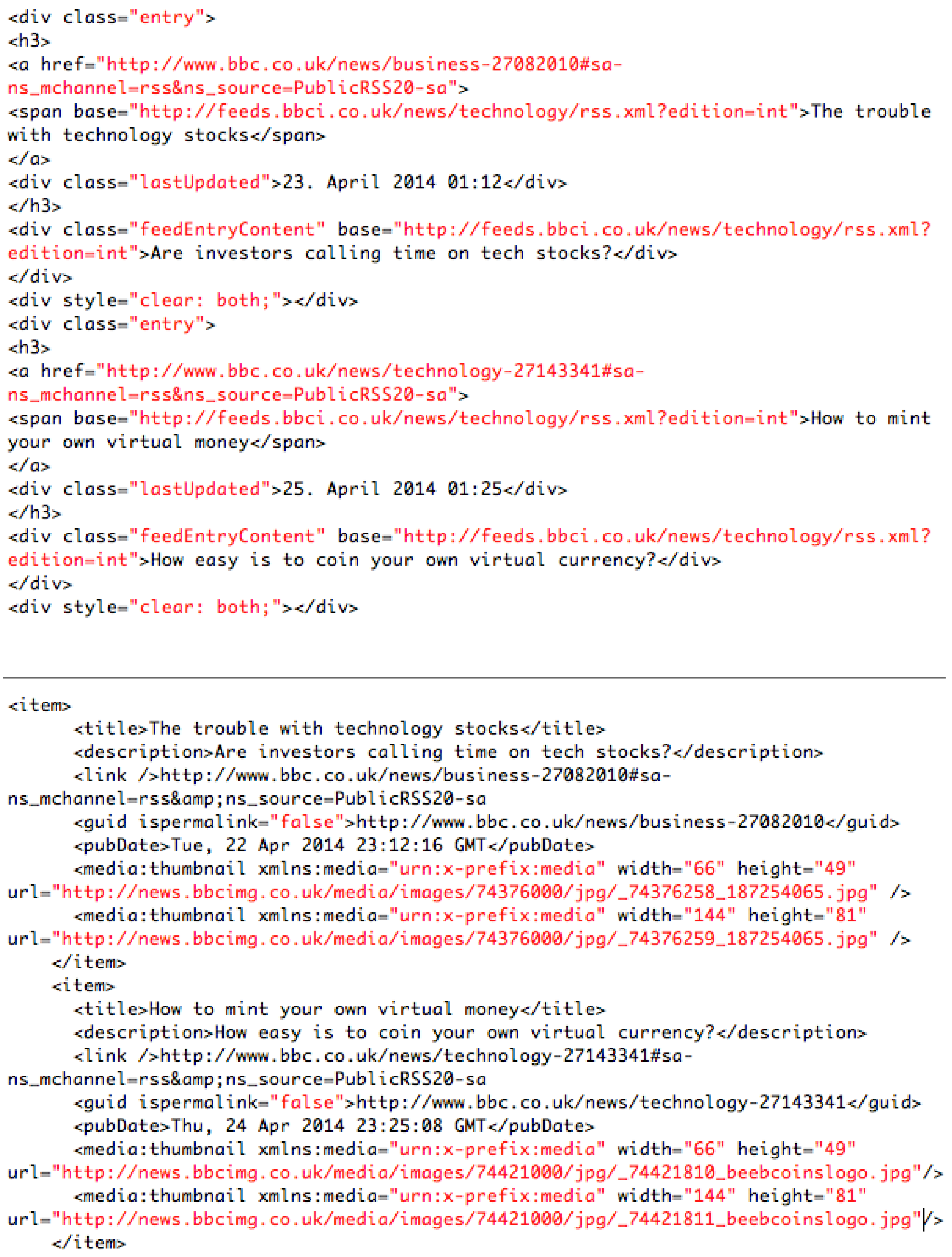
The trouble with technology stocks
23. April 2014 01:12

Are investors calling time on tech stocks?

How to mint your own virtual money
25. April 2014 01:25

How easy is to coin your own virtual currency?

Figure F: *An excerpt of a test website of BBC (http://feeds.bbci.co.uk/news/technology/rss.xml?edition=int) with some information. Based on this data one can get an impression how TagSoup and JDOM2 work.*



```
<div class="entry">
<h3>
<a href="http://www.bbc.co.uk/news/business-27082010#sa-
ns_mchannel=rss&ns_source=PublicRSS20-sa">
<span base="http://feeds.bbci.co.uk/news/technology/rss.xml?edition=int">The trouble
with technology stocks</span>
</a>
<div class="lastUpdated">23. April 2014 01:12</div>
</h3>
<div class="feedEntryContent" base="http://feeds.bbci.co.uk/news/technology/rss.xml?
edition=int">Are investors calling time on tech stocks?</div>
</div>
<div style="clear: both;"></div>
<div class="entry">
<h3>
<a href="http://www.bbc.co.uk/news/technology-27143341#sa-
ns_mchannel=rss&ns_source=PublicRSS20-sa">
<span base="http://feeds.bbci.co.uk/news/technology/rss.xml?edition=int">How to mint
your own virtual money</span>
</a>
<div class="lastUpdated">25. April 2014 01:25</div>
</h3>
<div class="feedEntryContent" base="http://feeds.bbci.co.uk/news/technology/rss.xml?
edition=int">How easy is to coin your own virtual currency?</div>
</div>
<div style="clear: both;"></div>
```

```
<item>
      <title>The trouble with technology stocks</title>
      <description>Are investors calling time on tech stocks?</description>
      <link />http://www.bbc.co.uk/news/business-27082010#sa-
ns_mchannel=rss&amp;ns_source=PublicRSS20-sa
      <guid ispermalink="false">http://www.bbc.co.uk/news/business-27082010</guid>
      <pubDate>Tue, 22 Apr 2014 23:12:16 GMT</pubDate>
      <media:thumbnail xmlns:media="urn:x-prefix:media" width="66" height="49"
url="http://news.bbcimg.co.uk/media/images/74376000/jpg/_74376258_187254065.jpg" />
      <media:thumbnail xmlns:media="urn:x-prefix:media" width="144" height="81"
url="http://news.bbcimg.co.uk/media/images/74376000/jpg/_74376259_187254065.jpg" />
    </item>
    <item>
      <title>How to mint your own virtual money</title>
      <description>How easy is to coin your own virtual currency?</description>
      <link />http://www.bbc.co.uk/news/technology-27143341#sa-
ns_mchannel=rss&amp;ns_source=PublicRSS20-sa
      <guid ispermalink="false">http://www.bbc.co.uk/news/technology-27143341</guid>
      <pubDate>Thu, 24 Apr 2014 23:25:08 GMT</pubDate>
      <media:thumbnail xmlns:media="urn:x-prefix:media" width="66" height="49"
url="http://news.bbcimg.co.uk/media/images/74421000/jpg/_74421810_beebcoinslogo.jpg"/>
      <media:thumbnail xmlns:media="urn:x-prefix:media" width="144" height="81"
url="http://news.bbcimg.co.uk/media/images/74421000/jpg/_74421811_beebcoinslogo.jpg"/>
    </item>
```

Figure G: *On the top, the HTML code of Figure F, extracted with Firebug, below the output of the JDOM2 document after applying TagSoup and Saxbuilder to the HTML code of the website*

With a simple, but comparable example one can illustrate how the conversion operates. Figure F shows a sample website containing some information comparable with the data of an issue report. In this form, the content is presented on the webpage. On the top of Figure G, the HTML code of Figure F is located, extracted with Firebug. To deal with these information in Java it is useful to convert this code to XML. After applying TagSoup and Saxbuilder we obtain the JDOM2 document as mentioned before. The structure and information of this document is at the bottom of Figure G.
Obviously we obtain a typical XML document with tree structure. The advantage is, that with the JDOM2 library we can operate on this document for example to search for our markers.

3.3.2 Search in XML

At first some words about XML and its structure. The Extensible Markup Language (XML) is a markup language for the representation of structured data in the form of text files. Every XML file has a logical structure that corresponds to a tree structure and therefore it is organized hierarchic.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com
    version="2.4">

    <display-name>HelloWorld Application</display-name>
    <description>
        This is a simple web application with a source code organization
        based on the recommendations of the Application Developer's Guide.
    </description>

    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>examples.Hello</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

</web-app>
```

Figure H: *The tree structure of a simple XML document example*

Figure H shows a standard XML document one can see this typical tree structure. The first line is the so called prologue, where the version of the XML is indicated. The part ”encoding” is optional and an attribute of the prologue. Generally we deal with elements. An element consists of a tag and a content. We consider the example in figure H. The line ” $< display - name > HelloWorldApplication <$

$/display - name >$ " is such an element. " $< display - name >$ " ist the start tag of the element, "$HelloWorldApplication$" the content of the element and " $< /display - name >$ " the end tag of the element.
Elements always have to be nested correctly. The start tag of an element contains the name of the element and optionally some attributes (for example the "encoding" in the prologue).[9] If we have a further element in the content of the origin element, we speak about a child element respectively a parent element. In Figure H for example the element "$web - app$" is the parent element of "$display - name$". As mentioned before, we deal with a JDOM2 document with XML structure. So we have a root element (the element, that has no parent element) and child elements. In the case of JDOM2 library we will say nodes instead of elements.
Given the obtained JDOM2 document, we have to search for our markers and ideally remember the path to them. To simplify the algorithm we will only search for one marker. If found, we will remember the path to it and search the next one.

Algorithm 1 SearchForMarker algorithm

```
procedure $search\_for\_marker(node\ n)$                ▷ n is the root element
    $p \leftarrow$ list of predecessors
    if $p.size \neq depth$ then
        $p.add(n)$
    else
        $p.removelast$
        $p.add(n)$
    end if
    for ($x = 0$, $x \leq$ number of childs of $n$, $x$++) do
        if child of $n$ has no children then
            if text of $n$ contains marker then
                finalpath $\leftarrow p$
                break
            end if
        else
            $search\_for\_marker$ (child of $n$)
        end if
    end for
end procedure
```

Algortihm 1 shows the pseudocode of the algorithm we used to find the marker. With the input of a node n, the algorithm uses depth-first search to retrieve the marker. p is the list of predecessors. Given, that we know the current *depth* of our search, we save all parent nodes up to n of our current node in the list of

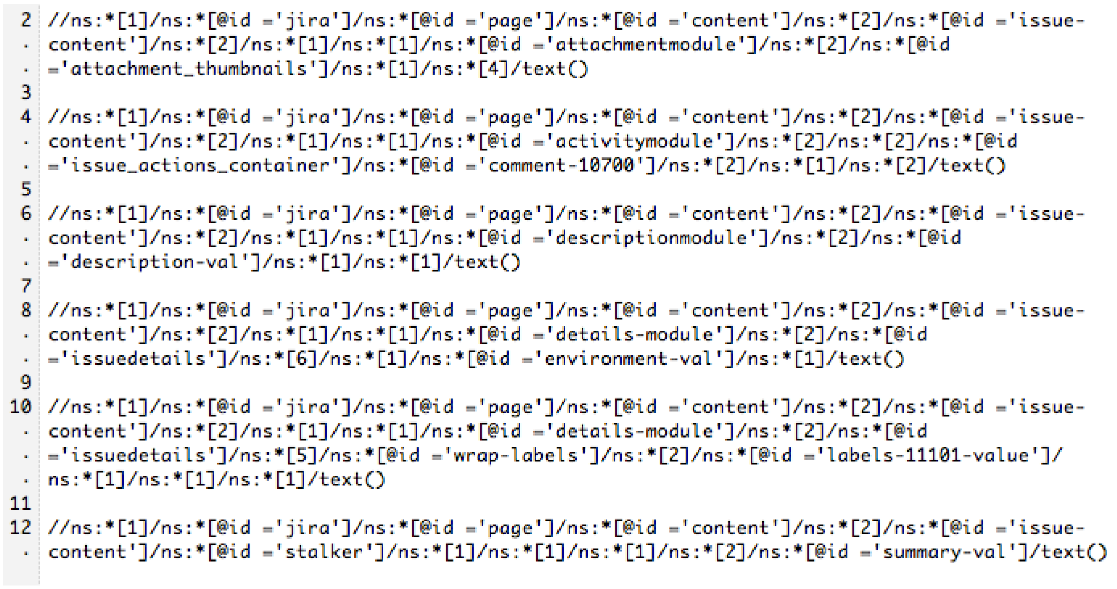
predecessors. In every recursive call of the procedure we check, if the size of the predecessor list is not equal to the depth. In the case $depth > p.size$, that means we are in a new plane of the tree structure and have to add our current to the list of predecessors.
If $depth = p.size$, we don't have a new plane and before adding the current node to the predecessor list, we have to delete the last one of it.
After that, in the for-loop we consider the children of node n. If the child x of n doesn't have any children, we test if the text of this node contains the marker we are looking for. If we found it, the predecessor list corresponds to the final path of the desired information. If the child x of n of the for loop has any children we call the procedure $search\_for\_marker$ with x instead of n.

So finally, if we found a marker, the $search\_for\_marker$ procedure yields a list with all the nodes of the XML document that are required to access the marker respectively the information of the corresponding node.

3.4 Generating a mining plan

Next step is to develop a method, which we are able to apply this obtained list to an issue report of the same bug tracking system as the origin report. That means, given the list, we need a possibility to access information of a XML document. To extract information of a bug report, we will convert the HTML code of its website again and as in 3.3.1 we will deal with an JDOM2 document with XML structure. The most obvious solution is to work with XPath. XPath, the XML Path Language, defined by the World Wide Web Consortium (W3C), is a query language for XML documents. Based on a tree representation of the XML document, the XPath language provides the possibility to navigate around the tree and select nodes. [10]
Ideally Xpath has been adopted by JDOM2, so that we can work with Xpath using the converted XML documents. Additionally we require Jaxen. Jaxen is an open source XPath library written in Java and it is adaptable to object models such as JDOM2.
After all we can use the XpathFactory. Given a XPath expression as a string this XPathFactory is able to generate a XPath object. With the help of another method of the XPath library we can evaluate this XPath object to the JDOM2 document of the concerning issue report. Finally this method provides the node, which contains the desired information respectively the marker in the origin issue report.
So we are able to access the content of a node, given a XPath expression as a

string. Therefore we have a small method, thats input is the predecessor list of 3.3 and the output is the corresponding XPath expression in the form a string.



```
//ns:*[1]/ns:*[@id ='jira']/ns:*[@id ='page']/ns:*[@id ='content']/ns:*[2]/ns:*[@id ='issue-content']/ns:*[2]/ns:*[1]/ns:*[1]/ns:*[@id ='attachmentmodule']/ns:*[2]/ns:*[@id ='attachment_thumbnails']/ns:*[1]/ns:*[4]/text()

//ns:*[1]/ns:*[@id ='jira']/ns:*[@id ='page']/ns:*[@id ='content']/ns:*[2]/ns:*[@id ='issue-content']/ns:*[2]/ns:*[1]/ns:*[1]/ns:*[@id ='activitymodule']/ns:*[2]/ns:*[2]/ns:*[@id ='issue_actions_container']/ns:*[@id ='comment-10700']/ns:*[2]/ns:*[1]/ns:*[2]/text()

//ns:*[1]/ns:*[@id ='jira']/ns:*[@id ='page']/ns:*[@id ='content']/ns:*[2]/ns:*[@id ='issue-content']/ns:*[2]/ns:*[1]/ns:*[1]/ns:*[@id ='descriptionmodule']/ns:*[2]/ns:*[@id ='description-val']/ns:*[1]/ns:*[1]/text()

//ns:*[1]/ns:*[@id ='jira']/ns:*[@id ='page']/ns:*[@id ='content']/ns:*[2]/ns:*[@id ='issue-content']/ns:*[2]/ns:*[1]/ns:*[1]/ns:*[@id ='details-module']/ns:*[2]/ns:*[@id ='issuedetails']/ns:*[6]/ns:*[1]/ns:*[@id ='environment-val']/ns:*[1]/text()

//ns:*[1]/ns:*[@id ='jira']/ns:*[@id ='page']/ns:*[@id ='content']/ns:*[2]/ns:*[@id ='issue-content']/ns:*[2]/ns:*[1]/ns:*[1]/ns:*[@id ='details-module']/ns:*[2]/ns:*[@id ='issuedetails']/ns:*[5]/ns:*[@id ='wrap-labels']/ns:*[2]/ns:*[@id ='labels-11101-value']/ns:*[1]/ns:*[1]/ns:*[1]/text()

//ns:*[1]/ns:*[@id ='jira']/ns:*[@id ='page']/ns:*[@id ='content']/ns:*[2]/ns:*[@id ='issue-content']/ns:*[@id ='stalker']/ns:*[1]/ns:*[1]/ns:*[1]/ns:*[2]/ns:*[@id ='summary-val']/text()
```

Figure I: *Here is an example of XPath strings that have been created of our method. The list of all pathes of fields we want to mine corresponds to the mining plan.*

A very important fact is to create unique XPath expressions. The nodes of a XML document have names. But these names are not unique, so we have to check, if the nodes have unique attributes. Such an unique attribute is the id of a node. But not every node is assigned to an id. Figure I shows six examples of XPath expressions. The double slash is the entree to the root node of the document. An expression like 'ns:*[1]' responds to the first node of the current depth. 'ns:*[@id ='jira']' responds to the node in the current depth with the id-attribute 'jira'. This query is unique and it is also a point for future work to find more unique attributes. The 'text()' term supplies the content of a node as a string.

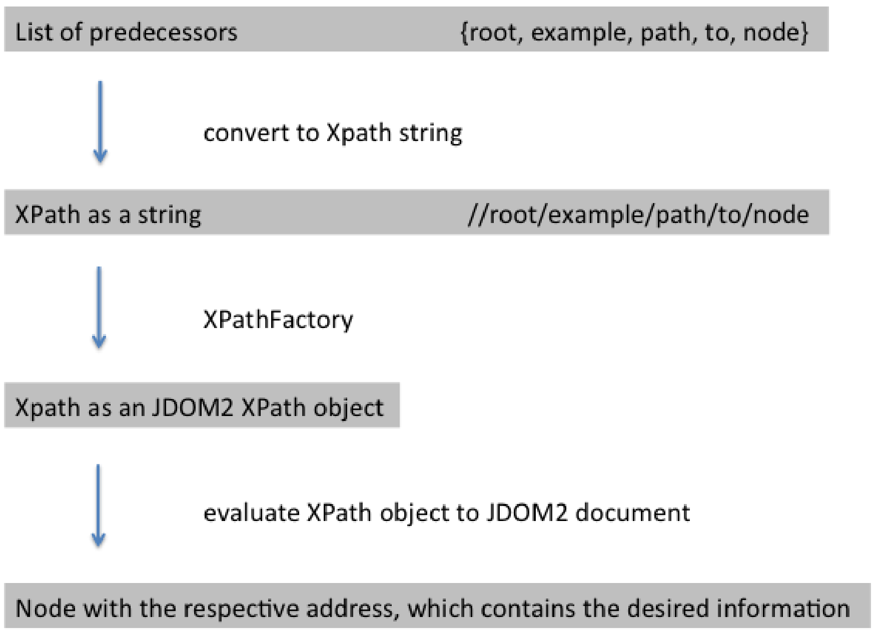
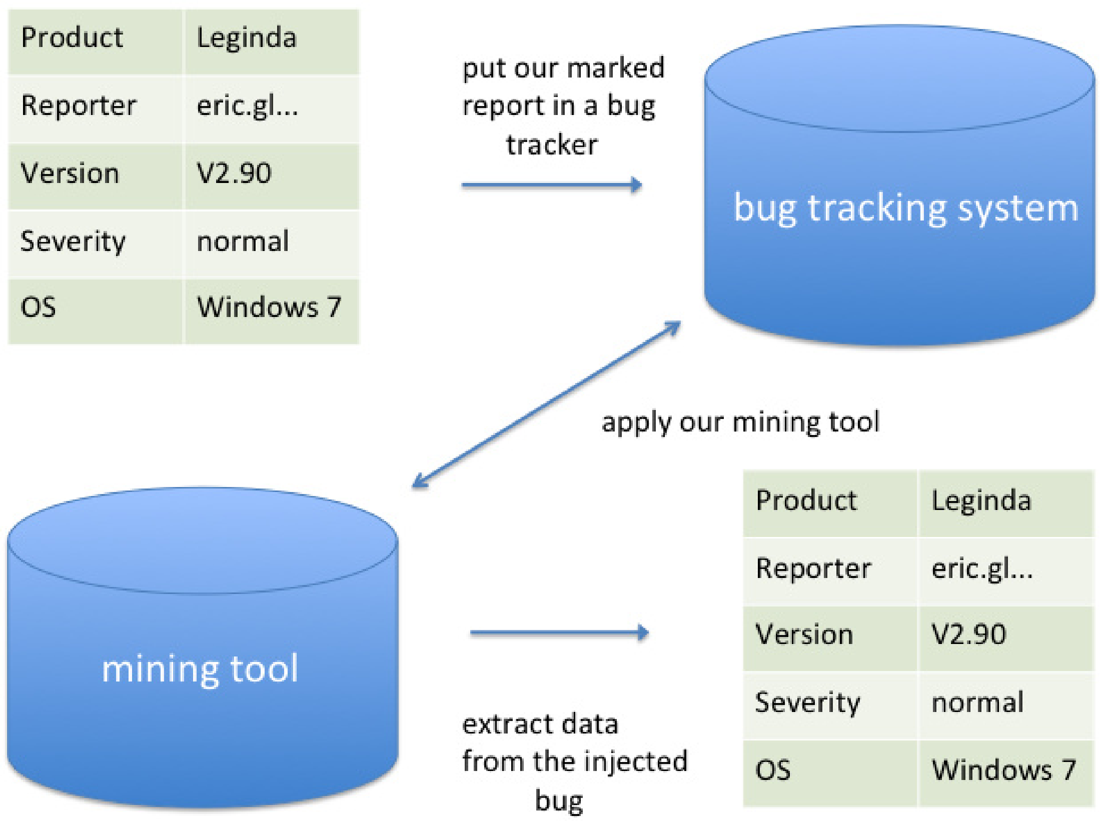

List of predecessors
{root, example, path, to, node}
convert to Xpath string
XPath as a string
//root/example/path/to/node
XPathFactory
Xpath as an JDOM2 XPath object
evaluate XPath object to JDOM2 document
Node with the respective address, which contains the desired information

Figure J: *The conversion of the predecessor list to a XPath expression which yields the desired node of the JDOM2 document*

Chapter 4

Evaluation and Analysis

After developing our methods, we can test their correctness by applying it to our benchmark set. As mentioned in chapter 3.2.2 we choose a set of random bug reports of our dataset and put them into two different bug tracking systems (JIRA and Bugzilla) and different version of these trackers. Then we will apply our methods to retrieve data to these bug reports and will compare the submitted results with our original data. If we get nearly the same information concerning the data sets after applying our methods, we can say that it is a good adaptive miner.


put our marked report in a bug tracker
bug tracking system
apply our mining tool
mining tool
extract data from the injected bug

Product	Leginda
Reporter	eric.gl...
Version	V2.90
Severity	normal
OS	Windows 7

Product	Leginda
Reporter	eric.gl...
Version	V2.90
Severity	normal
OS	Windows 7

Figure K: *A mining process of our methods without any loss of information.*

Figure K shows an optimal mining process. Ensuing from a bug report of our database equipped with our markers, we put this report in a bug tracking system, apply our mining methods and compare the result with our original bug report. We can attest ideal work to our methods if we obtain the same information as result, such as we put it in the bug tracking system.

Our data set for the tests consists of 17 bug reports, that cover a majority of fields, MOZKITO uses. Depending on the bug tracking system, we mined between 13 and 16 of this fields and analyzed the results. For reasons of simplification we restrict some fields. For example we analyzed reports with the maximum count of one concerning the fields attachments, comments and labels. The analysis of reports with several entries in these fields is a chapter for Future Work (5.2) and was not discussed in this thesis.

4.1 JIRA

This version of JIRA (v6.0.7) was the first bug tracking system, we deployed our data set. As mentioned before, we created 17 bug reports and analyzed the results concerning 13 fields. Altogether we achieved a rate of 88 % of retrieved data. Fields like Priority, Resolution or Status were very uncomplicated and achieved 100 % of retrieved data, in other words, no loss of data. But there are also fields, that a little bit more problematic. An example therefore is the information about the lastUpdatedTimestamp, thus the time of the last update of the report. The problem is the representation of the date and the time. In this version a timestamp has the form DD-MM-YYYY hh:mm. But if the report was created at the same day or the day before, the term 'DD-MM-YYYY' is replaced by 'Today' respectively 'Yesterday' and the XML structure of this report changes fractionally. So if we used a report with such a timestamp entry for the mining process, it would be difficult to find the timestamps of the other bug reports of our benchmark. During the tests for this version we achieved a value of 76 % of retrieved lastUpdatedTimestamps.
Another aspect affects the fields comment and label. As mentioned before, we are searching for attributes of nodes during the search in our XML documents such as the id. In the case of comment and label we come upon id-attributes as 'id=comment-1706' for example. That means that the id-attribute of this report is unique, but one will not find a node with this atrribute in a different bug report. We solved this problem by storing a value n. This n expresses that one can find the node that leads to the data at the n-th position in the concerning depth of the

Field	**Mined**	**Retrieved (in %)**
Attachment	✓	94
Comments	✓	94
Description	✓	100
Id	✓	0
LastUpdatedTimestamp	✓	76
PersonContainer	✓	94
Priority	✓	100
Product	✓	100
Resolution	✓	100
ResolutionTimestamp	X	—
Severity	X	—
Status	✓	100
Subject	X	—
Summary	✓	100
Type	✓	100
Keywords	✓	94
17 Fields		**88,6**

Table 4.1: This table shows the fields that have been mined and the percentage of retrieved data of each field

XML document. It is another point for future work to find a more elegant solution for this problem.
Altogether we can attest our methods satisfying results in the test runs of this version. However one has to admit, that this version also was similar to the test version during the development of the methods, so probably these mehtods achieve slightly better results concerning this version of JIRA than in other bug tracking systems or versions.

Chapter 5

Conclusion

5.1 Summary

During the development of the methods we used the JIRA Codehaus tracker respectively some reports of this bug tracking system for test intentions. JIRA Codehaus is a free bug tracking system. We used the version JIRA v6.1.6, powered by a free Atlassian JIRA open source license for Codehaus. This differs from the JIRA v6.0.7 tracking system of our evaluation. Concerning some points like number of attachments or comments, we had to lower one's sights, but this will be discussed in the chapter 5.2 Future Work.

Finally, a complete mining process proceeds in the following way. Starting from our CSV file with the data of reports and our markers, we read this file and memorize the number of reports and their fields respectively the value of the fields. Then we create reports in the bug tracking system, we want to test, by deploying the stored information as a HTTP post.
After that, we mine these reports on the basis of their URL. We apply our methods, that search for the markers or compare two similar bug reports. These methods supply rules in form of Xpath expressions for each field of a report, that indicates, how to find the value of the concerning field. That is the mentioned mining plan. By applying this plan to each of the created reports, we try to retrieve all data, our base set (CSV file) contains, and evaluate the results by comparing them with the base set.
At the beginning we spoke of different bug tracking systems or versions for test cases. Here we have to admit, that we did not test the methods in the bug tracking system Bugzilla for reasons of time. That has to be fetched later. The unexpected problem was the login procedure of Bugzilla that differed in some degree from login of JIRA and that took up a lot of time. It would be interesting to see, if the

perfomance of the methods in Bugzilla will be as good as in JIRA.
Altogether we can be contented with 88,6 % of retrieved data in JIRA and we can summarize, that the plans, we arranged, apart from the different versions for evaluation, have been realized and our methods yield the desired data with satisfying numbers.

5.2 Future Work

5.2.1 Enhancement of methods

At first we elaborate aspects of our methods that could be improved.
There are some points to improve the results of our methods. The first one is to advance the method CompareDocuments. This method yields the difference of two XML documents. We need it in the case, that we can not set a marker in a field of a bug report.
As mentioned before, we solve this quest by creating two similiar reports that only differ in the concerning field. Nevertheless this method often yields two or more results respectively different possibilities of pathes that lead to our field. This is caused by facts as time of creation, that we are not able to influence. So two different bug reports will always differ in this field. At this point one could develop an intelligent method to select the possibility with the highest probability to lead to the desired field. For example one could store the value of the time of creation and ignore it, if the CompareDocuments method yields this value as a difference of two documents. With this improve the value of retrieved data could be increased.
Another point to improve the results is to generate the xpath expressions based on more unique attributes of nodes in the XML documents.
At the moment we are only searching for the attribute id of a XML node. This is unique, but there are a lot of nodes in a XML document that do not have such an attribute. In this case we remember of what number the node was concerning the current depth of the document tree. If we would find some other attributes which we could store for our path to the field, we could improve the uniqueness of the generated xpath expressions.
A last point is to handle reports with more than one entry in fields like comment, attachment or label.

5.2.2 Absorption

The second point is to discuss possibilities to encross the thoughts of this work.
To pursue the idea of our methods, there is an interesting aspect. If we are able to

create a model or mining plan, that means we are knowing where and what data is delivered, we could analyze the evolution of the API. In other words themes like 'Analyzing Web Service Front-End Evolution' for example would be close to our methods.
If we continue this idea, we could contribute aspects like security and testing. Is data output, that has not been output before or is data output in other combinations as before?
The latter is an interesting aspect for web-testing. Given a set of data from which several users are able to see different parts of it. Then one wants to know, if its structure changes.
A last further idea is that generally it is known from database statistics, what the most important and the most prompted data of web-services are. Assume we would have a mining plan, we could see how easy or laborious it is to gain access to these data. That means, one could pursue the work of these methods in the direction of 'Interface Optimization' and 'Data Traceability'. This are advantages of the adaptive mining.

Bibliography

[1] A.Schroeder, T.Zimmermann, Rahul Premraj and A.Zeller: *If Your Bug Database Could Talk...*. In Proceedings of the 5th International Symposium on Empirical Software Engineering, ISESE 2006.

[2] N. Nagappan, T. Ball, and A. Zeller: *Mining metrics to predict component failures*. In Proceedings of the International Conference on Software Engineering, ICSE 2006.

[3] J. Sliwerski, T. Zimmermann and A. Zeller: *When do changes induce fixes*? In Proc. International Workshop on Mining Software Repositories (MSR), St. Louis, Missouri, U.S., May 2005.

[4] N. Bettenburg, S. Just et al.: *What Makes a Good Bug Report*? Universitt des Saarlandes, Saarbrcken, Germany, March 2008.

[5] N. Bettenburg, S. Just et al.: *Quality of Bug Reports in Eclipse*. Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange, ACM Press, New York, NY, USA, October 2007.

[6] Tim Berners-Lee: *Information Management: A Proposal*. CERN (March 1989, May 1990).

[7] Tim Berners-Lee: *First mention of HTML Tags on the www-talk mailing list*. World Wide Web Consortium. October 29, 1991.

[8] Confluence.atlassian.com: *JIRA 6.2 Release Notes*. Retrieved 18 April 2014.

[9] Mounia Lalmas: *XML Retrieval*. Morgan & Claypool Publishers, August 08, 2009.

[10] Randy Bergeron: *XPath Retrieving Nodes from an XML Document*. SQL Server Magazine, October 31, 2000.

[11] Y. Shafranovich: *Common Format and MIME Type for CSV Files*. Network Working Group (RFC 4180), October, 2005.