Heurac: A heuristic-based tool for extracting refactoring data from open-source software versions

Deepak Advani and Youssef Hassoun,
School of Computer Science, Birkbeck, University of London,
Malet Street, London, WC1E 7HX.

Steve Counsell, Department of Information Systems and Computing,
Brunel University, Uxbridge, Middlesex. UB8 3PH.

ABSTRACT

Refactoring is considered as a means to enhance software quality and maintainability. In pursuit of investigating trends in changes made via refactorings, this paper focuses on building a tool for detecting various refactorings performed on Java software. The tool automates the identification of refactorings as program transformations between consecutive software releases. For each release, Java source is parsed and key information about class entities is collected and saved as XML document. The XML tree representations corresponding to consecutive releases are compared according to a selected set of criteria, which define the refactorings to be identified. In doing so, our tool allows for empirical analysis of software development and aging from the perspective of refactorings transformations.

1 INTRODUCTION

Change is imminent to almost any Software system whether that is in the name of covering extra requirements, fixing bugs, improving performance, improving comprehension or for any other reason. However, changing the software haphazardly may adversely affect the software in which case the quality of the software gets degraded. At its best, the adverse effects could well be loss of comprehension caused by complicating the design and at worse unknowingly introducing new bugs, which sits like time bomb all set to blow up any moment. Refactorings [2, 5, 6] can be conceived as program transformations applied to existing software during its maintenance phase to prevent new bugs from creeping in. At the same, time refactorings aim at enhancing the design of software to improve its compression or perhaps apply standard practices.

Fowler [2] defines refactoring as “the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure”. Modifying the internal structure amounts to program transformations, however, these transformations preserve the external behaviour. When transformations are applied on the source code from this perspective, the motivation is either to improve the comprehension or perhaps apply standard practices. The former relates to increasing the human comprehensibility of a given source code while the later describes modifying source code in a certain way that is believed to be a best practise by the development community. Whatever be the reason, there has been a major hoo-ha in the developer community in regards to refactoring. Through this research, we intend to prove or disprove the extent to which refactoring technique is being applied to the software systems. As a part of our research, in this paper, we present a tool for detecting refactoring transformations in Java software systems.

In this paper, we present a tool for automating refactorings detection as software systems grow during the maintenance phase. The motivations behind analysing the growth of software systems from refactorings perspective are diverse. Automation allows us to address at empirically high scales many questions about the software growth and aging in relation to refactorings. For example, as refactorings aim at enhancing software comprehension and performance by improving the internal structure, the question is whether "more" refactorings means better software quality (in terms of cohesion, coupling, reuse etc.).

2 THE UNDERLYING MODEL

The underlying model of the tool is based on the concept of sets and can be described as follows. A Java software system is a collection of programming units (i.e. classes or interfaces). Each unit is viewed as a set containing various elements such as fields, methods, constructors, subclasses and superclasses. A field element is a set of two values that capture field's name and field's type. A method element is also a set of values such as method's access label (public, private or protected), return type, name and the method's parameter set. The constructor is modelled as set of values such as its name, access label and constructor's parameter set. As fields, methods and constructors are themselves modelled as sets of values, it follows that a class is a modelled as a set of higher order. Program transformations are then interpreted as changes in the set structure.

The set of values of the entire program are represented as an XML tree consisting of sequences of (sub) trees representing the individual types. Using this representation, XML data that appears to have been refactored can be identified by applying appropriate set of rules or criteria, which differs from one refactoring to another. The tool compares different releases of the same industrial software system according to a set of criteria thereby identifying the corresponding refactoring. Software system releases are direct result of bug-fixing and/or program re-structuring for the purpose of either maintenance or incorporating new features. Changes corresponding to refactoring transformations are sorted out by applying appropriate criteria.

3 SAMPLE DATA AND REFACTORINGS CONSIDERED

The tool was applied to seven different open source Java software systems of industrial standards chosen on random basis to cover a range of application domains including computer games (MegaMek and Tyrant), template engines (Velocity), compiler construction (Antlr), SQL databases (HSQLDB), documentation support (JasperReports) and PDF file manipulation (PDFBox). We applied the tool and followed the growth of each of these systems through the different releases. The systems considered are (with release identifiers between parenthesis):

1. MegaMek (MegaMek 029s1- s2, s3, s6, s7, s8, s9 & 029s10)
2. Tyrant (Tyrant 0309-10, 11, 12, 13, 14, 15, 16, 17 & 0318)
3. Velocity (Velocity 1.0- 1.0.1, 1.1, 1.2.rc3, 1.2, 1.3.rc1, 1.3, 1.3.1.rc2, 1.3.1 & 1.4)
4. Antlr (Antlr 2.6.0- 2.7.0, .1, .2, .3, .4 & 2.7.5rc3)
5. HSQLDB (HSQLDB 1.6.1- 1.7.0, 1.71, 1.7.2.11 & 1.7.3.1)
6. JasperReports (JasperReports 1.6.0- .1, .2 & 1.6.3)
7. PDFBox (PDFBox 0.6.1- .2, .3, .4, .5, .6 & 0.7a)

Detection of refactorings depends on the set-theoretic model described in the previous section. It also depends on a set of rules (or criteria), each of which defines a refactoring. At present, the model covers at least fifteen different refactorings ranging from simple (easy to implement) to more complex refactorings (require major changes). In the present version, the tool can detect the following refactorings: "Add Parameter", "Encapsulate Downcast", "Hide

Method”, “Rename Method”, “Remove Parameter”, “Encapsulate Field”, “Move Method”, “Move Field”, “Rename Field”, “Push Down Field”, “Push Down Method”, “Pull Up Field”, “Pull Up Method”, “Extract Subclass” and “Extract Superclass”. Fowler [2] divides refactorings into different groups according to the activity employed in transforming the system. According to Fowler’s classification, our refactorings are chosen from the groups: “Making Method Calls Simpler”, “Organising Data”, “Moving Features among Objects” and “Dealing with Generalisation”.

We note that some of the named refactorings are easier to implement than others. For example, renaming and hiding of fields and/or methods are relatively simple and do not require major program changes. Other refactorings, such as extracting sub- and/or superclasses are more complex and require structural changes involving class hierarchy.

4 IMPLEMENTATION OF TOOL

A software system, implemented in Java, consists of classes distributed over several packages underlying a hierarchy structure. Moreover, a class set can be represented as a tree where fields, methods, constructors, subclasses as well as superclasses denote either nodes or leaves; depending on whether an element is structured (has children) or simple (has no children). XML format has a tree structure and, therefore, it is natural to choose XML format to study the transformation between software systems releases using this format. Each program release is represented as an XML document file with the main package occupying the root. Each class is represented as a subtree having the name of the class and its classification (class or interface), in form of attributes[1]. Fields, methods, constructors, subclasses and superclasses constitute tree elements themselves.

Figure 1 displays the functionality of the tool in sequential order of execution. Phase-1 produces an XML document file in one-go for each release of each software system (referred to as API). Once any two consecutive releases of an API are parsed into XML files, we are ready for phase-2. In pahse-2, consecutive releases of all API’s are compared (one by one). Once all the consecutive releases of each and every API are compared, we are ready for phase-3. Phase-3 involves gathering statistics about how many refactorings have been performed.

Parsing involves extracting data that is relevant for detecting any of the fifteen refactorings (discussed above), formatting the data and saving it in an XML file. The resulting files are compared according to a particular type of refactoring, which dictates the criteria. The result of the comparison is reported (in XML format) if a refactoring appears to have been performed on the source code. The process can be repeated arbitrarily for different consecutive versions and for certain types of refactorings’. Theoretically, all refactorings can be detected using this approach. At present, however, our parser is hard coded to produce a limited XML repository, which can only serve the purpose of detecting above-mentioned 15 refactorings.

The tool makes use of recoder API [7]– an open source, object based, Java source code transformation tool – to parse java source code (Phase 1). Recoder API in turn depends on the JavaCC [3] compiler tool, which is customised to be used as a part of recoder API. The tool also makes use of JDOM API [4] to parse XML documents, build the java based XML documents and to traverse the XML tree structure (Phases 2 & 3).

[1] In XML, attributes are used to describe elements or to provide additional information about the elements.

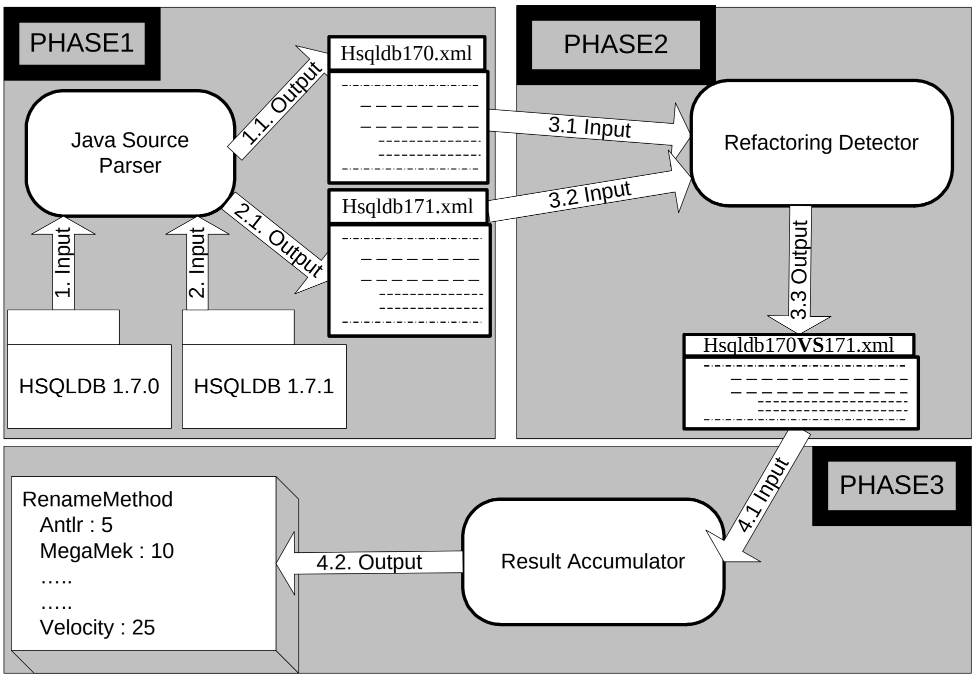
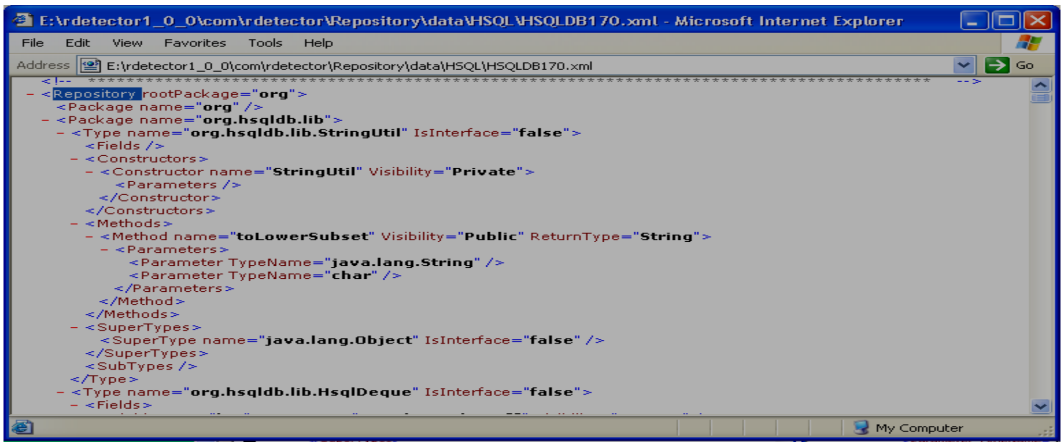


Figure 1: Pictorial representation of the tool

4.1 THREE DIFFERENT PROCESSING PHASES

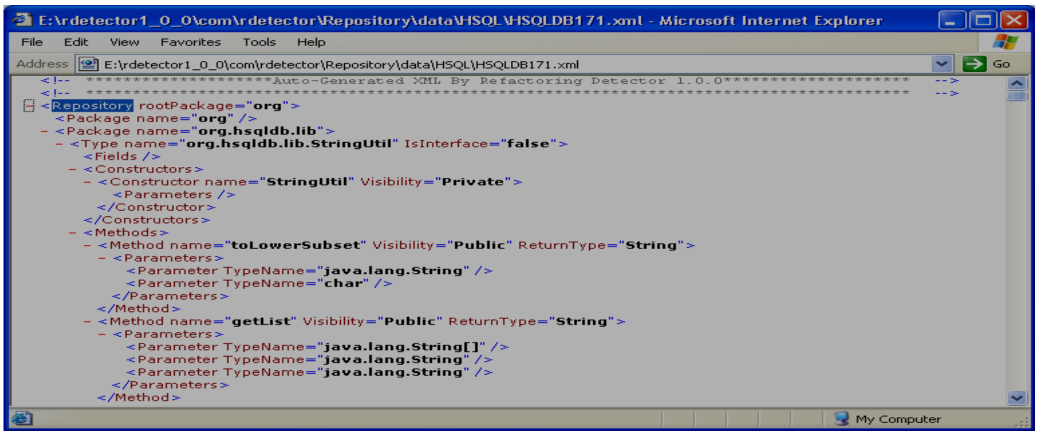
The first phase consists of parsing the code of consecutive software releases and generating XML representations. Data is extracted from the source code includes: root package, package name, type (class or interface) name, whether the type is an interface, the set of fields (if there are any) each with its name, type and visibility, constructors, their names, visibility and parameters, and lastly, methods with their names, visibility, return type and the set of parameters. Figures 2 and 3 show extracts of the of the XML tree generated by parsing 1.7.0 release and 1.7.1 releases of HSQLDB software, respectively.


E:\rdetector1_0_0\com\rdetector\Repository\data\HSQL\HSQLDB170.xml - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Address
E:\rdetector1_0_0\com\rdetector\Repository\data\HSQL\HSQLDB170.xml
Go
My Computer

```
  <!-- ********************************************************************************************* -->
- <Repository rootPackage="org">
    <Package name="org" />
  - <Package name="org.hsqldb.lib">
    - <Type name="org.hsqldb.lib.StringUtil" IsInterface="false">
        <Fields />
      - <Constructors>
        - <Constructor name="StringUtil" Visibility="Private">
            <Parameters />
          </Constructor>
        </Constructors>
      - <Methods>
        - <Method name="toLowerSubset" Visibility="Public" ReturnType="String">
          - <Parameters>
              <Parameter TypeName="java.lang.String" />
              <Parameter TypeName="char" />
            </Parameters>
          </Method>
        </Methods>
      - <SuperTypes>
          <SuperType name="java.lang.Object" IsInterface="false" />
        </SuperTypes>
        <SubTypes />
      </Type>
    - <Type name="org.hsqldb.lib.HsqlDeque" IsInterface="false">
      - <Fields>
```

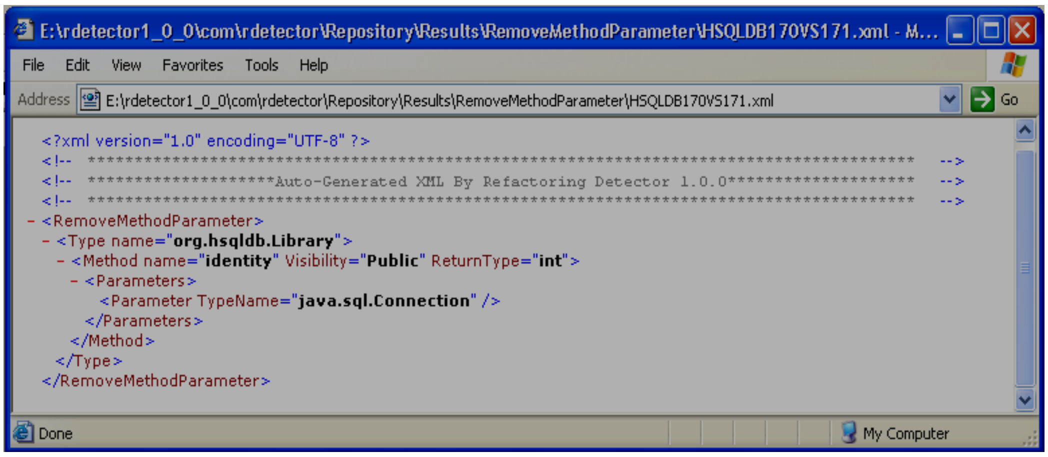
Figure 2: A partial imprint of parsed HSQLDB API release 1.7.0.

The second phase deals with comparing the resulting representations according to a particular refactoring. The result of this phase is an XML file containing specific information about class entity and the refactoring. For example, if a “Remove Parameter” has occurred, the XML tree structure looks like Figure 4.


E:\rdetector1_0_0\com\rdetector\Repository\data\HSQL\HSQLDB171.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address E:\rdetector1_0_0\com\rdetector\Repository\data\HSQL\HSQLDB171.xml
Go
My Computer

```
  <!-- ********************Auto-Generated XML By Refactoring Detector 1.0.0********************  -->
  <!-- **************************************************************************************  -->
- <Repository rootPackage="org">
    <Package name="org" />
  - <Package name="org.hsqldb.lib">
    - <Type name="org.hsqldb.lib.StringUtil" IsInterface="false">
        <Fields />
      - <Constructors>
        - <Constructor name="StringUtil" Visibility="Private">
            <Parameters />
          </Constructor>
        </Constructors>
      - <Methods>
        - <Method name="toLowerSubset" Visibility="Public" ReturnType="String">
          - <Parameters>
              <Parameter TypeName="java.lang.String" />
              <Parameter TypeName="char" />
            </Parameters>
          </Method>
        - <Method name="getList" Visibility="Public" ReturnType="String">
          - <Parameters>
              <Parameter TypeName="java.lang.String[]" />
              <Parameter TypeName="java.lang.String" />
              <Parameter TypeName="java.lang.String" />
            </Parameters>
          </Method>
```

Figure 3: A partial imprint of parsed HSQLDB API release 1.7.1.


E:\rdetector1_0_0\com\rdetector\Repository\Results\RemoveMethodParameter\HSQLDB170VS171.xml - M...
File Edit View Favorites Tools Help
Address E:\rdetector1_0_0\com\rdetector\Repository\Results\RemoveMethodParameter\HSQLDB170VS171.xml
Go
Done
My Computer

```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!-- **************************************************************************************  -->
  <!-- ********************Auto-Generated XML By Refactoring Detector 1.0.0********************  -->
  <!-- **************************************************************************************  -->
- <RemoveMethodParameter>
  - <Type name="org.hsqldb.Library">
    - <Method name="identity" Visibility="Public" ReturnType="int">
      - <Parameters>
          <Parameter TypeName="java.sql.Connection" />
        </Parameters>
      </Method>
    </Type>
  </RemoveMethodParameter>
```

Figure 4: Results of comparison between the two releases of HSQLDB API for "Remove Parameter" refactoring.

The third phase amounts to reading the XML files generated in the second phase and printing out the number of refactorings that took place.

To achieve high flexibility, configuration files are used to store parameter values needed for the processing of source code. The set of parameters include information about the paths of the system releases to be analysed, names of XML output files and name of the refactoring to be detected and so on. Specifically, in the first phase, we need to provide the parser with three parameters:

a) path to the location where the source code of a particular release of a software system or API (for example, HSQLDB's release 1.7.0) is saved;
b) the starting package name of the API that we want to inspect;
c) the absolute path and name of XML file, where we want the parsed contents to go.

In the second phase, we need four parameters. The first two parameters are absolute path names of the two XML source files that were produced by executing the first phase twice (the first time to parse a lower release and the second time to parse higher release of the same software system). For example, the first time we executed phase 1, we parsed HSQLDB's release 1.7.0 and the second time to parse release 1.7.1. The third parameter is the absolute path name of XML output file and the fourth is the refactoring that we want to detect. Finally, to execute phase 3, we need two parameters. The first is absolute location to where the hardcoded 43 XML files (containing results) must exist in each folder (resembling the refactoring name) and the second is either 1 refactoring name or "ALL" to indicate whether we want to collect statistics for 1 refactoring or all.

4.2 CODE PROCESSING AND ANALYSIS

The tool parses consecutive software releases implemented in Java and provides information about the number of fifteen different refactorings that has occurred during the transition from one release to the next. The processing of source code leading to the extraction of the number of refactorings that have taken place passes through three different phases, as depicted in the previous section.

4.2.1 PARSING PROBLEMS AND DEPENDANCIES

Parsing requires the availability of all dependencies of the given source code that is about to be parsed. Dependencies can be expressed as internal or external dependencies. Internal dependencies are auxiliary packages containing utility classes that are normally delivered with the software relaese. External dependencies are that on which a software system depends to get some work done but it is not a physical part of the software system. Such dependencies are normally available in form of jars. Both the internal and external dependencies are passed to the parser via a configuration file, which is again written in XML format. Various samples of the software systems investigated showed (as expected) increase in size and (consequently) increase in the number of dependencies. These dependencies must be identified before parsing can begin.

Apart from dependecies, the root location and the root package must be passed to the parser in the first phase. Root location is an absolute path, which represents the tip of root package. The root package is a package whose source code we want to parse. It may internally depend on auxiliary packages. Additionally, a root package may also depend on external other APIs. Finally, the output folder is used to store the tree representation of the system release in XML format.

Figure 5 depicts an example of parsing Antlr 2.6.0. which has internal dependencies but no external dependencies. Similarly, for each release of each API these inputs are to be determined before parsing can be carried out. The table A1 in Appendix A shows exactly what is required for each release of all sample software systems in order to parse.

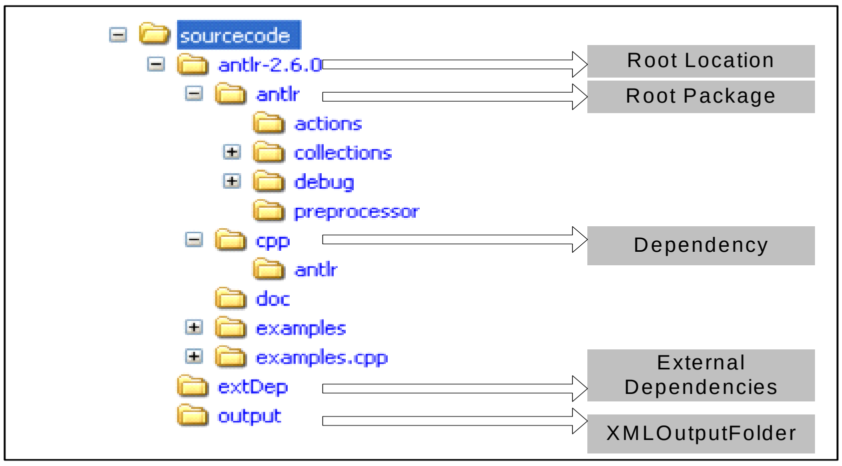


Figure 5: Illustrating inputs required by Parser.

We encountered some problems whilst parsing individual releases. These problems were either due to the offending statements that which the JavaCC parser (embedded in Recoder API) couldn't deal with or unupdated bytecodes in the external dependencies. For example, Tyrant 0.3.1.3 release and later are susceptible to an offending statement found in a class called “Scripts”. The statement is as follows:

```
String sn=new String[]  {"skill", "strength", …, "craft"} [Being.statIndex(stat)] ;
```

A work around involved modifying the source code to parenthesize it in following manner:

```
String sn=(new String[]  {"skill", "strength", …, "craft"}) [Being.statIndex(stat)] ;
```

Another example from parsing Velocity software releases 1.0, 1.01. and 1.1 which depend on jdom-b6.jar. The parser failed to process some of the bytecodes in jdom-b6.jar. These bytecodes must be replaced with its original source code, which is downloadble from jdom archives (available on www.jdom.org). The list of bytecodes that need to be replaced, belong to org.jdom, org.jdom.input and org.jdom.output packages are: Attribute, CDATA, Comment, DocType, Document, Element, Entity.class, IllegalAddException, IllegalDataException, IllegalNameException, JDOMException, SAXBuilder, SAXOutputter, XMLOutputter and ProcessingInstruction.

Similar parsing problems exist with Velocity software specifically with releases starting from 1.2-rc3 to 1.3.1. Only this time, the problems are related to jdom-b7.jar instead of jdom-b6.jar. However, the same solution can be applied as with jdom-b6.jar.

4.2.2 REFACTORING DETECTION CRITERIA: EXAMPLES

Each refactoring transformation is defined through a set of rules or criteria. In the second phase, XML tree representations of two consecutive releases are traversed and evaluated according to different criteria defining different refactorings. For example, to detect whether “Move Field” refactoring has taken place in the transition from one release to the next, the tool checks whether:

1. A field (name, type) that appears in a class type (belonging to older version) but appears to be missing i.e. dropped from the corresponding type (belonging to later version); and
2. the field (name, type) must not appear in any superclass or subclass of the original type; and
3. a similar field (name, type) appears to have been added to another type (belonging to later version); and
4. whose corresponding type in former version, if there is any, must not contain the field in question.

If entire criteria (clauses 1 to 4) is met for a field, under investigation, than the tool reports each such field as a “Move Field”. Each such field along with its type information qualifies as a part of results in XML format (See Figure 4 for an example of generated XML file on “Remove Parameter”). The criteria for “Move Method” refactoring are similar bearing in mind that a method representation includes (access label, name, return type, parameter list).

The criteria for “Extract Superclass” is as follows:

1. A class type whose unaccounted fields or methods are pulled up into a newly created superclass (that does not exists in former release) becomes an extracted superclass; and
2. the class from which this superclass was extracted becomes the base type.

The reported XML file includes the base class name with all the fields and methods that have been pulled up to define the new superclass in the later version. The criteria for “Extract Subclass” are same as that of “Extract Superclass” except that instead of pulling the fields or methods up the hierarchy, they are pushed down the hierarchy. In this case, we consider the source type as a source of extracted subclass.

4.2.3 RESULTS

In this section, we present a sample of results obtained from applying the tool. These results constitute part of the output of the third and final phase where refactorings are counted after they were detected and reported. Due to space restrictions we refrain from presenting full details and refer the reader to [1] where we give full account of results in different formats.

Table 1 shows the frequencies of refactorings occurred when comparing consecutive releases of the HSQLDB software. The first column refers to refactorings identified in the transition from release version 1.6.1 into 1.7.0. The next column to those identified in the changes occurred in transforming 1.7.0 into 1.7.1. The third and fourth columns follow the same pattern showing the number of refactorings between consecutive releases. We note the high frequency of simple refactorings such as “Rename Method” and “Rename Field” and relatively lower occurrences of more complex refactorings such as “Extract Subclass” and “Extract Superclass”. Remarkable is the absence of any of the fifteen refactorings in the last transition between 1.7.2 and 1.7.3. We notice a pattern of decline in the number of refactorings after reaching a “peak” (here, in 1.7.0 → 1.7.1 and 1.7.1 → 1.7.2) over all the software systems studied in this research.

HSQLDB	**1.6.1 → 1.7.0**	**1.7.0 → 1.7.1**	**1.7.1 → 1.7.2**	**1.7.2 → 1.7.3**
Encapsulate Downcast	0	0	0	0
Encapsulate Field	0	0	0	0
Hide Method	3	4	1	0
Rename Method	16	3	57	0
Rename Field	30	7	121	0
Move Method	13	1	25	0
Move Field	6	37	57	0
PushDown Method	0	0	0	0
PushDown Field	0	16	3	0
PullUp Method	0	0	4	0
PullUp Field	0	1	3	0
AddMeth. Parameter	9	6	24	0
RemoveMeth. Param.	3	1	3	0
Extract Superclass	0	1	6	0
Extract Subclass	0	2	3	0

Table 1. Refactorings identified between consecutive releases of HSQLDB

The frequencies of fifteen refactorings occurring over all the releases in the seven software systems are depicted in Table 2. A trend noticed in the HSQLDB software persists here, namely, the high number of simple refactorings compared to the number of more complex refactorings.

	Mega Mek	Jasper Reports	Antlr	Tyrant	PDF Box	Velocity	HSQL DB
Encapsulate Downcast	0	0	0	0	0	0	0
Encapsulate Field	0	0	0	2	1	9	0
Hide Method	0	0	0	1	0	4	8
Rename Method	1	1	8	12	14	55	76
Rename Field	0	7	7	9	5	23	158
Move Method	0	0	6	3	16	27	39
Move Field	0	2	6	0	6	21	100
PushDown Method	0	0	0	0	0	6	0
PushDown Field	0	0	0	0	0	7	19
PullUp Method	0	0	5	1	0	55	4
PullUp Field	0	0	0	0	0	10	4
AddMeth. Parameter	0	5	10	17	10	18	39
RemoveMeth. Param.	1	1	8	12	14	55	76
Extract Superclass	0	0	0	15	0	1	7
Extract Subclass	0	0	0	1	0	0	5

Table 2. Refactorings frequencies of all software systems over all releases

5 CONCLUSION AND FUTURE WORK

Refactoring transformations are meant to improve software comprehension and reduce software aging. This paper presents a tool for automating the analysis of software systems developments from refactorings perspective. Studying the refactorings trends as software systems grow, gives us a clue of the systems' quality. At present, the underlying model and consequently the tool cover a limited number of refactgorings. We are planning to extend to the scope of the tool to cover more refactorings. We are also interested in wrapping the tool in a GUI interface to make the investigation of refactoring trends in software development easier.

6 REFERENCES

[1] D. Advani, Y. Hassoun and S. Counsell "Refactoring trends across *N* versions of *N* Java open source systems: an empirical study", Technical Report BBKCS-05-02, Birkbeck College, School of Computer Science and Information Systems, 2005.
[2] M. Fowler (1999), K. Beck, J. Brant, W. Opdyke and D. Roberts (1999), *Refactoring: Improving the design of existing code*, Addison Wesley, 1999.
[3] JavaCC, *JavaCC Parser*, Available [ONLINE] from [https://javacc.dev.java.net/], Last Accessed 22nd August 2004.
[4] JDOM, *JDOM API*, Available [ONLINE] from [http://www.jdom.org], Last Accessed on 23rd August 2004.
[5] W. Opdyke. Refactoring object-oriented frameworks, Ph.D. Thesis, University of Illinois. 1992.
[6] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. Automated Software Engineering, 8:89-120, 2001.
[7] Recoder, *Recoder API*, Available [ONLINE] from [http://recoder.sourceforge.net/], Last Accessed on 23rd August 2004.

Appendix A – Release wise internal and external dependencies

	Release	Internal Dependencies.	External Dependencies
1	Antlr 2.6.0	\cpp	None
2	Antlr 2.7.0-.1,.2,.3,.4 & .5rc3	\lib	None
3	HSQLDB1.6.1	\lib	junit.jar
4	HSQLDB1.7.0, .1	None	servlet.jar,junit.jar
5	HSQLDB1.7.2.11 & .3.1	None	servlet.jar,junit.jar,jsse.jar
6	JasperReports 1.6.0, .1, .2 & .3	None	ant-1.5.1.jar, bsh-1.3.0.jar, commons-beanutils-1.5.jar, commons-collections-2.1.jar, commons-digester-1.3.jar, commons-logging-1.0.2.jar, commons-logging-api-1.0.2.jar, hsqldb-1.61.jar, itext-1.01.jar, poi-2.0-final-20040126.jar, servlet.jar, xalan.jar, xercesImpl.jar, xmlParserAPIs.jar
7	MegaMek029s1-s2, s3, s6, s7, s8, s9 & s10	None	collections.jar;Tinyxml.jar
8	PDFBox0.6.1, .2, .3, .4, .5, .6 & .7a	None	ant.jar, junit.jar, log4j.jar,lucene-1.2.jar, lucene-demos-1.2.jar
9	Velocity1.0	None	ant-1.3-optional.jar, antlr-runtime.jar, fop-bin-0_17_0.jar, JavaClass.jar, jdom-b6.jar, junit-3.2.jar, log.jar, log4j-1.0.4.jar, log4j-core-1.0.4.jar, oro.jar, servlet.jar, w3c.jar, werken.xpath.jar, xalan-2.0.0.jar, xerces-1.3.0.jar
10	Velocity1.0.1	None	// same as above plus ant-1.3.jar
11	Velocity1.1	None	Antlr-runtime.jar, commons-collections.jar, fop-bin-0_17_0.jar, JavaClass.jar, jdom-b6.jar, junit-3.2.jar, log.jar, log4j-1.0.4.jar, log4j-core-1.0.4.jar, oro.jar, servlet.jar, w3c.jar, werken.xpath.jar, xalan-2.0.0.jar, xerces-1.3.0.jar
12	Velocity1.2.rc3 & Velocity1.2	None	// as above with xalan-2.0.0.jar & xerces-1.3.0.jar missing
13	Velocity1.3.rc1- 1.3, 1.3.1.rc2, 1.3.1	None	antlr-runtime.jar, bcel-5.0rc1.jar, commons-collections.jar, fop-bin-0_17_0.jar, jdom-b7.jar, junit-3.7.jar, log4j-1.1.3.jar, log4j-core-1.1.3.jar, logkit-1.0.1.jar, oro.jar, servlet.jar, w3c.jar, velocity-dep-1.3.jar, werken.xpath.jar
14	Velocity1.4	None	// as above with jdom-b9.jar & velocity-dep-1.4.jar replacing jdom-b7.jar & velocity-dep-1.3.jar, resp.
15	Tyrant0309-10, 11, 12, 13, 14, 15	None	None
16	Tyrant0316-17, 18	None	junit.jar

Table A1: Internal and external dependencies of the chosen software systems