Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes

Robert Dyer Hridesh Rajan Tien N. Nguyen
Iowa State University
{rdyer,hridesh,tien}@iastate.edu

Abstract

Software repositories contain a vast wealth of information about software development. Mining these repositories has proven useful for detecting patterns in software development, testing hypotheses for new software engineering approaches, etc. Specifically, mining source code has yielded significant insights into software development artifacts and processes. Unfortunately, mining source code at a large-scale remains a difficult task. Previous approaches had to either limit the scope of the projects studied, limit the scope of the mining task to be more coarse-grained, or sacrifice studying the history of the code due to both human and computational scalability issues. In this paper we address the substantial challenges of mining source code: a) at a very large scale; b) at a fine-grained level of detail; and c) with full history information.

To address these challenges, we present domain-specific language features for source code mining. Our language features are inspired by object-oriented visitors and provide a default depth-first traversal strategy along with two expressions for defining custom traversals. We provide an implementation of these features in the Boa infrastructure for software repository mining and describe a code generation strategy into Java code. To show the usability of our domain-specific language features, we reproduced over 40 source code mining tasks from two large-scale previous studies in just 2 person-weeks. The resulting code for these tasks show between 2.0x–4.8x reduction in code size. Finally we perform a small controlled experiment to gain insights into how easily mining tasks written using our language features can be understood, with no prior training. We show a substantial number of tasks (77%) were understood by study participants, in about 3 minutes per task.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

Keywords visitor pattern; source code mining; Boa

1. Introduction

An extremely large wealth of information exists in software repositories, such as open-source repositories like SourceForge which contain over 250k projects, metadata about those projects, source code repositories for those projects including revision history, bug artifacts, etc. Mining this wealth of information is important for researchers in order to detect problems with existing development practices or to use for quantifying proposed solutions to software engineering problems. Many previous works mine these open-source software projects and software repositories to support use cases that include:

- guiding software evolution [23, 24, 35],
- discovering API usage [14, 21, 25, 33],
- fault prediction [10, 13, 17, 22],
- discovering code characteristics [8, 31],
- language feature usage [7, 11, 30], etc.

Source code mining is similar in nature to compilation and program analysis techniques and such techniques often use the visitor pattern [9]. This pattern allows easily traversing the structure of the underlying source code, which is represented as a graph or tree. For example, compilers represent source code as an abstract syntax tree (AST) and typically have several visitors (e.g. for type checking, semantic analysis, code generation). Many developers are familiar with compilers and/or program analysis techniques and have seen/used visitors before.

Despite this, existing approaches for mining source code typically use relational databases [4, 11, 19] and query with SQL. Other approaches provide the ability to mine source code using Datalog [12], Program Query Language (PQL) [20], JQuery [15], or even natural language queries [18]. The motivating question for this work is: can we present a more familiar interface to people interested in mining source code data?

In this paper we present domain-specific language features for mining source code. These features are inspired by the rich body of literature on object-oriented visitor patterns [1, 9, 26, 27, 34]. A key difference from previous work is that we do not require the host language to contain object-oriented features. Our *visitor types* provide a default depth-first search (DFS) traversal strategy, while still maintaining the flexibility to allow custom traversal strategies. Visitor types allow specifying the behavior that executes for a given node type, *before* or *after* visiting the node's children. The language also provides abstractions for dealing with mining of source code history, such as the ability to retreive specific snapshots based on date. We also show several useful patterns for source code mining that utilize these domain specific language features.

We show the feasibility of supporting these domain-specific features by realizing them in the Boa research infrastructure for mining software repositories [6]. Boa provides support for running mining tasks on a very large set of software repositores (699,332 projects from Sourceforge as of February 2013 [32]). As of May 2013, full support for the features described in this work is available

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/2517208.2517226

in the Boa research infrastructure and is being actively used by ourselves and other researchers for various mining tasks.

To evaluate the language features proposed in this work, we have reproduced two previous large-scale empirical studies using these new features [7, 11]. Together these require writing over 40 software repository mining tasks as programs. Both of these studies require fine-grained access to source code, e.g. Grechanik *et al.* [11] accesses expressions in source code. The study by Dyer *et al.* also requires access to full history information to examine usage of Java language features over time [7]. Finally, both studies require processing large data sets for higher confidence in the results. Using our new language features, a single student was able to reproduce both of these large-scale studies in 2 person-weeks.

Besides these studies, we have also written several other small and medium size mining tasks that are available from the Boa website [32]. These other use cases further increase our confidence in the usefulness of the proposed language features.

We also show some initial insights into the ease of comprehension of mining tasks written in our framework, in which over 75% of tasks were understood in about 3 minutes with no prior training in our language.

In summary, this paper presents the following contributions:

- Domain-specific language abstractions for easily writing source code mining tasks on billions of AST nodes.
- A data schema for representing source code in a language-agnostic manner.
- An implementation of the language features in the Boa infrastructure for software repository mining [6, 32].
- A partial reproduction of a large-scale empirical study on the Java language [11], with over 10 times more projects than the initial study.
- Initial insights into the ease of comprehension of mining tasks written in our framework, in which over 77% of tasks were understood in about 3 minutes each, with no prior training in the language. The same tasks written in Java had only 62% comprehension.

In the next section we motivate the need for large-scale, fine-grained source code mining via an example. We give necessary background on the data representation in Section 3 and detail our approach in Section 4. Our code generation strategy is described in Section 5. We then evaluate our approach in Section 6. In Section 7 we discuss related works. Finally we conclude in Section 8.

2. Motivation

Mining source code is extremely useful for researchers, allowing them to investigate if potential problems exist in reality and test their hypotheses on real-world software. For example, Okur and Dig mine the source code to over 600 programs to see how programmers use parallel libraries and if they use those libraries correctly [25]. Pinto *et al.* mine source code to investigate how test suites evolve over time [31]. Gabel and Su mine over 6k projects to determine how unique source code is [8]. These are but a few example use cases for mining source code. In this section we motivate the need for a domain-specific language for source code mining via a simple task.

Consider testing a simple hypothesis: a large number of bug fixes add checks for `null`. Null-pointer exceptions are a common source of bugs in object-oriented programs. A possible fix for some of these bugs may be to simply guard access to the variable with a check to ensure it is non-null. To investigate such a hypothesis, one may perform the following tasks:

1. Download candidate source code repositories (for example, from SourceForge [2]).
2. Write a program to scan all repositories and locate revisions that potentially fixed bugs.
3. Check out source code snapshots from the identified revisions and the previous snapshots (if any) of the code.
4. Write a program to compare each pair of files, and determine if the number of null checks has increased since the previous snapshot. If so, these files potentially represent a bug fix that added a null check.
5. Parallelize the previous program to support mining tens or hundreds of thousands of projects.

For the purposes of this paper, we assume step 1 has already finished and the repositories are available in a format most suitable for each query language used. This step by itself represents a significant challenge, but for simplicity of this example we will not go into detail on that step at the moment and just assume the data is already available in a suitable format.

The remaining steps, while sounding relatively simple, are very complex as well. For now, let's focus on a small portion of just step 4: let's write queries to find null comparisons in source code. Once we have such a query, we can of course extend it to look for such comparisons occurring inside an if-statement and apply that query to different versions of files, completing step 4.

One possible implementation could be in Java, using Hadoop [3] MapReduce [5] to parallelize the mining task. Such a program is shown in Figure 1. As with most Hadoop programs, there are three main sections: the job setup (lines 2–15), the mapper class (lines 16–138), and the reducer class (lines 139–146). The main portion of the mining task is inside the mapper's `map` method (lines 126–136).

This code is a mixture of several different features: the Hadoop code for efficient data parallelization (lines 2–15 and 139–146), code for traversing the structure of the source being mined (lines 17–124), and code for performing the mining (lines 126–136). Even if you only focus on the mining portion of the code and ignore the rest, this is a complex program.

Another possible implementation in Boa [6, 32] is shown in Figure 2. This program takes a project as input (line 1). It then declares a single visitor (lines 3–8) named `nullCheck`. When the visitor reaches a node of type `Expression` (line 4), it checks if that expression is a comparison operator (line 5) and if one of the operands is `null` (line 6). If it is, then it increments a counter (line 7). This visitor is used by starting a visit on the project using the declared visitor (line 9).

This code avoids the boilerplate code and complexity of the Hadoop version by abstracting away the details of step 5 from the user. Similar to the Hadoop version, it offers a visitor syntax which is easy to understand (as shown in Section 6.2) and familiar to developers. Expanding this query is straight-forward: simply add additional visitors and/or add more clauses to the existing visitor.

3. Background: Representing Data in Boa

Our approach builds on top of Boa [6, 32], a domain-specific language and research infrastructure for efficient, scalable software repository mining. In Boa, users write simple queries in a language that has abstracted away the details of how to write a MapReduce program, thus allowing users to focus on the mining task and not on how to parallelize their programs. Boa's compiler automatically generates a Hadoop [3] program from the source code. Users submit their program to Boa's website via the web interface shown in Figure 3, which compiles and executes it on a cluster. This cluster

```
1   class AddNullCheck {
2     static void main(String[] args) {
3       ... /* create and submit a Hadoop job */

15    }
16    static class AddNullCheckMapper extends
          Mapper<Text, BytesWritable, Text, LongWritable> {
17      static class DefaultVisitor {
18        ... /* define default tree traversal */

124     }

125     void map(Text key, BytesWritable value,
             Context context) {
126       final Project p = ... /* read from input */
127       new DefaultVisitor() {
128         boolean preVisit(Expression e) {
129           if (e.kind == ExpressionKind.EQ ||
                  e.kind == ExpressionKind.NEQ)
130             for (Expression exp : e.expressions)
131               if (exp.kind == ExpressionKind.LITERAL
                      && exp.literal.equals("null")) {
132                 context.write(new Text("count"),
                                  new LongWritable(1));
133                 break;
134               }
135         }
136       }.visit(p);
137     }
138   }
139   static class AddNullCheckReducer
           extends Reducer<Text, LongWritable,
                           Text, LongWritable> {
140     void reduce(Text key, Iterable<LongWritable> vals,
                    Context context) {
141       int sum = 0;
142       for (LongWritable value : vals)
143         sum += value.get();
144       context.write(key, new LongWritable(sum));
145     }
146   }
147 }
```

Figure 1. Finding null checks in Java/Hadoop.

```
1  p: Project = input;
2  count: output sum of int;

3  nullCheck := visitor {
4    before e: Expression ->
5      if (e.kind == ExpressionKind.EQ
           || e.kind == ExpressionKind.NEQ)
6        exists (i: int; isliteral(e.expressions[i],
                                   "null"))
7          count << 1;
8  };

9  visit(p, nullCheck);
```

Figure 2. Finding null checks in Boa.

already contains a cached copy of the software repositories to be mined. Once finished, the website provides the output to the user.

Boa represents all input data using a tree structure. This tree is rooted with the `Project` and contains information such as project metadata, the source code repositories (SVN, CVS, etc), and the actual source code data.

The types Boa provides for representing source code are: `Namespace`, `Declaration`, `Method`, `Variable`, `Type`, `Statement`, `Expression`, and `Modifier`. Several of these are shown in Figure 4 [1], along with the enumeration

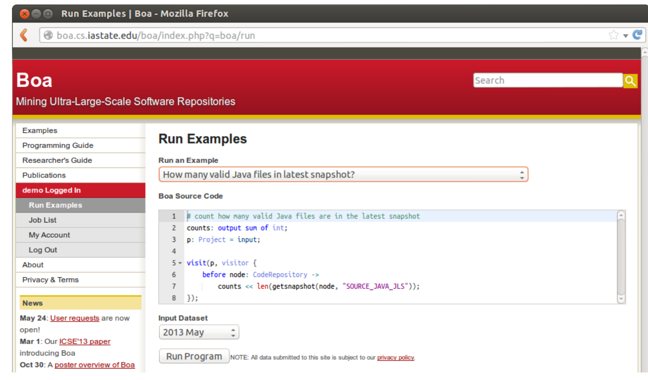

Figure 3. Boa's interface. Zero installation cost. Accessible interface to ease adoption in research, practice, and education.

`StatementKind`. The declaration, statement, and expression types are discriminated types, meaning they actually represent the union of many different record structures.

```
Namespace
name        : string
modifiers   : Modifier[]
declarations: Declaration[]
```

```
Declaration
name                : string
kind                : TypeKind
modifiers           : Modifier[]
methods             : Method[]
fields              : Variable[]
generic_parameters  : Type[]
parents             : Type[]
nested_declarations : Declaration[]
```

```
Statement
kind                 : StatementKind
statements           : Statement[]
initializations      : Expression[]
updates              : Expression[]
variable_declaration: Variable?
type_decl            : Declaration?
expression           : Expression?
```

```
enum StatementKind
OTHER, BLOCK, TYPEDECL, RETURN,
DO, FOR, ASSERT, SYNCHRONIZED,
WHILE, IF, BREAK, THROW, TRY,
CATCH, SWITCH, CASE, EXPRESSION,
LABEL, CONTINUE, EMPTY
```

Figure 4. Discriminated types for representing source code.

For example, consider the type `Statement` shown in Figure 4. This type has an attribute `kind`, which is an enumerated value. Based on the kind of statement, different attributes in the record will be set. For example, if the kind is `TYPEDECL` then the `type_decl` attribute is defined. However if the kind is `CATCH` then the `type_decl` is undefined. Representing these types as discriminated types allows Boa to keep the number of types as small as possible. This makes supporting future languages easier by only needing to provide a mapping from the new language to the small set of types in Boa. Existing mining tasks would immediately be able to mine source code from these new languages.

While Boa keeps these types as simple as possible, they are still flexible enough to support more complex language features. For example, consider the enhanced-for loop in Java:

```
1  for (String s : iter)
2    body;
```

which says to iterate over the expression `iter` and for each string value `s`, run the `body`. Boa's types do not directly contain an `ENHANCEDFOR` kind for this language feature.

Despite this design decision, an enhanced-for statement can be easily represented in Boa's schema without having to extend it. First, Boa generates a `Statement` of kind `FOR`. Inside that statement, Boa sets `expression` to `iter`. Boa also sets the `variable_declaration` for `String s` in the statement. Thus, if a statement of kind `FOR` has its `variable_declaration` attribute set it is a for-each statement. If that attribute is not defined, then it is a standard for-loop.

Currently, we have fully mapped the Java language to Boa's schema, attempting to simplify the schema as much as possi-

[1] For a full list of all data types and their attributes, please visit Boa's online documentation:
http://boa.cs.iastate.edu/docs/dsl-types.php

ble. This gives a simple, yet flexible, schema capable of supporting the entire Java language (through Java 7). As additional support for other source languages is added, if the schema is not capable of directly supporting a particular language feature the `StatementKind` or `ExpressionKind` enumerations can be easily extended.

4. Fine-grained Source Code Mining

Users must be able to easily express source code mining tasks. For users who are intimately familiar with compilers and interpreters, the visitor style is well understood. However, other users may find two aspects of visitor-style traversals daunting. First, it generally requires writing a significant amount of boiler-plate code whose length is proportional to the complexity of the programming language being visited. Second, this strategy requires intimate familiarity with the structure of that programming language.

To make source code mining more accessible to all users, we investigated the design of more declarative features for mining source code. In this section, we describe our proposed syntax for writing source code mining tasks. The syntax was inspired by previous language features, such as the before and after visit methods in DJ [27] and case expressions in Haskell [16].

visitor ::= `visitor` { *visitClause** }
visitClause ::= *beforeClause* | *afterClause*
beforeClause ::= `before` *typeList* -> *beforeClauseStmt*
afterClause ::= `after` *typeList* -> *stmt*
typeList ::= _ | *identifier* : *type* | *type* (, *type*)*
beforeClauseStmt ::= *stmt* | *stopStmt* | `visit` (*identifier*) ;
stopStmt ::= `stop` ;

Figure 5. Proposed syntax for easing source code mining.

The new syntax is shown in Figure 5. The top-level syntax for a mining task is a *visitor type*. Visitor types take zero or more *visit clauses*. A visit clause can be a *before* or an *after* clause. During traversal of the tree, a before clause is executed when visiting a node of the specified type. If the default traversal strategy is used, then the node's children will be visited. After all the children are visited, any matching after clause executes.

Before and after clauses take a *type list*. A type list can be a single type with an optional identifier, a list of types, or an underscore wildcard. The underscore wildcard provides default behavior for a visitor clause. This default executes for a node of type T if no other clause specifies T in its type list. Thus, the following code:

```
1  v := visitor {
2    before Project, CodeRepository, Revision -> { }
3    before _ -> counter++;
4  }
```

will execute the clause's body on line 2 when traversing nodes of type `Project`, `CodeRepository`, or `Revision`. When traversing a node of any other type, the default clause's body on line 3 executes. The result of this code is thus a count of all nodes, excluding those of the types listed. Thus we count only the source code AST nodes for a project.

Note that unlike pattern matching and case expressions in functional languages like Haskell, the order of the before and after clauses do not matter. A type may appear in at most one before clause and at most one after clause.

To begin a mining task, users write a `visit` statement:

```
visit(n, v);
```

that has two parts: the node to visit and a visitor. When this statement executes, a traversal starts at the node represented by `n` using visitor `v`.

4.1 Supporting Custom Traversals

To allow users the ability to override the default traversal strategy, two additional statements are provided inside `before` clauses. The first is the *stop statement*:

```
stop;
```

which when executed will stop the visitor from traversing the children of the current node. This is useful in cases where the mining task never needs to visit specific types further down the tree, allowing to stop at a certain depth. Note that stop acts similar to a return, so no statements after it are reachable.

If the default traversal is stopped, users may provide a custom traversal of the children with a *visit statement*:

```
visit(child);
```

which says to visit the node's `child` tree once. This statement can be called on any subset of the children and in any order. This also allows for visiting a child more than once, if needed.

Figure 6 illustrates a custom traversal strategy from one of our case studies [7]. This program answers the question *how many fields that use a generic type parameter are declared in each project?* To answer this question, the program declares a single visitor. This visitor looks for `Type` nodes where the name contains a generic type parameter (line 5). This visit clause by itself is not sufficient to answer the question, as generic type parameters might occur in other locations, such as the declaration of a class/interface, method parameters, locals, etc. Instead, a custom traversal strategy (lines 10–34) is needed to ensure only field declarations are included.

The traversal strategy first ensures all fields of `Declaration` are visited (lines 12–13). Since declarations can be nested (e.g. in Java, inside other types and in method declarations) we also must manually traverse to find nested declarations (lines 15–32). Finally, we don't want to visit nodes of type `Expression` or `Modifier` (line 34), as these node types can't possibly contain a field declaration but may contain a `Type` node.

Complex mining tasks can be simplified by using multiple visitors. For example, perhaps we only want to look for certain expressions inside of an if statement's condition. We can write a visitor to find if statements, and then use a second sub-visitor to look for the specific expression by visiting the if statement's children. We could perform this mining task with one visitor, however then we need to have flags set to track if we are in the tree underneath an if statement. Using multiple visitors keeps these two mining tasks separate and avoids using flags to keep it simple.

4.2 Mining Snapshots in Time

While our infrastructure contains data for the full revision history of each file, some mining tasks may wish to operate on a single snapshot. We provide several helper functions to ease this use case. For example, the function:

```
getsnapshot(CodeRepository [, time] [, string...])
```

takes a `CodeRepository` as its first argument. It optionally takes a time argument, specifying the time of the snapshot which defaults to the last time in the repository. The function also optionally takes a list of strings. If provided, these strings are used to filter files while generating the snapshot. The file's kind is checked to see if it matches at least one of the patterns specified. For example:

```
getsnapshot(CodeRepository, "SOURCE_JAVA_JLS")
```

says to get the latest snapshot and filter any file that is not a valid Java source file.

A useful pattern is to write a visitor with a before clause for `CodeRepository` that gets a specific snapshot, visits the nodes in the snapshot, and then stops the default traversal:

```
p: Project = input;
GenFields: output sum[string] of int;

genVisitor := visitor {
  before t: Type ->
    if (strfind("<", t.name) > -1)
      GenFields[p.id] << 1;

  # traversal strategy ensures we only reach Type
  # if the parent is a Variable, and
  # we only include Variable paths that are fields
  before d: Declaration -> {
  ######## check each field declaration ########
    foreach (i: int; d.fields[i])
      visit(d.fields[i]);

  ########### look for nested types ############
    foreach (i: int; d.methods[i])
      visit(d.methods[i]);
    foreach (i: int; d.nested_declarations[i])
      visit(d.nested_declarations[i]);
    stop;
  }
  before m: Method -> {
    foreach (i: int; m.statements[i])
      visit(m.statements[i]);
    stop;
  }
  before s: Statement -> {
    foreach (i: int; s.statements[i])
      visit(s.statements[i]);
    if (def(s.type_declaration))
      visit(s.type_declaration);
    stop;
  }

  ####### stop at expressions/modifiers ########
  before Expression, Modifier -> stop;
};
visit(p, genVisitor);
```

Figure 6. Using a custom traversal strategy to find uses of generics in field declarations.

```
visitor {
  before n: CodeRepository -> {
    snapshot := getsnapshot(n);
    foreach (i: int; def(snapshot[i]))
      visit(snapshot[i]);
    stop;
  }
  ...
}
```

This visitor will visit all code repositories for a project, obtain the last snapshot of the files in that repository, and then visit the source code of those files. This pattern is useful for mining the *current version* of a software repository.

4.3 Mining Revision Pairs

Often a mining task might want to locate certain revisions and compare files at that revision to their previous state. For example, our motivating example looks for revisions that fixed bugs and then compares the files at that revision to their previous snapshot. To accomplish this task, one can use the following pattern:

```
files: map[string] of ChangedFile;

v := visitor {
  before f: ChangedFile -> {
    if (def(files[f.name])) {
      ... # task comparing f and files[f.name]
    }
    files[f.name] = f;
  }
};
```

which declares a map of files, indexed by their path. The code on line 4 checks if a previous version of the file was cached. If it was, the code on line 5 executes where `f` refers to the current version of the file being visited and the expression `files[f.name]` refers to the previous version of the file. Finally, the code on line 7 updates the map, storing the current version of the file.

4.4 Bringing It All Together: Motivating Example

Recall the hypothesis in Section 2: a large number of bug fixes add checks for `null`. In that section, we focused on a very small sub-task of step 4. In this section, we describe a solution that incorporates all five steps required to answer the proposed hypothesis.

```
# STEP 1 - candidate projects as input
p: Project = input;
results: output collection[string] of string;

fixing := false;
count := 0;
files: map[string] of ChangedFile;

nullCheckVisitor := visitor {
  before e: Expression ->
    if (e.kind == ExpressionKind.EQ
        || e.kind == ExpressionKind.NEQ)
      exists (i: int; isliteral(e.expressions[i],
                                  "null"))
        count++;
};

visit(p, visitor {
  before r: Revision ->
    # STEP 2 - potential revisions that fix bugs
    fixing = isfixingrevision(r.log);

  before f: ChangedFile -> {
    if (fixing && haskey(files, f.name)) {
      count = 0;
      # STEP 3a - check out source from revision
      visit(getast(files[f.name]));
      last := count;

      count = 0;
      # STEP 3b - source from previous revision
      visit(getast(f));

      # STEP 4 - determine if null checks increased
      if (count > last)
        results[p.id] << string(f);
    }
    files[f.name] = f;
    stop;
  }

  before s: Statement ->
    if (s.kind == StatementKind.IF)
      visit(s.expression, nullCheckVisitor);
});
```

Figure 7. Finding in Boa fixing revisions that add null checks.

Consider the Boa program in Figure 7, which implements all five steps of the entire mining task. This program takes a single project as input. It then passes the program's data tree to a visitor (line 13). This visitor keeps track if the last `Revision` seen was a fixing revision (line 16). When it sees a `ChangedFile` it looks at the current revision's log message and if it is a fixing revision (step 2) it will get snapshots of the current file and the previous version of the file (step 3) and visit their AST nodes (lines 21 and 25).

When visiting the AST nodes for these snapshots, if it encounters a `Statement` of kind `IF` (line 34), it then uses a sub-visitor to check if the statement's expression contains a null check (lines 35 and 7–12) and increments a counter (line 11). Thus we will know the number of null checks in each snapshot and can compare (line

27) to see if there are more null checks (step 4). Note that this analysis is conservative and may not find all fixing revisions that add null checks, as the revision may also *remove* a null check from another location and thus give the same count.

This task illustrates several features mentioned earlier in this section. First, the second visitor shows use of a custom traversal strategy by utilizing a stop statement. Second, it makes use of a sub-visitor (`nullCheckVisitor`). Third, it uses the revision pair pattern to check several versions of a file.

Finally, writing this task required no explicit mention of parallizing the query. Writing the same task in Hadoop would require a lot of boilerplate code to manually parallelize the task, whereas the Boa version is automatically parallelized.

5. Code Generation Strategy

As noted previously, each Boa program is translated by the Boa compiler into a Map/Reduce program that runs using the Hadoop framework. We extended this compiler to support visitor types. In this section we outline the code generation strategy for supporting visitor types. For ease of illustration, we omit all code related to Map/Reduce to allow readers to focus on visitor types. The key to our strategy involves a default visitor (Figure 8) that we added to the Boa runtime.

```
1   public abstract class DeafultVisitor {
2     public final void visit(Project node) {
3       if (preVisit(node)) {
4         ... // call visit() on each of node's children

8         postVisit(node);
9       }
10    }
11    ... // similar visit() for each node type

205   ///////////////////////////////////////
206   // methods for before clauses

207   protected boolean defaultPreVisit() {
208     return true;
209   }

210   protected boolean preVisit(Project node) {
211     return defaultPreVisit();
212   }
213   ... // similar preVisit() for each node type

250   ///////////////////////////////////////
251   // methods for after clauses

252   protected void defaultPostVisit() { }

253   protected void postVisit(Project node) {
254     defaultPostVisit();
255   }
256   ... // similar postVisit() for each node type

295 }
```

Figure 8. Outline of the abstract default visitor.

The `DefaultVisitor` class contains a public `visit` method for each node type in the language. These methods contain a single if-statement which calls a `preVisit` method in the condition. If that method returns `true`, then the children of the current node are each visited and a `postVisit` method is called.

A `preVisit` and `postVisit` method is also generated for each node type in the language. The bodies of these methods simply call the `defaultPreVisit`/`defaultPostVisit` methods. These methods are virtual methods and are (possibly) overridden by the concrete visitor sub-classes.

Generating Visitors All visitors in the language:

```
var := visit { .. };
```

inherit from the `DefaultVisitor` (Figure 8):

```
var = new DefaultVisitor() { .. };
```

This inheritance provides the visitor with a default depth-first traversal strategy that will visit all nodes in the tree. The actions taken when visiting specific nodes are specified via the before and after visit clauses.

Generating Visit Clauses A before visit clause generates one or more method overrides for the `preVisit` methods. There are three possibilities for a before visit clause's type list. First, it may specify a specific type and an identifier:

```
before id: T -> body;
```

which is translated into:

```
1  protected boolean preVisit(T id) {
2    body;
3    [return true;] // if necessary
4  }
```

Since the method must return a value, the `body` is analyzed to determine if a `stop` statement occurs on all exit paths. If it does not, then a return statement is generated with a value of `true`.

The second form for a visit clause's type list is a list of types:

```
before T1, T2, .. -> body;
```

which is translated similar to before, where each type has its own `preVisit` method generated and the id is a fresh name.

The third form is an underscore wildcard:

```
before _ -> body;
```

which is translated into:

```
1  protected boolean defaultPreVisit() {
2    body;
3    [return true;] // if necessary
4  }
```

similar to the previous translation strategy.

Generation of after visit clauses is almost identical to before clauses, with two slight differences. First, the name of the generated method is changed to `postVisit`/`defaultPostVisit`. Second, since the method has a void return type no return statements are generated.

Generating Stop Statements Before visit clauses return a boolean value to indicate if the `DefaultVisitor` should visit the children of the node. Since `stop` statements can only appear in before visit clauses, they are transformed into:

```
return false;
```

which makes the if condition (Figure 8, line 3) `false` and stops the default traversal of the node's children. It also stops the execution of the before visitor.

Generating Nested Visit Calls There is no need to transform a nested `visit` call, as both the method name and arguments are identical in the generated code.

Optimizing Traversals By default, the generated code visits every node in the tree. For some visitors, this may not be optimal. By analyzing the visit clauses, we can determine the lowest type being visited and ensure the traversal stops at that point.

Stopping is accomplished by adding a `stop` statement to the end of the before clause for that type. If the type only has an after clause, then an empty before clause is first generated. If that type also has an after clause, it is merged into the body of the before clause, immediately before the `stop` statement.

6. Evaluation

We now evaluate the utility and comprehensibility of new features.

6.1 Utility of Declarative Visitors

To evaluate the language features proposed in this work, we reproduced two previous large-scale empirical studies [7, 11] using these new features. Together these require writing over 40 software repository mining tasks as programs. Both of these studies require fine-grained access to source code, e.g. Grechanik *et al.* [11] access expressions in source code. The study by Dyer *et al.* also requires access to full history information to examine usage of Java language features over time [7]. Finally, both studies require processing large data sets for higher confidence in the results.

Using our new language features, a single student was able to reproduce both of these large-scale studies in 2 person-weeks. Grechanik *et al.*'s study [11] took a bit over 1 person-week and Dyer *et al.*'s study [7] took under 1 person-week. This included the time to formulate, write, execute, debug, and analyze results.

6.1.1 Java Language Feature Usage

Dyer *et al.*'s study [7] performed a large-scale study on the use of Java language features. In that study, they wrote several mining tasks to identify the use of Java language features over time. For example, one such task mined source code to track the use of `assert` statements over time. These tasks were written in Boa without the syntax proposed in Section 4. For this paper, we rewrote those tasks using the new syntax and then measured the lines of code for both versions. The results are shown in Table 1.

Task	LOC (visitor)	LOC (old [7])
Annotations-define	17	63 (3.7x)
Annotations-use	17	80 (4.7x)
Assert	17	63 (3.7x)
Binary-lit	17	63 (3.7x)
Diamond	17	82 (4.8x)
EnhancedFor	17	63 (3.7x)
Enums	17	63 (3.7x)
Generics-define-field	39	77 (2.0x)
Generics-define-method	17	63 (3.7x)
Generics-define-type	17	63 (3.7x)
Generics-wildcard-extends	17	82 (4.8x)
Generics-wildcard-super	17	82 (4.8x)
Generics-wildcard	17	82 (4.8x)
Multicatch	17	63 (3.7x)
Safe-varargs	38	83 (2.2x)
Try-resources	17	63 (3.7x)
Underscore-lit	18	64 (3.6x)
Varargs	17	63 (3.7x)
Total	**350**	**1,262 (3.6x)**
Mean	**19**	**70 (3.6x)**

Table 1. Mining tasks on Java language feature usage.

The results clearly show that our proposed syntax improves the writing of fine-grained source code mining tasks. On average, the tasks required almost 73% fewer lines of code. Most of the difference in lines came from the lack of needing to manually specify the traversal strategy. These results give some insight into the utility and potential ease of declarative visitors for mining tasks.

6.1.2 Reproducing the Treasure Study

Grechanik *et al.* performed a large-scale empirical study on Java source code from 2,080 open-source projects [11]. The dataset used in their study were randomly selected projects from SourceForge. For their study they built an SQL database containing tables and attributes for storing (non)terminals from Java's grammar. They posed 32 different research questions and queried their database to answer those questions. To show the usefulness of our approach, we reproduced a portion of this study using Boa.

As the actual queries used are not available, we had to make a few assumptions about their study. First, we assumed that none of the projects in their study were empty and all contained at least one valid Java source file. This assumption was made on the basis that their minimum value for number of classes per application is 1. Thus, we filter our dataset to exclude any projects without at least one valid Java source file. This left 23,510 projects in our study.

Second, although their paper only mentions parsing the source code, we assume that since they were only working with releases of each project that they also had type resolution and bindings. This information is not yet available in Boa[2], so we do not reproduce the six tasks that rely on that information for accuracy.

Finally, we assume that the versions of each project used in their study were the latest versions. As Boa contains all revisions for projects but does not currently know what revision(s) map to specific releases, for our version of the study we simply take the latest snapshot of each project. We also filter out any obvious branches (in SVN, branches typically are rooted in the 'branches' folder). This gave a total of 8,360,673 changed files in this snapshot, or about one third of the total dataset.

The results of our study, as well as the values from the previous study, are shown in Table 2. The statistical values (mean, median, max, and min) are computed using the most logical container for each question. For example, the container for classes are projects, the container for methods are classes, etc. As can be seen, most values differ between the studies. This is to be expected, as there are over 11 times more projects in our study. However, note that the general trends are similar and in particular the order of magnitude between rows is maintained.

For our version of the study, some of the values in the max column seemed like they might be too high. We manually verified[3] these values to be correct. Some interesting results:

- Despite the Java VM having a limit of 255 arguments for a method, we located a class constructor with 262 arguments!
- We located a test class with over 32k (hopefully generated) `void` methods in it to exhaustively test a method's 16-bit integer argument.
- A compiler-generated X10 file with over 7k local variables.
- A class with 10k static fields as constant strings.

The one task where we differ substantially is for nested classes. Note the mean value is almost 1k times higher. We believe this is because their study averages nested classes by number of methods. However we disagree with this, as the most common container for a nested class is another class. Thus we opted to compute this value slightly different.

6.2 Comprehension of Mining Tasks

In this section, we outline a small controlled experiment to determine if our proposed framework and language extensions make it easier to understand source code mining tasks. Each participant was shown, one at a time, a set of 5 source code mining tasks written in Boa. For each task, they were asked to describe in their own words what the task does. They were given up to five minutes to study each task and forced to move on if no answer was given after five minutes. The five tasks were:

[2] This is a limitation of the data and types available in Boa, and not of the visitor syntax described in this paper.

[3] `http://goo.gl/bwGGC` `http://goo.gl/jf0Fy` `http://goo.gl/zuYoh` `http://goo.gl/ZgamQ`

	Total		Mean		Median		Max		Min	
Question	**Boa**	**Treasure**	**Boa**	**Treasure**	**Boa**	**Treasure**	**Boa**	**Treasure**	**Boa**	**Treasure**
Classes	11,822,321	270,973	503.68	96.8	89	33	139,668	2,071	1	1
Static Classes	569,501	7,368	24.25	6.7	0	0	23,744	1,035	0	0
Anonymous Classes	3,772,130	29,237	0.05	0.04	0	0	724	136	0	0
Nested Classes	1,218,213	14,270	51.86	0.06	3	0	30,576	61	0	0
`assert` Statements	612,166	2,047	0.01	0	0	0	374	9	0	0
Methods	68,062,962	938,779	5.89	3.5	2	4	32,774	1,175	1	1
Static Methods	5,696,065	231,647	0.48	0.36	0	0	4,853	289	0	0
Methods (interfaces)	4,712,116	84,130	6.13	3.4	3	3	10,000	558	1	1
Method Arities	66,778,747	544,324	1.59	1.5	1	1	262	30	1	1
`void` return Methods	35,988,971	414,953	3.54	5.1	2	3	32,772	1,172	1	1
Methods Returning Arrays	1,334,259	24,744	1.87	2	1	1	383	137	1	1
non-`void` return Methods	32,073,991	523,826	4.93	5.8	2	3	4,854	888	1	1
Fields	31,682,721	448,898	2.68	1.9	0	0	10,000	1,457	0	0
`this` Expressions	51,933,214	840,937	0.72	2.2	0	1	6,294	785	0	0
Static Fields	10,949,191	154,067	0.93	0.7	0	0	10,000	1,457	0	0
Volatile Fields	48,471	492	0	0	0	0	97	9	0	0
Conditional Statements	118,557,128	620,419	1.63	0.76	0	0	5,294	750	0	0
`String` Fields	6,425,161	231,647	0.54	0.3	0	0	3,473	432	0	0
`try` Statements	14,080,420	93,714	0.19	0.11	0	0	1,722	90	0	0
Exceptions Thrown From `catch`	4,559,274	110,740	0.3	0.26	0	0	34	5	0	0
Exceptions	12,631,996	818,358	0.17	0.9	0	0	1,086	40	0	0
Local Variables	79,057,404	818,358	1.09	0.87	0	0	7,005	1,055	0	0

Table 2. Reproducing a portion of the Treasure study [11], at a much larger scale.

Q1 Count AST nodes (Section 4)

Q2 Assert use over time (Table 1, Assert)

Q3 Annotation use, by name (Table 1, Annotations-use)

Q4 Type name collector, by project and file (not shown)

Q5 Null check (Section 2, motivating example)

These answers were graded on a fixed set of criteria. For each question, we determined a list of criteria that must all be mentioned in order for the answer to be marked correct. For example, for Q1 they had to mention counting only AST nodes (not all nodes) and grouping the count by project.

Then they were shown the same set of 5 tasks again in a random order, only this time instead of a free-form entry they were given a choice of four descriptions and asked to choose the one that best fit. Only one of the four descriptions was accurate while the other four varied slightly (to make them inaccurate). For example, for Q1 only half the responses mention grouping by project. Also only two responses mention counting only AST nodes.

The results are shown in Table 3. A 'Y' indicates the participant answered correctly, both in the free-form and the multiple choice. Similarly a 'N' indicates they answered incorrectly in both. An entry marked '-Y' indicates their free-form answer was incorrect while their multiple choice was correct. Conversely, a '+N' indicates their free-form answer was correct and multiple choice answer incorrect. For the multiple choice, they were also given a choice of 'I am not sure what this task does' which is indicated in the table as '?'. We count this as an incorrect answer.

On average it took 16 minutes to study these five tasks, or around 3 minutes to comprehend a mining task in Boa. The accuracy of the comprehension was at 77.5% on average. Note however that one of the tasks in particular (Q1) seemed to give difficulty. Feedback suggested they failed to understand the semantics of the wildcard. Excluding that task, the accuracy jumps to over 90%.

We repeated this experiment with the same participants six months later. In the repeated experiment, the same five mining tasks were used as in the previous experiment, but this time the source code implementing the tasks was Java+Hadoop code (similar to Figure 1). The results are shown in Table 4. Note that all results were anonymized so rows do not correlate to rows in Figure 3.

Q1	Q2	Q3	Q4	Q5	Total	Time
N	Y	Y	Y	Y	80%	12m32s
-Y	Y	Y	Y	Y	100%	11m22s
?	Y	Y	Y	Y	80%	19m22s
-Y	Y	Y	Y	Y	100%	18m21s
?	+N	Y	Y	N	40%	11m40s
N	Y	Y	Y	-Y	80%	23m01s
N	-Y	Y	Y	Y	80%	16m10s
N	+N	-Y	-Y	Y	60%	14m50s
				Mean	**77.5%**	**15m55s**

Table 3. Controlled experiment on comprehensibility of source code mining tasks in Boa.

Q1	Q2	Q3	Q4	Q5	Total	Time
-Y	-Y	N	-Y	-Y	80%	23m44s
?	-Y	-Y	-Y	N	60%	10m50s
-Y	Y	+N	Y	-Y	80%	23m48s
N	Y	N	-Y	N	40%	12m07s
N	-Y	N	N	N	20%	12m08s
-Y	Y	Y	Y	Y	100%	15m52s
N	N	Y	-Y	-Y	60%	18m14s
-Y	+N	Y	N	Y	60%	11m17s
				Mean	**62.5%**	**16m**

Table 4. Controlled experiment on comprehensibility of source code mining tasks in Java+Hadoop.

Again, participants spent 16 minutes on average to study these tasks. This time however, the accuracy of comprehension was lower at 62.5%, almost 15% lower than the Boa survey! Another interesting result was the number of '-Y' responses. There were 15 such responses in the second survey compared to only 6 in the Boa survey. This may indicate more guessing or possibly a memory effect where they recalled the answer from taking the Boa survey six months earlier.

The results from these studies are extremely promising for two reasons. First, it gives insight that in only a few minutes most peo-

ple can comprehend a source code mining task using our approach. Second, this comprehension comes with **no training at all** in the new language features! Based on the feedback, we believe that even a short training session on Boa's language features would have helped the participants understand Q1 better.

6.3 Threats to Validity

For our usability evaluation in Section 6.1.1, there is potential construct bias as all of the code for our lines of code comparison in Table 1 was written by us. This was unavoidable as at the time no other researchers had access to our infrastructure.

The results of our reproduction of the Treasure study in Section 6.1.2 may not generalize to Java development practices in industry, as all of the code in our study comes from open-source. This same threat applies to the original study [11]. We avoid generalizing our results and instead focus on if the trends we observe are similar to the trends the previous study observed.

Our comprehension study in Section 6.2 suffers from selection bias as all participants were graduate students. We try to offset this bias by selecting participants from several sub-fields of SE/PL. The study also suffers from testing effects, since each task is given to the participants twice. We offset this effect by randomizing the presentation order the second time tasks were shown. There is also possible construct bias as we chose which tasks to present and might inadvertently select only simple tasks. To counter this, we chose what we considered to be a range from easy to difficult tasks. Finally, there are additional testing effects since the Java+Hadoop portion of the survey was performed after the Boa portion. This could actually bias the results in favor of the Java+Hadoop approach.

7. Related Works

Source code analysis is often performed using a visitor-style pattern [9]. The visitor pattern is intended to allow easily adding additional functionality to a hierarchy of types, without having to modify each type. This is typically accomplished via a double-dispatch where each type to be traversed contains an `accept` method and the new analysis contains `visit` methods. By default visitors perform a depth-first traversal of the tree. There are other forms of the pattern, such as hierarchical visitors [1] which allow controlling the traversal and visitor combinators [34] to compose more complex visitors. There are also reusable visitor pattern libraries [26]. Other approaches make use of visitors, such as Ovlinger and Wand who define a language for recursive traversals [29].

Our language is similar to many of these approaches, however while these approaches are typically for object-oriented languages our host language has no notion of object (only simple record types). Visitors make use of dynamic dispatch in the underlying language, which is not available in procedural languages like Boa. Also, since there is no notion of inheritance, the number of types in the language are fixed, making the analysis in our compiler implementation much simpler and allowed for the optimization mentioned in Section 5.

Orleans and Lieberherr provide the language DJ [27], a purely Java-based library implementation of Demeter/Java [28]. In DJ, users provide a traversal strategy and declare visitors with before and after visit methods, similar to our approach. DJ's implementation uses reflection to implement the traversals, while our implementation uses the `DefaultVisitor` and has no reflection in the source or generated code.

Both the work on DJ [27] and recursive traversals [29] provide syntax for specifying traversals separate from the visitor code. Our approach provides a default depth-first traversal and if users need a custom traversal strategy they must specify it intermixed with the visitor code by using `stop` statements and `visit` calls. In the future we may investigate syntax for separating custom traversal strategies from the visitor syntax.

Martin *et al.* describe a program query language (PQL) [20] for easily analyzing source code. They provide a fixed set of events in the language, such as method call or field access and allow queries on those events. They provide static and dynamic matching algorithms. The query language lacks a visitor syntax.

There are also interactive tools for querying source code using natural language queries [18] and custom languages such as JQuery [15]. Since these tools are interactive, they are designed for searching a single codebase and not for mining source code across a large number of projects.

The Sourcerer project [19] provides project metadata source code for over 18k Java projects. Their data is stored in a SQL database, allowing for standard SQL queries on that data. They provide data on single snapshots of projects, including source code information which is represented in the database as entities and relationships. Entities include declarations, type references, and local variables. Relationships include full type resolution and binding of the entities, which our approach does not currently support. The Treasure study [11] built a database containing source code for over 2k projects. They take source code from releases of each project, and map it into their database schema. This schema is capable of representing the entire source code, down to the expression level. Bevan *et al.* proposed a centralized database and data schema called Kenyon [4] for storing mined software repository information. They provide an SQL interface for querying this dataset. All three of these approaches use SQL for mining source code, which gives the benefit of easily performing joins. However source code queries often require recursion (over the graph structure of the data), which is cumbersome to express in SQL [12].

Hajiyev *et al.* describe CodeQuest [12], which uses safe Datalog to query source code information. They map the Datalog queries to standard SQL and query a relational database containing source code information. Unlike SQL, Datalog allows easily specifying recursive style queries but lacks the visitor pattern that is familiar to researchers who have worked on or studied compilers and source code analysis previously.

Our previous work on the Boa language and infrastructure [6] provided a domain specific language for querying metadata on over 600k projects and an efficient infrastructure for executing those queries. The language abstracted away details of the underlying infrastructure such that users did not need to be aware of how to parallelize their queries. It also provided a set of domain-specific types for mining software repositories. Boa was extended to support source code mining on millions of Java files, however the language lacked the simple abstractions for easily traversing the structure of the data to perform source code mining tasks. This work has nicely filled that research gap.

8. Conclusion

Mining source code in large-scale software repositories should be easier! It is important for answering a large number of research questions [7, 8, 10, 11, 13, 17, 21, 23–25, 31]. In particular, having full history information for source code is necessary for research on fault prediction [10, 13, 17, 23], change dependency and change coupling analyses [14, 35], and temporal analysis on API and object usage [22, 33] among others.

This work described new domain-specific features to help with mining source code. Although host languages for mining source code may not always be object-oriented, we show how to build support for abstract syntax tree traversal in a style reminiscent of the familiar visitor design pattern. These features have a familiar look and semantics, but are flexible enough to support over 40 different mining tasks from two previous studies. We also give

insights into the ease of comprehending tasks written using these domain-specific language features, which showed that over 77% of tasks can be understood in about 3 minutes even with no prior training in the new language features.

To date, previous works offer only a sub-set of: full source code history, enough data for mining down to the expression level, and being capable of scaling to a large number of projects. No previous work offers all of these features. The implementation presented in this work supports all of these features. We provide a flexible data schema for representing source code, including full history information and entities down to the expression level.

Since these features are now available in the Boa research infrastructure, and are actively being used on a daily basis, in the next few years we anticipate having hundreds more example uses. Feedback from this use would help drive their evolution and provide a larger validation of their design in practice.

Acknowledgments

This work was supported by NSF grants CCF-13-49153, CCF-13-20578, TWC-12-23828, CCF-11-17937, CCF-10-17334, and CCF-10-18600.

References

[1] Hierarchical visitor pattern, c2 pattern repository. `http://c2.com/cgi/wiki?HierarchicalVisitorPattern`, 2012.

[2] Sourceforge website. `http://sourceforge.net/`, 2012.

[3] Apache Software Foundation. Hadoop: Open source implementation of MapReduce. `http://hadoop.apache.org/`, 2013.

[4] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with Kenyon. In *ESEC/FSE'05: 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, 2005.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: 6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[6] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE'13: 35th International Conference on Software Engineering*, pages 422–431, 2013.

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. A large-scale empirical study of Java language feature usage. Technical report, Iowa State University, 2013.

[8] M. Gabel and Z. Su. A study of the uniqueness of source code. In *FSE'10: 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 147–156, 2010.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26 (7):653–661, 2000.

[11] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *ESEM'10: International Symposium on Empirical Software Engineering and Measurement*, pages 11:1–11:10, 2010.

[12] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: scalable source code queries with datalog. In *ECOOP'06: 20th European conference on Object-Oriented Programming*, pages 2–27, 2006.

[13] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE'09: 31st International Conference on Software Engineering*, pages 78–88, 2009.

[14] K. Herzig and A. Zeller. Mining cause-effect-chains from version histories. In *ISSRE'11: 22nd IEEE International Symposium on Software Reliability Engineering*, pages 60–69, 2011.

[15] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD'03: 2nd international conference on Aspect-oriented software development*, pages 178–187, 2003.

[16] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[17] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In *ICSE'07: 29th International Conference on Software Engineering*, pages 489–498, 2007.

[18] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *ASE'11: 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 376–379, 2011.

[19] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, April 2009.

[20] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA'05: 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, 2005.

[21] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *TOSEM: ACM Transactions on Software Engineering and Methodology*, page To Appear, 2013.

[22] Y. Mileva, A. Wasylkowski, and A. Zeller. Mining evolution of object usage. In *ECOOP'11: 25th European Conference on Object-Oriented Programming*, 2011.

[23] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE'05: 27th International Conference on Software Engineering*, pages 284–292, 2005.

[24] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR'05: International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[25] S. Okur and D. Dig. How do developers use parallel libraries? In *FSE'12: 20th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 54:1–54:11, 2012.

[26] B. C. d. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA'08: 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 439–456, 2008.

[27] D. Orleans and K. J. Lieberherr. Dj: Dynamic adaptive programming in java. In *REFLECTION'01: 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 73–80, 2001.

[28] D. Orleans and K. J. Lieberherr. DemeterJ. Technical report, Northeastern University, 2001. URL `http://www.ccs.neu.edu/research/demeter/DemeterJava/`.

[29] J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *OOPSLA'99: 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–81, 1999.

[30] C. Parnin, C. Bird, and E. Murphy-Hill. Adoption and use of Java generics. *Empirical Software Engineering*, pages 1–43, 2012.

[31] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *FSE'12: 20th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 33:1–33:11, 2012.

[32] H. Rajan, T. N. Nguyen, R. Dyer, and H. A. Nguyen. Boa website. `http://boa.cs.iastate.edu/`, 2012.

[33] G. Udding, B. Dagenais, and M. P. Robillard. Temporal analysis of API usage concepts. In *ICSE'12: 34th International Conference on Software Engineering*, pages 804–814, 2012.

[34] J. Visser. Visitor combination and traversal control. In *OOPSLA'01: 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 270–282, 2001.

[35] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE'04: 26th International Conference on Software Engineering*, pages 563–572, 2004.