Empir Software Eng
DOI 10.1007/s10664-008-9077-5

Toward an understanding of bug fix patterns

Kai Pan · Sunghun Kim · E. James Whitehead Jr.

Editors: A. Hassan, S. Diehl and H. Gall

Abstract Twenty-seven automatically extractable bug fix patterns are defined using the syntax components and context of the source code involved in bug fix changes. Bug fix patterns are extracted from the configuration management repositories of seven open source projects, all written in Java (Eclipse, Columba, JEdit, Scarab, ArgoUML, Lucene, and MegaMek). Defined bug fix patterns cover 45.7% to 63.3% of the total bug fix hunk pairs in these projects. The frequency of occurrence of each bug fix pattern is computed across all projects. The most common individual patterns are MC-DAP (method call with different actual parameter values) at 14.9–25.5%, IF-CC (change in if conditional) at 5.6–18.6%, and AS-CE (change of assignment expression) at 6.0–14.2%. A correlation analysis on the extracted pattern instances on the seven projects shows that six have very similar bug fix pattern frequencies. Analysis of *if* conditional bug fix sub-patterns shows a trend towards increasing conditional complexity in *if* conditional fixes. Analysis of five developers in the Eclipse projects shows overall consistency with project-level bug fix pattern frequencies, as well as distinct variations among developers in their rates of producing various bug patterns. Overall, data in the paper suggest that developers have difficulty with specific code situations at surprisingly consistent rates. There appear to be broad mechanisms causing the injection of bugs that are largely independent of the type of software being produced.

Keywords Software bugs · Bug fix changes · Categorization of software faults · Software fault taxonomy · Software fault · Software error · Causes of software bugs · Algorithms · Measurement · Experimentation

K. Pan · E. J. Whitehead Jr.
Department of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, USA

K. Pan
e-mail: pankai@cs.ucsc.edu

E. J. Whitehead Jr.
e-mail: ejw@cs.ucsc.edu

S. Kim (✉)
Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
e-mail: hunkim@gmail.com

1 Introduction

The presence of bugs is a persistent quality of human created software. This is not intentional. From the dawn of software engineering and the coining of the software crisis, the creation of bug-free software has been a goal for engineer and researcher alike. One useful starting point towards the creation of bug-free software is a characterization of the problem. Specifically, we would like to know the most common kinds of software bugs for a specific system, and whether the frequency of bug kinds is similar across multiple systems. Once this information is known, it is possible to rank the kinds of bugs from most to least common, and then focus research attention on reducing the most common types of bug. Of course, this only makes sense if the most common bug types are similar across a broad range of systems.

Software engineering as a discipline still does not have general consensus on which kinds of software bugs are most common, and whether bug types have similar frequency distributions across multiple systems. The reason is not a deficit of research, but a lack of uniformity. Efforts to create taxonomies of common bug types date back to at least Endres, whose 1975 paper classifies faults in operating system code (Endres 1975). Since then, studies focused on categorizing bugs are generally consistent with the following pattern (Basili and Perricone 1984; Leszak et al. 2000; Li et al. 2006; Ostrand and Weyuker 1984; Perry and Stieg 1993; Potier et al. 1982). Researchers create a new taxonomy of bug types (or, less frequently, use an existing one), examine one (or very rarely two) software systems, and then categorize its bugs using the taxonomy. No examination is performed of human error that may creep in due to human subjectivity in assigning bug reports to bug categories. The end result (and current state of knowledge) is predictable. There are multiple taxonomies—each bearing some resemblance to the others—which are largely incommensurable (Marick 1990 exemplifies this point). The results are difficult to compare at anything but a broad level. Due to the inherent scalability limitations of manual fault classification, no one study has sufficient data to drawn conclusions concerning the similarity of bug type frequencies across projects.

This paper presents an alternative approach in which software history data is mined to find patterns in bug fix changes, thereby automatically categorizing bugs. Two characteristics of contemporary software development make this possible. First is the prevalent use of software configuration management (SCM) systems to record a fine-grain history of software changes during the development and maintenance of a software system. Second is the widespread convention of developers noting that a specific change fixes a bug when they write the change log message for a SCM commit. Together, they permit the automatic identification of bug fix changes in an SCM change history. Once the bug fix changes are known, it is possible to write a program that automatically examines them to find matches with existing categories of bug fixes. In this way, it is now possible to automatically classify bugs into specific bug types, avoiding the traditional problems of human bug categorization. The technique is automatic, and hence scalable far beyond techniques that require a human analyst. Since no human analyst is involved, the categorizations are reliable and stable; a researcher at a different site can independently reproduce the bug categorizations. Since the technique is scalable and reliable, it is now possible to examine whether bug type frequencies are similar across multiple projects.

The primary contribution of this paper is its definition of 27 automatically extractable bug fix patterns in Java software. These patterns are based on the syntax components and context of the source code involved in bug fix changes. Patterns were initially identified from a manual analysis of the bug fixes in open source Java projects. Subsequently, a bug

fix pattern extraction tool was developed that can automatically identify bug fix patterns. This tool is used to analyze the change history of seven open source projects, including two that were not used to develop the initial taxonomy of patterns.

These patterns, and the tool that extracts them from SCM repository data, can be viewed as a new kind of scientific instrument that provides improved visibility into the kinds of bugs fixed during a project. We use this new instrument, the bug fix extractor, to examine a series of research questions. Since source code can, at times, have a complex structure, one concern with automatic identification of bug fix patterns is what percentage of bug fixes matches a pattern. If very few bug fix changes actually match a pattern, further analysis of the patterns would not be useful. Assuming many bug fix changes actually do match a pattern, then the relative frequency of patterns is important, since this gives insight into the most common types of errors occurring in software systems. These observations motivate the first two research questions.

1. *Research question 1*. What percentage of bug fixes can be automatically categorized using the bug fix pattern extractor approach?
2. *Research question 2*. What are the most common types of bug fix patterns? Since bug fix patterns are tightly correlated with bugs, another way to view this question is, what are the most common types of bug?

Once bug fix pattern frequencies are known for several projects, we would like to know if the frequencies are similar across them. If the frequencies are similar, this would suggest that specific language features and constructs are an underlying cause of these error types, independent of application domain. However, if the frequencies vary, this would suggest that it is characteristics of the application domain which are the underlying driver for these errors. This leads to the third research question.

3. *Research question 3*. Is the frequency of bug fix pattern categories similar across multiple projects? For example, do fixes of *if* conditional errors occur with the same frequency in multiple projects, or do they vary substantially across projects?

In exploring research question 2, *if* conditionals are found to be the second most common bug fix pattern. This led to the development of a series of sub-patterns related specifically to *if* conditionals, and the fourth research question.

4. *Research question 4*. Are *if* conditional sub-pattern frequencies similar across projects? For example, do repairs to if conditionals that add (or remove) a variable occur with the same frequency across projects?

Since bugs are initially injected into code by individual developers, it raises the question of whether all developers commit bugs at the same rate. This leads to the final research question.

5. *Research question 5*. Is the frequency of bug fix patterns similar across developers within a project? For example, do all developers create *if* conditional bugs with the same frequency? Do developers have certain bug types they are more likely to create?

The primary motivation for these research questions is improving our understanding of the frequency characteristics of bug injection in projects, and starting the process of understanding the underlying causal factors. To the extent that the repair of bugs in bug fix changes provides a window on the frequency of bugs in the code, this data provides understanding of the relative frequency of bug types.

The main drawback of the bug fix patterns approach stems from its automation. As shown below in Fig. 2, the bug fix pattern extractor tool is not able to assign every bug fix to a category, and hence some bugs cannot be classified using this technique. Still, the bug frequency data collected from the 45.7–63.6% of bug fix hunk pairs that *can* be classified has several practical uses. Mutation testing is an approach for testing software that automatically seeds bugs in software using mutation operators. One concern that arises in mutation testing is the frequency at which bugs should be added to the code base. Recent research has mined open source software repositories for bug frequency data use to train mutation operators, albeit using a manual approach (Duraes and Madeira 2006). Engineers performing code inspections can use bug fix pattern frequency data to drive their code inspection activities, focusing attention on code patterns that fit those of high frequency bug fix patterns. Programming language designers could use bug fix pattern data to develop new language features that reduce or eliminate commonly occurring classes of bugs. Finally, developers can improve self awareness of the types of bugs they most frequently add to the code, potentially altering their coding practices to make these kinds of bugs less likely.

The paper is organized as follows. The first three sections describe the tools and pattern categories used to explore the research questions. Specifically, Section 2 explains the method used to identify bug fix changes in the change history of a project, Section 3 provides a catalog of bug fix patterns, and Section 4 describes the bug fix pattern extractor tool. The next four sections address the research questions. Section 5 examines the percentage of bug fix changes that match a big fix pattern. Section 6 then presents the frequency distribution of bug fix changes, and examines their similarity across projects. Section 7 examines bug fix sub-patterns of *if* conditionals, and Section 8 considers the distribution and similarity of bug injection patterns for five developers in the Eclipse project. The paper finishes with Section 9 discussing related work, Section 10 describing threats to validity, and Section 11 giving conclusions.

2 Identification of Bug Fixes

In this paper, the term bug describes a mistake made in software source code. It is a programmer error that manifests itself in the form of incorrect source code. We prefer "bug" to the equivalent term "fault" since it is more colorful, and is in widespread professional use.

Traditionally, bugs are identified in software by examining test executions for incorrect output, performing software inspections, or running static analysis tools. Our method for bug identification is somewhat different, in that we assume that developers have been using these traditional methods for bug identification throughout a project's evolution, and have been fixing the buggy code. For us, a bug is whatever a project developer has called a bug, via their action of declaring certain changes to be bug fixes. Hence, our task is to retrospectively discover which changes involved bug fixes, and which ones performed other kinds of software update.

A project revision containing changes to repair buggy code is a bug fix revision. The version before the bug fix revision is the bug version, and the version after the bug fix revision is the fix version. We identify bug fix revisions based on the log messages that are supplied with a revision. There are two approaches for identifying bug fix revisions. The first one is to look for keywords like "fixed" or "bug" in the change log, a technique introduced by Mockus and Votta (2000); the second approach is looking for references to

bug reports like "#42233" as introduced by Fischer et al. (2003) and by Cubranic and Murphy (2003). We generally use the first approach in this paper. That is, if a change log contains "bug", "fix", or "patch", the revision is found to be a bug fix revision. For the Eclipse project, we instead use the identifier for a bug report, since this project consistently uses bug tracking software. For Scarab, we also search for the keywords "issue number" in addition to "bug," "fix," and "patch," since this project, being a bug tracker, uses bug tracking software. One problem that arises is Scarab uses bug tracking software to track new features as well as bug fixes. As a result, the bug fix extractor algorithm pulls out both bug fix changes and new feature changes for this project, and hence results in this paper for Scarab should be interpreted as patterns in bug fix changes and new feature additions.

Changes to each file involved in a bug fix revision consist of bug fix hunk pairs. The bug fix hunk pairs are computed by the GNU diff (GNU 2003) tool, and represent the text difference of a file between the bug version and the fix version. The bug hunk indicates a code section in the bug version that is modified or deleted in bug fix revision, and its corresponding fix hunk indicates the code in the fix version that fixes the bug. We call a bug fix hunk pair a bug fix change. There are three kinds of bug fix changes: modification, addition, or deletion, as shown in Fig. 1. Modification bug fix changes have both changed code in the bug version (bug hunk) and changed code in the fix version (fix hunk). Addition bug fix changes only have a fix hunk, and deletion bug fix changes only have a bug hunk.

3 Bug Fix Patterns

Bug fix hunk pairs contain just the text difference between a section of code in the bug version and the corresponding code in the fix version. Some differences look random to us, others do not. We can sense some patterns in the changes from the changed syntax components in the bug fix hunk pair, the context the hunk code is in, and the code near the hunks.

To define a set of bug fix patterns, we manually analyzed part of the bug fix change history of five open source projects, ArgoUML, Columba, Eclipse, JEdit, and Scarab. The analysis involved inspecting the bug hunks and the corresponding fix hunks in the bug fix revisions, and classifying bug fix changes into different patterns (bug types) based on the syntax component kinds in the hunk pairs and their containing program context. These identified bug fix patterns are grouped into several categories including If-related (IF), Method Call (MC), Loop (LP), Assignment (AS), Switch (SW), Try (TY), Method Declaration (MD), Sequence (SQ), and Class Field (CF). We note that the If-related, Loop,

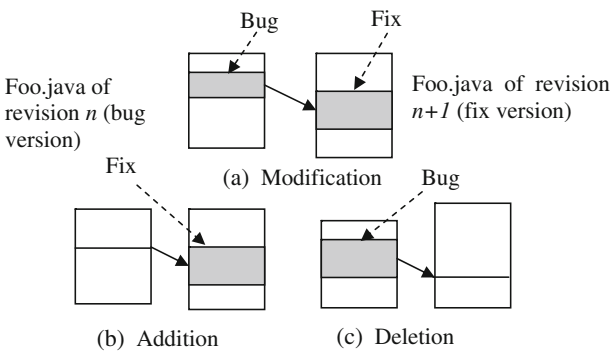


Fig. 1 Three kinds of bug fix changes to a file

and Switch categories together comprise a group of bug fix patterns concerned with logic errors, discussed further in Section 6.2.

The left hand side of Table 2 (presented below in Section 6.1) presents an overview list of bug fix patterns. In the following subsections, we present a catalog of observed bug fix patterns. Each pattern begins with a title, and an abbreviation code. A brief description is given of the pattern, followed by one or more source code examples taken from the examined projects. Example code with a leading "−" is from the bug hunk, while code with leading "+" is in the fix hunk. Unmarked code (without leading "−" or "+") is found in both bug and fix versions.

3.1 If-related (IF)

3.1.1 Addition of Precondition Check (IF-APC)

Description This bug fix adds an *if* predicate to ensure a precondition is met before an object is accessed or an operation is performed. Without the precondition check, there may be a NullPonterException error or an invalid operation execution caused by the buggy code. This kind of bug occurs when the developer did not consider the precondition before an operation is performed.

Example:

```
− lastChunk.init(seg,expander,x,styles,
−             fontRenderContext, context.rules.getDefault());
+ if (!lastChunk.initialized)
+       lastChunk.init(seg,expander,x,styles,
+             fontRenderContext, context.rules.getDefault());
```

3.1.2 Addition of Precondition Check with Jump (IF-APCJ)

Description This bug fix pattern is similar to the previous one, IF-APC, except it adds an *if* statement that encloses a jump statement, such as return, continue, or break. This causes the fixed code to skip the remaining code in the block if the precondition is not satisfied.

Example:

```
+ if (!comp.isShowing())  return true;
for (; ; ) {
   if (comp instanceof View) {
        ((View)comp).processKeyEvent(evt);
```

3.1.3 Addition of Post-condition Check (IF-APTC)

Description The fix code adds an *if* statement after an operation to check the result from the operation. The value modified by the operation must be used in the *if* condition. The bug fix extractor verifies this by performing a data-flow analysis between the operation and the *if* condition. One example is error checking, as shown in the example below. This kind of bug occurs when a developer fails to consider different return results from an operation.

Example:

```
− parentFolder.addFolder(name);
+ FolderTreeNode folder = parentFolder.addFolder(name);
+ if (folder = = null) success = false;
```

3.1.4 Removal of an If Predicate (IF-RMV)

Description The fix removes an if predicate from the code it encloses. This bug occurs where there is an unnecessary condition check. There are many fewer instances of this bug-fix pattern than of IF-APC.

Example:

```
− if (seg.array = = null || seg.array.length < len)
    seg.array = new char[len];
```

3.1.5 Addition of an Else Branch (IF-ABR)

Description The bug fix adds an else branch to an *if* statement to cover a condition not previously considered.

Example:

```
  else if (aname = = “PLUGIN”) depPlugin = value;
+ else if (aname = = “SIZE”) size = Integer.parseInt(value);
```

3.1.6 Removal of an Else Branch (IF-RBR)

Description This bug fix pattern is the opposite of IF-ABR. This bug fix removes an else branch, thereby freeing the code in the else body from the constraint of the *if* condition. There are many fewer instances of this bug-fix pattern than of IF-ABR.

Example:

```
− else addOptionGroup(pluginsGroup,rootGroup);
+ addOptionGroup(pluginsGroup,rootGroup);
```

3.1.7 Change of If Condition Expression (IF-CC)

Description This bug fix change fixes the bug by changing the condition expression of an *if* condition. The previous code has a bug in the *if* condition logic. This pattern is further explored in Section 5.3, which presents sub-patterns describing common changes within the conditional expression.

Example:

```
− if (getView().countSelected() = = 0) {
+ if (getView().countSelected() <=1) {
```

3.2 Method Call (MC)

3.2.1 Method Call with Different Number of Parameters or Different Types of Parameters (MC-DNP)

Description The bug fix changes a method call by using a different number of parameters, or different parameter types. This may be caused by a change of method interface, or use of an overloaded method.

Example:

```
− query = getLuceneQuery(filter.getFilterRule());
+ query = getLuceneQuery(filter.getFilterRule(), analyzer);
```

3.2.2 Method Call with Different Actual Parameter Values (MC-DAP)

Description The bug fix changes the expression passed into one or more parameters of a method call.

Example:

```
− tree.putClientProperty("JTree.lineStyle","Horizontal");
+ tree.putClientProperty("JTree.lineStyle","Angled");
```

3.2.3 Change of Method Call to a Class Instance (MC-DM)

Description The fix code calls a different member method of a class instance. This new method may have some name similarity to the one used in the bug version. This bug fix may be caused by a method being renamed or by a developer using an incorrect member method.

Example:

```
− Enumeration enum = windows.keys();
+ Enumeration enum = windows.elements();
```

3.3 Sequence (SQ)

3.3.1 Addition of Operations in an Operation Sequence of Method Calls to an Object (SQ-AMO)

Description The bug fix adds one or more method calls into a sequence of method calls to the same object. This kind of bug occurs when the developer missed one or more method calls.

Example:

```
importDeclaration.setSourceRange();
− importDeclaration.setOnDemand(importReference.onDemand);
+ importDeclaration.setName(name);
+ importDeclaration.setOnDemand(onDemand);
```

3.3.2 Removal of Operations from an Operation Sequence of Method Calls to an Object (SQ-RMO)

Description The opposite of SQ-AMO, this bug fix pattern removes one or multiple method calls in a sequence of method calls to the same object.

Example:

```
pathField.setPreferredSize(prefSize);
− pathField.addFocusListener(new FocusHandler());
```

3.3.3 Addition of Operations in a Field Setting Sequence (SQ-AFO)

Description This bug fix pattern is similar to SQ-AMO, except that the operation sequence involves setting object fields, not method calls.

Example:

```
cd.sourceEnd = sourceEnd;
cd.modifiers = modifiers & AccVisibilityMASK;
+ cd.isDefaultConstructor = true;
```

3.3.4 Removal of Operations from a Field Setting Sequence (SQ-RFO)

Description This bug fix pattern is the opposite of SQ-AFO. The code in the bug version contains an unnecessary field setting operation.

Example:

```
− ref.sourceEnd = intStack[intPtr−];
ref.sourceStart = intStack[intPtr−];
```

3.3.5 Addition or Removal of Method Calls in a Short Construct Body (SQ-AROB)

Description This bug fix adds or removes method calls from a construct body, such as method body, if body, or while body that only contains two or three statements. Unlike the SQ-AMO and SQ-RMO patterns, SQ-AROB does not require the method calls involved to be the calls to the same object variables. This kind of bug fix occurs when the developer missed operations or included unnecessary operations in an operation sequence.

Example:

```
if (evt.getClickCount() = = 2) {
      jEdit.showMemoryDialog(view);
   + memory.repaint();
}
```

3.4 Loop (SQ)

3.4.1 Change of Loop Predicate (LP-CC)

Description The bug fix changes the loop condition of a loop statement. The previous code has a bug in the loop condition logic.

Example:

```
− while (!buffer._isLineVisible(line,index)) line−;
+ while (!buffer._isLineVisible(line,index) && line>0) line−;
```

3.4.2 Change of the Expression that Modifies the Loop Variable (LP-CE)

Description The bug fix changes the expression that modifies the loop variable or adds a statement that modifies the loop variable.

Example:

```
while (comp ! = null) {
    …
  + comp = comp.getParent();
}
```

3.5 Assignment (AS)

3.5.1 Change of Assignment Expression (AS-CE)

Description The bug fix changes the expression on the right hand side of an assignment statement. The expression on the left-hand side is the same in both the bug and fix versions.

Example:

```
− interfacesRange[1] = bodyStart − 1;
+ interfacesRange[1] = superinterfaceEnds[
+                                superinterfaces.length − 1];
```

3.6 Switch (SW)

3.6.1 Addition/Removal of Switch Branch (SW-ARSB)

Description The bug fix adds or removes a case from a *switch* statement. The previous code missed a case situation or includes an unnecessary case situation.

Example:

```
+ case ClassFileStruct.ClassTag:
+    name = extractClassReference(
+              constantPoolOffsets,reader,i);
```

3.7 Try (TY)

3.7.1 Addition/Removal of Try Statement (TY-ARTC)

Description The bug fix adds a *try*/*catch* statement to enclose a section of code, or removes a *try*/*catch* construct from the code in the bug hunk.

Example:

```
+ try {
      mimePartTree = srcFolder.getMimePartTree(uid, wsc);
+}    catch (FileNotFoundException ex) {
+     return;
+}
```

3.7.2 Addition/Removal of a Catch Block (TY-ARCB)

Description The bug fix adds a catch block to a *try* statement, or removes a *catch* block from a *try* statement in the bug hunk. The previous code failed to capture a kind of exception caused by the code in the *try* block.

Example:

```
+} catch (InvocationTargetException ex) {
+    ex.getCause().printStackTrace();
+    throw ex;
```

3.8 Method Declaration (MD)

3.8.1 Change of Method Declaration (MD-CHG)

Description The bug fix changes the declared interface for a method. The interface change may increase or decrease the number of parameters, change parameter types, change the return type, or change a method access modifier. This kind of change usually also leads to changes at call sites to this method.

Example:

```
− public int fetchMessageCount() throws Exception
+ public int fetchMessageCount(WorkerStatusController
+                               worker) throws Exception
```

3.8.2 Addition of a Method Declaration (MD-ADD)

Description A method declaration is added in the fix version.

Example:

```
+ public void removeNotify(){
+    jEdit.setIntegerProperty("vfs.browser.splitter",
+          splitPane.getDividerLocation());
```

3.8.3 Removal of a Method Declaration (MD-RMV)

Description A method declaration is deleted from the bug version.

Example:

```
− public static Image getEditorIcon(){
− return ((ImageIcon)EDITOR_WINDOW_ICON).getImage();
−}
```

3.9 Class Field (CF)

3.9.1 Addition of a Class Field (CF-ADD)

Description A class field is added in the fix version.

Example:

```
+ private NameReference[] unknownRefs;
+ private int unknownRefsCounter;
```

3.9.2 Removal of a Class Field (CF-RMV)

Description A class field is removed from the bug version.

Example:

```
− private MarkerHighlight markerHighlight;
```

3.9.3 Change of Class Field Declaration (CF-CHG)

Description A class field declaration is changed in the bug fix revision.

Example:

```
− JPanel content = new JPanel(new BorderLayout());
+ JPanel content = new JPanel(new BorderLayout(12,12));
```

3.10 Ignored patterns

Besides the bug fix patterns identified above, there are still some bug fixes that have an obvious pattern, but are ignored because those bug fixes are trivial code changes and have insignificant semantic impact. Examples of these ignorable bug fixes include changes to comments, addition or removal of debug information, code cleanup, code formatting, addition or removal of output statements, and changes to import statements. We also omit bug fix patterns that require expensive program analysis to detect, including statement permutation, variable renaming, and removal of dead code.

4 Bug Fix Pattern Extractor

The bug fix pattern extractor tool consists of two major parts, a parser module and a pattern discovery module. The parser module is responsible for parsing the Java source code (using Eclipse JDT 2006) and collecting syntax information for each statement. The syntax information for a statement includes its AST tree and its structural context, such as if body, while body, or method body.

The pattern discovery module is responsible for recognizing the bug fix pattern from almost all bug fix hunk pairs. Since large bug fix hunk pairs generally look random and do not contain meaningful bug fix patterns, bug fix pattern analysis ignores large bug fix hunk pairs whose bug hunk or fix hunk contains more than seven statement lines. Specially, addition of a new file usually results in large bug fix hunk pairs, which are ignored by the extractor. Most bug fix hunk pairs (91% to 96%) are small ones, and hence ignoring large hunk pairs has minimal impact on the analysis.

The code syntax information from the bug hunk and fix hunk is used to determine what bug fix patterns this bug fix hunk pair contains. We describe the pattern discovery rule for several selected bug fix patterns below.

4.1 IF-APC (Addition of Precondition Check)

There is no *if* predicate in the bug hunk (the bug hunk can be empty), but the fix hunk does contain an *if* predicate. The code enclosed by the *if* predicate in the fix version has corresponding code in the bug version. If these conditions are met, the pattern discovery module believes that this bug fix hunk pair contains an IF-APC pattern instance.

4.2 MC-DM (Different Method Call to a Class Instance)

In the bug hunk there is one and only one method call to an object variable, and in the fix hunk there is a corresponding method call with a different method name to the same object variable. In this case, the pattern discovery module discovers an MC-DM pattern instance from the bug fix hunk pair.

4.3 SQ-AMO (Addition of Operations in an Operation Sequence of Method Calls to an Object)

In the fix hunk there is a method call to an object variable, but the same method call is not made in the bug hunk (the bug hunk can be empty). In code close to the method call in the fix version, there are other method calls to the same object variable that are found in both the bug and fix versions. If these conditions are met, the pattern discovery module discovers an SQ-AMO pattern instance.

4.4 TY-ARTC (Addition/Removal of Try Statement)

For simplicity, only the addition case is explained here. There is a *try* in the fix hunk, but no corresponding *try* in the bug hunk (the bug hunk can be empty). The statements in the *try* body in the fix version have corresponding statements, not enclosed in a *try* statement, in the bug version. Under these circumstances, the pattern discovery module discovers a TY-ARTC pattern instance.

5 Coverage of Bug Fixes by Bug Fix Patterns

Having described the bug fix patterns and the tool used to automatically detect them, the paper now shifts to consider the first research question concerning what percentage of bug fix patterns can be detected using the extractor tool described in the previous section.

5.1 Bug Fix Coverage Data

The pattern extractor tool analyzed the change history of seven open source projects, ArgoUML, Columba, Eclipse, JEdit, Scarab, Lucene, and MegaMek, summarized in Table 1.

Recall that a single SCM commit consists of changes to one or more files, and each file consists of one or more changed hunks. This suggests three ways to examine pattern coverage, at the commit, file and hunk level. The first, *commit coverage*, describes the percentage of SCM commits where at least one hunk in one file in the commit matches a pattern. This gives a sense of how many bug fix commits can be given at least a partial categorization (i.e., at least some of its hunks are categorized) using the patterns. As a result, this figure can be useful in comparing the efficacy of the bug fix patterns approach with other bug categorization work performed on commits or bug reports. The second, *file coverage*, examines the percentage of files that have at least one hunk containing a pattern. This provides insight into how many file level changes can be categorized. The final coverage type, *hunk coverage*, examines the percentage of hunks that contain at least one pattern. This provides visibility into the effectiveness of bug fix patterns in categorizing *all* hunk pairs. We expect file coverage to be larger since there is a greater chance that at least one (of usually many hunks) matches a pattern.

We examined file and hunk coverage for the bug fix changes in the change history of the projects in Table 1, with results displayed in Fig. 2. This permits us to answer the first research question, summarized below:

Pattern coverage—Research question #1
Commit coverage ranges from 66% to 91%.
File coverage ranges from 53% to 75%.
Hunk coverage ranges from 46% to 64%.
File coverage is greater than hunk coverage.

5.2 Discussion

The hunk pattern coverage data in Fig. 2 shows that the bug fix patterns account for approximately half of the hunk pairs within a project. This is encouraging, since it indicates

Table 1 Analyzed projects

Project	Period	No. of revisions	No. bug fix revisions
ArgoUML	01/1998–09/2005	4,685	1,310
Columba	11/2002–12/2005	2,362	797
Eclipse	06/2001–01/2006	6,394	2,807
JEdit	09/2001–01/2006	1,190	557
Scarab	12/2000–02/2006	2,962	535
Lucene	09/2001–02/2006	1,042	266
MegaMek	02/2002–09/2006	2,825	706

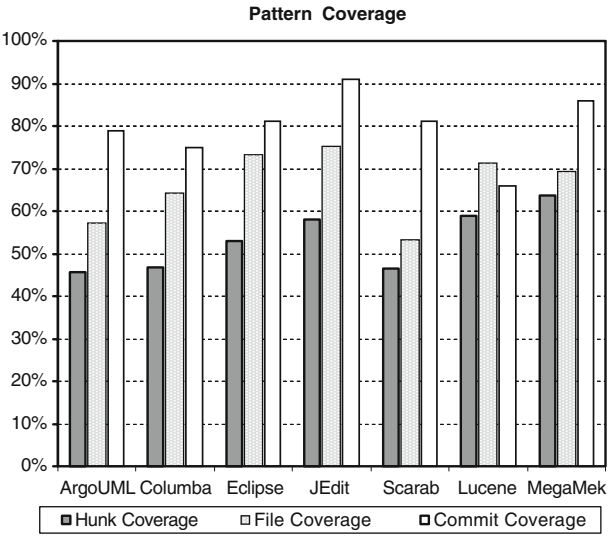


Fig. 2 Pattern coverage in the analyzed projects

a large number of bug fix hunk pairs can be categorized by the identified patterns, and this provides a large enough data set for further analysis. If hunk pattern coverage is too low, then performing any further analyses using the patterns would not be as useful, since there would be many hunk pairs that were unable to be categorized. At the same time, it indicates many bug fix hunk pairs have changes that either exhibit patterns not yet codified, are substantially random and not amenable to automatic classification, or are one of the ignored pattern types (e.g., comment changes).

The commit pattern coverage data shows that typically more than three quarters of all bug fix commits can be given at least a partial categorization. Since existing bug categorization research operates at the logical change level, commit coverage can be used to more directly compare the categorization capacity of bug fix patterns to this body of work.

In the remainder of the paper we examine characteristics of bug fix patterns, and it is worth remembering that these patterns encompass only about half of the bug fix hunk pairs, and between half and three quarters of all file changes observed in these projects. Especially for the bug fix pattern frequency data, hunk coverage indicates that the frequency values should be multiplied by the coverage value to find their overall contribution to bug production rates in a project. For example, the IF-CC pattern in ArgoUML is 10.8% of the observed bug fix patterns (value from Table 2). However, bug fix patterns cover only 46% of the hunk pairs in ArgoUML. As a result, IF-CC bugs are 4.97% of all hunk pairs in this project.

6 Cross-Project Similarity

The second research question asks about the most common types of bug fix patterns, and the third research question queries whether bug fix pattern frequencies are similar across projects. Addressing these questions involves first computing the frequency of bug fix patterns, and then computing measures to assess their similarity.

Table 2 Frequency of bug fix patterns in hunk pairs of analyzed projects

Category	Pattern Name	Short Name	ArgoUML (no., %)		Columba (no., %)		Eclipse (no., %)		JEdit (no., %)		Scarab (no., %)		Lucene (no., %)		MegaMek (no., %)	
Assignment (AS)	Change of assignment expression	AS-CE	685	8.5	159	6.0	1208	6.3	404	8.3	169	7.2	255	8.3	1120	14.2
			685	8.5	159	6.0	1208	6.3	404	8.3	169	7.2	255	8.3	1120	14.2
Class Field (CF)	Addition of a class field	CF-ADD	246	3	105	4.0	707	3.7	216	4.4	66	2.8	102	3.3	269	3.4
	Change of class field declaration	CF-CHG	230	2.9	85	3.2	422	2.2	94	1.9	44	1.9	144	4.7	227	2.9
	Removal of a class field	CF-RMV	132	1.6	42	1.6	208	1.1	108	2.2	42	1.8	106	3.4	61	0.8
			608	7.5	232	8.7	1337	7.0	418	8.6	152	6.5	352	11.4	557	7.1
If-related (IF)	Addition of an else branch	IF-ABR	78	1	29	1.1	208	1.1	88	1.8	22	0.9	17	0.6	103	1.3
	Addition of precondition check	IF-APC	318	3.9	148	5.6	1142	6.0	266	5.5	124	5.3	68	2.2	199	2.5
	Addition of precondition check with jump	IF-APCJ	295	3.7	90	3.4	726	3.8	164	3.4	34	1.5	60	1.9	196	2.5
	Addition of post-condition check	IF-APTC	131	1.6	102	3.8	285	1.5	131	2.7	37	1.6	16	0.5	89	1.1
	Change of if condition expression	IF-CC	868	10.8	149	5.6	3553	18.6	624	12.9	250	10.7	370	12.0	1157	14.7
	Removal of an else branch	IF-RBR	25	0.3	15	0.6	96	0.5	51	1.1	8	0.3	10	0.3	18	0.2
	Removal of an if predicate	IF-RMV	139	1.7	56	2.1	441	2.3	140	2.9	51	2.2	67	2.2	78	1.0
			1854	23	589	22.2	6451	33.9	1464	30.2	526	22.5	608	19.7	1840	23.4
Loop (LP)	Change of loop condition	LP-CC	114	1.4	23	0.9	297	1.6	59	1.2	31	1.3	84	2.7	230	2.9
	Change of the expression that modifies the loop variable	LP-CE	23	0.3	3	0.1	34	0.2	11	0.2	2	0.1	7	0.2	5	0.1
			137	1.7	26	1.0	331	1.7	70	1.4	33	1.4	91	3.0	235	3.0
Method Call (MC)	Method call with different actual parameter values	MC-DAP	1921	23.8	432	16.3	3416	17.9	725	14.9	596	25.5	515	16.7	1500	19.1
	Different method call to a class instance	MC-DM	75	0.9	59	2.2	256	1.3	105	2.2	27	1.2	105	3.4	224	2.8

	Method call with different number or types of parameters	MC-DNP	428	5.3	104	3.9	1365	7.2	233	4.8	151	6.5	166	5.4	181	2.3
			2424	30	595	22.4	5037	26.4	1063	21.9	774	33.1	786	25.5	1905	24.2
Method Declaration (MD)	Change of method declaration	MD-CHG	511	6.3	204	7.7	1108	5.8	289	6.0	149	6.4	176	5.7	275	3.5
	Addition of a method declaration	MD-ADD	540	6.7	254	9.6	1036	5.4	256	5.3	172	7.4	358	11.6	309	3.9
	Removal of a method declaration	MD-RMV	243	3	90	3.4	358	1.9	98	2.0	58	2.5	165	5.4	56	0.7
			1294	16	548	20.6	2502	13.1	643	13.2	379	16.2	699	22.7	640	8.1
Sequence (SQ)	Addition of operations in an operation sequence of field settings	SQ-AFO	18	0.2	3	0.1	132	0.7	31	0.6	1	0	5	0.2	707	9.0
	Addition of operations in an operation sequence of method calls to an object	SQ-AMO	418	5.2	203	7.6	562	2.9	267	5.5	103	4.4	62	2.0	458	5.8
	Addition or removal method call operations in a short construct body	SQ-AROB	255	3.2	99	3.7	376	2.0	203	4.2	77	3.3	24	0.8	67	0.9
	Removal of operations from an operation sequence of field settings	SQ-RFO	26	0.3	1	0.0	59	0.3	13	0.3	0	0	8	0.3	143	1.8
	Removal of operations from an operation sequence of method calls to an object	SQ-RMO	265	3.3	105	4.0	263	1.4	204	4.2	90	3.9	66	2.1	128	1.6
			982	12.2	411	15.5	1392	7.3	718	14.8	271	11.6	165	5.4	1503	19.1
Switch (SW)	Addition/removal of switch branch	SW-ARSB	3	0	9	0.3	305	1.6	28	0.6	0	0	72	2.3	55	0.7
			3	0	9	0.3	305	1.6	28	0.6	0	0	72	2.3	55	0.7
Try (TY)	Addition/removal of a catch block	TY-ARCB	41	0.5	51	1.9	450	2.4	23	0.5	23	1	44	1.4	15	0.2
	Addition/removal of a try statement	TY-ARTC	42	0.5	36	1.4	44	0.2	24	0.5	8	0.3	11	0.4	3	0.0
			83	1	87	3.3	494	2.6	47	1.0	31	1.3	55	1.8	18	0.2

6.1 Frequency and Similarity of Bug Fix Patterns

Using the extractor tool, the total number of bug fix patterns found in each category are counted for each of the seven projects in Table 1. That is, the table presents a count of how many instances of each pattern type are observed over the lifetime of each project. From this data, the frequency of each pattern type is computed. Table 2 presents this data (for each system, total counts are in the left hand column and computed frequency is in the right hand column), along with a summary of the number and frequency of bug fix pattern categories [e.g., If-related (IF), Loop (LP)].

With the pattern frequency data in hand, it is now possible to compute their similarity. Table 3 presents Pearson's correlations (Courtney and Gustafson 1992) between the ratios of bug fix pattern instances in different projects. The correlation values are all high (most of them are greater than 0.85, except for MegaMek), and all correlations are significant ($p < 0.001$). The quantitative inspection of the results shows that most of these projects have a very similar bug fix pattern distribution.

6.2 Discussion

Since the bug fix patterns are observing fixes to bugs, they provide substantial insight into the original bug itself. For example, a bug fix that involves a repair to an *if* conditional indicates that the conditional itself was involved in a bug prior to the fix. As a result, the frequency of bug fixes also provides the frequency of different bug types. Hence, the primary significance of the pattern frequency data lies in the improved visibility it provides into the relative frequency of different bug types.

Somewhat surprisingly, the results show two clear spikes in frequency. The Method Call (MC) and If-Related (IF) categories are by far the most prevalent bug fix patterns. Together they account for 44.6% to 60.3% of all bug fix pattern instances. In the Method Call category, most of the bug fixes to method calls are changes to the actual parameter expressions (MC-DAP). Getting parameter lists correct is the single largest source of programmer error observed. This is consistent with Basili and Perricone (1984), which identifies interface errors as the most common source of error in the software system examined.

Within the If-Related category, the If-conditional change (IF-CC) pattern has many more instances than other patterns in the IF category. Viewing this data in a different way, it is possible to lump together the loop condition pattern (LP-CC) with the Switch (SW) and If-Related (IF) categories to develop an aggregate sense of bug fixes that repair logic errors. This composite category accounts for 23.5% to 37.2% of all bug fix pattern instances. At

Table 3 Pearson's correlation between the ratios of pattern instances of patterns in different projects

	ArgoUML	Columba	Eclipse	JEdit	Scarab	Lucene	MegaMek
ArgoUML	1	0.91	0.89	0.93	0.99	0.89	0.70
Columba		1	0.75	0.85	0.91	0.82	0.56
Eclipse			1	0.94	0.89	0.86	0.65
JEdit				1	0.92	0.85	0.66
Scarab					1	0.88	0.66
Lucene						1	0.71
MegaMek							1

least for the observed projects, program logic is a notable source of error. This is somewhat consistent with several of the studies surveyed in Marick (1990), which found that logic errors were the most common source of error. Since the systems examined predate object-oriented software development, it is unclear how this might affect the frequency of interface errors.

Bug fixes in the Switch (SW) category are quite rare, but we note that the Eclipse project has a much higher ratio of Switch bug fixes than the other projects. The reason for this phenomenon is that that there are many case expressions in the Eclipse project. For example, the file 'jdt/internal/compiler/parser/Parser.java' in Eclipse contains hundreds of case expressions.

We manually examined the bug fix changes in MegaMek to find out why its bug fix pattern distributions differ from other projects. MegaMek is an online BattleTech board game. In its design, MegaMek defines Java classes for many kinds of weapons, ammunition, and music. Classes in the same group, e.g. weapons, are very similar to each other in design and have the same list of class fields. So, when there is a bug in one class, such as a bug in the field setting statement or addition of a field setting statement, the same bug will occur in many other classes in the same group. This is the reason MegaMek has a much higher ratio of AS-CE and SQ-AFO bug fix pattern instances than other projects.

We are now able to answer the second research question:

Most common bug fix pattern types—Research question 2

The most common categories of bug fix patterns are Method Call (MC, 21.9–33.1%) and If-Related (IF, 19.7–33.9%)

The most common individual patterns are MC-DAP (method call with different actual parameter values) at 14.9–25.5%, IF-CC (change in if conditional) at 5.6–18.6%, and AS-CE (change of assignment expression) at 6.0–14.2%.

The pattern similarity data in Table 3 allows us to answer the third research question:

Pattern frequency similarity—Research question 3

Bug fix pattern frequencies tend to be similar. With the exception of one project (MegaMek), Pearson similarity measures exceed 0.85 with $p<0.001$.

This result is surprising. Since the observed projects span many application types—UML modeling tool, email client, software development environment, text editor, change tracking system, search engine, and computer game—we would naively expect that differences among these application domains would lead to broadly dissimilar frequencies. Instead, just the opposite is observed. Despite the differences in application domain, bug fix pattern frequencies are very similar (with the exception of MegaMek). This is an important question. If projects are found to be broadly similar, then broad general mechanisms of bug production are at work. If dissimilar, the reasons for bug injection will tend to be more project-specific, and will not be as broad in their ability to predict the behavior of new projects.

The strong frequency similarity suggests many further questions. The first concerns the validity of the result. Is MegaMek truly an outlier, or is the apparent similarity an artifact of selecting a set of systems that just happen to be very similar. Though the current data set of seven observed projects is large compared to prior bug categorization studies, it is still not large enough to substantively answer this question.

Assuming the results are valid, there needs to be an explanation for why there is such strong similarity. One explanation is that the underlying frequency of statement types is also similar across projects, and each statement type carries an associated probability of bug introduction per statement instance. Assume, for example, that *if* statements occur in a known

frequency distribution of approximately 10% of statements, with probability of being a bug of 1% per statement instance per year. In a 100,000 statement project, 10,000 statements are an *if*, yielding 100 *if* related errors per year. If true, this suggests that project managers could predict the total number of specific kinds of bugs from a project size estimate.

A second explanation for the frequency similarity is the interplay of cognitive factors and specific statement types or fine-grain code patterns. For example, it may be that *if* conditionals are just inherently difficult to understand and this makes it difficult to write and modify them correctly. This explanation is potentially compatible with the first; it may be the case that cognitive issues with *if* conditionals lead to consistent error production rates for this type of statement.

Some bug types seem more related to a constant state of project evolution. The MC-DNP (method call with different number or types of parameters) category is best explained by developers accommodating to changes to the interfaces they develop against, as well as some errors in the use of overloaded methods. In this case, the similarity of error types may be more closely related to a common rate of change across projects.

The current paper provides two initial lines of inquiry into the broader issue of why the patterns are so similar. In the first, we examine sub-patterns of if-conditional (IF-CC) bugs. If it is the case that there are inherent qualities of specific statement types that make them bug prone, a more detailed exploration of a single pattern might reveal an explanation for why engineers inject that kind of bug. This is the rationale behind the exploration of *if* conditional sub-patterns in Section 7 below. If it is the case that developers have cognitive issues with specific statement types or code situations, then we would expect multiple developers to have broadly similar rates of bug injection. Section 8 below performs an initial exploration of bug production by five developers on the Eclipse project.

7 If Conditionals

The language keyword that is the single greatest individual source of bug fixes is *if*, with the If-conditional (IF-CC) pattern accounting for 5.6%–18.6% of all bug fix patterns. To better characterize this type of change, six sub-patterns were developed, described in Section 7.1 below. The frequency of these patterns is reported in Section 7.2, with discussion following.

7.1 If Conditional Patterns

Changes to *if* conditions are a mixture of regular and complex changes. Regular changes include addition of a condition clause, removal of a condition clause, addition of a variable in the condition expression, etc. Complex changes involve complete turnover of the conditional (all variables and operators changed), and change to a function call in the *if* condition (changed method parameters, changed method name, etc.).

We consider three factors, condition clauses, variables and operators in an *if* condition, and list the finger-grained bug fix patterns under the IF-CC pattern below. Note that the five sub-patterns may have overlaps.

7.1.1 Clause Added to Condition (IF-SUB-AC)

Description A new clause is added to a condition.

Example: if (*flag>5*) is changed to *if* (*flag> 5 && flag> 10*).

7.1.2 Clause Removed from Condition (IF-SUB-RC)

Description A clause is removed from a condition.

Example: if (*flag> 5 && flag> 10*) is changed to if (*flag> 5*).

7.1.3 Addition of Variable to Condition (IF-SUB-AV)

Description A new variable is used in the condition.

Example: if (*flag> 5*) is changed to *if* (*flag> 5 && length> 0*). This example also adds a new clause to the condition and hence satisfies IF-SUB-AC.

7.1.4 Removal of Variable from Condition (IF-SUB-RV)

Description There is a decrease in the number of variables used in the condition.

Example: if (*flag> 5 && length> 10*) is changed to *if* (*flag> 5*). This example removes a clause and hence also satisfies IF-SUB-RC.

7.1.5 Addition of Operator to Condition (IF-SUB-AO)

Description There is an increase of the number of operators in the condition. The operator factor indicates the complexity of a condition.

Example: if (*len > start*) is changed to *if* (*len > start+ 1*).

7.1.6 Removal of Operator from Condition (IF-SUB-RO)

Description There is a decrease in the number of operators in the condition.

Example: if (*len > start+ 1*) is changed to *if* (*len > start*).

7.2 If Conditional Pattern Frequency

A tool was developed to extract instances of the finer-grained IF-CC patterns, and was run on the seven projects in Table 1. The frequency distribution of the extracted patterns is shown in Table 4. The results show that generally 25% of IF-CC bug fixes involve addition or removal of condition clauses from an *if* condition (combination of IF-SUB-AC and IF-SUB-RC), with the exception of Lucene. The Lucene project has a much lower percentage of clause addition/removal subpatterns, perhaps due to the relatively small number of IF-SUB-AC and IF-SUB-RC instances (16 total out of 266 revisions).

7.3 Discussion

A strong trend that emerges from this data is a tendency towards increased complexity of conditionals. This can be seen by examining the relative frequency of addition and removal

Table 4 Bug fix distribution on finer-grained IF-CC pattern

	ArgoUML (%)	Columba (%)	Eclipse (%)	JEdit (%)	Scarab (%)	Lucene (%)	MegaMek (%)
IF-SUB-AC	13.1	28.9	20.8	23.1	20.8	4.3	16.1
IF-SUB-RC	11.5	8.7	6.9	11.2	3.6	4.3	2.7
IF-SUB-AV	8.3	14.1	14.3	23.7	16.4	11.4	19.5
IF-SUB-RV	12.0	5.4	9.7	15.1	16.8	10.5	11.6
IF-SUB-AO	22.4	37.6	22.3	38.0	26.8	13.2	27.8
IF-SUB-RO	14.6	11.4	15.1	21.0	10.4	11.6	11.8

categories within each category of sub-pattern (IF-SUB-AC vs IF-SUB-RC, IF-SUB-AV vs IF-SUB-RV, and IF-SUB-AO vs IF-SUB-RO). In six of the projects (all but Lucene), changes adding clauses (IF-SUB-AC) are more common than those removing them (IF-SUB-RC). For five of the projects (Columba, Eclipse, JEdit, Scarab, and MegaMek), the addition of clauses is substantially higher than removal of clauses (appx. 2 to 6 times as frequent). In five of the projects (Columba, Eclipse, JEdit, Lucene, and MegaMek), the number of variables in an *if* condition tends to increase (more IF-SUB-AV instances than IF-SUB-RV ones) when fixing a bug, whereas two projects (ArgoUML and Scarab) have the contrary trend. In all seven projects, the complexity based on the number of operators of if conditions tends to increase in bug fixes, since there are many more IF-SUB-AO instances than IF-SUB-RO ones.

This data indicates that an important factor in the production of if condition bugs is missing logic in the conditions. Possible causes for this include increasing logical complexity of a project over time, and lack of understanding of the complete set of logical conditions at the time the *if* statement was written. We now address research question 4.

If conditional frequency similarity—Research question #4
If conditional sub-patterns have substantial variability and are not similar across projects.
Projects generally have a trend towards increasing complexity of conditionals in bug fix changes.
Missing logic in conditions appears to be an important factor in production of if condition bugs. This conclusion requires further study.

8 Per-Developer Injection of Bugs

One possible explanation for the unusual level of similarity among bug fix patterns is that developers consistently have trouble with specific kinds of code situations, such as *if* conditionals. If this is the case, we would expect that developers would tend to have similar bug injection rates across the bug fix pattern types. At the same time, since developers are known to have wide variation in their level of productivity, it also seems reasonable that developers would exhibit some degree of individuality in their modes of bug injection. In this section we perform an initial exploration of the degree to which developers have similar or individual bug injection patterns.

8.1 Distribution and Similarity of Bug Injection in Eclipse by Developer

Our analysis examines five major development members of the Eclipse project. Across the project, a bug introduction analysis is performed to trace backwards from bug fix changes

to the initial change that injected the bug (the bug introducing change). Since the SCM system records which developer makes every change, it is possible to identify the specific developer that injects a bug into the code. From this bug introducing data, the bugs injected by the Eclipse developers can be determined.

Bug introduction analysis (Kim et al. 2006) is a refinement of the commonly used SZZ algorithm (Sliwerski et al. 2005), applied to buggy code that is covered by an instance of bug fix pattern. Bug introduction analysis traces backwards from a bug fix revision through a file's revision history to find the origin of each line containing a bug. This permits identification of the developer that introduced a problematic line, and in which revision. For example, given the line *if* (*foo.flag> 1*) that is covered by an IF-CC pattern instance in a bug hunk in revision 200, bug introduction analysis finds that this line is introduced by developer Bob in revision 50. Looking in the direction of increasing time, we can state that developer Bob introduced a buggy line in revision 50 that is solved by an IF-CC bug fix in revision 200.

The frequency of bug introduction for five Eclipse developers is shown in Fig. 3. To reduce clutter and focus the discussion, data from just 5 pattern types is shown (a more complete set of patterns can be found in (Pan 2006), p 61). In Fig. 3, the *x*-axis represents the bug fix patterns, and the *y*-axis represents the pattern distribution, i.e., each bar indicates the ratio of bug fixes of this pattern this developer has caused to all the bug fixes this developer's code has caused. Table 5 presents the Pearson's correlations among the pattern distributions of the five developers.

8.2 Discussion

The data presented in Fig. 3 and Table 5 provide support for both viewing developer bug introduction as being broadly similar, but also having noticeable individual qualities. Table 5 shows that the bug fix patterns have similar pattern distributions between different developers (most of the Pearson's correlations are greater than 0.85). No one developer is substantially different from the others in their production of bugs across the different categories. However, an examination of Fig. 3 clearly shows that some developers have spikes in the frequency of producing one or more types of bug. For example, developer *C*'s code caused fewer than average *if* condition checks errors (IF-APC), and more than average assignment statement bugs (AS-CE).

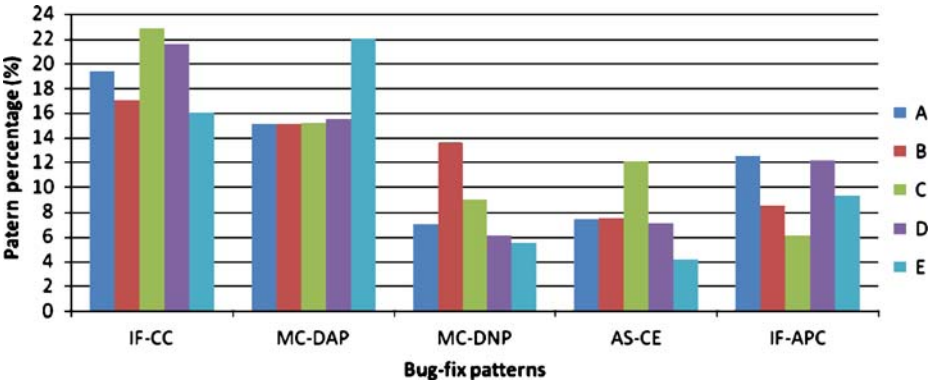


Fig. 3 Selected pattern distribution of bug introduction for five Eclipse developers (*A*, *B*, *C*, *D*, and *E*)

Table 5 Pearson's correlation between the pattern distributions for five Eclipse developers

	A	B	C	D	E
A	1	0.92	0.94	0.96	0.88
B		1	0.92	0.88	0.84
C			1	0.91	0.82
D				1	0.87
E					1

One finding is that per-developer bug introduction frequencies closely track the most common bug fix patterns. For Eclipse, the three most common bug fix patterns are IF-CC (18.6%), MC-DAP (17.9%), and MC-DNP (7.2%), which mirrors the most frequent per-developer patterns. This indicates that developers tend to have problems with certain kinds of code situations (such as method call parameters and *if* conditionals), and these situations are problematic for all developers equally. We caution that this must be viewed as a preliminary conclusion, since it is based solely on data from a single system, and only five developers for that system. Still, the observation is strong enough to warrant a detailed future examination.

We can now answer our final research question.

Per developer bug fix pattern frequency—Research question 5
There is substantial similarity in the frequency of bug introduction patterns across developers.
The frequency of bug introduction patterns for individual developers is similar to the overall project bug fix pattern frequency.
Individual developers do tend to have individual patterns of bugs they introduce more (and less) often than other developers.
There are indications that developers may consistently have trouble with specific code situations, and this leads to bugs being injected.

9 Related Work

Finding common bug patterns and using the patterns to detect bugs in advance is an active research area (Flanagan et al. 2002; Hovemeyer and Pugh 2004; PMD 2006). There are many static checkers that detect errors in program source code or binary executables using source code or binary patterns. FindBugs (Hovemeyer and Pugh 2004) defines 50 error-prone bug patterns, while ESC/Java (Flanagan et al. 2002) uses type checking and specification (a kind of pattern) to detect errors. PMD (2006) uses problematic source code style patterns, such as unused variables, empty catch blocks, and unnecessary object creation, to detect potential bugs. These systems base their bug patterns on well-known program errors, and use static analysis to detect bug pattern instances. These bug patterns are horizontal—general to all the projects—and carry no project-specific application knowledge. Our approach focuses on finding bug or fix patterns in the software change history, and the bug fix pattern instances automatically extracted are based on the real bugs identified by developers, i.e. the bug fix changes. Hence bug fix pattern instances are vertical, recording project-specific bug and fix information.

The literature contains several bug classification taxonomies. The IEEE Standard Classification for Software Anomalies presents a classification scheme of types of software

abnormalities, including logic problem, computation problem, interface/timing problem, data handling problem, data problem, documentation problem, document quality problem, and enhancement (IEEE 1993). Orthogonal defect classification (ODC; Chillarege et al. 1992) is a scheme for capturing the semantics of software defects whose types are associated with different phases of the software life cycle, such as design, development, test and service.

Ostrand and Weyuker (1984) developed a fault categorization scheme by examining change report forms for an interactive special-purpose editor system. The change report forms include several questions to reveal the causality of software errors. From the case study, they categorized the faults into seven major categories including data definition, data handling, decision and processing, decision (alone), system, documentation and unknown. Their fault distribution results show that data definition and data handling categories contain the most defects. Perry and Stieg (1993) surveyed software faults in a large real-time system through questionnaires. Design and code phase faults were classified into 22 types including language pitfalls, protocol, low-level logic internal functionality, interface complexity, etc. The characteristics of these faults were analyzed for several factors, including how difficult they were to fix, fault causality, and fault prevention. Leszak et al. (2000) studied a networking product to explore the causality of defects. In their study, they investigated defect modification requests and classified the defection into three classes, implementation, interface, and external. Each class has a set of appropriate defects. For example, the following defect types belong in the implementation class: algorithm, functionality, performance or language pitfalls. They computed the distribution of defect types on the defects in the system studied, and found the algorithm defect type to be the most prevalent type.

How do bug fix patterns differ from these existing fault taxonomies? Bug fix patterns are very syntax-driven, while existing taxonomies tend to be either cause-driven (what caused this bug), and/or document-driven (in what document is this cause located). A major benefit of bug fix patterns is they are automatically extractable, whereas existing taxonomies usually require human categorization of a bug into the taxonomy. Furthermore, since bug fix patterns are closely tied to the program text, the patterns are more concrete and less ambiguous than many existing fault categories that require human interpretation. This makes bug fix patterns far better suited for cross-project comparison of fault data.

Duraes and Madeira (2006) presented a statistical analysis of fault types across multiple open source projects and used this fault data to perform improved mutation testing. Though the primary goal of their work is to perform fault injection in a simulation model, the analysis of fault types based on program constructs in their work is similar to ours. However, the fault classification method in (Duraes and Madeira 2006) is a manual one, and their bug classification is coarser-grained than ours. Their work also assumes that all the changes are bug fixes, but it is not always true. Our approach automatically identifies bug-fix change and non-fix changes. Another difference is that their work targeted systems written in C, while the systems studied in our research are all “high level” software systems written in Java.

There is a rich literature for bug detection and prediction. Graves et al. assumed that modules *that were changed recently* are more fault-prone than modules that were changed a long time ago (Graves et al. 2000). They built a weighted time damp model to predict faults from changes over where recent changes are weighted over older ones. Hassan and Holt (2005) use a caching algorithm to compute the set of fault prone modules, called the top-ten list. They use four factors to determine this list: software that was most frequently modified, most recently modified, most frequently fixed, and most recently fixed. Fischer et al. (2003) and (Bevan and Whitehead 2003) identified code smell and instable modules based on co-change.

Combining static or dynamic analysis and mining repositories techniques to identify bugs are proposed. Williams and Hollingsworth (2005) use project histories to improve

existing bug finding tools. Using a return value without first checking its validity may be a latent bug. In practice, this approach leads to many false positives, as typical code has many locations where return values are used without checks. To remove the false positives, Williams and Hollingsworth use project histories to determine which kinds of function return values must be checked. For example, if the return value of *foo* was always verified in the previous project history, but was not verified in the current source code, it is very suspicious. Livshits and Zimmermann (2005) combined software repository mining and dynamic analysis to discover common use patterns and code patterns that are likely errors in Java applications.

These bug detection and prediction techniques focus on identifying latent bugs while our work tries to provide developer understandable patterns from previously identified bugs.

10 Threats to Validity

We identify the following threats to validity.

Systems Examined Might Not Be Representative Seven Java systems were examined in this paper. Since we intentionally chose systems for which we could identify fixes based on the change description log (required for determination of bug fix location), we might have a project selection bias. Analyzing only Java projects may also introduce bias, since the relative frequency of bug fix patterns is expected to vary across languages, since some bug fix patterns are specific to object-oriented languages. Extending the bug fix pattern analysis to other languages remains future work.

Systems Are All Open Source All systems examined in this paper are developed as open source. Hence they might not be representative of closed-source development since different development processes could lead to different bug-fix patterns. Despite being open source, several of the analyzed projects have substantial industrial participation.

Bug Fix Patterns Have Incomplete Coverage of Bug Fixes Only 45.7–63.6% of bug fixes contain at least one identifiable bug fix pattern, and hence there are many bug fix changes that are not accounted for by one of the patterns. Our experience is that the remaining bug fix changes do not have any readily identifiable patterns. Still, the potential exists for a more sophisticated analysis to discover additional patterns, thereby increasing coverage and potentially altering the observed pattern frequencies.

False Positives in Bug Identification The bug fix pattern extractor identifies program bugs by looking for the keywords “fixed,” “bug,” and “patch” in the change log. There is a limitation in this approach: it only uses the change log information, and change logs of some non-bug-fix changes may also contains these keywords. A more precise way for identifying bugs is to use bug tracking information together with change logs. This paper only used change logs for identifying bugs, which may cause some false positives in bug identification. For the Scarab system we do use bug tracking information, which leads to the problem that some new feature additions are also marked as bug fixes.

Bug Fix Data is Incomplete Even though we selected projects with a high quality of historic data, we still can only extract a subset of all faults (typically 40–60% of those reported in bug tracking systems).

11 Conclusion

This paper explored the underlying patterns in bug fixes mined from software project change histories. Through manual inspection of the bug fix change history of several open source projects, 27 bug fix patterns were identified, all of which are amenable to automatic detection by a bug fix pattern extractor tool. The bug fix pattern extractor was used to characterize the bug fix patterns of seven Java open source projects, Eclipse, Columba, JEdit, Scarab, ArgoUML, Lucene, and MegaMek. The results show that 45.7% to 63.6% of the bug fix hunk pairs are covered by the bug fix patterns.

The most common categories of bug fix patterns are Method Call (MC, 21.9–33.1%) and If-Related (IF, 19.7–33.9%). The most common individual patterns are MC-DAP (method call with different actual parameter values) at 14.9–25.5%, IF-CC (change in if conditional) at 5.6–18.6%, and AS-CE (change of assignment expression) at 6.0–14.2%. This agrees with prior bug classification work that identified interface errors and logic errors as the most common bug categories.

Bug fix pattern frequencies tend to be similar across all projects, a surprising result. With the exception of one project (MegaMek), Pearson similarity measures exceed 0.85. This indicates that developers may have trouble with individual code situations, and that frequencies of bug introduction are independent of program domain. Further evidence from a preliminary examination of individual developer frequencies of introducing bug patterns shows that per-developer bug introduction rates are also very similar, and mirror overall project bug pattern frequency. This also lends support to the view that developers have difficulty with specific code situations.

Analysis of If-conditional (IF-CC) sub-patterns finds that these bug fix changes commonly increase the complexity of the conditional by adding operators, variables, or condition clauses. This suggests some specific causes for *if* conditional errors, namely that projects have increasing complexity over time, and that developers have difficulty enumerating all of the possible conditions initially.

Taken together, the data presented in this paper suggests a new way of thinking about bugs in software. Approximately half of all project bugs appear to fall into well known pattern categories, with the frequency of errors relatively constant across projects. Instead of application domain-specific processes of bug production, there appear to be general processes that involve the interplay of cognitive errors and specific code situations. Due to the similarity of bug fix patterns across projects, these code situations cause difficulty despite their context, independent of the developers working on them. This is a hopeful result, since it suggests further study can uncover broad, general causes for a large fraction of project bugs. Once these general causative factors are known, it may be possible to craft interventions in current software engineering practice to reduce or eliminate broad classes of error. It also suggests the potential for detailed predictions of the number of different types of bugs in software projects, permitting better allocation of testing resources.

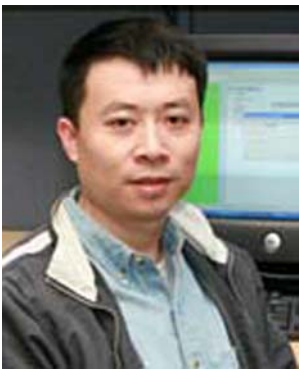
References

Basili VR, Perricone BT (1984) Software errors and complexity: an empirical investigation. Commun ACM 27(1):42–52

Bevan J, Whitehead EJ Jr (2003) Identification of software instabilities. Proceedings of the 10th Working Conference on Reverse Engineering. Victoria, BC, Canada, pp 134–145

Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY (1992) Orthogonal defect classification—a concept for in-process measurements. IEEE Trans Softw Eng 18(11):943–956

Courtney RE, Gustafson DA (1992) Shotgun correlations in software measures. Softw Eng J 8(1):5–13
Cubranic D, Murphy GC (2003) Hipikat: Recommending pertinent software development artifacts. Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon, pp 408–418
Duraes JA, Madeira HS (2006) Emulation of software faults: a field data study and a practical approach. IEEE Trans Softw Eng 32(11):849–867
Eclipse (2006) Eclipse Java Development Tools (JDT) Subproject Home Page. http://www.eclipse.org/jdt/
Endres A (1975) An analysis of errors and their causes in system programs. Proceedings of the International Conference on Reliable Software. Los Angeles, California, pp 327–336
Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. Proceedings of 2003 Int'l Conference on Software Maintenance (ICSM'03). Amsterdam, The Netherlands, pp 23–32
Flanagan C, Leino K, Lillibridge M, Nelson C, Saxe J, Stata R (2002) Extended static checking for java. Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Desing and Implementation. Berlin, Germany, pp 234–245
GNU (2003) GNU Diffutils. http://www.gnu.org/software/diffutils/
Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661
Hassan AE, Holt RC (2005) The top ten list: Dynamic fault prediction. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). Budapest, Hungary, pp 263–272
Hovemeyer D, Pugh W (2004) Finding bugs is easy. ACM SIGPLAN Notices 39(12):92–106
IEEE (1993) IEEE standard classification for software anomalies: IEEE Standard 1044–1993
Kim S, Zimmermann T, Pan K, Whitehead EJ Jr (2006) Automatic identification of bug-introducing changes. Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. Tokyo, Japan
Leszak M, Perry DE, Stoll D (2000) A case study in root cause defect analysis. Proceedings of the 22nd International Conference on Software Engineering. Limerick, Ireland, pp 428–437
Li Z, Tan V, Wang X, Lu S, Zhou Y, Zhai C (2006) Have things changed now? An empirical study of bug characteristics in modern open source software. Proceedings of 1st Workshop on Architectural and System Support for Improving Software Dependability. San Jose, California, pp 25–33
Livshits B, Zimmermann T (2005) DynaMine: Finding common error patterns by mining software revision histories. Proceedings of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005). Lisbon, Portugal, pp 296–305
Marick B (1990) A survey of software fault surveys. Technical Report, University of Illinois at Urbana-Champaign UIUCDCS-R-90-1651, December
Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. Proceedings of International Conference on Software Maintenance (ICSM 2000). San Jose, California, pp 120–130
Ostrand TJ, Weyuker EJ (1984) Collecting and categorizing software error data in an industrial environment. J Syst Softw 4(4):289–300
Pan K (2006) Using evolution patterns to find duplicated bugs. PhD dissertation. Department of Computer Science, UC Santa Cruz, p 61
Perry DE, Stieg CS (1993) Software faults in evolving a large. Real-time system: a case study. Proceedings of the Fourth European Software Engineering Conference. Garmisch, Germany, pp 48–67
PMD (2006) PMD home page. http://pmd.sourceforge.net/
Potier D, Albin JL, Ferreol R, Bilodeau A (1982) Experiments with computer software complexity and reliability. Proceedings of 6th International Conference on Software Engineering. Tokyo, Japan, pp 94–103
Sliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? Proceedings of the International Workshop on Mining Software Repositories (MSR 2005). Saint Louis, Missouri, pp 24–28
Williams CC, Hollingsworth JK (2005) Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans Softw Eng 31(6):466–480



Kai Pan graduated with Ph.D. degree in Computer Science from the University of California, Santa Cruz in October 2006. Before that, he received a Bachelor of Science in Computer Science from Peking University, Beijing, China in 1995, and a Master of Science in Computer Science also from Peking University, Beijing, China in 1998. Kai's research interests include software evolution analysis, data mining, program analysis, and software configuration management.

Sunghun Kim is an Assistant Professor of Computer Science at the Hong Kong University of Science and Technology. He completed his Ph.D. in the Computer Science Department at the University of California, Santa Cruz in 2006. He was a postdoctoral associate at Massachusetts Institute of Technology and a member of the Program Analysis Group. He was a Chief Technical Officer (CTO), and led a 25-person team at the Nara Vision Co. Ltd, a leading Internet software company in Korea for six years. His core research area is Software Engineering, focusing on software evolution, program analysis, and empirical studies.

Jim Whitehead is an Associate Professor of Computer Science at the University of California, Santa Cruz. Jim's research interests lies in the area of software evolution (bug prediction), automated software construction, and automatic generation of computer game levels. He has recently developed a new undergraduate degree program in computer games, the B.S. Computer Science: Computer Game Design. Jim received his Ph.D. in Information and Computer Science from the University of California, Irvine, in 2000.