Improving Bug Location Using Binary Class Relationships

Nasir Ali[1,2], Aminata Sabané[1,2], Yann-Gaël Guéhéneuc[1], and Giuliano Antoniol[2]
[1] *Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada*
[2] *SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada*
E-mail: {*nasir.ali,aminata.sabane,yann-gael.gueheneuc*}*@polymtl.ca,antoniol@ieee.org*

***Abstract*—Bug location assists developers in locating culprit source code that must be modified to fix a bug. Done manually, it requires intensive search activities with unpredictable costs of effort and time. Information retrieval (IR) techniques have been proven useful to speedup bug location in object-oriented programs. IR techniques compute the textual similarities between a bug report and the source code to provide a list of potential culprit classes to developers. They rank the list of classes in descending order of the likelihood of the classes to be related to the bug report. However, due to the low textual similarity between source code and bug reports, IR techniques may put a culprit class at the end of a ranked list, which forces developers to manually verify all non-culprit classes before finding the actual culprit class. Thus, even with IR techniques, developers are not saved from manual effort. In this paper, we conjecture that binary class relationships (BCRs) could improve the rankings by IR techniques of classes and, thus, help reducing developers' manual effort. We present an approach, LIBCROOS, that combines the results of any IR technique with BCRs gathered through source code analyses. We perform an empirical study on four programs—Jabref, Lucene, muCommander, and Rhino—to compare the accuracy, in terms of rankings, of LIBCROOS with two IR techniques: latent semantic indexing (LSI) and vector space model (VSM). The results of this empirical study show that LIBCROOS improves the rankings of both IR technique statistically when compared to LSI and VSM alone and, thus, may reduce the developers' effort.**

***Keywords*—Bug location, object-oriented, information retrieval, binary class relationships.**

I. INTRODUCTION

Preliminary to any software maintenance task, a developer must understand the source code. Due to incomplete/missing documentation, software evolution, and software aging, developers may have difficulties to comprehend the source code. Program comprehension-related activities are involved in 50% to 90% of all software maintenance activities [1]. Thus, these difficulties to understand source code increase maintenance costs.

A particular task during which these difficulties slow down developers is that of bug location. To fix a bug, developers must locate the bug in the source code of a program: they must identify the classes (and–or parts thereof) describing the unexpected behavior of the program. Therefore, they must understand the source code enough to identify the culprit classes and modify them to fix the bug. Effectively automating this task could reduce maintenance costs by reducing developers' effort [2].

There exist several semi-automated bug-location techniques (BLTs) for object-oriented programs [4], [5], [6], [7]. These techniques typically analyse a bug report and the source code of a program to report a list of the classes ordered by their likelihood to be related to the bug. These techniques mainly divide into three sets: static, dynamic, and hybrid. To locate a bug, static BLTs [4], [5], [8] use textual information extracted using static analyses on the program source code whereas dynamic BLTs [7] use textual information extracted from the execution traces of a program. Hybrid BLTs [6] use both static and dynamic analyses. In the following, we consider only static BLTs because some bugs may prevent a program to compile or it may not be possible to retrieve execution traces from a program because it is not yet completed. Therefore, static BLTs have advantages over dynamic ones because they do not require a compilable program and can be applied at any stage of its development or maintenance.

Static BLTs often use Information Retrieval (IR) techniques [2], [6] to link bug reports to classes. IR-based techniques, in particular the commonly-used Latent Semantic Indexing (LSI) [9] and Vector Space Model (VSM) [4], have proven useful for bug location [2]. An IR technique takes as input some text extracted from a program class through static analyses. Then, it computes the textual similarity between the class and the bug report. A high textual similarity means that the class and bug report share several concepts [4], *i.e.*, they are likely to be related to one another. Developers should consider classes highly-similar to a bug report first during their bug location task because they are more likely related to the bug and, hence, cause of the bug.

However, due to the often low similarity between the textual information extracted from source code and bug reports, IR techniques may have poor accuracy, in particular IR techniques may put the classes most related to a bug—from the developers' point of view—very low in the ranked list. Consequently, many researchers proposed to use additional information to improve the accuracy of IR techniques in general [6], [10], and for bug location in particular [2]. They conjectured that other information combined with IR techniques could be used to re-rank the classes for each bug to bring the most relevant classes higher in the ranked list.

Yet, to the best of our knowledge, no previous authors

considered binary-class relationships (BCRs). BCRs include use, association, aggregation, composition, and inheritance. We conjecture that the existence of BCRs among classes is a useful information to re-rank the list of culprit classes. Our conjecture stems from the observations that developers express the design and implementation of their programs in terms of classes and the BCRs among these. For example, the classes playing a role in implementing the “find/replace” feature in a program would probably be related through some BCRs. We therefore study the impact of using BCRs to improve the ranking of classes to help developers during their bug-location tasks. BCRs are extracted from source code using static analyses. We exclude from our study composition BCR that, in general, require dynamic analyses.

To test our conjecture, we propose LIBCROOS, an approach that uses LInguistic (textual) and BCRs of Object Oriented Programs, to improve the accuracy of IR techniques. In LIBCROOS, an IR technique creates a set of so-called baseline links between a bug report and the classes of a program. Then, each BCR among the classes acts like an expert to vote on the baseline links. The higher the number of experts voting [6], [10] for a baseline link, the higher the confidence in the link, and the higher LIBCROOS puts that class in its ranked list. Thus, LIBCROOS is a complementary approach to any IR technique, which is, to the best of our knowledge, the first approach to use BCRs as experts to vote on the links between bug reports and classes.

We perform an empirical study to compare LIBCROOS with two IR techniques alone: LSI and VSM. We evaluate the effectiveness of our proposed approach on four programs—Jabref, Lucene, muCommander, and Rhino. Our results show that LIBCROOS can statistically improve the accuracy of the two IR techniques. Furthermore, we analyse each BCR separately; we observe that the inheritance and aggregation relationships help to improve the accuracy of the two IR techniques more than the other relationships. Results of our empirical study suggests that developers must use more relationships among classes contributing to a same feature.

This paper is organised as follows. Section II provides a brief description of the state-of-the-art BLTs. Section III describes proposed approach in details and sketches our implementation. Section IV provides details on our empirical study. Section V reports the results, discussions, and threats to the validity of our findings. Finally, Section VI concludes with future work.

II. Related work

Information retrieval (IR) techniques, bug, concept, feature location as well as binary class relationships are related to our research work.

Bug or feature location is a search activity, whether a developer searches the source code to find the classes that are playing a role to implement a feature or cause a bug. In the search results, developers tend to look at the top few results only [6]. Many researchers have proposed automated and semi-automated approaches to facilitate developers to locate a feature or a bug. All the proposed approaches could be divided into three categories, dynamic, static, and hybrid. Static approaches have a benefit over dynamic and hybrid approaches that they do not require compilable program. Static analyses [8], execution traces [11], and IR techniques [4], [5] have been used by researchers since the early works on feature location [11]. Often, IR techniques [4], [5], [6], use vector space models, probabilistic rankings, or a vector space model transformed using latent semantic indexing. Whenever available dynamic data [11], [6] proved to be complementary and useful for traceability recovery by reducing the search space. Recently, high-level documentation was mapped into source code features using a variety of information sources [6].

Poshyvanyk et al. [6] formulated the feature location problem as combination of the opinions of different experts. They used a scenario-based probabilistic ranking of event and an IR technique as experts to locate features in the source code. Marcus et al. [5] proposed a LSI-based approach to locate features in the source code. Their approach allows developers to formulate queries in natural language and results are returned as a list of source code elements ranked by the relevance to the query. Ali et al. [12] proposed COPARVO to recover traceability links between object oriented programs and requirements. Authors partitioned source code in four parts, in particular class, method, variable name, and comments. Each source code part then voted on recovered link by VSM. Their results show that all source code partitions have their own importance in RT.

Robillard [8] proposed an approach that analyses the topology of structural dependencies in a software system in order to propose relevant program elements for the developer to investigate. It takes as input a set of program elements of interest to a developer and produces a fuzzy set describing other elements of potential interest. Shao and Smith [13] combined IR and static control flow information for feature location. They used LSI to rank all the methods in a software system by their relevance to a query. Then, for each method in the ranked list, a call graph is constructed and assigned a call graph score. The call graph counts the method’s direct neighbours that also appeared in LSI ranked list. Finally, they combined the LSI cosine similarity and call graph score to produce a new ranked list. Zhao et al. [14] proposed SNIAFL, a static, non-interactive approach to feature location, technique. SNIAFL combines IR techniques with a branch-reserving call graph (BRCG) for feature location. They used VSM to generate an initial set of methods related to a feature, and then they use BRCG to find more relevant methods for the feature.

Object Oriented Programming is a method of implementation in which programs are organised as a collection of

collaborative objects [15]. As the real world entities, classes in object oriented programs do not exist in isolation; they cooperate through the BCRs between them. The main BCRs are inheritance, association, aggregation, and using (use relation)[15]. In our study, we will consider these four BCRs. In object oriented programs, relationships are as important as the objects themselves [16]. Class relationships allow classes to share data, to define more complex structures or to participate in the implementation of a program feature[15], [17]. Therefore, classes involved in the implementation of a feature (program behaviour) are probably linked by class relationships. Program behaviour that deviates from its specification is called a bug[3]. Thus, to locate relevant classes involved in the occurrence of a bug is similar to locating classes involved in the implementation of the feature that has not been correctly implemented. Based on this observation, we believe that using information about BCRs to locate a bug can be helpful.

Although class relationships are essential in the implementation of features and for program comprehension tasks, they are not all explicit in the source code [18], [19]. It is not an obvious task to recover class relationships in source code. Indeed, many researchers propose various approaches [20], [18] to extract class relationships in the source code. Part of our approach is based on the approach proposed by Guéhéneuc [18]. In this approach, authors formalised BCRs based on 4 independent-language properties: exclusivity, receiver type, life-time and the number of instances. Using these properties and specific algorithms, they were able to recover class relationships in the source code. They provided this technique in the ptidej tool suite[1] [21].

To the best of our knowledge, none of previous work performed experiment to analyse what are important BCRs, in particular at class-level, for bug location. In addition, how BCRs could be combined with IR techniques to improve the accuracy, in terms of ranking, for bug location. The work presented in this paper is complementary to the existing IR BLTs, because it exploits the BCRs of object-oriented programs to improve the accuracy of IR techniques.

III. LIBCROOS

We now present the details of our approach for bug location using BCRs, which uses textual information and BCRs extracted from the source code to link classes to bug reports. Figure 1 shows the high-level view of LIBCROOS. It has three main modules, *i.e.*, an IR engine, a relation model, and a ranker. We give some details about the abstract model of LIBCROOS and then explain each module.

A. LIBCROOS Abstract Model

Let $B = \{b_1, \ldots, b_N\}$ be a set of bug reports, $C = \{c_1, \ldots, c_M\}$ be a set of classes, and $\mathcal{R} = \{r_1, \ldots, r_P\}$ be a set of BCRs.

[1]http://www.ptidej.net/download

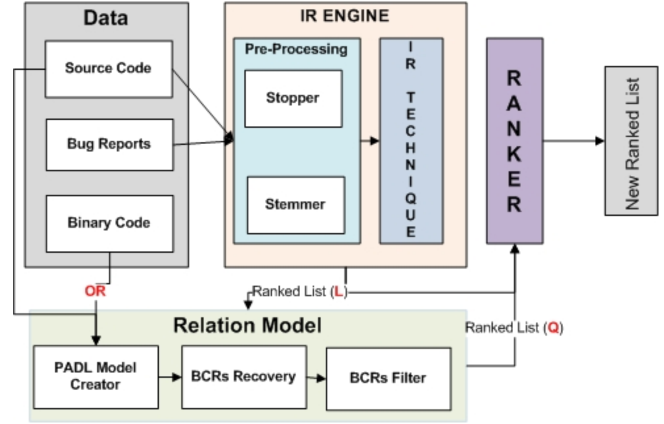

Figure 1. High-level Diagram of LIBCROOS

Let $\mathcal{L} = \{L_1, \ldots, L_N\}$ be a set where each L_i is a set of classes $\{c_1, \ldots, c_j\}$ linked to a bug b_i; whose cosine similarity is greater than 0. Let $Q = \{Q_{1,1}, \ldots, Q_{N,P}\}$ be a set in which each $Q_{i,k}$ is a subset of L_i, in which all classes are linked by r_k. Q is produced by the relation model.

Finally, let us define four functions α, β, δ, and γ: the first function, $\alpha(b_i, c_j)$, returns the similarity score $\sigma_{i,j}$ between a class c_j and a bug report b_i computed by the IR engine. The function $\beta(b_i)$ returns the set L_i of classes linked to a bug b_i. The function $\delta(L_i, r_k)$ returns the set $Q_{i,k}$, and the function $\gamma(c_j, L_i, r_k)$ returns 1 if $c_j \in Q_{i,k}$ and 0 otherwise.

We also define $\psi_{i,j}$, a function that computes the final similarity between a class c_j and a bug b_i by combining the vote of each BCR, *i.e.*, inheritance, association, aggregation, and use relation, and IR technique, as:

$$\psi_{i,j} = \lambda \frac{\sum_{k=1}^{|R|} \gamma(c_j, L_i, r_k)}{|R|} + (1 - \lambda)\, \sigma_{i,j} \qquad (1)$$

where $\lambda \in [0, 1]$, represents the weight of the IR technique. The higher the evidence, *i.e.*, number of BCRs ($\sum \gamma(c_j, L_i, r_k)$), the higher the new similarity $\psi_{i,j}$. In the contrary, little evidence decreases $\psi_{i,j}$ relatively to the similarities of other $c_j \in L_i$. LIBCROOS model is thus similar to the Trustrace model [10], the higher the evidence for a link, the higher the similarity value of the link.

B. IR Engine

LIBCROOS uses some IR techniques as an engine to create links between bug reports B and classes C. LIBCROOS is not dependent on a particular IR technique, any IR technique could be used with LIBCROOS. IR techniques consider both bug reports and classes as textual documents. For source code, we use a parser, *e.g.*, a Java parser, to extract all source code identifiers. The parser discard extra information, *e.g.*, data types, from the source code [12]. IR techniques extract all the terms from the documents and compute the similarity between two documents based on the similarity of their terms and–or the distributions thereof.

With any IR technique, a high similarity value between two documents suggests a potential link between them.

IR techniques take some pre-processed documents, as explained in the following, as input to build a $m \times n$ term-by-document matrix, where m is the number of all unique terms that occur in the documents and n is the number of documents in the corpus. Then, each cell of the matrix contains a value $w_{p,s}$, which represents the weight of the pth term in s^{th} document. A weight represents the importance of a term in the corpus of all terms. Various term weighting schemes are available to compute the weight of a term [4], [22]. In this paper, we use the commonly-used TF/IDF [4] weighting scheme.

The similarity between two documents is measured by the cosine of the angle between their corresponding vectors of terms' weight. Cosine values are in $[-1, 1]$ but negative values are discarded and a link has thus a value in $]0, 1]$, because similarity cannot be negative between two documents and 0 similarity means that two documents would not share *any* textual information. Different IR techniques [4], [22], [5] can be used to compute the similarity between bug reports and source code files. In this paper, we use the IR techniques LSI and VSM. Thus, LIBCROOS IR engine generates a set $\mathcal{L}$.

C. Relation Model

The relation model is the part of LIBCROOS that provides the BCRs between classes. In LIBCROOS, each BCR is treated as an expert that votes on the links recovered by the IR engine. The relation model takes as input the set $\mathcal{L}$ provided by the IR engine and the source code or binary code of the program. It produces as output the set Q using analyses based on models of the source code [18].

Step 1: PADL Model Creation (PADL Model Creator): We use the Ptidej tool suite[21] and its PADL meta-model (Patterns and Abstract-level Description Language) to build PADL models of object-oriented programs. A PADL model is a representation of a program compliant with the PADL meta-model. Such model includes the main constituents that represent the structure and part of the behaviour of a program, *i.e.*,ie classes, interfaces, member classes and interfaces, methods, attributes, inheritance.

The Ptidej tool takes as input the C++ or Java source code or binaries of programs and generates PADL models of these programs. The Ptidej tool suite essentially divides into a set of parsers, an implementation of the PADL meta-model, and language-dependent extensions to the meta-model to integrate, within a PADL model, constituents particular to a programming language. For example, the PADL meta-model does not define a constituent to describe C++ destructors, this constituent is provided along with the C++ parser to allow the modelling of C++ programs with possible highest precision. The C++ and Java source code parsers are based on the parsers provided by the CDT and JDT Eclipse plug-ins. The Java binary code parser uses the BCEL library.

Step 2: BCRs among classes recovering (BCRs Recovery): Using PADL and based on an extensive literature review [18], we implemented analyses to uncover BCRs in the source code of programs and make them explicit in their PADL models.

Theoretically, we assume that a BCR exists between two classes if any method of one of the two classes invokes at least one method of the other. Then, we define four properties of any potential BCR but inheritance: we exclude inheritance because it is explicit in the source code of programs in C++ (through the `:` syntax) and Java (through the `extends` keyword). We need dedicated analyses to recover all BCRs but inheritance because, in mainstream programming languages, such as C++ and Java, these relationships are not explicit in the source code but implemented by developers from the design documents using various idioms.

These properties are: (1) exclusivity of the participation of the instances of the classes involved in the BCR, (2) the types of the receivers of the messages exchanged, (3) the life-time of the instances of the classes involved in the BCR, and (4) the multiplicity of the instances of the classes involved in the BCR. We use these properties to define uniquely each BCR, from the least constraining in terms of the values of the properties to the most constraining: use, association, aggregation, and composition.

We do not recall here the sets of values for each properties for lack of space and because the reader may find all details in a previous work [18]. Also, we do not further consider composition because it requires dynamic information that would either be gathered through dynamic analyses or through incomplete static analyses. These properties and their values essentially allow identifying the various idioms used by developers to implement BCRs.

When defining the properties of any BCR (but inheritance and composition), we make sure that we can identify these values of the properties using PADL constituents, in particular: classes, methods, and fields, and method invocations between methods. Thus, our analyses mainly consist in identifying potential BCR among classes and then refining these candidates using the values of their properties. These analyses are source code analyses, because they use essentially information extracted from the source code. However, we abstract these analyses to make them operate on PADL models so that we can recover BCRs from various programming languages.

When applying these analyses on a PADL model, we obtain a new PADL model that contains all the constituents from the original model plus constituents representing explicitly the BCRs: instances of the Use, Association, Aggregation, and Implementation constituents of an extension to the PADL meta-model.

Step 3: Linked Classes Extraction (BCRs Filter): Based on the model built in Step 1 and refined in Step 2 and the set $\mathcal{L}$ provided by the IR engine, we build the set Q. For each BCR r_k of R, we iterate over each L_i of $\mathcal{L}$. If the PADL model of the program indicates for a class c_j of L_i a BCR r_k between c_j and another class c_l of L_i, then c_j and c_l are selected as elements of $Q_{i,k}$. At the end, of this step, we associate to each bug b_i four sets:

- $Q_{i,1}$, the classes linked by an inheritance relationship and linked to the bug b_i;
- $Q_{i,2}$, the classes linked by a use relationship and linked to the bug b_i;
- $Q_{i,3}$, the classes linked by an association and linked to the bug b_i;
- and, $Q_{i,4}$, the classes linked by an aggregation and linked to the bug b_i.

D. Ranker

The ranker assigns weights to different BCRs and similarity computed by IR engine to re-rank the classes linked to a bug. The ranker takes the set $\mathcal{L}$ generated by the IR engine and the set Q generated by the relation model as input. It uses the function $\gamma(c_j, L_i, r_k)$ to get the total number of BCRs for each c_j. For example, if a Bug_1 is linked to a class c_1 and the relation model indicates that $c_1 \in Q_{1,1}$, *i.e.*, there exist an inheritance relationship between c_1 and another class of L_1 while $c_1 \notin Q_{1,2}$, $c_1 \notin Q_{1,3}$, and $c_1 \in Q_{1,4}$, then the ranker observes that the total number of relationships found for c_1 is 2, *i.e.*, $\sum_{k=1}^{4} \gamma(c_j, L_i, r_k) = 2$. Then, the ranker assigns weights to the similarities $\sigma_{i,j}$ computed by the IR engine and $\gamma(c_j, L_i, r_k)$ provided by the relation model. Based on Equation 1, the ranker computes the final similarity $\psi_{i,j}$ for each link and then re-ranks the classes linked to a bug b_i based on their similarities.

IV. Empirical Study

We perform an empirical study on four programs and with two state-of-the-art IR techniques to assess the accuracy of our proposed approach for bug location. This study provides data to assess the accuracy, in terms of ranking, of the improvement brought by LIBCROOS over two "traditional" IR techniques, using LSI and VSM alone.

The *goal* of our empirical study is to evaluate the effectiveness of our novel approach for bug location against traditional LSI and VSM-based approaches. In addition, we analyse which BCR helps to improve IR technique accuracy the most. The *quality focus* is the ability of LIBCROOS to link a bug report to appropriate classes in the source code in terms of ranking [6]. The *perspective* is that of practitioners and researchers interested in locating bugs in source code with greater accuracy than that of currently-available bug-location approaches based on IR techniques. The *objects* of our empirical study are four open-source programs, Jabref[2], Lucene[3], muCommander[4], and Rhino[5].

A. Research Questions, Hypothesis, and Variables

The research questions that our empirical addresses are:

***RQ1*:** Does LIBCROOS provide better accuracy, in terms of ranking, than IR techniques?

***RQ2*:** What are the important BCRs that help to improve IR techniques accuracy more than the others?

To answer our research questions, we perform four experiments on Jabref, Lucene, muCommander, and Rhino using LIBCROOS, LSI, and VSM. We use a measure of ranking [6] to measure the accuracy of proposed and IR techniques to answer our research questions. LSI and VSM return ranked lists of classes for each bug in descending order of the textual similarities between the bug report and the classes. LIBCROOS returns a similar ranked list in descending order of the similarities computed using Equation 1. If a culprit class is lower (has a higher rank) in the ranked list then a developer must spend more effort to reach this actual culprit class because she must assess more candidate classes to solve the bug. Thus, the higher is a culprit class in a ranked list (decreased rank), the less is the developers' effort and the more accurate is an approach.

For RQ1, we consider the four BCRs, *i.e.*, use, association, aggregation, and inheritance, to analyse how much LIBCROOS can decrease the rank of classes to put culprit classes closer to the top of the list and, hence, can decrease a developer's effort. Consequently, we apply LIBCROOS, LSI, and VSM approaches on the four programs seeking to reject the two null hypotheses:

H_{01}**:** There is no statistical difference in terms of ranking between LIBCROOS and VSM.

H_{02}**:** There is no statistical difference in terms of ranking between LIBCROOS and LSI.

For RQ2, we use LIBCROOS with only one BCR at a time to observe which relationship helps more than the other to put the culprit classes at the top. Consequently, we apply LIBCROOS, LSI, and VSM approaches on the four programs seeking to reject the two null hypotheses:

H_{03}**:** All the BCRs equally improve accuracy, in terms of ranking, over VSM.

H_{04}**:** All the BCRs equally improve accuracy, in terms of ranking, over LSI.

We use the approaches, either a LIBCROOS, LSI, or VSM, as independent variables and the rankings of the approaches as dependent variables to empirically attempt rejecting the null hypotheses.

[2]http://jabref.sourceforge.net/
[3]http://lucene.apache.org/core/
[4]http://www.mucommander.com/
[5]http://www.mozilla.org/rhino

B. Objects

We select the four open-source programs, Jabref, Lucene, muCommander, and Rhino because they satisfy several criteria. First, we select open-source programs, so that other researchers can replicate our experiments. Second, we avoid small programs that do not represent programs handled by most developers. Third, three of the programs have been used in previous studies by other researchers [23], [24] for possible comparisons. Finally, three programs come with independent, manually-built oracles, which mitigates some of the threats to the validity of our results. Only for Lucene, we recovered links between bugs and source code.

JabRef is an open-source bibliography reference manager. The native file format used by JabRef is BibTeX, the standard LaTeX bibliography format. JabRef runs on the Java VM (version 1.5 or newer), and should work equally well on Windows, Linux and Mac OS X. Jabref version 2.6 has 579 classes, $287,791$ LOC, and 36 bug reports. There are 108 manually built links between bug reports and classes.

Lucene is an open-source high-performance text-based search-engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Lucene version 3.1 has 434 classes, $111,117$ LOC, and 89 bug reports. We select Lucene $v3.1$ because it contains more closed bugs than other versions and these were linked to its SVN repository. There are 235 manually built links between bug reports and classes.

muCommander is a lightweight, cross-platform file manager with a dual-pane interface. It runs on any operating system with Java support. It supports 23 international languages. muCommander supports virtual filesystem as well as local volumes and the FTP, SFTP, SMB, NFS, HTTP, Amazon S3, Hadoop HDFS, and Bonjour protocols. muCommander version $0.8.5$ has $1,069$ classes, $124,944$ LOC, and 81 bug reports. There are 231 manually built links between bug reports and classes.

Rhino is an open-source JavaScript engine entirely developed in Java. Rhino converts JavaScript scripts into objects before interpreting them. It works in compiled as well as in interpreted modes. It is intended to be used in server-side systems. Rhino can be used as a debugger by making use of the Rhino shell. Rhino version $1.5R4.1$ has 111 classes, $94,078$ LOC, and 41 bug reports. There are 92 manually built links between bug reports and classes.

C. Preprocessing Documents

We now detail how we prepare the input data necessary to answer our research questions.

Generating Corpora: We download the source code of Jabref v2.6, Lucene v3.1, muCommander v0.8.5, and Rhino v1.5R4.1 from their respective CVS/SVN repositories. We generate corpora of all the programs for IR techniques to link bug reports to classes. To generate corpora, we extract source code identifiers using a dedicated Java parser [12] to extract all source-code identifiers. The Java parser builds an abstract syntax tree (AST) of the source code that can be queried to extract required identifiers, *e.g.*, class, method names, and so on. Only for Lucene, we download bug reports from the JIRA bug repository. For all the other programs, we used the bug reports provided by the other researchers [23], [24]. We only use long description of bug reports.

Oracles: We use previously built oracles to verify if a class linked to a bug is true positive or false positive link. Some researchers [23], [24] manually and–or semi-automatically created links between bug reports and methods to create oracles. In this paper, we link bug reports to classes, thus we converted the oracles at class level. In the case of Lucene, we download bug reports and SVN logs from JIRA repository. Developers usually write bug IDs in SVN logs [25] when they fix a bug. We automatically extract bug IDs from SVN logs to link bug reports to the classes.

Preprocessing the Corpora: We remove non-alphabetical characters from the terms gathered from the source code and the bugs and then use the classic Camel Case and under-score algorithms to split identifiers into terms. Then, we perform the following steps to normalise bug reports and source code: (1) convert all upper-case letters into lower-case and remove punctuation; (2) remove all stop words (such as articles, numbers, and so on); and, (3) perform word stemming using the Porter Stemmer bringing back inflected forms to their morphemes.

D. Linking Bugs Reports and Classes using VSM

We use VSM [4] to index the corpora generated in the previous step. For each bug report, VSM generates a ranked list of classes. For example, if there are 10 bugs then there would be ten ranked lists. In VSM, bug reports and classes are viewed as vectors of terms. Different term weighting schemes can be used to construct these vectors. The most popular scheme is TF/IDF. Term frequency (TF) is described by a $t \times d$ matrix, where t is the number of terms and d is the number of documents in the corpus. TF is often called local weight. The most frequent term will have more weight in TF but it does not mean that it is an important term. The inverse document frequency (IDF) of a term is calculated to measure the global weight of a term: $(TF/IDF)_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \times \log_2 \left(\frac{|D|}{d:|t_i \in d|} \right)$, where $n_{i,j}$ is the occurrences of term t_i in document d_j, $\sum_k n_{k,j}$ is the sum of occurrences of all terms in document d_j, $|D|$ is the total number of documents in the collection, and $|d : t_i \in d|$ is the number of documents in which the term t_i appears. Once all bug reports and source code documents have been represented in the vectors, we compute the similarities between bug reports and classes to link them.

E. Linking Bugs Reports and Classes using LSI

Latent Semantic Indexing (LSI) is an information retrieval technique based on the VSM. For each bug report, LSI

		LSI vs. LIBCROOS						VSM vs. LIBCROOS					
		Mean		Median		SD		Mean		Median		SD	
	Relationship	LSI	LIBC.	LSI	LIBC.	LSI	LIBC.	VSM	LIBC.	VSM	LIBC.	VSM	LIBC.
Jabref	ALL	22.19	5.47	8	2.5	32.86	7.59	16.14	2.5	6.5	2	23.30	2.09
	Aggregation	22.19	14.86	8	6	32.86	20.64	16.14	7.97	6.5	4	23.30	9.93
	Association	22.19	19.92	8	8	32.86	28.47	16.14	13.5	6.5	6.5	23.30	18.77
	Inheritance	22.19	10.72	8	4	32.86	15.33	16.14	5.75	6.5	3	23.30	7.59
	Use	22.19	19.56	8	8	32.86	27.997	16.14	12.97	6.5	6.5	23.30	18.35
Lucene	ALL	26.22	9.66	8	4	43.11	15.16	24.90	9.40	6	2	46.08	16.66
	Aggregation	26.22	16.42	8	6	43.11	25.47	24.90	15.47	6	5	46.08	28.74
	Association	26.22	24.90	8	7	43.11	40.50	24.90	23.88	6	6	46.08	43.57
	Inheritance	26.22	15.85	8	5	43.11	25.45	24.90	14.87	6	3.5	46.08	25.80
	Use	26.22	23.16	8	7	43.11	37.21	24.90	22.12	6	6	46.08	40.18
muCommander	ALL	36.72	11.59	7.5	3.5	89.33	24.96	39.88	13.23	9.5	3.5	105.20	32.01
	Aggregation	36.72	19.54	7.5	4.5	89.33	43.76	39.88	20.37	9.5	5.5	105.20	50.87
	Association	36.72	31.35	7.5	6.5	89.33	73.59	39.88	34.05	9.5	7.5	105.20	87.69
	Inheritance	36.72	20.08	7.5	5.5	89.33	46.26	39.88	22.21	9.5	5	105.20	55.64
	Use	36.72	29.94	7.5	6.5	89.33	69.59	39.88	32.69	9.5	8	105.20	83.66
Rhino	ALL	11.34	3.25	4.5	1	17.13	4.36	9.47	2.78	3	1	16.45	4.38
	Aggregation	11.34	6.97	4.5	3	17.13	9.55	9.47	5.47	3	2	16.45	8.76
	Association	11.34	9.41	4.5	3.5	17.13	14.66	9.47	7.72	3	2.5	16.45	13.87
	Inheritance	11.34	6.31	4.5	2.5	17.13	9.61	9.47	4.5	3	2	16.45	8.02
	Use	11.34	9.19	4.5	3.5	17.13	14.53	9.47	7.72	3	2.5	16.45	13.54

Table I
DESCRIPTIVE STATISTICS LIBCROOS, LSI, AND VSM. LIBC. AND SD REPRESENT LIBCROOS AND STANDARD DEVIATION RESPECTIVELY

generates a ranked list of classes. LSI assumes that there is an underlying or latent structure in word usage for every document set in corpus [5]. The processed corpus is transformed into a term-by-document matrix. The matrix is then decomposed using *Singular Value Decomposition (SVD)* [5] to derive a particular latent-semantic structure model from the term-by-document matrix. In SVD, each term and artifact could be represented by a vector in the k space. The choice of k value, *i.e.*, the SVD reduction of the latent structure, is critical and still an open issue in the natural language processing literature. We want a value of k that is large enough to fit all the real structures in the data but small enough so we do not also fit the sampling error or unimportant details. In this study, we tried various values of k and $k = 200$ for Jabref, muCommander, and Lucene, and $k = 50$ for Rhino provide better accuracy than the other k values. Once all bug reports and source code documents have been represented in the LSI sub-space, we compute the similarities between bug reports and classes using cosine similarity as explained in Section III-B.

F. Linking Bugs Reports and Classes using LIBCROOS

In LIBCROOS, we take the preprocessed corpora as input to link the bug reports with classes. Any IR technique could be used in LIBCROOS to link bugs to classes, as base line to start the process. In this paper, we use the LSI and VSM IR techniques. For each bug report, LSI and VSM generate a ranked list of classes. Each ranked list contains potential culprit classes in descending order. We use our relation model to analyse the relationships between the classes of each bug's ranked list. We put the classes in four sets, *i.e.*, $Q_{i,1}$, $Q_{i,2}$, $Q_{i,3}$, and $Q_{i,4}$ (see Section III-C), depending on the BCR by which they are linked to other classes in the ranked list. If a class does not have any BCR with other classes, we do not keep that class in any of the sets. Thus, each set is treated as a different ranked list. Indeed, we have four ranked lists for each bug. We combine all ranked list as described in III-D using Equation 1.

To select λ values, we simply assigned 0.1 weight to each BCR for the corpora, *i.e.*, Jabref, Lucene, muCommander, and Rhino, and both IR techniques. We observed that using $\lambda = 0.1$ for each BCR yield to a good accuracy of LIBCROOS on the corpora. Furthermore, we analysed that assigning between 0.07 to 0.15 λ value to each BCR statistically increases the accuracy. More experiments are required with other corpora to generalise the weights for each BCR. Choosing a weight is still an open research question [6]. However, we observe that using weights between 0.07 to 0.15 for all the copora decreases the ranking of culprit classes.

G. Analysis Methods

We performed the following analysis on the LIBCROOS, LSI, and VSM ranked lists to answer our research questions by attempting to reject our null hypotheses.

We do not use precision and recall, because, on the one hand, the IR techniques would link all the bug reports to all the classes and recall would always be equals to 100% and because, on the other hand, using a high threshold value [6] based on textual similarity and linking only some bug reports to classes would yield a high precision. Thus, to compare LIBCROOS, LSI, and VSM, we use the rank of the first related class to a bug report as a measure of accuracy [6]. To answer RQ1, we compare LIBCROOS generated ranked list with LSI and VSM generated ranked lists of each bug. To answer RQ2, we use LIBCROOS with only one BCR at a time to compare its ranked list with LSI and VSM's ranked lists. Each approach used the same bug reports and classes. All null hypotheses have been tested using a paired, nonparametric test Mann-Whitney test. We use a significance level of 95% for all the statistical test. We use the Mann-Whitney test to assess whether the differences

in accuracy, in terms of rankings, are statistically significant between the LIBCROOS, LSI, and VSM. Mann-Whitney is a nonparametric test and, therefore, it does not make any assumption about the distribution of the data.

V. Results and Discussion

This section presents the results of our experiments.

Figure 2 shows the box plots of the accuracy measure, in terms of ranking, of LIBCROOS, LSI, and VSM applied on the corpora. we use manually built oracles to analyse the rank of actual culprit class in the ranked list generated by LIBCROOS, LSI, and VSM. For all the corpora, LIBCROOS assigns a lower rank to the culprit classes in terms of lower quartile, mean, median, standard deviation, and upper quartile. The results illustrated in Figure 2 provide a high-level view of the accuracy of LIBCROOS. The smaller boxes represent the decreases in rankings, *i.e.*, the culprit classes at the top in the ranked lists.

Table I reports results for LIBCROOS using all the BCRs and using only one BCR at a time. Results show that LIBCROOS statistically decreases the ranks of culprit classes up to 67%. For example, in the case of muCommander (VSM), on average across bug reports, LIBCROOS decreases the rank of culprit from 39.88 to 13.23. Standard deviation values show that all the ranks tend to be very close to the mean for LIBCROOS, whereas for LSI and VSM, they are spread out over a large range of values.

We have statistically significant evidence to reject the H_{01} and H_{02} hypotheses. The $p-$values, for all the comparison of LIBCROOS vs. LSI and LIBCROOS vs. VSM, are below the standard significant value, *i.e.*, 0.05. Results of RQ1 support our conjecture that adding BCRs with IR techniques helps to improve the accuracy, in terms of ranking, of IR techniques. Thus, we conclude that using BCRs among different classes that implement a same feature (or participate to a same bug) with IR techniques yields better accuracy than "traditional" IR techniques alone in bug location.

> Thus, we answer RQ1 as follow: LIBCROOS helps to decrease the rank of culprit classes and put culprit classes higher in the ranked lists when compared to "traditional" IR techniques alone.

To answer RQ2, we use LIBCROOS with only one BCR at a time. We assign 0.1 weight to λ in Equation 1 (see Section IV-D. Table I shows the detailed results of each BCR separately. In the majority of the programs, we can see that inheritance is the most important relation, then aggregation, use, and association, for bug location. However, using other weights for each BCR could yield different results. We performed experiments using weights λ between 0.07 to 0.15 for each BCR and observed the same importance as mentioned before. We performed statistical tests to compare LIBCROOS using one BCR with LSI and VSM and test our null hypotheses H_{03} and H_{04}.

We have statistically-significant evidence to reject H_{03} and H_{04}. The $p-$values, for all the comparison of LIBCROOS using one BCR vs. LSI and vs. VSM, are below the standard significant value. The results show that using only one BCR also improves the accuracy, in terms of ranking, of IR techniques for bug location. LIBCROOS using only one BCR helps to decrease the ranks of culprit classes and put culprit classes higher in the ranked lists, when compared to "traditional" IR techniques alone. However, all the BCRs have different importance for bug location.

> Thus, we answer RQ2 as follow: inheritance is the most important BCR to decrease the ranking, then aggregation, use, and association.

A. Threats to Validity

Several threats potentially limit the validity of our empirical study. We now discuss potential threats and how we control or mitigate them.

Construct validity: Construct validity concerns the relation between theory and observations. The degree of imprecision of automatic bug reports to class traceability link was quantified by means of a manual validation of the ranked lists generated by the approaches, using manually-built oracles. We did not build the oracles for three programs. Thus, there is no bias in the links between bug reports and classes. All three corpora have been used by various researchers and manual oracles have been verified to avoid imprecision in the measurement. Only for Lucene, we automatically built the oracle by mining JIRA software repository. We used Lucene to mitigate the threat of imprecision of manually built oracles.

Internal Validity: The internal validity of a study is the extent to which a treatment effects change in the dependent variable. The internal validity of our empirical study could only be threatened by our choice of the λ value: other values could lead to different results. To mitigate this threat, we use the same λ for all the corpora and approaches. We used $\lambda = 0.1$ for every BCR in all the experiments. In addition, we used λ values between 0.07 to 0.15 to analyse; we found statistical improvement on these λ values too. However, it is possible using other λ values combination provide different results. Thus, more experiments are required.

External Validity: The external validity of a study relates to the extent to which we can generalise its results. Our empirical study is limited to four programs, Jabref, Lucene, muCommander, and Rhino. Yet, our approach is applicable to any other object-oriented programs. However, we cannot claim that the same results would be achieved with other programs. Different programs with different usage patterns of BCRs, source code structure, and identifiers may lead to different results. However, the four selected programs have different source code size, different number of BCRs, and identifiers. Our choice reduces the threat to the external

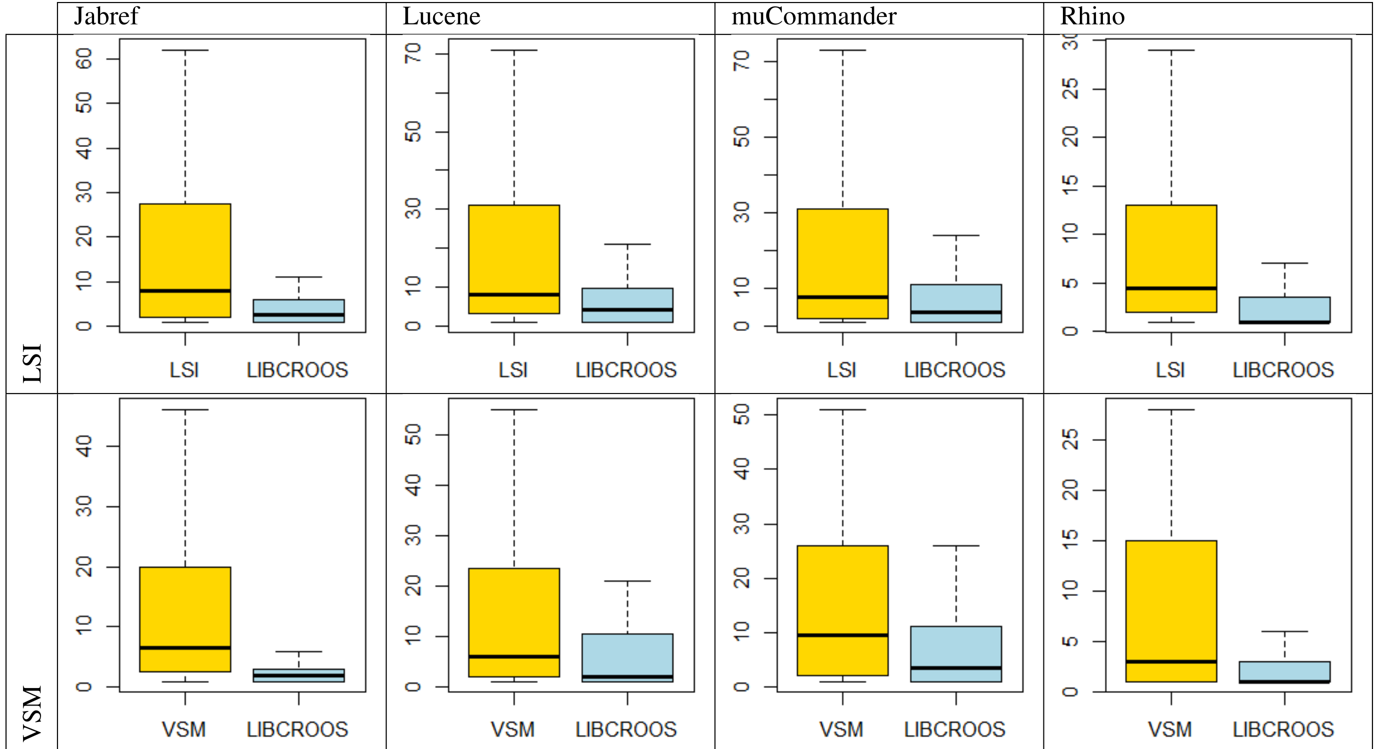

Figure 2. LIBCROOS, LSI, and VSM Results Boxplot Graph

validity. The results suggest that inheritance helps better to put the culprit classes at the top. We explain this observation by the fact that inheritance is the most accurate BCR in the model because inheritance is explicit in the source code and actually expresses exactly what the developers want. For other BCRs that are not explicit in source code and recovered with the help of some algorithms, it may not always be the developers' intent to have these BCRs. Also, the analyses may miss some relationships or add more relationships than expressed in the design. Thus, more case studies are required to generalise our results.

Conclusion validity: Conclusion validity threats deals with the relation between the treatment and the outcome. The appropriate non-parametric test, Mann-Whitney, was performed to statistically reject the null-hypotheses, which does not make any assumption on the data distribution.

VI. Conclusion and Future Work

The literature [4], [5], [6] showed that information retrieval (IR) techniques are useful to link features, *e.g.*, bug reports, and source code, when a bug is defined as a feature that deviates from the program specification [3]. However, IR techniques lack accuracy in terms of ranking [26]. In this paper, we conjectured that whenever developers implement some features, they use some BCRs among the different classes implementing the feature. Thus, combining BCRs with IR techniques could increase their accuracy in terms of ranking, to help developers with their bug location tasks. To verify our conjecture, we proposed a new approach, LIBCROOS, that uses BCRs and textual information extracted from source code to link classes and bug reports. To the best of our knowledge, LIBCROOS is the first approach to use BCRs as experts to vote on the links recovered by an IR technique. In this paper, we considered four commonly-used relationships, *i.e.*, use, association, aggregation, and inheritance, and two IR techniques, *i.e.*, LSI and VSM, to link classes and bug reports.

To evaluate the effectiveness of our proposed approach, we performed an empirical study on four software programs, *i.e.*, Jabref, Lucene, muCommander, and Rhino. We compared LIBCROOS-generated ranked lists of classes with the ranked lists produced by LSI and VSM alone. The results achieved in the reported empirical study showed that, in general, LIBCROOS improves the accuracy of LSI and VSM. In all experiments, LIBCROOS improved the accuracy of the IR-based technique and could reduce developers' efforts by putting actual culprit classes at the top in the ranked lists. We also used LIBCROOS with only one BCR at a time to analyse which BCR helps more to improve the accuracy of the IR techniques. In the majority of the programs, we observed that inheritance is a more important relation than aggregation, use, and association.

We also observed that, as the size of the source code increases, IR techniques produce larger ranked lists, which increase the difficulty of developers' bug-location tasks, because they must manually iterate through more potential culprit classes. LIBCROOS automatically increases the rank

of non-culprit classes and brings actual culprit classes closer to the beginning of the ranked lists. Our experiments showed that the size of the source code does not impact the accuracy of LIBCROOS. In the contrary, the more relationships among classes, the greater the accuracy of LIBCROOS. Our empirical study helps developers and practitioners to understand how BCRs could be used for bug location. In addition, it suggests that developers must use more relationships among classes contributing to a same feature.

There are several ways in which we are planning to continue this work. First, we will consider more BCRs to analyse the impact of different relationships on bug location. Second, we will apply our approach on different software comprehension and maintenance activities, *e.g.*, requirements traceability. Third, we will analyse more datasets to increase the generalisability of our findings. We will also perform an in-depth analysis of the λ value effect on other datasets. Lastly, we will use other IR techniques, *e.g.*, Jensen Shannon divergence, to quantify the improvement using our proposed approach.

References

[1] H. A. Muller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000, pp. 47–60.

[2] S. Lukins, N. Kraft, and L. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, oct. 2008, pp. 155 –164.

[3] E. Allen, *Bug Patterns in Java*. APress L. P., 2002.

[4] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[5] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of 25th International Conference on Software Engineering*. Portland Oregon USA: IEEE CS Press, 2003, pp. 125–135.

[6] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[7] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Towards employing use-cases and dynamic analysis to comprehend mozilla," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2005, pp. 639–642.

[8] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 18:1–18:36, 2008.

[9] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004, pp. 214 – 223.

[10] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Trust-based requirements traceability," in *Proceedings of the* 19th *International Conference on Program Comprehension*, S. E. Sim and F. Ricca, Eds. IEEE Computer Society Press, June 2011, 10 pages.

[11] N. Wilde and C. Casey, "Early field experience with software reconnaissance technique for program comprehension," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 1996.

[12] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Requirements traceability for object oriented systems by partitioning source code," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, oct. 2011, pp. 45 –54.

[13] P. Shao and R. K. Smith, "Feature location by ir modules and call graph," in *Proceedings of the 47th Annual Southeast Regional Conference*, New York, NY, USA, 2009, pp. 70:1–70:4.

[14] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, pp. 195–226, 2006.

[15] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[16] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," *SIGPLAN Not.*, vol. 22, pp. 466–481, 1987.

[17] J. Purdum, *Beginning C# 3.0: An Introduction to Object Oriented Programming (Wrox Beginning Guides)*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[18] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: putting icing on the uml cake," *SIGPLAN Not.*, vol. 39, pp. 301–314, 2004.

[19] D. J. Pearce and J. Noble, "Relationship aspects," in *Proceedings of the 5th international conference on Aspect-oriented software development*, New York, NY, USA, 2006, pp. 75–86.

[20] D. Jackson and A. Waingold, "Lightweight extraction of object models from bytecode," in *Proceedings of the 21st international conference on Software engineering*, New York, NY, USA, 1999, pp. 194–202.

[21] Y.-G. Guéhéneuc, "Ptidej: Promoting patterns with patterns," in *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*. Springer-Verlag, 2005.

[22] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, 2008, pp. 103 –112.

[23] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[24] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, 2011, pp. 11 –20.

[25] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 97–106.

[26] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, *Factors Impacting the Inputs of Traceability Recovery Approaches*, A. Zisman, J. Cleland-Huang, and O. Gotel, Eds. New York: Springer-Verlag, 2011.