Micro Pattern Evolution

Sunghun Kim
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
hunkim@cs.ucsc.edu

Kai Pan
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
pankai@cs.ucsc.edu

E. James Whitehead, Jr.
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
ejw@cs.ucsc.edu

ABSTRACT

When analyzing the evolution history of a software project, we wish to develop results that generalize across projects. One approach is to analyze design patterns, permitting characteristics of the evolution to be associated with patterns, instead of source code. Traditional design patterns are generally not amenable to reliable automatic extraction from source code, yet automation is crucial for scalable evolution analysis. Instead, we analyze "micro pattern" evolution; patterns whose abstraction level is closer to source code, and designed to be automatically extractable from Java source code or bytecode. We perform micro-pattern evolution analysis on three open source projects, ArgoUML, Columba, and jEdit to identify micro pattern frequencies, common kinds of pattern evolution, and bug-prone patterns. In all analyzed projects, we found that the micro patterns of Java classes do not change often. Common bug-prone pattern evolution kinds are 'Pool → Pool', 'Implementor → NONE', and 'Sampler → Sampler'. Among all pattern evolution kinds, 'Box', 'CompoundBox', 'Pool', 'CommonState', and 'Outline' micro patterns have high bug rates, but they have low frequencies and a small number of changes. The pattern evolution kinds that are bug-prone are somewhat similar across projects. The bug-prone pattern evolution kinds of two different periods of the same project are almost identical.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Algorithms, Measurement, Experimentation

1. INTRODUCTION

Software evolution research examines the development history of a software project to learn facts about the software, and better understand its qualities. After examining the history of many different software projects, ideally we would like to be able to make claims like, if we observe evolution pattern *X*, then the consequences for one or more software qualities are *Y* and *Z*.

MSR '06, May 22-23, 2006, Shanghai, China.

Most software repository mining research examines software by subdividing it into parts using physical distinctions, such as modules, directories, files, and methods. Researchers examine the evolution of these physical elements, and then correlate various software properties with traits of the observed evolution. For example, researchers have examined revision histories to determine correlations between changes and bugs [13]. Though there has been much success in correlating software properties with the evolution of physical elements within a project, the ability to apply these results to other projects has been limited. This is due to the use of the software's existing physical distinctions, which limits the applicability of results to just a single project. Knowing something about the evolution of the methods in a specific Java class does not typically provide any insight into other classes, since different classes have different source code.

To make more generalizable observations requires some means for abstracting away from the physical elements into abstract categories. These categories need to be concrete enough to capture important aspects of the behavior of the software, yet sufficiently general that one can observe the same abstract categories across multiple projects. The classic software design patterns [6] fit this description, and suggest the possibility that we can deeply understand the evolutionary behavior of specific design patterns. To perform such analysis in a scalable way, we need an automated mechanism for extracting software design patterns from source code. Unfortunately, to date there is no accurate mechanism for identifying design patterns in code, with existing approaches suffering from large amounts of false positives or false negatives.

Recent work by Gil and Maman has introduced the concept of micro patterns [7], which are "Java class-level traceable patterns." These are more fine-grained design patterns than the classic patterns, and have been designed to always be automatically extractable from source code (or bytecode). Micro patterns express more fine-grained design idioms than classic patterns. For our purposes, what is important is that we now have a reliable, automatic way to extract a set of general design abstractions from Java projects. This now allows us to explore whether evolution characteristics can be correlated with the abstractions inherent in these micro patterns, and make generalizable conclusions about specific evolution patterns.

In this paper we analyze the micro pattern evolution of three open source projects, ArgoUML, Columba, and JEdit, shown in Table 1. Our goal in doing so is to examine whether there are any correlations between the evolution of micro patterns and the likelihood of having bugs. Ideally we wish to identify micro pattern evolution kinds that are consistently fault prone across projects, and hence allow us to make general conclusions about this kind of evolution that have broad applicability.

Table 1. Analyzed projects, ArgoUML, Columba, and jEdit. # of revisions is the number of revisions we analyzed. # of class changes indicates the number of corresponding source code (Java) changes. # of bug changes indicates the number of changes that introduce bugs identified by mining change logs and SCM history [13]. % of bug rate is the rate of bug-introducing changes over all changes.

Project	Software type	Period	# of revision	# of class changes	# of bug class changes	% of bug rate
ArgoUML	UML design tool	01/2002 ~ 03/2003	1,262	4,179	1,245	29.8
Columba	Email Client	11/2002 ~ 01/2006	1,652	11,138	1,604	14.4
jEdit	Editor	09/2001 ~ 01/2006	1,449	5,526	2,456	44.5

After examining the micro pattern evolution history of the three open source projects, we found that micro patterns do not typically change when a class file changes. For example, the most common pattern evolution kinds are 'Limited Self → Limited Self', 'Implementor → Implementor', and 'Sink → Sink' (these micro patterns are briefly described in Section 2). In all these cases the micro pattern is the same before and after the class change. Only 4-6% of class file changes cause micro pattern changes, examples being 'Implementor → NONE' and 'Stateless → RestrictedCreation'.

For each project we identified the micro pattern kinds that were most bug-prone. We additionally found the most bug-prone pattern evolution kinds of the three projects, and found that they are somewhat similar. Furthermore, we observed that the bug-prone evolution kinds for two different periods of the same project are almost identical. For example, micro pattern evolution kinds such as 'Pool → Pool', 'Implementor → NONE', and 'Sampler → Sampler' are bug-prone in jEdit. We found that 'Box', 'CompoundBox', 'Pool', 'CommonState', and 'Outline' micro patterns have high bug rates, but they have low frequencies and a small number of changes. In contrast, 'Overrider' and 'Sink' micro patterns have relatively lower bug rates.

We anticipate that these findings can be used by software quality engineers to identify areas of a software project that are more bug-prone, and apply more testing and verification resources to those areas. We could also make software developers aware that they are working on a bug-prone pattern, or kind of pattern transition, and thereby encourage more defensive coding and more extensive unit testing.

In the remainder of the paper, we explain micro patterns (Section 2) and describe our experimental setup (Section 3). Following are results from our experiments (Section 4), along with discussion of the results (Section 5). Rounding off the paper, we end with related work (Section 6) and conclusions (Section 7).

2. JAVA MICRO PATTERNS

Micro patterns capture idioms of Java programming languages such as the use of inheritance, immutability, data wrapping, data management, and modularity [7]. Micro patterns include Box, Compound Box, Sampler, Canopy, Immutable, Implementor, Pseudo Class, Pool, Restricted Creation, Overrider, Sink, Stateless, Common State, Outline, Function Pointer, Function Object, Joiner, Designator, Record, Taxonomy, PureType, Augmented Type, Extender, Data Manager, Trait, Cobol Like, State Machine Recursive, and Limited Self [7]. While the reader is strongly encouraged to examine [7] for a detailed description, we describe a few micro patterns here to provide a flavor of these patterns:

Pool: A class has only final static fields and no methods.

Box: A class has exactly one instance field, which can be modified by methods in the class.

Sampler: A class that has at least one public constructor and at least one static field whose type is the same as that of the class.

Limited Self: Suppose class '*foo*' is a subclass of class *'bar'*. If *'foo'* does not introduce any new fields, and all self method calls in *'foo'* are calls to methods in *'bar'*, then *'foo'* is a Limited Self pattern class.

Recursive: A class that has at least one field whose type is the same as that of the class. For example, java.util.LinkedList is a recursive pattern class.

Sink: a class whose declared methods do not call instance methods or static methods.

Implementor: a non-abstract class such that all of its public classes are implementation of its super abstract class.

We use micro patterns for our pattern change analysis for three reasons: (1) they are traceable, (2) they are close to the source code, (3) and they capture non-trivial design idioms of the Java language.

3. EXPERIMENT SETUP

In this section, we describe the data used in our study and explain how it was extracted. We use the Kenyon [3] infrastructure to automatically extract project revisions and class changes from the SCM repositories for ArgoUML, Columba, and jEdit. Bug-introducing changes are identified by mining change logs and project history data using techniques described in [13]. Micro patterns are extracted using a pattern extraction tool developed by Gil and Maman [7], after compiling each revision.

3.1 Micro Pattern Extraction

We extract software histories including all revisions and all files from SCM systems such as CVS [2] using the Kenyon infrastructure [3]. After checking out each project revisions, we compile the revision and generate a jar file. We feed the jar file into the micro pattern extraction tool [7]. The tool automatically reads all class files in the jar file and extracts the pattern(s) matched by each class file. We persistently store these extracted micro patterns for each Java class file for all revisions of all three projects.

3.2 Pattern Changes and Bug Changes

Now we have the micro patterns for all Java class files (corresponding Java source files) of each revision. Using the standard *diff* tool, we can easily identify Java class file changes. To determine bug-introducing changes, we mine SCM change logs and project history data [13]. We then observe the micro pattern changes in each Java class file and compute bug introduction rates for these changes.

For example, consider the change history for the file '*foo.java*' (*foo.class*) as shown in Figure 1. The change log at revision 6 (Rev 6) states "Fixed issue #355", which indicates that it is a fix change. It means the file at revision 5 has one or more problematic lines, which are fixed in revision 6 by changing the problematic lines. When were the problematic lines added in the first place? SCM systems such as CVS [2] and Subversion [1] provide an annotation feature that shows information about when each line of a file was modified, and by whom. Using SCM annotation, we can find out when the problematic lines were initially added. Suppose the problematic lines were added in revision 3. This means the file at revision 2 does not have the problematic lines, so they were added in the change between revision 2 and 3. This change introduced a bug into the software, and hence we call it a bug-introducing change.

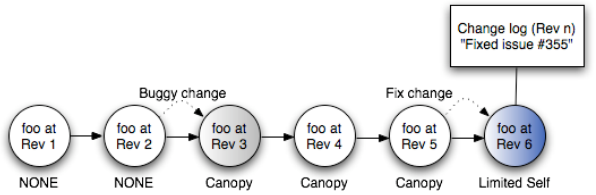

Figure 1. Example of pattern evolution kinds and a bug-introducing change.

The micro patterns for each revision are shown in Figure 1. As an example, the pattern evolution kind for '*foo.class*' between revisions 1 and 2 is 'NONE → NONE'. In Figure 1, we see the following micro pattern evolution kinds: '*NONE → NONE*' (1 time), '*NONE → Canopy*' (1 time), '*Canopy → Canopy*' (2 times), and '*Canopy → Limited Self*' (1 time). We count the number of bug-introducing changes and compute the bug introduction rate for each micro pattern evolution kind. For example, the bug-introducing change count of the '*NONE → Canopy*' kind is 1, and occurs 1 time, so it has a 100% bug introduction rate. General categories of pattern evolution kinds are described in Table 2.

Table 2. Categories of pattern evolution kinds

Category		Description
Pattern unchanged		Pattern remains the same after a class change. E.g. Canopy → Canopy
Pattern changes	Change to other pattern	Pattern changes to other patterns. E.g. Stateless → RestrictedCreation
	Losing pattern	Pattern changes to NONE. E.g. LimitedSelf → NONE
	Adding pattern	Pattern changes from NONE. E.g. NONE → Stateless

When we compute the bug introduction rates of micro pattern evolution kinds, we filter out the total count if it is less than 10 (outliers). If a micro pattern evolution kind occurs less than 10 times, we believe it is hard to make general conclusions about its bug introduction rate, and it is possible that a small number of bugs can affect the bug introduction rate substantially.

4. RESULTS

We first present micro pattern frequencies of a project snapshot (the latest revision). We next show the list of micro pattern evolution kinds, their counts, and ratios. The bug-prone micro pattern evolution kinds are shown using contour graphs. We compare common bug-prone evolution kinds of three projects and two periods of the same project. Finally, we compare frequencies, the number of changes, and bug rates of each micro pattern.

Table 3. Java micro pattern frequencies of analyzed projects. The * marked patterns do not exist or are rare in the analyzed projects; we exclude them in further analysis.

Micro Patterns	ArgoUML (%)	Columba (%)	jEdit (%)
Box	1	3	2
Compound Box	1	2	6
Sampler	1	2	1
Canopy	8	10	22
Immutable	3	5	10
Implementor	28	31	32
*Pseudo Class	0	0	0
Pool	1	3	1
Restricted Creation	2	2	1
Overrider	8	7	22
Sink	5	10	10
Stateless	8	9	5
Common State	1	1	3
Outline	1	0	0
Function Pointer	1	2	1
Function Object	5	7	19
*Joiner	0	0	0
*Designator	0	0	0
Record	0	0	1
Taxonomy	1	2	2
PureType	4	9	4
*Augmented Type	0	0	0
Extender	3	11	5
Data Manager	1	3	1
*Trait	0	0	0
Cobol Like	0	1	0
State Machine	1	2	1
Recursive	0	0	2
Limited Self	16	20	12
Coverage	55	79	81

4.1 Pattern Frequencies

We compute micro pattern frequencies of a project snapshot (the latest revision), with results shown in Table 3. 'Canopy', 'Implementor', 'Overrider', 'Function Object', and 'Limited Self' are the most prevalent micro patterns. 81% of classes have one or more micro patterns in jEdit, 79% for Columba and 55% for ArgoUML. The remaining classes do not match any micro pattern (NONE). Some micro patterns, such as 'Joiner' or 'Pseudo Class,' do not exist in the latest revision.

The micro pattern distributions of three projects are quite similar. For example, the Pearson's correlation coefficient [5] of the micro pattern frequencies of ArgoUML and jEdit is 0.96. Even though these two projects have different physical features in their source code, they have similar pattern frequencies, suggesting that any correlations between patterns, or pattern evolution kinds found in these two projects would have applicability to both projects, and perhaps others as well.

4.2 Pattern Evolution Kinds

We count all micro pattern evolution kinds of each Java class file change across the project histories. Table 4 shows the top 20 micro pattern evolution kinds, their counts, and relative frequency (percentage of all observed pattern evolutions) of each pattern change. The most common micro pattern evolution kind is 'NONE → NONE'. Other common micro pattern evolution kinds are 'LimitedSelf → LimitedSelf', 'Implementor → Implementor', 'Overrider → Overrider', and 'Extender → Extender'. The common micro pattern evolution kinds are similar for the three projects.

Table 4. Top 20 most common pattern evolution kinds of the analyzed projects.

Rank	ArgoUML Pattern evolution kind	ArgoUML change # (change %)	Columba Pattern evolution kind	Columba change # (change %)	jEdit Pattern evolution kind	jEdit change # (change %)
1	NONE → NONE	1830 (33%)	NONE → NONE	4245 (31%)	NONE → NONE	1738 (24%)
2	LimitedSelf → LimitedSelf	931 (17%)	LimitedSelf → LimitedSelf	1684 (12%)	LimitedSelf → LimitedSelf	803 (11%)
3	RestrictedCreation → RestrictedCreation	490 (8.8%)	Implementor → Implementor	1589 (12%)	Overrider → Overrider	751 (10%)
4	Implementor → Implementor	375 (6.8%)	Extender → Extender	1294 (9.5%)	CommonState → CommonState	582 (7.9%)
5	Overrider → Overrider	342 (6.2%)	Overrider → Overrider	749 (5.5%)	Implementor → Implementor	575 (7.8%)
6	Extender → Extender	262 (4.7%)	Stateless → Stateless	578 (4.2%)	Canopy → Canopy	452 (6.2%)
7	Sink → Sink	203 (3.7%)	Sink → Sink	538 (4%)	Recursive → Recursive	317 (4.3%)
8	Stateless → Stateless	189 (3.4%)	CommonState → CommonState	237 (1.7%)	Extender → Extender	286 (4%)
9	Sampler → Sampler	142 (2.6%)	Immutable → Immutable	223 (1.6%)	Sampler → Sampler	274 (3.7%)
10	Common State → Common State	91 (1.6%)	Box → Box	201 (1.5%)	Immutable → Immutable	223 (3%)
11	Immutable → Immutable	77 (1.4%)	PureType → PureType	178 (1.3%)	CompoundBox → CompoundBox	216 (2.9%)
12	Compound Box → Compound Box	70 (1.3%)	Taxonomy → Taxonomy	163 (1.2%)	FunctionObject → FunctionObject	183 (2.5%)
13	Implementor → NONE	70 (1.3%)	DataManager → DataManager	163 (1.2%)	Sink → Sink	170 (2.3%)
14	NONE → Stateless	50 (0.9%)	CompoundBox → CompoundBox	161 (1.2%)	Pool → Pool	140 (1.9%)
15	Canopy → Canopy	45 (0.8%)	Canopy → Canopy	145 (1.1%)	Stateless → Stateless	112 (1.5%)
16	Outline → Outline	42 (0.8%)	Outline → Outline	143 (1%)	PureType → PureType	63 (0.9%)
17	Box → Box	30 (0.5%)	RestrictedCreation → RestrictedCreation	127 (0.9%)	Box → Box	39 (0.5%)
18	LimitedSelf → NONE	27 (0.5%)	FunctionPointer → FunctionPointer	114 (0.8%)	DataManager → DataManager	37 (0.5%)
19	Pool → Pool	25 (0.5%)	Pool → Pool	97(0.7%)	Outline → Outline	24 (0.3%)
20	NONE → Implementor	24 (0.4%)	FunctionObject → FunctionObject	82 (0.6%)	Taxonomy → Taxonomy	17 (0.2%)

Also note that the patterns in the top pattern evolution kinds are not the same as the most frequent patterns shown in Table 3. For example, the most common pattern in jEdit is 'Implementor', but the most common pattern evolution kind is 'LimitedSelf → Limited Self' (excluding 'NONE → NONE'). The fourth ranked pattern evolution kind, 'CommonState → CommonState', is a relatively rare micro pattern in jEdit (only 3%).

Overall, micro patterns in Java class files do not frequently transition to new micro patterns. If a Java class file exhibits characteristics of a given micro pattern, the class file tends to stick to the original micro pattern as the class file changes. Table 5 shows the counts and percentages of pattern evolutions that change patterns, and those that do not. Only 4 to 6% of Java class file changes result in micro pattern changes.

Note that the total pattern evolution kind count (Table 5) is greater than the total class file change count (Table 1), since a class file can have more than one pattern and a class change includes more than one pattern evolution kind. The multiplicity of micro patterns are explained in [7].

Table 5. Ratio of pattern evolution kinds the three projects.

	ArgoUML	Columba	jEdit
Pattern unchanged	5,238 (94%)	12,977 (95%)	7,403 (95.9%)
Pattern changes	313 (6%)	643 (5%)	287(4.1%)

4.3 Bug-prone Pattern Evolution Kinds

We count bug-introducing changes for each micro pattern evolution kind, and compute the bug change rate for each kind. After computing all bug introduction rates for all pattern evolution kinds, we draw contour graphs to indicate the common bug-prone pattern evolution kinds. Figure 2 shows the bug introduction rates for each micro pattern evolution kind for ArgoUML. The x-axis indicates to-patterns and y-axis indicate from-patterns. For example, the left-bottom cross indicates the bug rate of the 'NONE → NONE' pattern evolution kind. The order of micro patterns along the x-axis and y-axis is the same as the ordering in Table 3, excluding the infrequently occurring * marked patterns. The contour line density indicates the bug rates. Note that the value associated with each contour line varies by chart, since each chart scales the contours to improve presentation. Contour graphs show the overview properties of bug-prone pattern evolution kinds. Denser contour lines indicate higher bug rates.

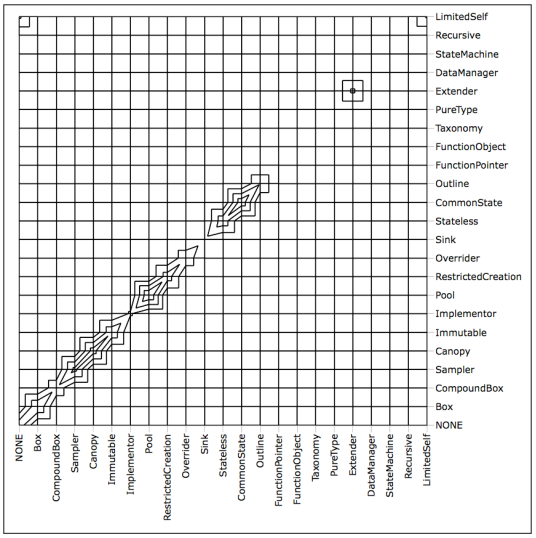

Figure 2. ArgoUML bug-prone pattern evolution kinds

Figure 3 shows bug introduction rates for each micro pattern evolution kind for jEdit. The two contour graphs (Figure 2, and Figure 3) show that the bug-prone micro pattern evolution kinds of the two projects are somewhat similar, but not identical. For example, the 'Sampler → Sampler' micro pattern evolution kind is bug-prone in all projects. However, the 'CompoundBox → Canopy' micro pattern evolution kind is bug-prone in jEdit, but not in ArgoUML. Table 6 shows the top 20 most bug-prone pattern evolution kinds of all three projects.

Table 6. Top 20 most bug-prone pattern evolution kinds.

Rank	ArgoUML Pattern evolution kind	bug rate	Columba Pattern evolution kinds	bug rate	jEdit Pattern evolution kinds	bug rate
1	Pool → Pool	40	Implementor → NONE	26	Sampler → Sampler	72
2	CommonState → CommonState	37	RestrictedCreation → RestrictedCreation	22	Recursive → Recursive	63
3	Canopy → Canopy	36	CompoundBox → CompoundBox	21	CommonState → CommonState	58
4	Sampler → Sampler	33	Immutable → Immutable	21	FunctionObject → NONE	53
5	Box → Box	30	CobolLike → NONE	20	CompoundBox → CompoundBox	52
6	Immutable → Immutable	29	NONE → Implementor	19	LimitedSelf → LimitedSelf	51
7	NONE → NONE	27	NONE → NONE	18	NONE → NONE	49
8	Stateless → Stateless	27	LimitedSelf → NONE	18	Pool → Pool	48
9	RestrictedCreation → RestrictedCreation	26	Recursive → Recursive	18	Immutable → Immutable	43
10	Extender → Extender	23	Overrider → NONE	17	Outline → Outline	42
11	LimitedSelf → NONE	22	NONE → Extender	17	Immutable → NONE	40
12	LimitedSelf → LimitedSelf	21	NONE → Overrider	15	Implementor → NONE	38
13	Outline → Outline	19	CommonState → CommonState	14	Sink → NONE	38
14	NONE → CobolLike	18	FunctionObject → FunctionObject	13	NONE → LimitedSelf	36
15	CompoundBox → CompoundBox	17	Box → Box	13	Stateless → Stateless	34
16	Overrider → Overider	17	Stateless → Stateless	13	CompoundBox → NONE	33
17	Implementor → Implementor	11	Extender → NONE	13	PureType → PureType	32
18	NONE → CommonState	10	Extender → Extender	12	Implementor → Implementor	32
19	NONE → FunctionObject	10	Canopy → Canopy	12	Extender → Extender	31
20	Stateless → RestrictedCreation	9.1	CobolLike → CobolLike	12	Overrider → Overrider	31

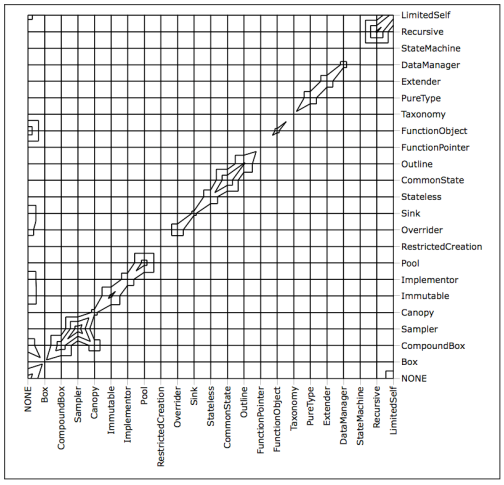

Figure 3. jEdit (rev 1-1449) bug-prone pattern evolution kinds

We observe bug-prone micro pattern evolution kinds in two different periods of the same project, jEdit. The bug rates of each micro pattern evolution kind of the two periods are shown in Figure 4 (revisions 1-500) and Figure 3 (revisions 1-1449). The bug introduction rates of the two periods are almost identical. We conclude that the bug rates of micro pattern evolution kinds of different projects are typically, but not always, similar. Bug-prone pattern evolution kinds from two different periods of the same project are very similar. We expect that quality assurance personnel could use already observed bug-prone micro pattern evolution kinds in a project to predict future bug-prone pattern evolution kinds for that same project.

4.4 Frequencies, Pattern Evolution Kinds, and Bug Rates

Since the Java class changes that cause pattern changes are infrequent (only 4 to 6% in Table 5), in this section we observe only class file changes that do not change micro patterns, such as 'NONE → NONE', 'Box → Box', and Sampler → Sampler.' We compare the frequencies, the number of the evolution kinds, and bug rates of these micro patterns. To permit cross-project comparison, we normalize each value (i.e., frequency, the number of the evolution kinds, and bug rate) by dividing each value by the sum of the values. For example, each change count is divided by the total number of changes to compute a normalized change count. The sum of normalized values is 1. The normalized values show the distribution of values among micro patterns. Figure 5-Figure 7 show the normalized values of each pattern of the three projects. For example, in Figure 5, 35% of the micro pattern evolution kinds are 'NONE → NONE,' as shown in the middle bar for 'NONE' (this is slightly higher than the 33% value for NONE → NONE in Table 4, since we have eliminated rates of class file changes that change micro patterns, and then recomputed frequencies). However, the bug introduction rate of 'NONE → NONE' is relatively low. Though Table 6 indicates that 27% of these transitions are buggy, they are only 6% of total project bugs. In contrast, the 'CommonState→CommonState' transition is found in only 1.6% of changes (see Table 4), but it contributes 9% of total project bugs. Clearly this is a dangerous type of change.

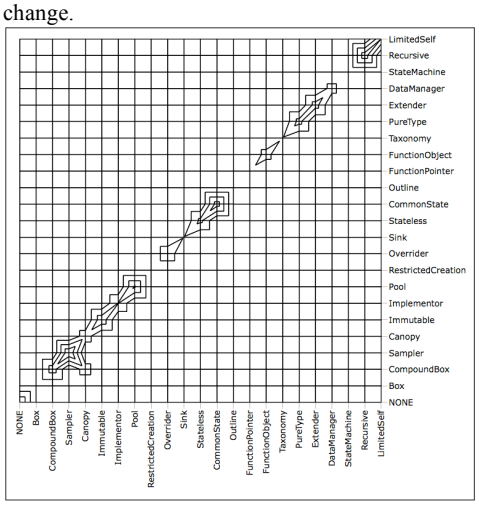

Figure 4. jEdit (rev. 1-500) bug-prone pattern evolution kinds

We observe that, in general, the fact that a pattern frequently occurs in the source code does not necessarily mean that it frequently changes. Similarly, the number of pattern changes and the bug introduction rate are not strongly correlated. Some patterns have many changes, but low bug introduction rates. There are common patterns, which occur less frequently and have small change numbers, but high bug introduction rates. For example, the 'Box', 'CompoundBox', 'Pool', 'CommonState', and 'Outline' micro patterns have high bug rates, but their frequencies and change counts are low. In contrast to that, 'Overrider' and 'Sink' micro patterns have comparatively lower bug rates.

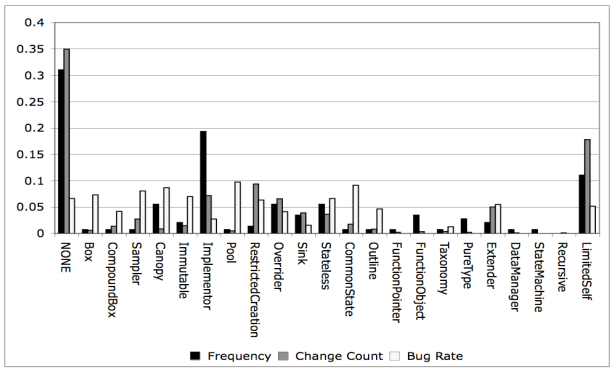

Figure 5. Micro pattern distributions, the number of changes, and bug rates of ArgoUML

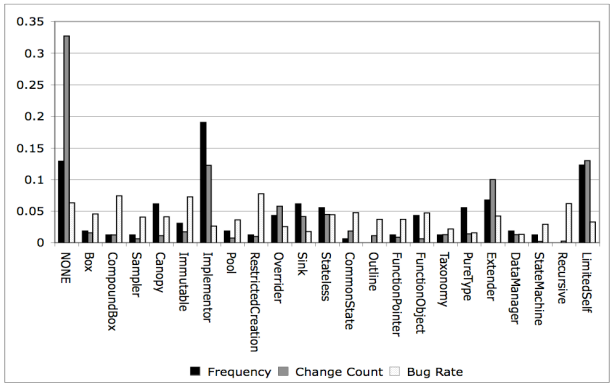

Figure 6. Micro pattern distributions, the number of changes, and bug rates of Columba

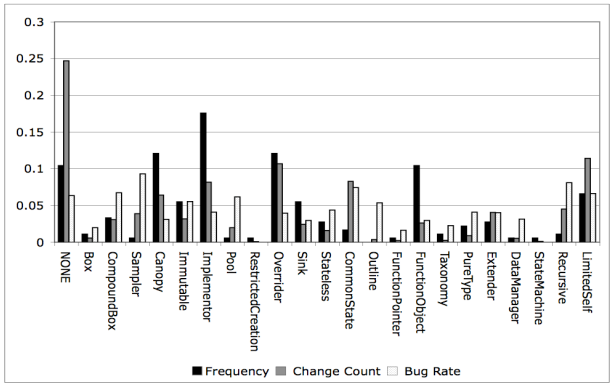

Figure 7. Micro pattern distributions, the number of changes, and bug rates of jEdit

5. DISCUSSION

5.1 Generalization

We identified the common pattern evolution kinds and bug-prone micro patterns of three projects. We showed that the three projects share some common properties, but they are not identical. We analyzed only these three projects, so it is hard to determine if our findings are broadly generalizable.

However, we showed that the bug-prone pattern evolution kinds of two different periods of the same project are very similar. This indicates that the common pattern evolution kinds and bug-prone micro patterns discovered in part of a project's history can be generalized for the remainder of the project's history.

5.2 Bug-prone Patterns

We identified common bug-prone micro patterns of the three analyzed projects, and summarize our findings in Table 7. Why are some micro patterns more bug-prone than others? Understanding bug-prone micro patterns may lead to a deeper understanding of the causes of bug-introducing changes. We also note that, since a class changes micro patterns so infrequently, most of our results are really noting correlations between individual micro patterns and bug-proneness, and not correlations between changes of micro pattern and being bug-prone.

Table 7. More/less bug-prone micro patterns

Category	Micro Pattern evolution kinds/micro patterns
Bug-prone Pattern evolution kinds	Pool → Pool, Implementor → NONE, Sampler → Sampler, CommonState → CommonState, Canopy→Canopy, Recursive → Recursive
High bug rate patterns	Box, CompoundBox, Sampler, Pool, Outline, CommonState
Low bug rate patterns	Overrider, Sink

Identifying pattern specific bugs may provide insight into the causes of bug creation. However, since identifying micro pattern specific bugs requires manual project analysis, it is very labor-intensive. In our limited explorations to date, we have not found strong examples or trends in pattern-specific bugs. Identifying trends in micro pattern specific bugs remains as future work.

5.3 Threats to Validity

There are four major threats to the validity of this work.

Systems examined might not be representative. We examined three systems. It is possible that we accidentally chose systems that have similar (or different) micro design patterns and evolution properties. Since we intentionally only chose systems that had some degree of linkage between change tracking systems and the text in the change log (so we could determine bug-introducing changes), we have a project selection bias. It certainly would be nice to have a larger dataset.

Systems are all open source and written in Java. The systems examined in this paper all use an open source development methodology and are written in Java, and hence might not be representative of all development contexts. It is possible that the stronger deadline pressure of commercial development could lead to different micro pattern change properties.

Some revisions are not compilable. To extract micro patterns from Java source code, we need to compile them and create class files

first. Analyzed open source projects contain revisions that cannot be compiled, with reasons ranging from syntax errors to missing library files. We skipped non-compilable source code, which may affect the results.

Bug-introducing change data is incomplete. We rely on the change logs to identify bug-introducing changes. Even though we selected projects that have good quality change logs, we still are only able to extract a subset of the total number of bugs. The bug change identification relies on the heuristic algorithm given in [13], so it may have false positives and false negatives.

6. RELATED WORK

Patterns in software design and implementation have been explored by many research efforts. In object-oriented designs, design patterns describe the relationships and interactions between classes or class instances and the template to manage them. In [6], Gamma et al. discussed some design patterns that are categorized into creational patterns, structural patterns, and behavioral patterns. Heuzeroth et al. [8] explored automatic design pattern detection in legacy code using static and dynamic analyses, in which patterns like Observer, Composite, Mediator, etc. are identified from Java code. In [12], Prechelt et al. presented a system called Pal that discovers structural design patterns in C++ software by examining the C++ header files. Livshits and Zimmermann combined software repository mining and dynamic analysis to discover common usage patterns and code patterns that likely encounter violations in Java applications [10]. Code-Web [11] discovers library reuse patterns in the ET++ application framework through data mining. Micro patterns are at an abstract level between design patterns and implementation patterns. Compared to design patterns, micro patterns are extractable; compared to implementation patterns that need static or dynamic analysis to discover them, micro patterns require less computation to extract.

Gil and Maman perform analysis on the prevalence of micro patterns across the Sun JDK versions 1.1, 1.2, 1.4, 1.4 and 1.4.2. They only compared distributions in each release and conclude that pattern prevalence tends to be the same in software collections [7]. We analyzed not only distributions, but also pattern evolution kinds and bug-prone change kinds.

Signature change pattern analysis [9] is similar to ours in that they try to observe signature change patterns over revisions. However, they observed only signatures change patterns, while our approach analyzes micro pattern evolution, which includes non-trivial idioms of each Java class. We also identify bug-prone patterns among identified patterns.

7. CONCLUSIONS AND FUTURE WORK

We observed the micro pattern evolution properties of three open source projects, including frequencies of micro patterns, common micro pattern evolution kinds, and bug-prone micro patterns. We found that the micro pattern distributions and common change kinds of analyzed projects are similar. The bug rates of patterns of different projects are somewhat similar. However, the bug rates of two different periods of the same projects are almost identical. We conclude that the identified bug-prone patterns from a part of a project history can be used to predict or raise awareness of the future pattern changes for the project.

We need to analyze more software projects to see if our findings can be generalized to other projects. The micro patterns are not originally designed to identify more/less bug-prone modules. We need to mine or develop new patterns to easily identify more/less bug-prone patterns. In addition, we need to mine finer granularity patterns for use at the function/method level. The software pattern evolution analysis methodology used in this paper can be reusable for other software patterns.

8. ACKNOWLEDGMENTS

Our thanks to Itay Maman and Joseph (Yossi) Gil for allowing us use the micro pattern extraction tool and for their valuable feedback.

9. REFERENCES

[1] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," 2005, http://subversion.tigris.org/.

[2] B. Berliner, "CVS II: Parallelizing Software Development," Proc. Winter 1990 USENIX Conf., Washington, DC, pp. 341-351, 1990.

[3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[4] J. W. Cooper, *The Design Patterns: Java Companion*: Addison-Wesley, 1998.

[5] R. E. Courtney and D. A. Gustafson, "Shotgun Correlations in Software Measures," *Software Engineering J.*, v. 8, pp. 5 - 13, 1992.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[7] J. Y. Gil and I. Maman, "Micro Patterns in Java Code," proceedings of the 20th Object Oriented Programming Systems Languages and Applications, San Diego, CA, USA, pp. 97 - 116, 2005.

[8] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic Design Pattern Detection," Proc. 11th IEEE Int'l Workshop on Program Comprehension, pp. 94, 2003.

[9] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Analysis of Signature Change Patterns," Proc. Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, MO, USA, pp. 64-68, 2005.

[10] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," Proc. 2005 European Software Engineering Conf. and Foundations of Software Eng. (ESEC/FSE 2005), Lisbon, Portugal, pp. 296-305, 2005.

[11] A. Michail, "Data Mining Library Reuse Patterns in User-Selected Applications," Proc. 14th International Conference on Automated Software Engineering, Cocoa Beach, Florida, USA, pp. 24–33, 1999.

[12] L. Prechelt and C. Krämer, "Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns," *J. Universal Computer Science*, vol. 4, pp. 866-882, 1998.

[13] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, MO, USA, pp. 24-28, 2005.