Graph-based Mining of In-the-Wild, Fine-grained, Semantic Code Change Patterns

Hoan Anh Nguyen*, Tien N. Nguyen†, Danny Dig‡, Son Nguyen†, Hieu Tran†, and Michael Hilton§
*Computer Science Department, Iowa State University, USA, Email: hoan@iastate.edu
†Computer Science Dept., The Univ. of Texas at Dallas, USA, Email: {tien.n.nguyen,sonnguyen,trunghieu.tran}@utdallas.edu
‡Computer Science Department, Oregon State University, USA, Email: digd@eecs.oregonstate.edu
§School of Computer Science, Carnegie Mellon University, USA, Email: mhilton@cmu.edu

***Abstract*—Prior research exploited the repetitiveness of code changes to enable several tasks such as code completion, bug-fix recommendation, library adaption, etc. These and other novel applications require accurate detection of semantic changes, but the state-of-the-art methods are limited to algorithms that detect specific kinds of changes at the syntactic level. Existing algorithms relying on syntactic similarity have lower accuracy, and cannot effectively detect semantic change patterns. We introduce a novel graph-based mining approach, CPatMiner, to detect *previously unknown* repetitive changes in the wild, by mining *fine-grained semantic code change patterns* from a large number of repositories. To overcome unique challenges such as detecting *meaningful* change patterns and scaling to large repositories, we rely on fine-grained change graphs to capture program dependencies.**

We evaluate CPatMiner by mining change patterns in a diverse corpus of 5,000+ open-source projects from GitHub across a population of 170,000+ developers. We use three complementary methods. First, we sent the mined patterns to 108 open-source developers. We found that 70% of respondents recognized those patterns as their meaningful frequent changes. Moreover, 79% of respondents even named the patterns, and 44% wanted future IDEs to automate such repetitive changes. We found that the mined change patterns belong to various development activities: adaptive (9%), perfective (20%), corrective (35%) and preventive (36%, including refactorings). Second, we compared our tool with the state-of-the-art, AST-based technique, and reported that it detects 2.1x more meaningful patterns. Third, we use CPatMiner to search for patterns in a corpus of 88 GitHub projects with longer histories consisting of 164M SLOCs. It constructed 322K fine-grained change graphs containing 3M nodes, and detected 17K instances of change patterns from which we provide unique insights on the practice of change patterns among individuals and teams. We found that a large percentage (75%) of the change patterns from individual developers are commonly shared with others, and this holds true for teams. Moreover, we found that the patterns are not intermittent but spread widely over time. Thus, we call for a community-based change pattern database to provide important resources in novel applications.

***Keywords*-Semantic Change Pattern Mining, Graph Mining**

I. Introduction

Prior studies have confirmed that *software changes are repetitive* [2], [25], [28]. Several researchers have exploited the repetitiveness of code changes to detect change patterns, and leverage them to support several tasks such as adapting client code to the changes in a library [5], [6], [31], language migration [47], recommending bug-fixes and code examples [19], [16], [32], [46], code completion [4], [26], [27], [39], and usage patterns [44]. Previous approaches focused on mining the patterns of specific types of changes (*e.g.,* refactorings, bug fixes, API migration) at the *syntactic level*. Thus, the mining's reliance on syntax alone lowers its accuracy. This limits the potential exploitation of repeated code changes.

In software development, engineers make changes with a certain high-level programming task in mind. Thus, mining the high-level change patterns will be meaningful and beneficial to developers as well as for tool builders. However, *semantic change patterns with relations among program elements might not be well captured by fine-grained syntactic changes*. For example, *"adding a null check for an argument of a method call"* consists of multiple atomic syntactic changes. Only if the variable used in null checking is directly passed as an argument of a method call, those syntactic changes can form that semantic change. Moreover, *to identify a higher-level semantic change pattern, one might have to also include unchanged statements* in order to completely make up a meaningful pattern. For example, the method call in the above example could be unchanged, but needs to be included in that pattern. However, this adds more noise in the mining process. In other cases, the atomic changes of a semantic change pattern might not be in contiguous source code due to irrelevant changed statements (as we show in Section II). This further increases the amount of noise.

To advance the field and enable novel applications, in this work, we detect previously unknown semantic change patterns in source code. Specifically, we introduce a novel graph-based mining approach to detect *in-the-wild, high-level, semantic code change patterns* from an *ultra large number of atomic code changes* in several open-source projects.

To catalyze fundamental advances that exploit repeated changes, one has to overcome unique challenges. First, how can we find changes that are *meaningful* to developers and encode a *semantic change pattern*? Previous work [10], [13], [25] has recorded key strokes as the programmer types in an IDE, and then grouped fine-grained atomic changes that make up a change pattern. However, in a version control repository, the order of such atomic changes is not recorded, leading to two challenges: i) a pattern detection algorithm operating on sequences of changes, *e.g.,* the frequent sub-sequence mining algorithm, does not work as the order of code elements might not be the order of changes, and ii) the actual semantic dependencies among changed code elements are lost if a mining algorithm, *e.g.,* frequent itemset or association rule mining, treats the changes as belonging to an un-ordered set of independent items. In our formative experiment, the frequent itembag mining algorithm picked up trivial patterns that represent unrelated, though common changes. For example, *adding a simple name (a variable)* and *adding a* null *literal* are two of the most common atomic changes. In many cases, they were detected as a pattern of two frequent items even though those two atomic changes did not actually go together to compose any meaningful pattern such as *"adding a null check"*. As a consequence, meaningful patterns were ranked lower than trivial, meaningless ones. Removal of common atomic changes does not help because they are key components of true patterns. Moreover, how can we detect changes that are *common and representative* across the developer community? Our algorithm must handle a *large number of fine-grained changes*.

To address those challenges, our *graph-based* approach connects program elements involved in the change, when they have data or control dependencies. The use of graph is a departure point from AST-based approaches [23], [25], enabling the detection of semantic change patterns. A fine-grained change graph (a *change graph* for short) has two sub-graphs representing the code before and after changes. The nodes in each sub-graph represent program elements similarly to the nodes in a program dependence graph, but at the expression level [8]. The edges among the nodes in a sub-graph represent the relations among the program elements, *e.g.,* the data and control dependencies. The edges across two sub-graphs represent the correspondences between the nodes in the old and new code.

We implemented our algorithm in a tool, CPATMINER, and used it on a corpus of 5,832 repositories to analyze 297M SLOCs, with real-world code changes encapsulated in 1.1M fine-grained change graphs with 11M nodes. For evaluation, we employ 3 complementary methods that assess its performance and usefulness from different angles. First, to assess if CPATMINER finds patterns that are meaningful to real developers, we survey open-source developers. We sent 451 requests about the mined change instances and patterns to their respective authors. We received 108 responses and 70% of them confirmed that change patterns discovered by CPATMINER are correct as part of their frequent code changes. 79% of them provided the names for the patterns and 44% indicated that they like to have tools to automate such fine-grained change patterns in the future IDEs. To study the diversity of instances of change patterns, using our survey responses, we manually investigated and classified all mined patterns into the development activities: *adaptive* (9%), *perfective* (20%), *corrective* (35%) and *preventive* (36%). Among the preventive changes, 35% are *refactoring*.

Second, we conducted a comparative study to evaluate our tool against the syntax-based technique used in the existing approaches [7], [9], [22], [23], [25]. Our result shows that our tool detects 2.1x more meaningful patterns than the baseline.

Third, to evaluate the effectiveness of CPATMINER and gain a thorough understanding about the practice of change patterns, we use CPATMINER to mine for patterns in a corpus of 88 GitHub repositories with *longer version histories* consisting of 164M SLOCs from 13K developers. CPATMINER detected 17K change patterns. We further studied the characteristics of change patterns in both space and time dimensions. For space, we studied the *commonality* and *uniqueness* of the change patterns among individual developers in the same/different teams and among individual teams. For time, we studied the *temporal distribution* of the instances of change patterns over a period of time. We found that 45% of the developers have more than 90% of their patterns repeated by others. 46% of the change patterns have been repeated for more than a month period.

Based on these results, we launch a community call to action related to fine-grained change patterns. We offer several actionable implications for researchers, tool builders, and developers. First, for researchers, we confirm that repetitive code changes exist in a much larger, varied dataset of thousands of projects and across tens of thousands of developers. We now have the evidence that previous tools [3], [4], [5], [6], [16], [19], [26], [27], [31], [32], [36], [39], [44], [46], [47] that exploited the repetitiveness of code changes are *empirically justified*. Researchers can replace their own database of change patterns with our significantly larger (25,000 change patterns) and varied (170,000+ contributors) database, thus making existing applications more powerful. Researchers can also use our tool/dataset to build novel applications, *e.g.,* tutoring systems for teaching a novice programmer how to learn programming, or teaching an expert programmer how to use advanced language features [14], [18]. Language and API designers can use our tool/dataset to learn how programmers commonly change their code to use new features, harnessing feedbacks from crowdsourcing. Tool builders could find the inspiration from common change patterns to automate several repetitive tasks in IDEs. This paper makes the following contributions:

A. Representation: A graph-based fine-grained change representation to mine *semantic* change patterns by capturing both the changes to code elements and their interdependencies.

B. Evaluation Results:

- **Meaningful patterns.** A high percentage of respondents in our survey confirmed that the detected patterns are meaningful and indicated preferences to have automated tools.
- **Commonality.** The majority of change patterns of a developer or in a team is commonly shared with others.
- **Temporal Distribution.** Same developer or team repeats the same changes over time, even over a month or a year.

C. Implications: We present practical, actionable implications of our findings for researchers, language and API designers, tool builders, and developers.

D. Community-based change database: To significantly improve existing applications and to catalyze novel applications of change patterns, we offer our dataset of 17K change patterns as an invaluable resource for our community, available anonymously at: https://nguyenhoan.github.io/CPatMiner/.

II. MOTIVATING EXAMPLES

We first illustrate the challenges of mining semantic, fine-grained change patterns and then explain the motivation for

Before change

```
1
2
3  File  f  = new File("Output.txt");
4  FileOutputStream s = new FileOutputStream(f);
5  i = str.indexOf("pattern",  start);
```

After change

```
1+ if (start <= 0)
2+    return;
3  File  outFile  = new File("Output.txt");
4  FileOutputStream outStream = new FileOutputStream(outFile);
5  i = str.indexOf("pattern",  start);
```

Fig. 1: Adding a Negative Check on an Argument of a Method Call

Before change

```
1  i = str.indexOf("pattern",  start);
2
3
4  sub = str.substring(i);
```

After change

```
1  i = str.indexOf("pattern",  start);
2+ if (i <= 0)
3+    return;
4  sub = str.substring(i);
```

Fig. 2: Adding a Negative Check on the Return Value from a Method Call

Before change

```
1
2
3  i = str.indexOf("pattern",  start);
```

After change

```
1+ if (mode <= 0)
2+    return;
3  i = str.indexOf("pattern",  start);
```

Fig. 3: A Change Contains Unrelated Atomic Changes

our solution. Fig. 1 displays a change pattern in which a developer *"adds a negative check on the second argument of a call to* `String.indexOf` *in Java"* before calling the method (lines 1–2, After change). Notice that the statements on lines 3–4 changed as well, though those changes are irrelevant to this change pattern and only add noise. On the other hand, despite that the statement `'i = str.indexOf...'` did not change, it is actually part of this change pattern. Thus, a semantic change pattern could include syntactically unchanged statements.

(**Observation O1:**) *syntactically un-changed program statements could carry important semantic change elements for a semantic change pattern.*

A mining approach that captures only the changes to syntactic units (*e.g.,* using tree editing operations between the ASTs before and after the change), would miss the method call at line 5 (After change). Moreover, notice that the changes at lines 3–4 are nearby yet irrelevant in the change pattern.

(**Observation O2:**) *the atomic changes of the same pattern might not be on the contiguous lines of source code.*

Thus, we need a change representation that relates syntactically un-changed program elements with the changed elements.

Fig. 2 displays another change pattern *"Adding a negative check on the return value from a call to* `String.indexOf`*"*. An approach that represents an atomic change via the tree editing operations will encode this change as the following set: 1) `adding if`, 2) `adding an identifier ID`, 3) `adding an operator OP <=`, 4) `adding literal 0`, and 5) `adding return`. This set *exactly matches* the set of tree editing operations representing the change in Fig. 1. Thus, that approach will (mis)classify the change in Fig. 1 and the change in Fig. 2 as the instances of the same change pattern.

(**Observation O3:**) *an approach using only syntactic changes for change representation will misclassify the instances of different change patterns.*

Therefore, we cannot treat the atomic changes as individual elements in a set, and we need a *representation able to semantically connect the atomic changes to form a meaningful pattern.*

Fig. 3 shows another example in which the unrelated atomic changes could be mistakenly grouped into a pattern if they are represented as a set of atomic tree editing operations as in existing work [25], [28]. In this example, the set of editing operations that occurred on lines 1–2 (After change) matches the sets of operations for the previous two examples (Figs 1 and 2). Note that, in this example, the editing operations such as `adding if`, `adding ID`, `adding the OP <=`, `adding literal 0`, and `adding return` are popular atomic changes, yet non data-dependent, and should not be grouped to form the same change pattern as before.

(**Observation O4:**) *an approach that relies only on syntactic changes would incorrectly classify frequent but un-related atomic changes into an incorrect pattern.*

From **O1-O4**, we propose a graph representation that encodes **data** and **control dependencies** to connect relevant atomic changes. In Fig. 1 (After change), the dependencies help connect the addition of the ID `start` at line 1 with the method call `String.indexOf` at line 5, despite that they are far apart in the source code and that the statement at line 5 does not change. In fact, with the integration of dependencies, *the assignment at line 5 (After change) in Fig. 1 is considered as semantically changed with regard to program dependencies/flows because after change, that assignment is only executed in the false branch of the* `if` *statement at line 1.* Dependencies/flows help differentiate also the instances in Fig. 1 and Fig. 2 because the first change relates to the argument of the method call `indexOf()`, while the second one relates to the return value of the call. Finally, for the change in Fig. 3, the algorithm does not group the changes at lines 1–2 with the statement at line 3 since they do not have data dependency. Next, we present our mining tool CPATMINER.

Before change

```
− for ( Iterator <Element> it=o.getElements().iterator() ; it .hasNext();){
−    myElements.add(it.next());
  }
```

After change

```
+ for (final Element element1 : o.getElements()) {
+    myElements.add(element1);
  }
```

Fig. 4: A Change Pattern: "Iterator Replaced With `foreach`"

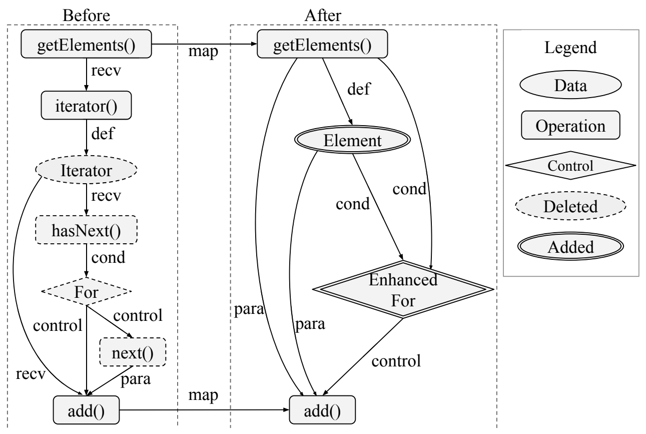


Fig. 5: Graph Representation fgPDG for Change in Fig. 4

III. Semantic Fine-grained Change Representation

A. Fine-grained Code Representation

1) Graph-based Representation: From our observations, we choose a graph representation that captures program dependencies. We create a graph-based program representation that captures the control/data dependencies as in Program-Dependence Graphs (PDGs) and uses the API elements as in Object Usage Graphs (GROUMs) [33]. Our representation, called *fine-grained program dependence graph* (fgPDG), is augmented with richer types of nodes and edges to capture fine-grained program elements at the expression level and the dependencies, including usages with exception handling. Let us detail fgPDG.

2) Fine-grained PDG: A fgPDG is a directed graph that represents the data/control dependencies among fine-grained program elements at the expression granularity. It also contains the nodes representing API elements. A node represents a method call, a variable declaration, an operation, or a control statement (*e.g.,* iteration or condition). An edge coming into a node represents the data on which the node depends or the control flow on which the execution of the node depends. There are three types of nodes: *data*, *operation*, and *control* nodes, and two types of edges: *control* and *data* edges. Fig. 5 shows the fgPDG for the code in Fig. 4 before and after the changes.
Data Nodes represent variables, field accesses and constants. In Fig. 5, Element in the subgraph after the change is a data node.
Operation Nodes represent operations on data, *e.g.,* arithmetic, relational, logical, bit-wise, type comparison, cast operations, array accesses, and method calls. Their labels are the operations' types, *e.g.,*!=, except for a call whose label is its declaring type and name, *e.g.,*Iterator.hasNext() in Fig. 5 (type not shown).
Control Nodes represent control statements, which include If for branching, For, EnhancedFor (see Fig. 5), While and Do for loops, and Try and Catch for exception handling. In our representation, we transform a switch statement into nested if statements.
Control Edges represent control relation between the statements/operations and the control nodes on which their executions depend. In Fig. 5 (before change), there are 2 control edges from the For node to the method call nodes next() and add().
Data Edges represent the data flow between fgPDG nodes. Each edge has a label specifying the type of the data flow. A **def** edge connects a data node or an operation node to another data node as its definition in an assignment. This definition node, in turn, could be connected to its references via **ref** edges. A **recv** edge connects a receiver object to a method call, *e.g.,* in Fig. 5, Iterator object is the receiver of the method call add(). Similarly, we defined **para, cond**, and **qual** for parameters, conditions in a condition statement, and qualified names.
We use control and data edges to connect nodes with dependencies to address the issues listed in *Observations* **O3** and **O4**.

B. Fine-grained Change Representation

1) Change Graph: We represent the changes from the old version to the new one of a fragment in a change graph by connecting the two fgPDGs of the fragment before and after the changes. An edge labeled **map** connecting a node u in the fgPDG before the changes to a node v in the fgPDG after if u and v are the corresponding un-changed nodes. We also annotate the nodes that are removed and inserted in the two graphs before and after the changes, respectively.

2) Fine-grained Change Extraction: To derive the fine-grained changes in each changed method, we first compute the tree-based differences using an AST differencing algorithm [29]. Given a pair of methods before and after the change, we parse them into ASTs and finds the mapping between two sets of nodes in two trees. The key idea is to map two nodes based on their node types and the structural similarity between the two subtrees rooted at them. The unmapped nodes are considered as *deleted* in the old tree or *added* in the new tree. Along with the mapping, the algorithm provides the information on whether the mapped nodes have change in their labels or in their descendants. From the mapped, deleted, and added AST nodes, our algorithm links the corresponding nodes in the two fgPDGs before and after the changes. For example, in Fig. 5, the two fgPDGs before and after the changes are named Before and After. The edge labelled **map** connects the two corresponding un-changed nodes labelled getElements().

As motivated in *Observation* **O1**, *syntactically un-changed program elements could be part of a semantic change pattern*. We identify these elements by leveraging their data and control dependencies with the changed elements as follows. We mark an operation node as changed if it has a changed node incoming to it via a data edge because the data flow into it has been changed. Moreover, we also consider an operation or control node as changed if it is controlled by a deleted or added control node because the control flow to it has been changed. Its mapped node in the other version of the changed code is also considered changed and connected to it via a *map* edge.

3) Change Closure: We build the change closure by connecting edges between change nodes having data dependencies transitively via intermediate data and/or operation nodes. This strategy helps address the issue in *Observation* **O2** (atomic changes of the same pattern might not be on contiguous lines of code, but have data or control dependency via intermediate (un-)changed elements). It also deals with the problem where the same code change could be realized into different code fragments that are different only in the use of temporary variables for computations, thus, could be represented by different change graphs. In the change graph for the code after the change in Fig. 5, the operation node getElements() is transitively connected to both the operation node add() via a *para* edge and the control node EnhancedFor via a *cond* edge even though it is not a direct parameter of the call nor the condition of the controlling loop statement. For clarity, we did not include transitive edges in the change graph in Fig. 5.

C. Change Pattern Mining Algorithm

*Definition 1 (*Semantic Change Pattern): *A change is a semantic change pattern if its corresponding fgPDG is a connected graph and repeated (isomorphic to each other) at least σ times (user-configurable) in the change graph dataset. All of its repeated occurrences are counted as instances of the pattern.*

We define two changes as being repeated if and only if their corresponding change graphs are isomorphic. Our mining algorithm (Listing 1) takes as input a set of change graphs G constructed from all change commits in a version control repository and produces the frequent subgraphs as patterns. We first identify frequently occurring nodes to build the set of size-of-1 patterns (line 2). Then, we recursively extend each size-of-1 pattern to find larger patterns (line 3). To extend a pattern p of size k, we start with all occurrences of p (lines 7–8) and generate from them all possible subgraphs of size $k+1$. To generate the subgraphs of size $k+1$ from a subgraph i of size k, we explore i's adjacent nodes, one at a time (line 18).

We have two key ideas to make it scalable. First, to extend a subgraph, the algorithm adds an *adjacent node*, *i.e.,* a node that is connected to one of the nodes in the subgraph by an edge. CPatMiner distinguishes different types of nodes. The intuition is that different types have different semantics and some extensions lead to meaningful larger patterns, while others do not. The algorithm relies on the subgraph and the type of the adjacent node to decide whether to extend that node. For each adjacent node, we decide whether it is suitable to extend i as follows: 1) for operation nodes, method-call nodes are always suitable (line 19), and other operations are suitable only if they have at least one incoming and one outgoing edge to i (line 20), because thus they consume and produce some data in i; 2) for data and control nodes, they are suitable only if

Listing 1: CPatMiner Change Pattern Mining Algorithm

```
1  function CPatMiner($G$: Set[fpPDG])
2    $P$ = {size-of-1 patterns from $G$}
3    for $p$ in $P$: extend($p$, $P$)
4    return $P$
5
6  function extend($p$: Pattern, $P$: Set[Pattern])
7    $E = \emptyset$
8    for $i$ in $p$.instances: $E = E \cup$ generate_extensions($i$)
9    $C$ = isomorphic_clusters($E$)
10   for $c$ in $C$: if frequency($c$) $< \sigma$ or $c \in P$: $C = C \setminus \{c\}$
11   if $C \neq \emptyset$
12     $c$ = most_frequent($C$)
13     extend(Pattern($c$), $P$)
14   else:   $P = P \cup \{p\}$
15
16 function generate_extensions($i$: fgPDG)
17   $extensions = \emptyset$
18   for $n$ in adjacent_nodes($i$)
19     if  is_method_call($n$)
20       or (is_operation_node($n$) and has_in_out_connection($n$, $i$))
21       or (is_data_node($n$) and has_out_connection($n$, $i$))
22       or (is_control_node($n$) and has_out_connection($n$, $i$))
23     $extensions = extensions \cup \{i \oplus n\}$
24   return $extensions$
```

they have an outgoing edge to i (line 21), because a data or control node should not be a sink in a pattern graph. If an added node has a mapped node (via its *map* edge), the mapped node is also added so that the pattern contains the change data of that pair of mapped nodes. Deleted and inserted nodes do not have their mapped nodes. When a node is added, all edges connecting from/to it are added to the extended subgraph.

Second, the algorithm does not try all possible extensions of a frequent subgraph, but only the most frequent one. Since different occurrences may be extended with different adjacent nodes having different labels, not all extensions are isomorphic to one another. Therefore, in the next step, we cluster extensions into groups of isomorphic graphs. To reduce the complexity of graph isomorphism detection, we use a heuristic from an existing work [30] that combines graph vectorization and hashing to efficiently cluster graphs into isomorphic buckets. From all clusters, we remove ones that do not have enough occurrences to be patterns or that are already discovered (line 10). After this filtering, if there still exist clusters of extension graphs, we select the one that has the most occurrences, create a pattern, and recursively search for its extensions (lines 11–13). This greedy strategy avoids the combinatorial explosion problem of exhaustive search with backtracking and makes our mining scale to a large number of large graphs. When p can no longer be extended (*i.e.,* maximal), it is added to the set of final patterns P (line 14).

IV. Empirical Methodology

We empirically evaluate our tool with the following questions:

RQ1: ***Meaningful Patterns.*** Do the change patterns mined by CPatMiner make sense to the developers in their programming tasks? This question addresses the quality of the change patterns mined by our tool, and their relevance to developers.

RQ2: ***Comparison.*** What is the contribution of our *graph-based representation* and *graph mining* to our tool's effectiveness? This evaluates the *improvement brought by our graph-*

based approach compared with the syntax-based representation in the state-of-the-art approaches [25], [22], [23], [9], [7].

RQ3: ***Activities.*** What types of activities (adaptive, perfective, corrective, preventive) do the change patterns belong to? This question addresses whether change patterns are diverse.

RQ4: ***Commonality of Patterns.*** How much do individual developers or teams share the change patterns? This question helps understand the generality of patterns and their utility for a wide population of developers. It decides if developers would be served by building a community database of such patterns.

RQ5: ***Temporal Distribution.*** What is the temporal distribution of change patterns? Are the instances of change patterns scattered over a long period of time or clustered within a short period? This addresses the feasibility of novel applications, *e.g.,* those using change patterns to recommend new code changes.

A. Surveying Real Developers

To determine if CPATMINER detects meaningful patterns, we ask real developers who are the authors of the changes if our mined patterns are relevant for their tasks. This allows us to get an answer from the developers who know the intention of their code and can explain the rationale for the changes. We first used CPATMINER to mine the change patterns in a corpus from GitHub. To eliminate toy or experimental repositories, we used GitHub APIs to search for Java repositories that have at least 5 stars (as given by individual GitHub users). This gave us 21,746 repositories. We further filtered out the ones that were not active during our investigation period, that is, we kept only the repositories with at least 50 commits and at least one recent commit. At the end of each week in the period, we ran our tool on the 50 most recent commits of the repositories in the corpus. We only ran it on those recent commits so that the changes are still fresh in the developers' memories. We then contacted the developers who had committed these changes, and asked them the four questions:

- Q1 *Is the change at these lines similar to another change in the past? (yes, no, not sure)*
- Q2 *Can you briefly describe the change and why you made it? (for example, checking parameter before calling the method to avoid a Null Pointer Exception)*
- Q3 *Can you give it a name? (for example, Null Check)*
- Q4 *Would you like to have this change automated by a tool? (Yes, No, Already automated)*

The purpose of *Q1* is to determine if the author believes this change to be repetitive. The change in the past could belong to the same or different authors. We wanted to study whether the author recalled having made a similar change in the past. Being aware of patterns of changes increases the chance of profitability of such changes in future, novel tools. It also indicates that the detected patterns are meaningful. With *Q2*, we aim to uncover the rationale behind the change. We asked *Q3* to determine if this change is meaningful to developers to the point that they could describe by a succinct name. A name for such an in-the-wild change would indicate a useful operation in future IDEs. *Q4* determines the developers' desire to have this change automated by future, novel tools.

TABLE I: Collected Datasets

	Survey Corpus	Depth Corpus
Projects	5,832	88
Total source files	35M	1M
Total SLOCs at last snapshots	4.1B	164M
Developers with analyzed commits	171K	13K
Java code change commits	292K	88K
Total changed files	592K	291K
Total SLOCs of changed files	216M	81M
Total changed methods (graphs)	824K	322K
Total AST nodes of changed methods	92M	34M
Total changed graph nodes	8M	3M

We extracted developers' emails from Git commits. We only contacted developers one time for our study, even if they have authored multiple commits. We contacted 451 developers and received 108 responses, for a response rate of 24%. To achieve this high response rate, we used the state-of-the-art *Firehouse interview technique* [24], [43], in which one would wait for an event to happen and follow up immediately with involved persons. The details on the event are still fresh in the persons' minds, thus, they are more willing and freshly remember them.

B. Mining Code Corpus

1) Data Collection: To answer more in-depth questions about the practice of change patterns in the developer community, we use a different corpus than the one we used in the survey. For the survey we wanted the changes to be still fresh in the mind of developers, thus, we used the recent change histories of the files (including up to the last 50 commits). To answer the commonality and temporal distribution of patterns, we need projects with a long development history. Thus, we analyze fewer projects but mine patterns deep in the version history of changes. To obtain high-quality, team projects from GitHub, we kept only Java projects that satisfy three criteria: 1) have at least 100 stars, 2) have at least 10 committers, and 3) have at least 1,000 commits with Java source code changes. This gave us 88 projects. Since the numbers of commits greatly vary among these projects (from thousands to hundreds of thousands), we wanted to normalize the number of commits. Thus, for each project we collected the most recent 1,000 commits with Java code changes. Table I (Depth Corpus column) shows our depth corpus. In their last snapshot, these projects contain 1M Java source files and 164M SLOCs. CPATMINER processed all commits and parsed 291K changed source files. For each of these changed files, it built fgPDGs, thus the total size of analyzed code is 81M SLOC. CPATMINER detected 322K changed graphs with a total of 3M nodes.

2) Experimental Settings and Metrics: We set the minimum frequencies of repeated graphs σ to be 3 adheres to the Rule of Three [42], a common recurrence measure in pattern analysis. We aimed to study *commonality* and *temporal distribution* of change patterns. We posit that the change patterns performed on different projects and by different programmers have different degrees of commonality and temporal distribution. Thus, to evaluate the impact of the project's culture and individual developer's habits, we designed two settings.

Developer Edition. We considered all the changes from all developers in the projects collected in the depth corpus. We used our tool to mine all change patterns and grouped them into the patterns corresponding to individual developers. We distinguished developers based on their full names and emails.

Team Edition. We also mined the patterns from all the changes in all the projects but we placed the mined change patterns into the groups corresponding to the individual projects.

For each of the two editions, we studied the commonality and temporal distributions of patterns for individual developers and teams. To study commonality of change patterns, we consider two instances of change patterns as repeated, if their *fine-grained change graphs* are isomorphic. For the developer edition, we compared the patterns of an individual against the ones from others and measured the percentage of the patterns from each individual that are commonly shared. For the team edition, we measured the same percentage but for the patterns of a project/team. To study temporal distribution of patterns, we measured the numbers of days between the instances of the same patterns. We then computed the percentages of the patterns that are repeated within different intervals.

3) Comparative study: For RQ3, we implemented a baseline approach (BASE) using the *syntax-based representation of changes* as in [25], [22], [23], [9], [7] with the frequent itembag mining algorithm [25] to group the changes. We first ran our tool and BASE on the same survey corpus. We wrote a tool to search for the instances of patterns mined by both approaches that occurred in the same methods of the same files in the same commits. The rationale was that one could easily compare the quality of the patterns *mined by the two tools when the pattern instances are reported on the same code and change commits*. We randomly selected 100 pairs of pattern instances. In each pair, each instance is from one tool and both instances are in the same method of the same file in the same commit. For each method, we chose only one instance. For pattern evaluation, we recruited two human subjects with more than 8 years of programming experience, who were not involved in the solution development. Each subject was given 1) the original commit and method, and 2) a pair of pattern instances (A, B) from the tools with their identities hidden. Each subject was asked to examine the commit to see what are the changes for that method and their purposes. (S)he evaluated which pattern was more meaningful based on the criteria that which one reflects better the original purpose of the commit without knowing the source of any pattern. There are 4 outcomes: A is better than B, B is better than A, both are the same, and undecided.

V. Empirical Results

A. Survey Results (RQ1)

Results. Table II summarizes the answers for RQ1. The majority of respondents (69% of the developers) confirmed that CPATMINER did indeed *correctly detect* their repeated changes as patterns. This validates that it does find meaningful, repeated change patterns that developers recalled having made before.

TABLE II: Survey Responses (108 Total Answers)

Q1 *Is the highlighted change at these lines similar to another in the past?*				
Yes	**No**	**Not Sure**		**No Answer**
69% (74)	22% (24)	6% (7)		3% (3)
Q3 *Can you give it a name?*				
Yes	**No**			**No Answer**
79% (85)	18% (20)			3% (3)
Q4 *Would you like to have this change automated by a tool?*				
Yes	**No**	**Not Sure**	**Already Automated**	**No Answer**
44% (48)	34% (36)	9% (10)	10% (11)	3% (3)

It is important that the detected change patterns be meaningful (Q3). To determine if our tool finds meaningful patterns, we asked each developer to give the detected pattern a succinct name. As seen in Table II, 79% of developers were able to name the detected patterns. Despite that some developers did not recall their changes, they actually recognized the meaningful patterns by giving them succinct names. Note that the catalog of refactorings to design patterns [11], [12] represent good practice of code restructuring changes, *e.g.,"Move Embellishment to Decorator"*, *"Replace Conditional with Polymorphism"*. These detected in-the-wild change patterns with succinct names could be promoted to be standard code refactorings to design patterns.

For Q4, we asked developers whether they would want this change pattern to be automated by a tool. 43% of responders indicated their preferences to have the detected change patterns to be automated. While 13% is unsure, 34% were not enthusiastic about the changes being automated. Upon further examination of their explanations, we found that some developers conflated their desire for tool automation with their belief whether the automation is feasible. For example, *D15* said: *"No, the tool would need to be able to make decisions based on the end use of the API which I would assume is not possible... (I might be wrong :))"*. Participant *D108* wrote *"I think it is very challenging which makes your research promising."*.

B. Diversity of Change Patterns (RQ3)

We manually classified all mined change patterns into different development activities: *adaptive* (9%), *perfective* (20%), *corrective* (35%) and *preventive* (36%). Among preventive changes, *35%* of them are *refactoring*. Interestingly, 11% participants stated that the detected change patterns have already been automated by IDEs (*e.g.,"Replace functor with lambda"*, *"Iterator replaced with foreach"*). Many other change patterns mined by our tool have not been automated yet, thus could be considered to be promoted as standard operations in IDEs.

C. Examples from our Results

Let us show some of the change patterns in our results. Fig. 6 shows the change from project anHALytics in GitHub. For better exception handling with properly closing resources, the developer manipulated the SQL statement before sending to the server. (S)he removed statement.close() and introduced closeQuietly(statement) inside the newly added finally statement. (S)he repeated this pattern in three other methods for updating, deleting, and finding a person in a database. (S)he confirmed the change as common and gave it the name *"closing resources"*. (S)he also indicated preference to have it automated by a tool.

Before change

```
1 boolean isIdentifierIfAlreadyExisting (...) throws SQLException {
2   ...
3
4   ...
7   ResultSet rs = statement.executeQuery();
8   ...
12  statement.close();
```

After change

```
1 boolean isIdentifierIfAlreadyExisting (...) throws SQLException {
2   ...
3+ try {
4     ...
7     ResultSet rs = statement.executeQuery();
8     ...
11+ } finally { closeQuietly(statement); }
```

Fig. 6: "Closing resources" Change Pattern in Project anHALytics

Before change

```
1  String id = Jobjet_event.getString("id");
2  if (!this.eventDao.isExisting(id)) {
3
4
5      ...
6      this.eventDao.add(event);
7  }
```

After change

```
1  String id = Jobjet_event.getString("id");
2  if (this.eventDao.isExisting(id)) {
3+         this.eventDao.update(event);
4+ } else {
5      ...
6      this.eventDao.add(event);
7  }
```

Fig. 7: "Adding a Condition Branch" Change Pattern in Project ApplETS

Before change

```
1 Collection<HighlightInfo> highlights1 = action1.getHighlights();
2 ArrayList<HighlightInfo> list = new ArrayList<HighlightInfo>();
3  for (HighlightInfo info : highlights1){
4      list.add(info);
5 }
```

After change

```
1 Collection<HighlightInfo> highlights1 = action1.getHighlights();
2 List <HighlightInfo> result = new ArrayList<HighlightInfo>();
3 result.addAll(highlights1);
4
5
```

Fig. 8: "Replacing Iterating and Adding one Element at a Time with addAll" Change Pattern in Project IntelliJ

Fig. 7 shows an example in which a developer fixed event handling. At line 2, (s)he negated the condition of the if statement and added a new update statement for the event in the true branch. (S)he moved the old code into the newly created false branch. This change pattern was repeated several times in the methods for updating, searching, and checking existence of events. (S)he wrote that *"I would definitely love a tool that checks my code and propose me advanced refactoring"*.

Fig. 8 shows an instance of another popular change pattern in project IntelliJ. In the previous version, the element of HighlightInfo was collected into a list with the use of ArrayList.add inside a for loop. In the new version, the developer replaced the for loop with the call to List.addAll operating on the collection action1. We also have renaming of the variable list into result (line 2).

This illustrates two important implications. First, if the library has provided the API addAll and a developer did not know it, this mined pattern is a good resource to learn that alternative. Second, if the library did not have this feature yet, its designer could learn from the mined pattern on how developers use the API add(.) and need such a new feature as addAll(.).

D. Comparative Study (RQ2)

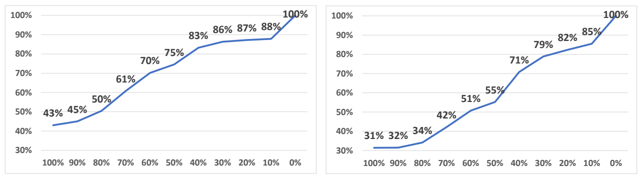
As seen in Table III, while both tools reported the mined patterns on the same code in the same commits, human subjects found that in 50% of the cases, the patterns mined by our tool reflect better the original purposes of the changes reported in the commit logs. The BASE tool won only in 3% of the cases while in 30% of them, both tools reported equally meaningful patterns. Detailed results are presented in our website [1]. We further studied the patterns that the syntax-based BASE tool cannot detect while CPATMINER can. We found that BASE cannot detect 51% of the patterns mined from CPATMINER because the change pattern instances fall into the cases explained in Observations 2-4: *the changed code elements need program dependencies between them to form meaningful change patterns*. For example, in Fig. 6, BASE cannot relate the addition of closeQuietly(statement) on line 11 with the statement statement.executeQuery() on line 7 to form a meaningful pattern of closing resources used with executeQuery(). That is because it does not encode data dependencies and missed an important part of a change pattern. We also examined the 65% of the cases that BASE incorrectly detected as patterns. We found that 59% of the cases, BASE groups the atomic syntactic changes that have no relation or dependencies, *e.g.,* the trivial frequent changes such as adding a name or a null literal.

TABLE III: Comparing effectiveness of CPATMINER and BASE

	CPATMINER	BASE	Draw	Undecided
#Cases	50	3	30	17
Percentage	50%	3%	30%	17%

E. Commonality of Patterns (RQ4)

Fig. 9a shows the accumulated percentage of the change patterns shared among the developers. As seen, 43% of the developers in our corpus have 100% of their change patterns



(a) Over developers. (b) Over projects.

Fig. 9: Accumulated % of Shared Patterns (x-axis) over Developers and Projects (y-axis)

shared with other developers in the same/different teams, while 75% of them have at least 50% of their patterns shared by others. 88% of them shared at least 10% common patterns.

Figure 9b shows the accumulated percentage of the change patterns shared among the projects/teams. As seen, 31% of the projects in our corpus have 100% of their patterns shared with other projects. 55% of them have at least 50% of their patterns shared by others. 85% of the projects have at least 10% common patterns. This result shows that *fine-grained change patterns are pervasive among developers and teams*.

F. Temporal Distribution of Patterns (RQ5)

The previous RQ4 determined that developers repeat their changes. Does a developer accumulate a large number of similar changes within a short period of time, and then never makes similar changes, or does she remember and perform similar changes over a long period of time? If the change patterns are memorable and repeated long after the developer performed previously, it makes sense to create a repository of change patterns from which the developer can *reuse previous knowledge* about how and why to change code.

We first have to isolate the instances of patterns that come from each developer. For each developer we first take only her patterns. For each pattern, we create a timeline of its instances (based on the commit-time from GitHub) and then we compute the time intervals between any two consecutive instances. Since we study temporal distribution of patterns, we discard pairs of instances occurring in the same commit. We repeat this process for all the patterns for all developers, and then we put all intervals into bins as shown in Fig. 10. For example, the first bin (same day) shows how many pairs of consecutive instances appear in different commits but within 24-hour interval.

Fig. 10 shows that 46% of the pattern instances are repeated by the same developer at least a month apart, and 17% of them are repeated at intervals longer than a year. Thus, change patterns are widely distributed in time. This also holds for the change patterns for the same team/project (not shown).

G. Running Time

Experiments were conducted on a desktop with Windows 10, Intel(R) Core i5 Quad-Core 3.4 GHz CPU and 32 GB of RAM. Change graph extractions were run with 4 threads while mining used single thread. The maximum of 16 GB memory was used. The running time for the survey and depth corpora was less than 8 and 4 hours, respectively (Table IV).

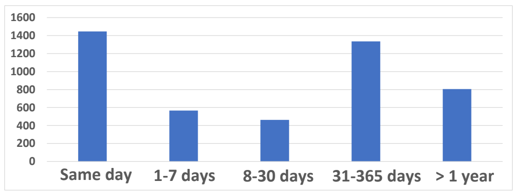


Fig. 10: Histogram of Change Patterns over Time

TABLE IV: Running Time and Statistics of Change Patterns

Corpus	Change Graphs				Pattern Size				Frequency			Time		
	All	Min	Max	Avg	All	Min	Max	Avg	Min	Max	Avg	Extr	Mine	Total
Survey	824K	3	100	24	8K	3	107	4	3	262	5	356	90	446
Depth	322K	3	100	32	17K	3	105	4	3	371	4	98	126	224

This is reasonable for running the tool offline every week to build the database. The time for extracting change graphs from the survey corpus was longer than that from the depth corpus, which is proportional to their numbers of commits and extracted graphs. However, the mining time for the latter was longer since there are more patterns in the depth corpus.

H. Threats to Validity

The two corpora in our studies might not be representative. The survey and comparative study involve human evaluation, which might have errors. We mitigated that by asking the real authors on their recent changes. In our comparative study, two human subjects must discuss to reach consensus when disagreements occurred. Our mining tool is currently for Java.

VI. Implications

Using CPatMiner, we collected and made available a dataset of 25K fine-grained change patterns. The database (Table IV) contains 8K patterns from a broad range of projects in our survey corpus and 17K patterns from large projects with large teams and long histories in our depth corpus. We now present actionable implications of our database and research tool/results from the perspective of four audiences: researchers, language and API designers, tool builders, and developers.

Researchers. Several researchers have exploited the repetitive nature of code changes [3], [4], [5], [6], [16], [19], [26], [27], [31], [32], [36], [39], [44], [46], [47]. They have shown that their tools work well in smaller datasets of task-specific changes (*e.g.,* bug fixes, porting, etc). In this paper, we confirm the widespread nature of repetitive code changes (see results of RQ3) for many more kinds of changes than previously studied, on a *much larger scale* (across 5,000+ projects and 170,000+ developers), at a *higher semantic level*, and across a wide span of time intervals (see results of RQ5). Thus, our research shows that these previous tools are *empirically justified.*

In addition to validating previously-held beliefs of existing tools/techniques, CPatMiner and our change pattern database can also enhance them. First, if one replaces an existing tool's database of repeated changes with ours (which is significantly larger), this could increase the tool's effectiveness. For example, Nguyen *et al.* [26]'s APIRec mines the fine-grained changes from 50 projects in GitHub to build its change database to recommend API usages. With our much larger database of fine-grained changes, especially at a higher semantic level, APIRec is expected to reduce the number of unseen changes, thus recommend more API usages (*i.e.,* increasing its recall).

Second, the task-specific recommendation tools, *e.g.,* for bug-fixing patterns, could benefit from our tool/dataset. Running our tool on bug-fixing changes, a researcher could mine complementary bug-fixing patterns to be used in their tool because 1) incorporating the dynamic dimension of changes (as opposed to previously only considering static code snapshots), 2) many more patterns mined in the wild, and 3) handling non-contiguous, multi-line change bug fixes. Moreover, our tool can improve the effectiveness of systems for program transformations by example [41], [46]. In such systems, the user first needs to write examples of the transformation, which can be time consuming. Using our tool, a user could easily find several examples from our database, feed these examples to a program synthesizer to synthesize a more complete transformation.

Researchers can also use our tool/dataset to build *novel* applications. One area that we envision is tutoring systems for teaching a novice programmer how to learn programming. For example, using our tool to mine changes from programming sessions that follow Test Driven-Development, a tutor system could recommend novice programmers how to change their programs from hard-coded values to generic, parameterized statements. Moreover, a tutor system that uses our tool to mine patterns from expertly-developed programs, can teach a novice programmer how to use advanced language features (*e.g.,* from Effective Java [18]), or APIs (*e.g.,* advanced concurrency [14]).

Language and API Designers can use our tool/dataset to learn how programmers commonly change their code to use the newly introduced features, thus harnessing the feedback from crowdsourcing. For example, the C# language designers have used the change patterns data from a work [34] to improve the default behavior of asynchronous constructs. Moreover, designers can observe how programmers refine the usage of an existing API, and get inspiration for new features that are driven by empirically-justified needs. For example, the previous study [20] on change patterns when using concurrent collections inspired adding new API methods in Java's ConcurrentHashMap.

Tool Builders can find new inspiration from commonly occurring change patterns, and automate them in IDEs. For example, IDE-integrated refactoring tools [21], [34] were inspired from change patterns. The patterns with succinct names are invaluable for researchers in Design Patterns [12] and tool builders to create new refactorings to design patterns.

Developers can educate themselves using our dataset on how to use advanced features of APIs or languages. Developers can use our tool to harness best practices or common mistakes across a wide variety of projects in our dataset. Moreover, they can study such practices within their own organization (by running our tool to create their own dataset), or on a corpus for different domains (*e.g.,* platform, problem, program domains).

VII. Related Work

The closest related work is by Negara *et al.* [25]. The authors aim to detect *high-level, in-the-wild frequent code change patterns* from a *fine-grained* sequence of code changes recorded from IDEs. There are key advances in our work. First, as CPatMiner does not rely on recorded editing changes from within IDEs, the dataset of fine-grained changes is significantly larger as we mine from an ultra-large number of changes in open-source projects. The frequent itemset mining algorithm performs worse than CPatMiner (Section V-D) in the scenario of mining from repositories where no order among the fine-grained changes in the same commit is recorded.

Other approaches learn systematic editing to support porting [22], [37], [38] or automatic similar changes [23]. They create context-aware edit scripts, identify the locations, and then transform the code. In comparison, their representation of changes is syntax-based in the same manner as in the AST differencing techniques [9], [7]. As shown in Section V-D, our tool outperformed the syntax-based change pattern mining technique as used in [25], [22], [23], [9], [7].

Nguyen *et al.* [28] performed an empirical study on repetitiveness of changes. Barr *et al.* [2] reported that it is possible to compose a new change from previous commits. Both of their goals are not to find semantic change patterns.

Other researchers detect the *patterns of changes* for program auto-repairing [19], [15] or software evolution [45], [48], [40]. Some approaches mine the bug fixing patterns and suggest fixes for other places [19], [32], [35]. Some other approaches use genetic programming [15] or search-based software engineering [17]. In comparison, they focus on bug-fixing change patterns for a domain-specific task, while CPatMiner is general for any fine-grained change patterns.

VIII. Conclusion

We introduce CPatMiner, a novel graph-based approach to detect repetitive changes in the wild, by mining fine-grained semantic code change patterns from a large number of open-source repositories. With the graph-based representation for fine-grained changes, it is able to overcome the limitations of existing mining approaches at the syntactic level, and is able to detect the patterns that were even recognized as meaningful by open-source authors/developers. The dependencies connecting (un)changed program elements help form semantic changes, facilitating CPatMiner detect meaningful patterns.

Our findings on the commonality of change patterns suggest that change patterns are pervasive in both time and space. Thus, we call for a community-based pattern database and present actionable implications of our findings for researchers, language/API designers, tool builders, and developers.

Acknowledgment

This work was supported in part by the following US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263 for Tien N. Nguyen, and CCF-1553741 and a Google Faculty Research Award for Danny Dig.

References

[1] https://docs.google.com/spreadsheets/d/1lK_UkwJP-W_8jcDde1L1DDrigFTs3oG6rwryNOApy60/edit#gid=0.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317. ACM, 2014.

[3] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA'15, pages 257–269. ACM, 2015.

[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222. ACM, 2009.

[5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 481–490. ACM, 2008.

[6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 404–428. Springer-Verlag, 2006.

[7] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324. ACM, 2014.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[9] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[10] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering, ICSE'12*, pages 222–232, 2012.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] X. Ge, Q. L. DuBose, and E. R. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering, ICSE'12*, pages 211–221, 2012.

[14] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[15] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[16] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, Dec. 2006.

[17] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 295–306. IEEE Computer Society, 2015.

[18] J. Kerievsky. *Effective Java (2nd Edition)*. Addison-Wesley, 2008.

[19] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811. IEEE Press, 2013.

[20] Y. Lin and D. Dig. Check-then-act misuse of Java concurrent collections. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 164–173. IEEE Computer Society, 2013.

[21] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15*, pages 224–235, 2015.

[22] N. Meng, M. Kim, and K. S. McKinley. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 440–443. ACM, 2011.

[23] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511. IEEE Press, 2013.

[24] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, Jan 2015.

[25] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 803–813. ACM, 2014.

[26] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522. ACM, 2016.

[27] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 69–79. IEEE Press, 2012.

[28] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 180–190. IEEE Press, 2013.

[29] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Trans. Softw. Eng.*, 38(5):1008–1026, Sept. 2012.

[30] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 440–455. Springer-Verlag, 2009.

[31] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'10, pages 302–321. ACM, 2010.

[32] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324. ACM, 2010.

[33] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, 2009.

[34] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 1117–1127. ACM, 2014.

[35] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 343–347, Feb 2014.

[36] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439. ACM, 2016.

[37] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11. ACM, 2012.

[38] B. Ray, C. Wiley, and M. Kim. REPERTOIRE: a cross-system porting analysis tool for forked software projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 8:1–8:4. ACM, 2012.

[39] R. Robbes and M. Lanza. How program history can improve code completion. In *ASE '08*, pages 317–326. IEEE CS, 2008.

[40] T. Rolfsnes, L. Moonen, S. D. Alesio, R. Behjati, and D. W. Binkley. Improving change recommendation using aggregated association rules. In *MSR '16: Proceedings of the 2016 international conference on Mining software repositories*. ACM, 2016.
[41] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415. IEEE Press, 2017.
[42] http://wiki.c2.com/?RuleOfThree.
[43] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870. ACM, 2016.
[44] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal API usage patterns. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 456–459. IEEE Computer Society, 2011.
[45] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, Sept. 2004.
[46] T. Zhang and M. Kim. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 665–676. IEEE Press, 2017.
[47] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE'10, pages 195–204. ACM, 2010.
[48] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572. IEEE Computer Society, 2004.