Innovations Syst Softw Eng (2011) 7:171–189
DOI 10.1007/s11334-011-0154-3

ORIGINAL PAPER

Rank-based refactoring decision support: two studies

Liming Zhao · Jane Huffman Hayes

Received: 2 December 2010 / Accepted: 19 July 2011 / Published online: 5 August 2011

Abstract Refactoring can result in code with improved maintainability and is considered a preventive maintenance activity. Managers of large projects need ways to decide where to apply scarce resources when performing refactoring. There is a lack of tools for supporting such decisions. We introduce a rank-based software measure-driven refactoring decision support approach to assist managers. The approach uses various static measures to develop a weighted rank, ranking classes or packages that need refactoring. We undertook two case studies to examine the effectiveness of the approach. Specifically, we wanted to see if the decision support tool yielded results similar to those of human analysts/managers and in less time so that it can be used to augment human decision making. In the first study, we found that our approach identified classes as needing refactoring that were also identified by humans. In the second study, a hierarchical approach was used to identify packages that had actually been refactored in 15 releases of the open source project Tomcat. We examined the overlap between the tool's findings and the actual refactoring activities. The tool reached 100/86.7% recall on the package/class level. Though these studies were limited in size and scope, it appears that this approach is worthy of further examination.

Keywords Refactoring · Maintainability · Decision support · Software engineering

L. Zhao · J. H. Hayes (✉)
Department of Computer Science, University of Kentucky,
Lexington, KY 40506, USA
e-mail: hayes@cs.uky.edu

L. Zhao
e-mail: lzhao2@uky.edu

1 Introduction

Refactoring is an approach to improving the design of software to make it easier to maintain without changing its external behavior [23]. Refactoring has been widely applied, such as in the Agile Process [60], and has also been validated by various researchers [19,52]. There is rarely enough time or money to apply all the desired refactorings to a software application. As a result, the following process is often followed by a software team performing refactoring:

1. identify code segments that need refactoring,
2. evaluate or predict the possible cost and benefit of each refactoring,
3. develop a refactoring plan according to the available project resources and use the results from the above steps, and
4. apply the refactorings.

Opdyke [56] stated that the software team is the one who makes the final decision and performs the actual refactoring. Proper tool support can make this process easier, faster, and more accurate. Commercial tools are available for applying refactoring automatically. However, effective tool support is still needed for the other three steps listed above.

Refactoring is aimed at reducing the complexity of certain code units. Programmers refactor a code unit to make it simpler. Programmers may also use indirections (such as extracting a method and calling the method) to hide the complexity of a given code unit. A code unit can have high complexity due to its size or internal logic as well as due to interactions with other code units. We focus on the former two in this study, examining the size and complexity metrics of code units to assist in identifying code that requires refactoring.

One way to prioritize the order in which code is refactored is by examining the benefits that will be gained from the refactoring. Improved maintainability of a code unit is one such benefit. Various techniques have been used to predict the maintainability of code and/or to identify code that is not easy to maintain (often called "bad smells" [23]). These techniques include maintainability models, similarity measures, logic queries, etc. [47,48,63,67]. Our maintainability (or refactoring target) prediction uses a hierarchical approach. Code maintainability is measured on multiple levels such as package level, class level and method level. The suggested refactoring unit that we examine is class. We assume that the same programmer wrote the code of a given class. Moreover, we assume that the code of a class is cohesive. We assume that if a programmer solves the maintainability problems of a given class, the experience gained from that can directly benefit the programmer when he/she works on a next such problem.

This paper introduces a hierarchical approach to identifying and prioritizing/ranking refactorings that could be undertaken. The refactorings are ranked based on predicted improvement to the maintainability of the software. We undertook two studies to investigate the usefulness of our approach as a means of augmenting human reviewers, one using a Java program written by graduate students and one using Apache Tomcat. The studies showed that our approach can support human reviewers and accurately identify code needing refactoring. Though these studies were limited in size and scope, it appears that this approach is worthy of further examination.

The paper is organized as follows. Section 2 discusses our approach to predicting classes in need of refactoring. Section 3 discusses a tool that implements the approach. Section 4 discusses the design and execution of the two studies, respectively. Section 5 presents related work. Conclusions and future work are presented in Sect. 6.

2 The refactoring decision support methodology

Programmers refactor software to make it easier to change, to debug and/or to understand. From the manager's viewpoint, refactoring is an approach to saving large amounts of maintenance effort in the long run at the price of the cost of the refactoring itself. Therefore, the first task of determining refactoring needs is to identify code units that require more maintenance effort than others.

Maintenance effort estimation may be performed based on project size and complexity, team skills, process maturity, software architecture, size of change, previous experiences, etc. (as discussed in Sect. 5). Many effort estimation models require a measure of size. The size measures how much code the maintenance programmers need to comprehend and/or modify:

$$\text{Weighted Size} = \alpha \times \text{SizeU} + \beta \times \text{SizeC} \quad (1)$$

where SizeU is the size of the code the programmers need to examine and understand and SizeC is the size of the code that has been changed (added, deleted or edited). Note that coupling is an important measure regarding how much code a programmer must comprehend. The size of change can be measured in two ways: the size of components under change and/or the size of the actual lines of code changed (edited, added, or deleted). Here, α and β are two constants related to complexity, self-expressiveness, etc.

In summary, maintenance effort is mainly proportional to the size of the code under maintenance and to the size of the code coupled (measured by coupling metrics such as RFC and CBO [13]). Due to the limited human resources of any given project, it is desirable to build an automated means, based on these metrics, to find the classes that most require refactoring.

There is a need for practical approaches and/or empirical work on how to make refactoring decisions based on relevant software metrics. The Maintainability Index (MI) provides a way to combine different metrics [68]. However, Welker [68] pointed out that "after all people maintain the software. Automated maintenance is not a reality yet. Therefore, it only makes sense that there are some characteristics of software construction that require a person to quantify for attributes such as maintainability. Determining maintainability purely by objective measures can be deceiving." Based on this idea, a refactoring decision support tool should reveal the characteristics of the software to the human users, and let them make the final decision based on their experience and available resources. In addition, a metric value will not necessarily make sense to a human analyst unless it is examined in the context of the software project/source code. For example, Welker [68] also stated that "…though object-oriented systems by nature have a fairly high MI due to the typical smaller module size. Naturally, smaller modules contain less operators and operands, less executable paths, and less lines of comments and code; therefore, the MI tends to be higher."

Based on the above analysis, it appears that a table providing all aspects of information obtained from related metrics can support refactoring decision making. Further, it seems intuitive that the information should be rank ordered to assist the human decision maker. Specifically, a weighted (or unweighted) sum of the ranking information can provide guidance for refactoring decisions. Let us examine an example.

In Table 1a, the cell in the second row and second column indicates that metric 1 of Class A ranked second (in needing refactoring) of the four examined classes. The second row, fifth column indicates that the ranks of Class A total

Table 1 Decision table based on metric ranking

Class	Metric 1	Metric 2	Metric 3	Total
(a) Unweighted decision table				
A	2	3	1	6
B	1	1	2	4
C	3	2	4	9
D	4	4	3	7
Class	Metric 1 (×2)	Metric 2 (×0)	Metric 3 (×3)	Total
(b) Weighted decision table				
A	4	0	3	7
B	2	0	6	8
C	6	0	12	18
D	8	0	9	17

six. Based on the total column, Class B is most in need of refactoring, with a value of four.

In Table 1b, the metrics have been weighted. Metric 1 has a weight of two, metric 2 has a weight of zero, and metric 3 has a weight of three. The weighted rank of metric 1 is four (column two, row two). We can see that Table 1b indicates that Class A needs refactoring the most with the lowest weighted rank sum of seven (row two, column five).

Next, we examine measures of interest.

2.1 Code metrics

We use size, coupling and complexity measures to estimate maintenance cost and to identify 'hot spots' or areas of particular interest for refactoring consideration. There are two unique characteristics of our approach: first, we use the *class* as the basic unit for making a refactoring decision; second, we follow a hierarchical approach to locate the class(es) of interest.

2.1.1 Coupling measures

Coupling is an important measure of class interdependency. The higher the coupling measure for a class that needs to be changed, the more classes that will need to be examined and/or changed in the maintenance process. Coupling includes direct and indirect coupling. We first examine direct coupling.

A *server class* is one whose attributes or methods are used by another (or others). A server class is directly coupled to the classes which use its services. Changes in a server class are easily propagated to other classes that are using its services [11]. Changes to such a class, therefore, will often require programmers to examine or change other coupled classes. SizeU and/or SizeC from formula (1) are thus not limited to the size of the server class. Also, if a class uses a service(s) provided by another class by method invocation, interface adapting or inheritance, it is often necessary to read and comprehend the source code of the service-providing class to understand the behavior of the class being maintained.

Direct coupling can be captured by a simple metric called Coupling Between Objects (CBO), one of the Chidamber–Kemerer or CK metrics [11]. Many researchers have shown that CK metrics can be used to predict maintenance effort [45].

Indirect coupling is the complement of direct coupling. Inheritance is an example of indirect coupling. It introduces important interdependencies between classes. The number of dependencies (in classes) and number of dependents (in classes) capture the interdependencies introduced by both coupling and inheritance. Indirect coupling can be measured by calculating the transitive closure of direct interaction relationships. It should be noted that the coupled classes might form a circular chain [11].

Two coupling metrics, afferent coupling and efferent coupling, are used in our approach to measure the dependency of a package [26]. Afferent coupling of a package is the number of types outside the package being maintained that depends on types within the package being maintained. High afferent coupling indicates that the package being maintained has many responsibilities. Efferent coupling counts the number of types inside the package being maintained that depends on the types outside the package being maintained. High efferent coupling indicates that the examined package is dependent [26].

2.1.2 Size and other measures

Size measures are very useful when estimating cost or productivity (effort). The number of lines of code (LOC), weighted methods per class (WMC), number of classes (NC), and number of public methods (NPM) can all be used to measure size. Some researchers use Halstead's software science measures to examine the characteristics of a piece of software, including its size. Halstead metrics[1] [28] are computed directly from operators and operands in the code.

In our approach, Halstead's *length* (total number of operators and operands), *vocabulary* (total number of unique operators and operands) and *effort* are used to represent the length, understandability and complexity of each class. Halstead metrics and WMC were used in Study 1 and Study 2, respectively (in Study 2, we used two open source metrics tools,

[1] We are aware that some researchers do not support the use of Halstead metrics or cyclomatic complexity. There are many researchers, however, who found that these metrics apply well in their environment [17,46,69,62,70]. Our work to date has also shown that these metrics assist greatly in maintainability predictions [33,34].

which contained a different set of metrics). Each metric is a good indicator of code size, as validated by much prior research.

In Study 1, we also used two variants of WMC to represent the accumulated complexity and size of a class. Long methods per class (LMC) is used to count the number of large classes and complex methods per class (CMC) is used to count the methods having an undesirable (high) cyclomatic complexity. The "undesirable" threshold can be set according to different project environments [73]. Our approach also uses several other CK metrics [12] such as depth of the inheritance tree (DIT) and lack of cohesion in methods (LCOM).

Next, process data and historical data are discussed.

2.2 Process and historical data

Version control systems (such as Subversion and CVS) have been widely applied to track the evolution of software. When a programmer uploads a version of a file (or files) to the VCS repository, it is called a *commit* action. A VCS repository stores every successfully committed version of a file by recording its relative difference with the previous version. It also records the date and time of each commit. Programmers can write comments for each commit. By utilizing a VCS, a software project allows access to snapshots of its evolution history. In practice, there is often a lack of good documentation for most software projects. Often, consultant companies have to work on a legacy project that is undocumented. Version control systems, thus, offer a very important resource for analyzing the quality and characteristics of a project. In our study, we use extracted revisions of Tomcat 6.0.x that contain refactoring records and commit logs (we do this by keyword matching), and we use the diff command to find all the files involved in the particular revision. We thus can determine: the locations of files of related changes, the number of files involved in a change and the lines of code that have been changed. In Case Study 2 (Sect. 4.2), this information has been used as the baseline (of what was actually refactored).

2.3 Interpretation of metrics and decision heuristic

As discussed earlier (Table 1), when the software team tries to make a decision on which classes (or packages) most need refactoring, the relative maintainability can be more important to them than the values of individual metrics. This is because different metrics have different foci. Ranks of individual metrics give the software team an idea of how a class performs based on diverse attributes (size, complexity, etc.) relative to other classes in the same development environment. Also, the classes that have a high rank based on the highest number of individual metrics should be a high priority for refactoring. Therefore, a comprehensive rank is needed. We call it the *weighted maintainability rank (WMR)*:

$$\mathrm{WMR} = \sum \mathrm{Weight}_i \times \mathrm{rank}_i \tag{2}$$

where $i = 1{-}n$ and n are the number of selected metrics. We will explain how we weight each metric in Case Study 2 (Sect. 4.2).

3 Tool support

To provide refactoring rankings using the information described above, we need a framework with the following functionality:

- parsing and analysis
- metrics collection
- repository analysis, and
- user interaction.

The overall architecture of the refactoring decision support tool is shown in Fig. 1. There are three major components: Code Repository Analyzer, Maintainability Prediction Component and Refactoring Planning Component. We discuss each in turn. As the tool is aimed to assist refactoring decision making for Java, it is called the Java Refactoring Inspection Assistant (JRIA). The tool has two major versions. The first version was used in Study 1 and the second version was used in Study 2.

The Code Repository Analyzer parses the source code, discovers structural characteristics of the code and collects metrics. JRIA version 1 mainly used size-based metrics (because these metrics were readily available at that time). In Version 2 of JRIA, several open source metrics tools were integrated to obtain coupling metrics and measures at the package level. The Code Repository Analyzer gets the necessary code information (size and coupling measures) by traversing the abstract syntax tree (AST).

The collected data are then sent to the Maintainability Prediction Component. We do not combine the raw metrics. Instead, we use individual metrics to rank classes separately. A ranking matrix is generated for the classes to give a comprehensive view of the relative maintainability attributes of all the classes. At the same time, a weighted sum of the ranks of different predictors yields an integrated WMR as described in Sect. 2.3.

Based on the ranking obtained from the Maintainability Prediction Component and a threshold provided by the human analyst, a refactoring decision/plan can be generated by the Refactoring Planning Component.

For JRIA Version 2, we investigated several existing open-source Eclipse-plugins including CKJM, Metrics [65] and JDepend [14] to understand how to obtain all the needed

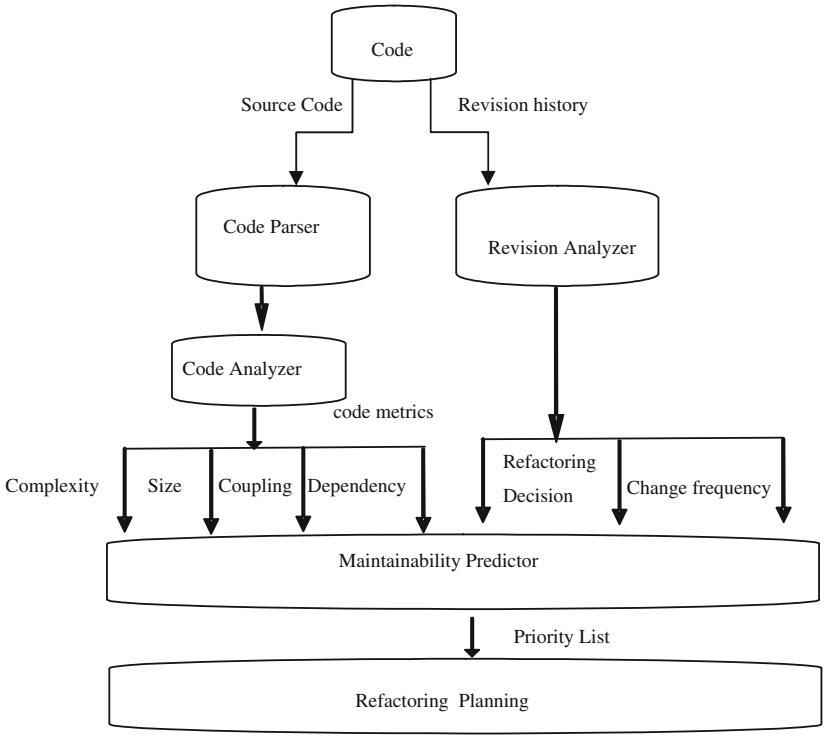

Fig. 1 Refactoring support tool framework

Table 2 Descriptive statistics of collected metrics

	Min	Max	Mean	SD
Halstead_L	4	491	211.4	155.6
Halstead_V	4	189	86.8	54.2
Halstead_E	0	72,044	20,727.3	21,727.0
MI	37.5	125.3	65.1	23.2
LMC	0	11	3.75	2.59
CMC	0	5	0.65	1.27

metrics and refactoring data. We found that none of the open source tools provide all the metrics we need, though they do provide some needed metrics. Further, none of these tools provide information on the history of revision record or analysis (RA). Also, the tools do not analyze the obtained metrics to provide high level decision support—the users have to do all the analysis themselves (Table 2).

In JRIA Version 2, we integrated package-level dependency information from JDepend and class-level information from CKJM using a two-phase approach discussed further in Sect. 4 (Fig. 2).

4 Empirical validation

This section presents two case studies undertaken to examine the refactoring decision support approach.

4.1 Study 1: pilot study

A pilot study was undertaken to evaluate the feasibility, practicality and usefulness of using data obtained from source code to predict refactoring needs. In the study, computer science graduate students were asked to review a Java application (consisting of 20 classes) and identify classes needing refactoring. The code had been written as a project for a graduate software engineering class at the University of XXX (made anonymous). Metrics were collected from the source code, and the refactoring decision support methodology was applied to find the classes in need of refactoring and the priority of the classes in need. This result was compared with the decisions made by human reviewers. Note that all data used for this study can be found in the Predictive Models for Software Engineering (PROMISE) repository.[2] We follow the guidelines for case studies developed by Kitchenham et al. [41] and discuss the study context, hypotheses, planning and analysis of the results below.

4.1.1 Study context

The study context consists of three parts [42]: objectives, baseline and constraints. The objective of the study was to examine the usefulness of the tool. Instead of simply concluding "the tool is better than human reviewers" or vice versa, we were interested in observing differences and whether or

[2] http://promise.site.uottawa.ca/SERepository/.

Fig. 2 The hierarchical approach to locating refactoring candidates

(a) Package level selection

Package level analysis starts
Set threshold (NT_{NRE}) on efferent coupling.
Set threshold (NT_{NRA}) on afferent coupling.
Rank packages again using efferent coupling.
Select package set SP_{NRE} based on NT_{NRE}
Rank packages again using afferent coupling.
Select package set SP_{NRA} based on NT_{NRA}
Return $SP_{NRE} \cup SP_{NRA}$
Package level analysis ends

(b) Class level selection

(For each selected package)
Class Level analysis starts
Set a threshold (NGRank)
Get CBO, RFC, WMC for each class
Calculate ranks of CBO, RFC, WMC, of each class inside this package
Calculate weighted Sum of these ranks (Grank)
Return a list of classes with Grank ≥ NGRank × ClassCountPKG
Class Level analysis ends.

not the manual and automated approaches supported each other. We thus posed five research questions:

RQ1. Does the JRIA output agree with that generated by humans?
RQ2. Is JRIA faster than humans?
RQ3. Does JRIA support humans?
RQ4. Can human beings identify all the refactoring needs? and
RQ5. Can human reviewers be replaced by JRIA?

4.1.2 Hypotheses

The study examined the following hypotheses (each null hypothesis is followed by an alternative hypothesis):

$H1_0$: When identifying the set of classes in need of refactoring, JRIA does not agree with humans.
$H1_a$: When identifying the set of classes in need of refactoring, JRIA agrees with humans.
$H2_0$: JRIA does not agree with humans when prioritizing the set of classes in need of refactoring.
$H2_a$: JRIA agrees with humans when prioritizing the set of classes in need of refactoring.
$H3_0$: There will be no difference in the amount of time spent (efficiency) on the refactoring identification task between JRIA and the human reviewers.
$H3_a$: JRIA and the human reviewers will spend different amounts of time on the refactoring identification task.
$H4_0$: When identifying the set of classes in need of refactoring, JRIA will discover all the bad smells identified by human reviewers.
$H4_a$: When identifying the set of classes in need of refactoring, JRIA will not discover all the bad smells identified by human reviewers.

4.1.3 Planning

Planning accomplished for the study is discussed next.

4.1.3.1 Subjects and objects The object of the study was a Java speech therapy application. It provides children who have speech impediments an entertaining way to practice their speech therapy exercises. If the child vocalizes in the proper pitch and loudness range, visual rewards are provided (a monkey climbs a tree). The program consists of 20 Java classes and was written as part of a graduate-level software engineering course at a US university. The subjects were graduate students who participated (as volunteers) in the code refactoring review.[3]

4.1.3.2 Study design The graduate student volunteers first performed some pre-reading and completed a short questionnaire. Based on their answers, we decided if they could participate in the study (some students had no experience with Java or maintenance programming and were omitted). We then gave the selected (seven) volunteers instructions for the study. They were asked to complete two reports for the study. They were instructed to read the Java code and look for and record "bad smells" (which were defined in the pre-reading). In the first report, we allowed them to select a bad smell from a provided list or write in a problem that they observed that was not on the list. The volunteers were asked to decide which class had the most serious problems and should be refactored first. In the second report, they provided us with a prioritized list of all the classes that they thought required refactoring. They also recorded the time that they spent reviewing/reading each class. Six students completed the study. One student did not complete the study because of an unexpected time conflict.

[3] The study has an approved institutional review board (IRB) protocol at the US University.

We also ran our refactoring decision support tool to generate a class-based priority list (JRIA does not generate an individual list of bad smells). The results from the manual inspection and JRIA were then compared.

There are a number of threats to the validity of our study. One threat to external validity was the use of students versus professional programmers. In partial mitigation of this threat, all selected reviewers had over 1 year of professional programming experience (four of the six had over 3 years of professional experience). Also, Host et al. [36] found that students perform the same as professionals on small tasks of judgment.

An additional threat was that volunteers may have lacked experience with refactoring. We tried to limit this threat by screening the graduate students.

A threat to the generalizability of results was that we had a very small set of volunteers. Such a small sample is not unheard of for pilot studies, but we plan to recruit a larger number of volunteers in future studies.

A possible threat to construct validity is the use of various models to measure maintainability. The threat was reduced by using common metrics (such as efferent coupling) in our model and an algorithm based on ranking and filtering instead of simply adding values from different dimensions together (Sect. 2.3). We also tried to use metrics verified in our previous research [33,34] or by other maintainability researchers [16,66].

We classified the difficulty level of identifying bad smells based on the time spent by the reviewer. That is, we tried to label a subjective measure with an objective value. This is another threat to construct validity.

A threat to internal validity is that of confounding variables. We minimized this threat by making sure the volunteers had not previously reviewed the study subjects and by advising the volunteers not to talk to each other during the study. Another threat to internal validity is the incompleteness of reported data—one of the reviewers did not record the time spent as required. In that case, we omitted the data point.

4.1.4 Analysis and discussion of results

All six reviewers turned in the two requested reports. As introduced in the study design (Sect. 4.1.3.2), the first report lists individual bad smells or design problems that the reviewers identified in each class. The second report provides a prioritized list of classes in need of refactoring. We observed that reviewers used different criteria to identify the most important bad smells. For example, results from the first reports showed that some reviewers looked for classes of unusual size, some looked for classes with high complexity, and some were interested in classes with dead code or duplicate code.

Table 3 Reviewer selection of classes to refactor compared to tool selection

Priority	Rvwr1	Rvwr 2	Rvwr 3	Rvwr 4	Rvwr 5	Rvwr 6	Tool
#1	I1	A	G	A	A	G	A
#2	F	W	T	P	E	H	LO
#3	W	LO	A	S	G		H
#4	G	V	L	W	H		P
#5	P	S		M	LO		F
#6					P		L

A AnimationScreen, *F* FileOutPut, *L* LoginScreen, *P* PitchAnalyzer, *S* SessionReport, *E* ErrorWindow, *G* Graphic, *LO* LoudnessAnalyzer, *R* RegistrationScreen, *T* TestScreen, *F* FileInputHandler, *H* HistoryReport, *M* MenuScreen, *W* Welcome, *I1* ImagePanel1, *I2* ImagePanel2, *V* Voice

Table 4 Metrics of class AnimationScreen

Halstead_L	Halstead_V	Halstead_E	MI	LMC	CMC
491	189	72,044	37.5	5	2

Table 2 presents descriptive statistics for the metrics collected. Table 3 shows the classes identified by the reviewers. The class name abbreviations are shown below the table. The first column gives the priority rank. The first row represents six reviewers and JRIA (Rvwr1 represents Reviewer 1, for example). Any of the other cells represent a class on which the reviewer made a decision. For example, the cell in the second row and the third column has the value "A" and indicates that reviewer 2 selected AnimationScreen as the class that needs to be refactored first (priority #1).

We found that 50% (three out of six) of the reviewers and JRIA agreed that Class A is priority #1 for refactoring. In fact, 67% (four out of six) of the reviewers put AnimationScreen (Class "A") on the priority list. We examined the metrics collected for AnimationScreen and found that it has the largest value of Halstead length, Halstead vocabulary, Halstead effort, and the smallest value of MI (the smaller the MI, the worse was the maintainability). In addition, the LMC and CMC (Sect. 2.1.2) of this class were above average (Table 4). This indicates that both JRIA and the reviewers agreed on what was the most complex class. This small study lends some initial support to the notion that complexity and size are among the major factors that add difficulty to programmers' comprehension of existing code and can be used to predict classes needing refactoring.

We also found that it was the case that 28% of the bad smells identified by the reviewers were not on the short list of bad smell types in the study handouts that we provided to them. Some of these bad smells were also not discovered by JRIA. We made the following observations from the reports from the human reviewers. First, reviewers were more likely to pick things off the instruction list than to identify

Table 5 Agreement analysis using an information retrieval ranking

Row entry compared to column entry	tf–idf relevance							
	Tool	Rvw1	Rvw2	Rvw3	Rvw4	Rvw5	Rvw6	Mean
Tool	na	0.3252	0.1414	0.3129	0.0748	0.341	0.3162	0.25192
Rvw1	0.4932	na	0.1601	0.0763	0.2369	0.1503	0.1807	0.21625
Rvw2	0.2105	0.1571	na	0.0267	0.5332	0.1928	0	0.18672
Rvw3	0.5326	0.0856	0.0305	na	0.0314	0.1236	0.2183	0.17033
Rvw4	0.1157	0.2418	0.5546	0.0286	na	0.106	0	0.17445
Rvw5	0.5646	0.1641	0.2144	0.1202	0.1134	na	0.6885	0.31087
Rvw6	0.3162	0.1191	0	0.1283	0	0.4159	na	0.16325
Mean	0.37213	0.18215	0.1835	0.1155	0.16495	0.2216	0.23395	

a bad smell not on the list. Second, though reviewers did make similar decisions on which classes should be refactored first, the decisions were often based on very different criteria. Code complexity appears to be the code characteristic that leads most reviewers to identify the same groups of classes. However, the background and experience of the individual reviewer appeared to be the main determinant of the kinds of bad smells for which they searched. The root cause needs to be investigated in future studies.

Although only five or six classes were selected by our reviewers from a total of 20, there were some classes that were selected by both a reviewer and JRIA. For example, the classes AnimationScreen and LoudnessAnalyzer appear in both Reviewer 2 and JRIA's priority list (Table 3).

To further examine the overlap among the choices of the reviewers, we examined the kappa interrater agreement measures, including Cohen's kappa and Krippendorf's alpha. These cannot be applied because of a number of factors:

- the reviewers did not provide observations of all 20 Java classes,
- some reviewers provided observations on six methods, some provided observations on anywhere from two to six methods, and
- there is no way in the interrater agreement measures to give credit for the fact that "making the list" is more important than whether the rankings of #1–#6 agreed. That is to say, if method A shows up in all six reviewer lists and that of JRIA, then JRIA "succeeded" in identifying a method that needs refactoring. The fact that one reviewer ranked method A as #6 and another reviewer ranked it as #1 is less important than the fact that it was picked by both reviewers out of the 20 methods.

Due to this, a statistics professor and researcher was consulted. He assisted in designing a method for assessing the agreement level of the reviewers and JRIA. The method uses information retrieval (IR) techniques. In this approach, called vector space model with term frequency–inverse document frequency (tf–idf) weighting, the list of all 20 classes as well as the returned lists from each reviewer and JRIA are treated as a document collection. Each reviewer's list (of anywhere from two to six methods) is, in turn, used as a query for which the document collection is searched. The tf–idf method returns a relevance score ranging from 0 to 1, where 0 means no relevance. The higher the score, the more relevant is the list. The results from this method are shown in Table 5. The columns are as follows: Tool, Reviewer 1 (Rvw1), through Reviewer6 (Rvw6) as well as Mean. The rows are the same. The table is read as follows, the cells represent the ranking in the list when the row was used as the query into all the other reviewer lists. For example, the entry of 0.3252 in JRIA row and Rvw1 column means that when JRIA's list was used as the query, the relevance weight for Reviewer 1's list was 0.3252. (Note: in general, information retrieval methods applied to software engineering problems treat relevance of 0.2 as a filter point, so that items above 0.2 are much more likely to be a "match") In comparison, Reviewer 4's list did not have much in common with JRIA's list, returning a relevance score of only 0.0748. The Mean relevance of all the other reviewer's list when compared with the list of JRIA is 0.25192 (the Mean column). This shows that the decision made via the refactoring decision support approach overlap with that made by the human reviewers. The row that is labeled Mean has the mean of the columns. The Mean relevance score of the JRIA column (how relevant all the other lists found JRIA list when they were used as the query) is 0.37213. The two highest means for the column and the rows have been highlighted. It can be seen that JRIA had the most relevant column score by far at 0.37213—meaning that JRIA list was very relevant to the other lists (the decision choice of reviewers).

Next, we examined a count of agreement on common methods between the lists (Table 6).

Each agreeing rank of common methods gets a count of 1. For example, JRIA had an item in the same rank as Reviewer

Table 6 Agreement count on rank of identified classes

Row compared to column	Only get credit (1) when rank of common classes agrees							
	Tool	Rvw1	Rvw2	Rvw3	Rvw4	Rvw5	Rvw6	Mean
Tool	na	0	1	0	1	1	0	0.5
Rvw1	1	na	0	0	0	0	0	0.166667
Rvw2	1	0	na	0	1	1	0	0.5
Rvw3	0	0	0	na	0	0	1	0.166667
Rvw4	1	0	1	0	na	1	0	0.5
Rvw5	1	0	1	0	1	na	0	0.5
Rvw6	0	0	0	1	0	0	na	0.166667
Mean	0.666667	0	0.5	0.166667	0.5	0.5	0.166667	

Table 7 Approximate time spent for finding bad smells in classes

Class	Rv1	Rv2	Rv3	Rv4	Rv6
Animation Screen		F	M	F	
FileOutput	F + M				
Graphic	S		F		M
HistoryReport					M
ImagePanel1	F + M				
ImagePanel2	M				
LoginScreen		F	M		
MenuScreen				S	
PitchAnalyzer	F	F		F + M	
RigistrationScreen		M			
SessionReport		F		F	
TestScreen			F		
Welcome	F + M	F		S	
WriteToFile	M	F			

2, Reviewer 4 and Reviewer 5. The values across the row were then averaged. Again, JRIA had the highest mean when compared with the other reviewer lists. Also, Reviewers 2, 4 and 5 had fairly high means for their row.

We also found that reviewers spent significant time in understanding the scope of the source code, reading documents and reading the source code to perform this study. Five of the six reviewers reported (one reviewer did not record the time spent as we noted Sect. 4.1.3.2) their total time spent, with a minimum of 1 h and a maximum of more than 3 h. In contrast, it took no more than a few seconds to run JRIA on the small-sized project source code used in the study.

The reviewers were asked to record the time that they spent on each problem (bad smell) that they found. Table 7 indicates how long it took for the reviewers to identify a bad smell. In the table, F (fast) means <1 min spent, M (medium) means 1–5 min were spent, and S (slow) means that more than 5 min were spent. An entry of F + M means that identifying one problem took <1 min and that identifying a second problem took 1–5 min. Each column represents a reviewer's ratings for the problems he/she identified in classes. For example, the cell in the third row and the second column has the value F + M. This means that two design problems (one took time F and one took time M) were identified by Reviewer 1 in the class FileOutput. In total, 29 problems were recorded by the reviewers. Only three (10%) of these problems were classified as S (slow) based on time spent.

This shows that code reviewers/inspectors are more likely to identify problems that can be found quickly and thus it is possible that they may miss problems that require more time to detect. This provides evidence in favor of tool support, especially for industry-sized projects. On the other hand, reviewers identified bad smells such as dead code, duplicate code and unused class that JRIA is not yet able to identify. This provides preliminary support that it is worthwhile to continue the design and development of JRIA so that it can identify more individual bad smells.

In summary, we found evidence to support rejection of $H1_0$ in favor of $H1_a$, as there was overlap between the set of classes identified by the refactoring decision support approach and by the human reviewers. We also found evidence to support rejection of $H2_0$ in favor of $H2_a$, as there was overlap in the priorities (ranks) identified by JRIA and the human reviewers. Moreover, we found evidence to support rejection of $H3_0$ in favor of $H3_a$ as JRIA ran in no more than a few seconds as opposed to the human reviewers who took from 60 to 180 min to perform the work. In addition, it was found that human reviewers may miss some deep problems in the code, possibly due to time constraints, lack of experience, etc. Finally, reviewers found some problems that were not discovered by JRIA. This indicates that JRIA cannot completely replace human reviewers. This lends support for the rejection of $H4_0$ in favor of $H4_a$.

4.2 Case Study 2: Apache Tomcat project

This section presents the empirical results of a second case study that further demonstrates the feasibility, practicality

and effectiveness of the refactoring decision support approach by using the data from the source repository of an open source project: Apache Tomcat [1]. In the case study, the results from the decision support approach were compared with the actual refactoring that had been performed by the open source developers.

4.2.1 Case study context

The objective of the case study was to examine how well the hierarchical refactoring decision approach predicts the classes that were actually refactored in an open source project. The baseline used for comparison was the actual refactoring performed by the open source developers of Tomcat.

4.2.2 Hypotheses

The case study examined the following hypothesis (followed by the alternative hypothesis):

- $H5_0$: There will be no overlap between the set of classes identified as needing refactoring by JRIA and the set of classes actually refactored by the Tomcat developers.
- $H5_a$: There will be overlap between the set of classes identified by JRIA and those refactored by the Tomcat developers.
- $H6_0$: JRIA is not better than a random selection in identifying the refactoring candidates (therefore it is not useful at all).[4]
- $H6_a$: JRIA is better than a random selection in identifying the refactoring candidates.

4.2.3 Planning

Planning accomplished for the second case study is discussed next.

4.2.3.1 Subjects and objects Fifteen releases (Apache Tomcat 6.0.0–6.0.14) were used for the case study. The source code of Tomcat 6.0.0 was imported to an Eclipse project from the Subversion source repository of the Apache Tomcat project (version 6.0.0). The project consists of over 100 Java packages. Package, class and method-level metrics were calculated as the basis for refactoring decision making. The Subversion source repository stores information about Apache Tomcat's change history. The revision history from version 6.0.1 through 6.0.14 was examined to find all the revisions with commit logs related to refactoring.

[4] We compare to random selection to ensure that JRIA captures more than noise information.

4.2.3.2 Case study design Multiple levels of code metrics (package, code and method level) were collected from the Apache Tomcat source code (version 6.0.0). A hierarchical decision approach was followed to locate the hot spots (i.e., classes in need of refactoring). That is, packages were first examined and then classes within interesting packages were further analyzed. We assume that classes within the same package can "tolerate" more coupling than is possible between packages. This is because the functionality of classes in the same package should be more cohesive. Following a similar process used at the package level, we further examined the metrics of classes inside these interesting packages (packages that were located by our package-level approach and had actually been refactored). This is a top down approach. This process aims to allocate resources (for code reviewing, for example) to the packages most in need of refactoring. The algorithm for this process will be discussed later (Fig. 2).

In software design, we prefer not to have strong coupling and high complexity. A number of metrics were used to measure coupling and complexity, and the metrics were categorized according to the level of code unit to which they were applied. We selected efferent coupling and afferent coupling as the package-level metrics. Efferent coupling of a package is the number of packages the package references; afferent coupling is the number of packages by which the package is referenced [24]. High efferent coupling means that a package has too much dependency; it is highly possible that changes in another package will have a ripple effect on this package. On the other hand, high afferent coupling within a package shows that many packages depend upon it. Change in this package could require updates in many other packages.

At the class level, we looked at size, complexity, coupling and cohesion such as WMC, CBO, RFC and LCOM metrics [13].

As mentioned earlier, a hierarchical approach was followed to locate classes of interest (Fig. 2). We first used JDepend [16] to obtain package-level metrics such as efferent coupling, afferent coupling, instability, etc.; then CKJM [65] were used to collect the class-level metrics. We developed a tool (JRIA) to integrate the two open source software tools.

The selection is based on the ranking of metrics instead of the metric value itself. For example, instead of using the coupling value as the criteria, we sorted the packages by efferent coupling and selected the packages with the highest ranks.

A threshold value can be determined based on expert estimate (subjective) and/or modeling estimation. Research reveals that model-based estimation is not always more precise than expert-based estimation. The imprecision of expert estimation can be improved by combining the estimation of multiple experts/approaches [49]. Expert-based weighting was used by other researchers in decision making regarding

cost/productivity estimation [43,67]. In this paper, we used expert-based estimation.

For the package level, we combined the search results of two coupling measures—efferent coupling and afferent coupling. For the class level, a weighted-based sum was used to combine the search results based on the ranks of multiple metrics.

Here, instead of using the rank itself, a normalized value is used. That is, for the package level,

$$\mathrm{NR} = \frac{\mathrm{Rank}}{\mathrm{PackageCount}}, \tag{3}$$

where NR means normalized rank and PackageCount stands for the total number of packages.

For example, suppose we use efferent coupling as the selection criteria, we can set 0.25 or 25% as the selection threshold. If there are 100 packages in the code, then the top 25 packages (with efferent coupling values greater than other packages) would be selected.

In addition, since both efferent coupling and afferent coupling were important aspects of dependency and maintenance cost, we tried to avoid type II errors since type I errors are relatively more tolerable. This is because it was assumed that we would rather examine more classes (even if some are false positives) than ignore classes that need refactoring.

NRE stands for formula (3) for efferent coupling or normalized rank for efferent coupling. TNRE is the threshold value of NRE; SPNRE (SP means selected packages) is the set of the N selected packages based on TNRE. Suppose that there are N packages, then,

$$\mathrm{SPNRE} = \{\text{Package } i \mid \mathrm{NREi} < \mathrm{TNRE}\}, \quad i = 1, \ldots, N. \tag{4}$$

Similarly, we define TNRA as the threshold value of NR on afferent coupling; SPNRA is the set of M selected packages based on TNRA. Then,

$$\mathrm{SPNRA} = \{\text{Package } i \mid \mathrm{NREi} < \mathrm{TNRA}\}, \quad i = 1, \ldots, M. \tag{5}$$

The selected packages are:

$$\begin{aligned} \mathrm{SPNR} &= \mathrm{SPNRE} \cup \mathrm{SPNRA} \\ &= \{\text{Package } i \mid \mathrm{NRA} < \mathrm{TNRA} \| \mathrm{NRE} < \mathrm{TNRE}\}, \\ &\quad i = 1, \ldots, K. \end{aligned} \tag{6}$$

Here, $k \geq M$, $K \geq N$. When $\mathrm{SPNRE} \cap \mathrm{SPNRA} \neq \emptyset$, then $K < M + N$. Note that TNRE and TNRA can be normalized by the number of packages.

$$\mathrm{NT_{NRE}} = \frac{T_{\mathrm{NRE}}}{\mathrm{PackageCount}} \tag{7}$$

Similarly, we can get NTNRA.

$$\mathrm{NT_{NRA}} = \frac{T_{\mathrm{NRA}}}{\mathrm{PackageCount}} \tag{8}$$

For the class level, we made the selection decision based on the ranking of multiple metrics. Here, we calculated a weighted sum for the ranking of each metric. Suppose there were M metrics, a class i ($i = 1 \ldots N$) had a rank of each metric j ($j = 1 \ldots P$). That is,

$$\mathrm{Grank}_i = \sum_{j=1}^{P} W_{ij} \times \mathrm{Rank}_{ij} \tag{9}$$

We selected CBO, RFC, LCOM and WMC to measure size, coupling and cohesion to represent different aspects that influence cost. We assumed that these metrics were similarly important. We tried not to include two metrics that capture the same information. For example, as NOC (number of classes), LOC (lines of code) and WMC all measure size, we used only one of them in the estimation. Therefore, we assigned a weight of 1 to WMC, 1 to RFC, 1 to CBO and 1 to LCOM, and 0 weight to other metrics. However, it was not our intention to provide a unified prediction model in the study. The optimization/validation of weights/thresholds for formulae (5, 9) is left for future work.

Further, we defined a metric called NGrank to normalize Grank:

$$\mathrm{NGrank} = \frac{\mathrm{Grank}}{\mathrm{ClassCountPKG}} \tag{10}$$

where ClassCountPKG stands for the total number of classes in the package. Since Grank $\leq$ ClassCountPKG, NGRank $\in [0, 1]$. The closer NGrank is to zero, the more need for refactoring is suggested by our approach. For the package level, we needed to get recall as high as possible, since a missing package can result in multiple missing refactoring target classes.

Then, the actual refactoring activities were examined: (1) logs of revisions regarding refactoring were extracted from the Subversion source repository, and (2) the files changed in the revisions were located. The actual changed code units were compared with the search results from the refactoring decision support tool.

There are a number of threats to validity in our study. A threat to internal validity is that the actual refactorings found in the Tomcat repository may not reflect the strategic type of planning that the refactoring decision support tool seeks to provide. Rather, these actual refactorings may only reflect opportunistic refactorings that were undertaken by very few (or maybe even just one) programmers. This threat, however, introduces bias 'against' JRIA. A threat to external validity is that only one project, open source, from one domain was used. This threat is accepted as Tomcat is a very large, popular, industrial project. Also, we had access

Table 8 The search result comparison by varying coefficients

NTNRE	NTNRA	Recall (package) (%)	Precision (package) (%)	Recall (class) (%)	Precision (class) (%)	Expected precision (package) (%)	Expected recall (%)	NGrank	Expected recall (class)	Expected recall (class)	Total classes	Expected correctly identified class
0.50	0.50	100	10.9	86.7	2.1	8.5	68	0.50	0.3255	0.0080	1,874	5
0.50	0.50	100	10.9	66.7	2.5	8.5	68	0.33	0.2113	0.0080	1,874	4
0.50	0.50	100	10.9	60.0	2.8	8.5	68	0.25	0.1697	0.0080	1,874	3
0.50	0.50	100	10.9	33.3	3.3	8.5	68	0.10	0.0800	0.0080	1,874	2
0.33	0.33	100	15.5	86.7	2.6	8.5	48	0.50	0.2636	0.0080	1,874	4
0.33	0.33	100	15.5	66.7	3.1	8.5	48	0.33	0.1718	0.0080	1,874	3
0.33	0.33	100	15.5	60.0	3.5	8.5	48	0.25	0.1371	0.0080	1,874	3
0.33	0.33	100	15.5	33.3	4.2	8.5	48	0.10	0.0630	0.0080	1,874	1
0.25	0.25	71.4	13.2	66.7	2.1	8.5	40	0.50	0.2455	0.0080	1,874	3
0.25	0.25	71.4	13.2	53.3	2.7	8.5	40	0.33	0.1600	0.0080	1,874	3
0.25	0.25	71.4	13.2	46.6	2.9	8.5	40	0.25	0.1275	0.0080	1,874	2
0.25	0.25	71.4	13.2	33.3	4.5	8.5	40	0.10	0.0582	0.0080	1,874	1

to 15 releases of this project, which were "refactored in the wild". A possible construct threat is the way we identify refactoring logs (by searching for the keyword "refactor"). This might introduce type I or type II errors. In our case, if a change log which was not refactor related was incorrectly classified as "refactored", a type I error occurs; if a refactor-related change log was classified as non-refactor, a type II error occurs. In the process of analyzing and minimizing this threat we found: (1) there was some valuable research regarding identifying refactoring activities according to metrics changes [50], but no mature commercial or open-source solutions ready for public use (to precisely identify or locate refactoring activities in source code) yet, and (2) the authors of Tomcat source code kept significantly precise logs regarding refactoring activities. Therefore, this approach of identifying refactoring activities in Tomcat source repository is acceptable in this context.

4.2.4 Analysis and discussion of results

Applying formulae (2)–(6) to the algorithm in Fig. 2, we can get a list of packages/classes that might need refactoring. Table 8 shows the result of the search by comparing the precision/recall of each combination.

To evaluate the effectiveness of the selection, we used two metrics: precision and recall. Precision is the percentage of actual matches to the total number of candidates found and recall is the percentage of actual matches found.

For example, the second row says that when we applied NTNRE 0.50 (second row, first column) and NTNRA 0.50 (second row, second column) to the package level and selected the first 50% of classes with the lowest GRANK (NGRANK = 0.50 or 50%), we got 100% recall (second row, third column) and 10.9% precision (second row, fourth column); at the same time, for the class level, the recall and precision were 86.7% (second row, fifth column), and 2.1% (second row, sixth column), respectively. When NTNRE, NTNRA and NGRANK are 0.33. 0.33 and 50, respectively, the search result achieved better performance with higher precision and recall (sixth row) for both package level (100% recall and 15.5% precision) and class level (86.7% recall and 2.6% precision). We would like to see higher recall and precision values at the same time. However, we favor recall when we cannot achieve satisfying results for both metrics at the same time. This is because we do not want to miss refactoring needs.

When the selection criteria for the package level got stricter (NTNRE and NTNRA drop to 0.25), class-level recall dropped to at most 2/3 ($\leq 66.7\%$).

To further confirm the usefulness of this approach, we compared the search result with that of a random approach. The precision and recall of a random approach can be calculated as follows. The number of successful candidates identified is a random variable that satisfies the following formula [62]:

$$P(X = x) = \frac{C_x^b C_{n-x}^r}{C_n^{b+r}}, \quad x = \max(0, n-r), \ldots, \min(n, b), \tag{11}$$

where n stands for the number of classes (or packages) selected, b for the number of refactoring candidates, and r for the classes (or packages) not identified as candidates. The mean value and variance are:

$$\mu = \frac{nb}{b+r}, \tag{12}$$

Table 9 Packages having files refactored in Tomcat 6.0.x as indicated by commit logs

Package name	NRE	*NRA*	NFG
org.apache.tomcat.uti.net	0.29787234	0.20212766	2
org.apache.coyote.http11	0.106382979	0.563829787	3
org.apache.catalina.valves	0.180851064	0.244680851	3
org.apache.catalina.core	0.010638298	0.074468085	4
org.apache.catalina.startup	0.042553191	0.138297872	1
org.apache.catalina.tribes.transport	0.074468085	0.244680851	7
j.org.apache.catalina.tribes.transport.bio	0.29787234	0.744680851	3
org.apache.catalina.tribes.transport.nio	0.29787234	0.276595745	3

$$\delta^2 = \frac{nbr(b+r-n)}{(b+r)^2(b+r-1)}, \tag{13}$$

respectively, and precision is

$$\text{Precision} = \frac{f}{n}, \tag{14}$$

and

$$\text{Recall} = \frac{f}{b}, \tag{15}$$

where f stands for the number of candidates found (found by our approach). Here, $f \leq n$ and $f \leq b$. Note that μ in formula (11), the expected number of found candidates, was used in formulae (13) and (14) as f to calculate the expected precision/recall (Table 8, column 7,8,10,11).

Paired t tests were performed on Table 9 to determine if our approach was more effective than a random selection. For both precision and recall for class and package levels, JRIA outperformed random search.

We first examined precision. For the class level, the mean increase (actual precision compared to expected precision) was 0.022 and standard deviation (SD) 0.0002 was statistically significantly greater than zero. The two-tail p value was 5.7×10^{-7} and the t value was 10.26. Similarly, for the package level, the mean increase was 0.047 with SD 0.00056. The two-tailed p value was 4.59×10^{-6} with $t = -8.3$.

We then looked at recall. For the class level, the mean increase was 0.041 with SD 0.0032 and was statistically significantly greater than zero. The two-tailed p value was 6.2×10^{-8} with $t = -12.7542$. For the package level, we found the mean increase (M 38.4%, SD 0.0289) was also statistically significantly greater than zero. The two-tailed p value was 3.93×10^{-8} with $t = -13.32$.

As can be seen, the paired t test showed that our approach was significantly better than random search. This rejects the hypothesis $H6_0$ and favors $H6_a$. At the same time, it rejects $H5_0$ and favors $H5_a$ as it is an evidence of agreement between the JRIA search and the true result.

Table 9 lists the packages with refactoring activities that were found in commit logs. In the table, NFG stands for the number of files changed in the refactoring activities. As mentioned earlier, NRE and NRA are between 0 and 1. The closer the value is to 0, the more is the need of refactoring relative to other packages. For example, on the fifth row in Table 9, it can bee seen that org.apache.catalina.core has very small NRE and NRA values. This shows that both its efferent and afferent coupling values are large compared to most of the other packages.

We further examined the characteristics of the refactored classes closely. As shown in Table 10, most of the refactored (12/15 or 80%) classes had NGrank above average (i.e., NGRank <0.5 or 50%). For example, in the second row, it is shown that class StandardWrapper (row 2, column 1) in package org.apache.catalina.core (row2, column 2) has NGRank 0.10 (row 2, column 5). In this row, we can also see that in the package there are 29 classes (ClassCountPKG) (row 2, column 3), the Grank of the package is 3 (row 2, column 4); the rank of the package for WMC, RFC, CBO and LCOM are 3, 2, 3 and 3, respectively (row 2, columns 6–9). Moreover, for these three classes that have higher NGRank, we found that all of them had abnormal coupling (CBO) ranks (3/29 or 3 out of 29 packages for StandardWapper; 14/35 for AprLifecycleListener; 1/5 for BioReplicationThread, and 2/5 for BioReceiver).

In summary, we found that there is overlap between the classes suggested by our refactoring support approach and those actually refactored. Therefore, we have evidence to support the rejection of $H5_0$ in favor of $H5_a$.

5 Related work

Related work can be categorized into the following fields: software maintenance effort prediction, software metrics, refactoring decision support and mining of software repositories. We will address each of these in turn.

5.1 Software maintenance effort prediction

There has been much research undertaken on software maintenance effort prediction based on size measured in actual number of or percentage of changes. Boehm et al. [8,9] provided a model in COCOMO 2 that can be used to predict the effort of modifying reusable software using two types of parameters: size measures and the percentage of modification. Note that Boehm used information (such as effort data) provided by numerous closed source projects to build his model. Antoniol et al. [4] estimated the change effort by predicting impacted classes based on analyzing the traceability

Table 10 Class metric rank of refactored class within its package

Class name	Package	ClassCountPKG	Grank	NGRank	Rank of WMC	Rank of RFC	Rank of CBO	Rank of LCOM
StandardWrapper	org.apache.catalina.core	29	3	0.10	3	2	3	3
ApplicationDispatcher	org.apache.catalina.core	35	12	0.34	16	8	4	13
AprLifecycleListener.	org.apache.catalina.core	35	24	0.69	24	21	14	22
Http11Processor	org.apache.coyote.http11	31	5	0.16	6	4	3	6
ErrorReportValve	org.apache.catalina.valves	17	8	0.47	9	5	3	8
AccessLogValve	org.apache.catalina.valves	17	3	0.18	2	2	5	2
ExtendedAccessLog Valve	org.apache.catalina.valves	17	1	0.06	1	1	3	1
WorkerThread	org/apache/catalina/tribes/transport	12	3	0.25	4	8	4	5
NioReplicationThread	org/apache/catalina/tribes/transport/nio	7	3	0.43	4	4	1	5
NioReceiver.java	org/apache/catalina/tribes/transport/nio/	7	1	0.14	2	1	3	3
BioReplicationThread.java	org/apache/catalina/tribes/transport/bio/	5	3	0.60	4	3	1	5
/BioReceiver.java	apache/catalina/tribes/transport/bio	5	3	0.60	3	2	2	2
ThreadPool.java	java/org/apache/catalina/tribes/transport	12	4	0.33	3	3	4	6
ReceiverBase.java	org/apache/catalina/tribes/transport	12	2	0.17	1	1	2	1
TldConfig	java.org.apache.catalina.startup	47	5	0.11	5	8	5	3

between change request and existing classes in the source code. Their prediction of effort is based on the estimation of change size.

Most of the effort models involve some measure of size as an important component of the equation. The size measure can either capture the size of the component being changed or the size of the changes. For example, Niessink and Vliet [53] found that the size of the component needing change rather than the size of the change itself had a larger impact on change effort. Ramil and Lehman [59] examined the predictive power of a set of size measures such as the number of modified modules in a case study. Hayes et al. [32] analyzed the relationship between effort and a set of candidate metrics and found that the number of changed operators performs well in predicting adaptive maintenance effort.

Software maintenance effort is influenced by multiple factors. Besides size measures, metrics regarding complexity, coupling, cohesion, etc., are believed to be related with maintenance effort. Some prediction approaches examine or use multiple measures. Briand et al. [12] provided a cost estimation model based on productivity and size estimation. In their approach, productivity was estimated based on the prediction of overhead cost, which captured the deviation in cost from an ideal project. They assumed a linear relationship between size and effort. Fioravani and Nesi [24] used complexity and size measures to build prediction models for adaptive maintenance effort for object-oriented systems.

Sneed et al. [61] considered both size and quality in estimating error correction cost; in estimating enhancements cost, they examined function points, complexity, quality, etc. Yu [71] studied the relationship between lag time (the time period between the submission and the closing of a maintenance task) and the maintenance effort. Their work was based on maintenance records instead of VCS repository data.

Boehm et al. [9,10] pointed out that algorithmic models are repeatable and objective. On the other hand, as they are calibrated or modeled from past experience, the model might not apply well to the future or projects with different environments or exceptional conditions. In addition, it is not clear how to choose the best model. Menzies et al. [49] provided an approach to compare and adjust estimation models. They formalized a number of rejection rules for comparing two models.

Previous research has also examined metrics that are available in the early phases of the software life cycle that can be used to predict software maintenance effort, such as function [2,3,53]. Often, these measures are used to build an empirical estimation model.

5.2 Maintainability prediction

Maintainability is a subjective metric that is defined as "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment [37]". As indicated by the definition, maintainability could impact or influence the maintenance effort. Similar to predicting maintenance effort, size, complexity, coupling, etc. have been used to evaluate maintainability.

Hayes and Burgess [30] used textual complexity measures to locate segments of code that are difficult to change and thus need additional documentation. In their tool Partially Automated In-line Documentation (PAID), the location of abnormal complexity is the focus. JRIA also checks for structures such as forward declarations and recursion.

Following Weyuker's software metrics evaluation criteria and focusing on class design, Chidamber and Kemerer [13] developed and empirically evaluated several object-oriented (OO) metrics, often referred to as the CK metrics suite. Metrics in the suite include WMC, DIT, NOC, CBO, RFC and LCOM. Chidamber and Kemerer analyzed the metrics from their own definition and evaluated them in experiments.

Li and Henry [45] were interested in those metrics that reveal the interconnections between software components. They hypothesized the relationship between maintainability and a set of object-oriented metrics, including five from the CK metrics suite and several of their own, such as MPC (message-passing coupling) and DAC (number of abstract data types defined in a class). They validated their hypotheses using data from two commercial systems. In their experiment, maintenance effort was measured as the number of lines changed per class.

Kemerer and Slaughter [39] realized that certain modules have a higher probability of needing changes than others and tried to find "predictable maintenance patterns"; they found that complexity, age and size are related to the maintainability of software modules.

Models were developed to provide quantitative evaluations of maintainability. Welker [68] suggests measuring software's maintainability using an MI. MI is a combination of multiple metrics, including Halstead metrics [28], McCabe's cyclomatic complexity [McCabe, 1989], lines of code and number of comments:

$$\begin{aligned}\text{Maintainability Index} = {} & 171 - 5.2 \times \ln(\text{aveV}) - 0.23 \\ & \times \text{aveV}(g') - 16.2 \times \ln(\text{aveLOC}) \\ & + 50 \times \sin(\text{sqrt}(2.4 \times \text{perCM}))\end{aligned}$$

where aveV is the average Halstead volume per module, aveV(g') is the average extended cyclomatic complexity per module, aveLOC is the average lines of code per module, and perCM is average percent of lines of comments per module.

Briand et al. [11] compared several frameworks for measuring coupling. Several aspects of coupling were identified, including the type of connection, direction, granularity, etc. Based on their research, it is important to distinguish different types of coupling; the selection of granularity depends on the goal of measuring instead of always using measurement at the lowest possible level. In addition, they found that the direction of coupling is very important for several aspects of maintainability such as understandability, error-proneness, etc.

Maintainability has also been studied in the context of software process in addition to analysis of static measures. Hayes [31] hypothesized that complexity increase with maintenance was the cause of code decay and used course projects to validate this. A number of metrics were investigated such as weighted method per class (WMC), cyclomatic complexity, system complexity, lack of cohesion in methods (LCOM), depth of inheritance tree (DIT), etc. Complexity was found to increase in the study.

Hayes et al. [32,33] measured "perceived maintainability" (PM) by asking the maintaining software engineer, after all the changes had been made, to assign a value from 1 to 10 to each component modified, where 10 indicated code that was very easy to change. They also calculated the maintainability product (MP) as the product of effort for the change and percentage of the program that was changed:

$$\text{MP} = \text{Change scope} \times \text{Effort} \times 100.$$

Maintenance effort is generally measured in person-hours. In formula (8), as a relative term, effort is evaluated on a scale of 0–1 as a percentage of the total change effort required for a given release. MP is measured on a scale of 0–100 where 0 is highly maintainable and 100 is highly un-maintainable.

Hayes and Zhao [34] verified that effort to implement changes correlated with maintainability and found the RDC ratio (the sum of requirement and design effort divided by code effort) to be a good predictor for maintainability using regression analysis on multiple student projects. They also found that using indirect measures can yield accurate prediction results.

Khoshgoftaar et al. [40] performed cost–benefit analysis of a classification model predicting fault-prone modules. In their analysis, cost and benefit were defined as the "direct costs of reliability enhancement of all the modules recommended by the model" and "cost-avoidance of maintenance-phase fixes for the fault-prone modules that are recommended by the model". The count of Type I and Type II errors were used in calculating these costs.

Offutt et al. [55] investigated the use of four types of coupling to measure software quality. In their studies of open-source projects, coupling and size of code had statistically significant correlation. Also, they found that the coupling

measure increases with version numbers, which agrees with early research.

5.3 Mining software repositories

Mining software repositories is an approach to learning the quality and/or other attributes of a software project by studying the source code and/or its evolution. Though not revealing all the project history (such as how much time a programmer spent on a maintenance task), a software repository provides an approximate approach for replaying the development and/or maintenance process.

Researchers found that repositories provide rich data that can help reveal information about the code evolution and software process. Ball et al. [5] pointed out that version control systems, such as CVS, can provide, as a by-product information about software development history. They illustrated this idea by exploring relationships between groups of classes in a compiler project. Draheim and Pekacki [19] proposed an approach that analyzes CVS repository data to gain knowledge about the software process, such as information regarding programmers' productivity, collaboration in the team and individual programmer's behavior characteristics.

Information from version control repositories has been used to predict software quality. For example, Fischer et al. [24] introduced their approach of combining information from version control systems and bug tracking data to provide more complete information about software evolution. Nikora and Munson [52] developed an approach for predicting the fault content of software using measures obtained from the software structural evolution. To obtain repeatable counts, they developed a standard for enumeration of faults and a framework that automates the measurement of those faults.

Zimmermann and Weisgerber [73] discussed four preprocessing tasks that make access to repository data faster. The tasks included data extracting, transaction restoring, mapping changes to entities and data cleaning.

5.4 Refactoring (decision support)

Refactoring is an approach to making a program easier to understand and change without altering its external behaviors (that is, no functionalities are added or removed). Opdyke [54] studied how refactoring can support the iterative design of object-oriented application frameworks. He identified a set of refactorings that people apply to object-oriented frameworks and introduced how to preserve the external behavior while refactoring under certain preconditions. He pointed out that applying arbitrary refactoring to a program is more likely to corrupt a system than to improve it, and that refactoring cannot be completely automated since refactoring tools cannot decide the type of refactoring to apply.

Fowler et al. [23] introduced refactoring as an approach to improve the design of existing code to make it more understandable and changeable, and suggested using a set of bad smells to decide when and where to apply refactoring. They illustrated some low-level refactorings (such as Move Method) as well as some more complicated refactorings (which consist of multiple low-level refactorings). He referred to these refactorings as "little refactorings". He also introduced "big refactorings", lacking the "little refactorings value" [Folwer et al. 1999], including predictability, visible progress and instant satisfaction.

Mens et al. [47] designed a tool used to detect places that need refactoring and decide which refactoring should be applied. They did so by detecting the existence of "bad smells". In their approach, programmers need to choose the entities that need analysis. In their experiments, they demonstrated the use of their tool to detect three kinds of "bad smells" including "inappropriate interface", "unused parameters" and "duplicated code." Their implementation is based on logic meta programming (LMP). Our approach differs from this in that we identify the entities that require refactoring.

Metrics have been used in refactoring support. Simon et al. [62] measured the similarity between class members to decide which attributes and methods should be put together. The similarity/distance metrics can be used to identify design abnormalities that disobey the cohesion principle and could therefore indicate entities needing refactoring. They used visualization to show the distance calculated between two class members. Our approach differs in that we examine other code characteristics besides cohesion to identify refactoring needs.

Studies also observe code metrics to reveal how refactoring changes the code attributes. Kataoka et al. [43] used a quantitative analysis to evaluate how certain refactoring(s) enhanced the maintainability of the target program. They mainly focused on coupling measures in their experiment. They computed three categories of coupling and combined them using a polynomial. They designed an experiment to observe how refactorings, such as Extract Method, changed the combined coupling metric. Nacer [52] conducted an empirical study using an object-oriented tool to refactor itself and observe the metrics change during the evolution. He found that the overall complexity of the code was reduced and the average number of methods per class was reduced. Du Bois et al. [20] analyzed how refactoring helped improve code by examining the coupling/cohesion metrics.

Research has been performed in the area of refactoring code into aspects too. For example, Binkley et al. [8] provided tool support for extracting the "crosscutting functionalities" into aspects to refactor the original code into aspect-oriented programming (AOP). Compared to our work, they focused on

the transformation instead of locating the refactoring opportunities.

Demeyer et al. [18] applied a series of heuristics based on change of metrics to identify refactoring activities in software repositories. They chose a small set of metrics of size, complexity and inheritance and several refactorings such as Splitting into Superclass. They analyzed false positive rates for each refactoring examined. Moser et al. [50] studied an approach to identifying refactoring activities in software repositories by using metrics. They applied widely used metrics such as lines of code, number of methods in a class, McCabe's cyclomatic complexity and aggregate values (such as average and max of a class). Instead of studying individual refactoring, they investigated "refactoring activity". They verified their approach using software repositories where refactoring activities are well documented. In our empirical study, we used the Tomcat repository where refactoring activities were well documented and the refactoring logs could be used as direct evidence of individual refactoring.

Our contribution is that we defined a metric ranking-based approach to support refactoring decision making and validated it empirically by comparing to human reviewers as well as by comparing to historical refactoring data for a very large software system.

6 Conclusions and future work

We have introduced an approach to identifying classes and packages that require refactoring based on static measures. This is useful to managers who are trying to allocate scarce resources on tight schedules and who want to perform some refactoring for improved future maintainability of code. Our approach provides a ranked or prioritized list of such classes and/or packages. We further proposed a hierarchical approach that starts from locating the packages with high coupling and then examines classes within these packages, mainly based on a weighted sum of the rank of multiple cost crucial measures.

We presented two case studies. In the first, a speech therapy project developed by graduate students was evaluated by our tool as well as by human reviewers. We found strong overlap between the classes identified by the human reviewers as most in need of refactoring and what JRIA recommended. In the second study, multiple releases of Apache Tomcat [6.0.x] were examined and refactoring that had actually been applied was mined. Then, JRIA was run twice on the source in the version control system. A hierarchical approach was followed to locate the packages and classes in need of refactoring, respectively. By varying the parameters in our approach (formulae 3–10, Table 8), optimized precision and recall can be obtained.

In our two studies, we found strong evidence in favor of tool-related hypotheses. The studies have shown encouraging results on: time savings when obtaining information to assist with refactoring decision making; similarity between reviewer or manager's identified refactoring targets and targets identified by JRIA; and reduction of the search space for human reviewers who decide what needs to be refactored and in what order.

The studies also provided answers to our research questions regarding the agreement between JRIA output and that generated by humans (RQ1). The rejection of $H4_0$ for $H4_a$ suggests that JRIA is faster than human reviewers (RQ2). We have shown that human reviewers missed some refactoring needs, but found others missed by JRIA (RQ4 and RQ5). This lends support that our approach is useful for managers performing decision making for refactoring, especially when deciding where to refactor and what to refactor first.

As future work, we plan to examine and/or automate weight selection for the weighted sum of rank. A learning algorithm can be used to achieve this goal. In addition, we can investigate previous versions of Tomcat such as Tomcat 5.x.x so that we can train/validate our model with more data. Also, we need to perform more studies using larger applications in multiple languages and domains. We also need to perform studies where a larger group of human reviewers examine applications.

Acknowledgments We thank the EDG Group for providing JFE for Study 1, the graduate student volunteers for participating in Study 1 and Dr. Stromberg for his suggestion on how to compare JRIA and human reviewer results.

References

1. Apache Tomcat. http://tomcat.apache.org
2. Albrecht, A.: Measuring application development productivity. In: Proceedings of the SHARE/GUIDE IBM applications development symposium, Monterey, CA, 14–17 October 1979
3. Ahn Y, Suh J, Kim S, Kim H (2003) The software maintenance project effort estimation model based on function points. J Softw Maint Res Pract 15(2):71–85
4. Antoniol G, Canfora G, Lucia A (1999) Estimating the Size of Changes for Evolving Object Oriented Systems: a case study. In: IEEE metrics, pp 250–258
5. Ball T, Kim J, Porter AA, Siy HP (1991) If your version control system could talk. In: ICSE workshop on process modeling and empirical studies of software engineering
6. Basili V, Briand L, Condon S, Kim Y-M, Melo WL, Valett JD (1996) Understanding and predicting the process of software maintenance releases. In: Proceedings of the 18th international conference on software engineering, Berlin, Germany, pp 464–474
7. Basili V, Lindvall M, Costa P (2001) Implementing the experience factory concepts as a set of experience bases. In: Proceedings of the SEKE 2001 conference, Buenos Aires, Argentina
8. Binkley D, Harman M (2006) Tool-supported refactoring of existing object-oriented code into aspects. IEEE Trans Softw Eng 32:9
9. Boehm B, Clark B, Horowitz E, Westland C, Madachy R, Selby R (1995) Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. Ann Softw Eng 1:57–94 (special volume)

10. Boehm B, Horowitz E, Madachy R Reifer D, Clark BK, Steece B, Brown AW, Chulani S, Abts C (2000) Software cost estimation with Cocomo II. Prentice-Hall, Englewood Cliffs
11. Briand L, Daly J, Wuest J (1999) A Unified framework for coupling measurement in object-oriented systems. IEEE Trans Softw Eng
12. Briand L, Emam Ei K, Bomarius F (1998) A hybrid method for software cost estimation and risk assessment. In: IEEE international conference on software engineering (ICSE), Osaka, Japan
13. Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493
14. Clarkware.com. http://www.clarkware.com/software/JDepend.html
15. Coupling, abstractness, stability—measuring and applying code metrics (2007) Retrieved 20 April 2010. http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/15/129265.aspx
16. Dagpinar M, Jahnke JH (2003) Predicting maintainability with object-oriented metrics—an empirical comparison. In: Proceedings of the 10th working conference on reverse engineering, November, pp 155–163
17. DeFee JM (1994) Integrating analysis complexity tool output with formal re-engineering estimation processes. In: Proceedings of the second annual McCabe users group conference. Baltimore, MD
18. Demeyer S, Ducasse S, Nierstrasz O (2000) Finding refactorings via change metrics. In: Proceedings of OOPSLA '2000 (international conference on Object-Oriented Programming Systems, languages and applications)
19. Draheim D, Pekacki L (2003) Process-centric analytical processing of version control data. In: IWPSE '03 proceedings of the 6th international workshop on principles of software evolution
20. Du Bois B, Demeyer S, Verelst J (2004) Refactoring—improving coupling and cohesion of existing code. In: Proceedings 11th working conference on reverse engineering, 8–12 November 2004, pp 144–151
21. Edison Design Group. JFE, a Front End for the Java Language. http://www.edg.com/jfe.html
22. Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: IEEE international conference on software maintenance
23. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (2001) Refactoring: improving the design of existing code. Addison-Wesley, Reading
24. Fioravanti F, Nesi P (2001) Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. IEEE Trans Softw Eng 27(12):1062–1084
25. Garrido A, Johnson R (2003) Refactoring C with conditional compilation. In: 18th IEEE international conference on automated software engineering
26. Glover A In pursuit of code quality: code quality for software architects. http://www.ibm.com/developerworks/java/library/j-cq04256/
27. Graves TL, Mockus A (1998) Inferring change effort from configuration management data. In: Metrics 98: fifth international symposium on software metrics, Nov 1998, Bethesda, MD, pp 267–273
28. Halstead, MH (1977) Elements of software science, operating, and programming systems series, vol 7. Elsevier, New York
29. Hassan A, Goseva-Popstojanova K, Ammar H (2005) UML based severity analysis methodology. In: 2005 annual reliability and maintainability symposium, Alexandria, VA
30. Hayes JH, Burgess C (1988) Partially automated in-line documentation (PAID): Design and implementation of a software maintenance tool. In: Proceedings of the 1988 IEEE conference on software maintenance, Phoenix, Az, pp 60–65
31. Hayes JH (2002) Energizing software engineering education through real-world projects as experimental studies. In: Proceedings of the 15th conference on software engineering education and training (CSEET), Covington, KY
32. Hayes JH, Mohamed N, Gao T (2002) The observe-mine-adopt model: an agile way to enhance software maintainability. J Softw Maint Evol
33. Hayes JH, Patel S, Zhao L (2004) A metrics-based software maintenance effort model. In: Proceedings of the 8th european conference on software maintenance and reengineering, Tampere, Finland, pp 254–258
34. Hayes JH, Zhao L (2005) Maintainability prediction: a regression analysis of measures of evolving systems. In: Proceedings of the 21th international conference on software maintenance, Budapest, Hungary, pp 601–604
35. Hayes J, Huffman, Dekhtyar A, Osborne J (2003) Improving requirements tracing via information retrieval. In: Proceedings of the RE 2003, pp 138–147
36. Høst M, Regnell B, Wohlin C (2000) Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. Empir Softw Eng 5(2):210–214
37. IEEE, ANSI/IEEE Std. 729-1983 (1990) IEEE standard glossary of software engineering terminology
38. Jorgensen M (2004) A review of studies on expert estimation of software development effort. J Syst Softw 70(1–2):37–60
39. Kemerer CF, Slaughter R (1999) An empirical approach to studying software evolution. IEEE Trans Softw Eng 25(4)
40. Khoshgoftaar T, Allen E, Jones W, Hudepohl J (2001) Cost–benefit analysis of software quality models. Softw Qual J 9(1)
41. Kitchenham B, Pickard L, Pfleeger S (1995) Case studies for method and tool evaluation. IEEE Softw 7:52–62
42. Knoernschild K (2006) Using metrics to help drive Agile software. Agile J. http://www.agilejournal.com/articles/the-agile-developer/using-metrics-to-help-drive-agile-software.html. Accessed 07 June 2006
43. Kataoka Y, Imai T, Andou H, Fukaya T (2002) A quantitative evaluation of maintainability enhancement by refactoring. In: ICSM 2002, pp 576–585
44. Stroggylos K, Spinellis D (2007) Refactoring—does it improve software quality? In: ICSE Workshops 2007. Proceedings of the fifth international workshop on software quality, 20–26 May 2007
45. Li W, Henry S (1993) Object-oriented metrics that predict maintainability. J Syst Softw 23(2):111–122
46. McCabe TJ, Watson, AH (1994) Software complexity. Crosstalk. J Def Softw Eng 7(12):5–9
47. Mens T, Tourwé T, Muñoz F (2003) Beyond the refactoring browser: advanced tool support for software refactoring. In: Proceedings of the international workshop on principles of software evolution, Helsinki, Finland, 2003
48. Mens T, Taentzer G, Runge O (2007) Analysing refactoring dependencies using graph transformation. Softw Syst Model 6:269–285
49. Menzies T, Chen Z, Hihn J, Lum K (2006) Selecting best practices for effort estimation. IEEE Trans Softw Eng
50. Moser R, Pedrycz W, Sillitti A, Succi G (2008) A model to identify refactoring effort during maintenance by mining source code repositories. In: PROFES 2008. LNCS, vol 5089, pp 360–370
51. Mukhopadhyay T, Kekre S (1992) Software effort models for early estimation of process control applications. IEEE Trans Softw Eng 18(10):915–924
52. Boudjlida N (1999) An experiment in refactoring an object oriented CASE tool. An experiment in refactoring an object oriented CASE tool. http://www.loria.fr/~nacer/PUBLI/Mcseai98.ps.gz
53. Niessink F, Van Vliet H (1997) Predicting maintenance effort with function Points. In: Proceedings of the international conference on software maintenance (ICSM '97), 01–03 Oct 1997, Bari, Italy

54. Nikora AP, Munson JC (2003) Developing fault predictors for evolving software systems. In: Proceedings of the ninth international on software metrics symposium, 3–5 Sept 2003, pp 338–350
55. Offutt J, Abdurazik P, Schach RS (2008) Quantitatively measuring object-oriented couplings. Softw Qual J 16(4):489–512
56. Opdyke WF (1992) Refactoring object-oriented frameworks, Ph.D. dissertation, Univerisity of Illinois at Urbana-Champaign
57. Patel V (2004) Building a static analysis tool for java programs, master's project report, University Of Kentucky, April 2004
58. Pressman RS (2001) Software engineering: a practitioner's approach, 5th edn. McGraw-Hill, NY
59. Ramil JF, Lehman MM (2000) Metrics of software evolution as effort predictors—a case study. In: Proceedings of the 16th IEEE international conference on software maintenance, San Jose, CA, USA, 2000, pp 163–172
60. Martin RC (2002) Agile software development principles, patterns, and practices. Prentice-Hall/Pearson Education Inc, Englewood Cliffs
61. Sneed HM, GmbH A, Austria V (2004) A cost model for software maintenance & evolution. In: Proceedings of the 20th IEEE international conference on software maintenance, pp 264–273, 11–14 Sept 2004
62. Simon F, Steinbrückner F, Lewerentz C (2001) Metrics-based refactoring. In: Fifth European conference on software maintenance and reengineering, 2001
63. (2008) Sourceforge.net. Metrics 1.3.6-Getting started. http://metrics.sourceforge.net/
64. Spiegel MR, Schiller JJ, Srinivasan AV (2002) Probability and statistics
65. Spinellis D ckjm—Chidamber and Kemerer java metrics. http://www.spinellis.gr/sw/ckjm/
66. McCabe TJ, Watson AH Software compleixty. http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp
67. Walston CE, Felix CP (1977) A method of programming measurement and estimation IBM Syst J 16(1)
68. Welker KD (2001) Software maintainability metrics model an improvement in the coleman. In: SCST Crosstalk, Aug 2001
69. Welker KD, Oman PW (1995) Software maintainability metrics models in practice. Crosstalk. J Def Softw Eng 8(11):19–23
70. Williamson ES (1993) Determination of redundancy using McCabe complexity metrics. In: Proceedings of the first annual McCabe users group conference, Baltimore, MD, 1993
71. Yu L (2004) Indirectly predicting the maintenance effort of open-source software. J Softw Maint Evol Res Pract 18(5):311–332
72. Zimmermann T, Weißgerber P (2004) Preprocessing CVS data for fine-grained analysis. In: Proceedings of the first international workshop on mining software repositories (MSR 2004), Edinburgh
73. Zhao L, Hayes J (2006) Predicting classes in need of refactoring: an application of static metrics. In: Proceedings of 2nd international PROMISE workshop, Philadelphia, 2006