Mining Continuous Code Changes to Detect Frequent Program Transformations

Stas Negara, Mihai Codoban, Danny Dig, Ralph E. Johnson
University of Illinois at Urbana-Champaign
{snegara2, codo, dig, rjohnson}@illinois.edu

ABSTRACT

Identifying repetitive code changes benefits developers, tool builders, and researchers. Tool builders can automate the popular code changes, thus improving the productivity of developers. Researchers would better understand the practice of code evolution, advancing existing code assistance tools even further. Developers would particularly benefit if such tools can learn and support repetitive code changes that are in progress. Unfortunately, the existing tools that aim at detecting frequent code change patterns predominantly focus on analyzing the static source code of an application rather than dynamic code changes, and thus, they can not learn from the changes *on-the-fly*.

We present the first approach that identifies previously *unknown* frequent code change patterns from a *continuous* sequence of code changes. Our novel algorithm effectively handles two major challenges that distinguish continuous code change pattern mining from the existing data mining techniques: *overlapping* transactions and transactions containing multiple instances of the same item kind. We evaluated our algorithm on 1,520 hours of code development collected from 23 developers, and showed that it is effective, useful, and scales to big amounts of data. We analyzed some of the mined code change patterns and discovered ten popular kinds of high level program transformations.

1. INTRODUCTION

It is widely known that at least two-thirds of software costs are due to evolution [2, 3, 7, 12, 16, 23, 42], with some industrial surveys [11] claiming 90%. Change is the heart of software development. For example, software evolves to add features, fix bugs, support new hardware (e.g., multicores), new versions of operating systems and libraries, and new user interfaces for new platforms (e.g., web or mobile devices). However, many code changes are repetitive by nature, thus forming code change *patterns*.

Frequent pattern mining [17] was successfully applied in a broad range of domains. For example, Amazon.com recommends related products based on "customers who bought this also bought that". Netflix recommends new movies based on "customers who watched this also watched that" movie. Similar frequent pattern mining has revolutionized other services such as ITunes, GoodReads, social platforms, etc. More recently, data mining techniques became popular in the domain of genetics [30, 32, 34]. In particular, these techniques are employed to identify similar sequences of genes, which is a common task in DNA studies. We conjecture that mining frequent code changes can be similarly transformative for software development.

Identifying frequent code change patterns benefits Integrated Development Environment (IDE) designers, code evolution researchers, and developers. IDE designers can build tools that automate execution of frequent code changes, thus improving the productivity of developers. Researchers would better understand the practice of code evolution and also would be able to focus their attention on the most popular development scenarios. Library developers can notice and fix the common mistakes in the library API usage.

Application developers would particularly benefit from the ability of an IDE to identify repetitive code changes *on-the-fly*. An IDE that learns code change patterns as soon as the developer types them can (i) perform the corresponding change automatically the next time a developer needs it or (ii) detect inconsistencies in the code changes if the developer continues to perform them manually. Developers would also benefit if an IDE can learn frequent library usage patterns and offer intelligent code-completion [4,27,28] based on most common scenarios. Finally, such an IDE can learn code changes from more experienced programmers and suggest them to novices, thus creating a *virtual* pair-programming environment that does not incur the limitations of forcing both programmers to be collocated in space and time.

The existing research [4–6, 19, 22, 25, 33, 35, 38, 39, 43] predominantly detects frequent code change patterns either analyzing the static source code of an application or comparing the application's Version Control System (VCS) snapshots. In our previous study [26], we showed that data stored in VCS is *imprecise*, *incomplete*, and makes it *impossible* to perform analysis that involves the time dimension inside a single VCS snapshot. However, the most important limitation both of the snapshot-based techniques and the static source code analysis is that such approaches can not learn code changes *on-the-fly*, i.e., when the changes are in progress. Recent research [13, 15] aimed at providing on-the-fly code change assistance to developers, but their code change identification techniques were limited in two ways: (i) they were looking for a single kind of code change patterns — refactorings, (ii) they considered only a small subset of *a-priori* known kinds of refactorings.

In this paper, we propose to apply data mining techniques to detect previously *unknown* frequent code change patterns from a *continuous* sequence of code changes. Since our approach works on a continuous sequence of code changes, it can be applied either on-the-fly, or on a previously recorded sequence.

There are unique challenges posed by our problem domain of program transformations, which render previous off-the-shelf data mining techniques [17] inapplicable. First, for program transformations, we need to mine a *contiguous sequence* of code changes that are ordered by their timestamps, without any a-priori knowledge of where the boundaries between patterns of transformations are. In contrast, standard data mining techniques operate on a database of transactions with well known boundaries (e.g., the user is checking out all items in the shopping cart). Thus, we need to divide the contiguous sequence of program changes into individual transactions. One way to perform such division is to make the transactions disjoint and size them according to the maximum length of a pattern, *max_length*. Unfortunately, this approach does not account for patterns that cross the boundary of two transactions. To address this concern, we employ *overlapping* transactions whose size is $2 * max\_length$. The size of the overlap between two neighboring transactions is *max_length*. This approach ensures that our mining algorithm finds all patterns whose length does not exceed *max_length* as well as some patterns whose length is in between *max_length* and $2 * max\_length$.

Second, unlike the standard frequent itemset mining, when mining frequent code change patterns, a high level program transformation corresponding to a given pattern may contain several instances of the same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references of the renamed variable. Consequently, in our mining problem, a transaction may contain multiple instances of the same item kind, thus forming itembags rather than itemsets.

In this paper, we present our novel frequent code change patterns mining algorithm. Our algorithm employs the vertical data format [40] to directly access the transaction identifiers while computing new itemsets, which is crucial for effective handling of overlapping transactions and itembags. Our approach is inspired by several ideas from CHARM [41], the state-of-the-art algorithm for frequent closed itemsets mining that uses the vertical data format. In particular, our algorithm extends the notion of itemset-tidset tree (IT-tree) and adapts several optimization insights of CHARM.

We applied our novel algorithm on a large corpus of real world data that we collected during our previous user study [26], in which we accumulated 1,520 of code development from 23 developers working in their natural settings. Our evaluation shows that our algorithm is effective, useful, and scales well for big amounts of data. In particular, our algorithm mined more than a million of item instances in less than six hours. We analyzed some of the frequent code change patterns detected by our algorithm and identified ten kinds of popular high level program transformations. On average, 32% of the pattern occurrences reported by the algorithm led to high level program transformation discoveries.

This paper makes the following contributions:

- **Algorithm:** We designed a novel algorithm that effectively addresses the challenges of frequent code change patterns mining.
- **Tool:** We implemented our algorithm as part of CodingTracker. CodingTracker is open source and is available at http://codingtracker.web.engr.illinois.edu.
- **Evaluation:** We evaluated our algorithm on a large corpus of real world data and showed that it is effective, useful, and scalable.
- We analyzed some of the mined code change patterns and identified ten kinds of popular high level program transformations.

Table 1: An example of a transaction database.

Transaction identifier (*tid*)	Set of items
1	a, c
2	b, d
3	a, b, d
4	c
5	b, c

2. BACKGROUND

In this section, we introduce the canonical problem of mining frequent closed itemsets and present the three basic methodologies of mining such itemsets. In the following section, we discuss how our problem of mining frequent code change patterns differs from the canonical one and describe our algorithm, which is inspired by one of the basic techniques for mining frequent closed itemsets.

A *transaction* is a tuple $< tid, X >$, where *tid* is a unique transaction identifier and X is a set of items. A *transaction database* D is a set of transactions. Let I be a set of all items and T be a set of all *tid*s that are present in a database D. A non-empty set of items X that is a subset of I is called an *itemset*. Correspondingly, a non-empty set of transaction identifiers Y that is a subset of T is called a *tidset*. An itemset that contains n items is called an n-itemset.

A transaction $< tid, X >$ contains itemset Y if Y is a subset of X. Let $t(X)$ denote the set of *tid*s of all transactions that contain itemset X. In a given transaction database D, the *support* of an itemset X, which we denote as $sup(X)$, is the number of transactions in D that contain X. That is, $sup(X) = |t(X)|$. An itemset X is a *frequent* itemset if $sup(X) \geq min\_sup$, where *min_sup* is a user-specified minimum support threshold. An itemset X is a *closed* itemset if there exists no proper superset Y such that $sup(X) = sup(Y)$. In contrast to mining frequent itemsets, mining frequent closed itemsets produces a significantly more compact result while preserving its completeness [29].

Table 1 shows an example of a transaction database that contains four items, $I = \{a, b, c, d\}$, and five transactions, $T = \{1, 2, 3, 4, 5\}$. For the convenience of presentation, when the order of items does not matter, we order them alphabetically. Table 2 shows several itemsets from the transaction database in Table 1. The second and the third columns of Table 2 present the set of the containing transaction identifiers and the support for each itemset.

One of the fundamental distinctions between different approaches to mining frequent itemsets is whether mining is performed with or without candidate generation. Apriori [1] is an exemplar algorithm for mining with candidate generation. Apriori first scans the transaction database to detect the frequent 1-itemsets. These 1-itemsets are used to generate 2-itemsets, and the algorithm scans the database again to see which of these 2-itemsets are frequent. This iterative process continues until it reaches iteration n, for which no

Table 2: Example itemsets from the transaction database in Table 1.

Itemset X	$t(X)$	$sup(X)$
$\{a\}$	$\{1, 3\}$	2
$\{a, b\}$	$\{3\}$	1
$\{b, d\}$	$\{2, 3\}$	2
$\{a, b, d\}$	$\{3\}$	1

Table 3: The transaction database from Table 1 represented in the vertical data format.

Item	Tidset
a	$\{1, 3\}$
b	$\{2, 3, 5\}$
c	$\{1, 4, 5\}$
d	$\{2, 3\}$

more frequent n-itemsets can be found.

Mining with candidate generation has two major drawbacks: a) it generates redundant itemsets that are found to be infrequent; b) it repeatedly scans the transaction database while progressing through the iterations. Mining without candidate generation addresses both these limitations. Such mining can be broadly divided into mining using *horizontal data format* and mining using *vertical data format*. The horizontal data format represents a transaction database as a set of tuples $< TransactionID, itemset >$. Table 1 illustrates a database in this format. FP-growth method [18] is an exemplar of mining data in the horizontal data format without candidate generation. This approach is based on constructing a *frequent-pattern tree* (FP-tree) out of frequent 1-itemsets and then growing the FP-tree recursively.

The vertical data format represents a transaction database as a set of tuples $< item, tidset >$, where *tidset* is a set of identifiers of transactions that contain the corresponding *item*. Table 3 shows the transaction database from Table 1 in the vertical data format. Eclat [40] is the first algorithm for mining data in the vertical data format without candidate generation. The basic idea of the algorithm is to compute $(n + 1)$-itemsets from n-itemsets by intersecting their tidsets. The algorithm starts with frequent 1-itemsets and finishes when no more frequent itemsets can be found. For example, let's consider two 1-itemsets, $\{a\}$ and $\{b\}$, from Table 3 (note that for any item x there is a corresponding 1-itemset $\{x\}$). The algorithm computes the tidset of a 2-itemset $\{a, b\}$ by intersecting the tidsets of $\{a\}$ and $\{b\}$: $t(\{a, b\}) = t(\{a\}) \cap t(\{b\}) = \{3\}$. The support of the itemset $\{a, b\}$ is 1: $sup(\{a, b\}) = |t(\{a, b\})| = 1$. If the minimum support threshold is greater than 1, then itemset $\{a, b\}$ is discarded. Otherwise, it is added to the results and consequently, it is considered for computing 3-itemsets.

The vertical data format enables a mining algorithm to directly access the transaction identifiers while computing new itemsets. This feature is crucial for accommodating such properties of code change patterns mining as overlapping transactions and itembags. Therefore, our approach is inspired by several ideas from CHARM [41], an advanced algorithm for mining data in the vertical data format, which introduces the notion of itemset-tidset tree (IT-tree), employs several optimizations, and searches for closed itemsets, thus considerably reducing the size of the mining result. In particular, our algorithm extends the notion of IT-tree and adapts several optimization insights of CHARM. In this section, we present the CHARM's definition of IT-tree. We discuss the specifics of our approach in the next section.

The nodes in an IT-tree are pairs *itemset* : *tidset*. The root of the tree represents an empty itemset, and thus, its tidset is T, the set of all *tids*. The immediate children of the root node are 1-itemsets that are computed by scanning the transaction database. The immediate children of a non-root node are computed by intersecting this node's tidset with the tidsets of the *not yet considered* 1-itemsets, traversing them from left to right. If the resulting tidset's size falls below the minimum frequency threshold, the new node is not added to the IT-tree. The IT-tree is completed when no more nodes can be added to it. Figure 1 shows the IT-tree for the transaction database from Table 3. Three itemsets in this IT-tree are not closed, and thus, are not included into the mining result: itemsets $\{a, b\}$ and $\{a, d\}$ are subsumed by the proper super itemset $\{a, b, d\}$, while itemset $\{d\}$ is subsumed by the proper super itemset $\{b, d\}$.

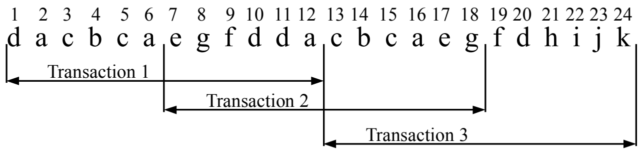

Figure 2: An example of a sequence of code changes. Distinct kinds of code changes are represented with different characters.

3. OUR MINING ALGORITHM

Handling overlapping transactions and itembags. We mine frequent code change patterns from a sequence of code changes that are ordered by their timestamps. To populate our transaction database, we divide this continuous sequence into individual transactions. Making transactions disjoint and sizing them according to the maximum length of a pattern, *max_length*, does not account for patterns that cross the boundary of two transactions. Therefore, we use *overlapping* transactions whose size is $2 * max\_length$. The size of the overlap between two neighboring transactions is *max_length*. As a result, our mining algorithm finds all patterns whose length does not exceed *max_length* and some patterns whose length lies in between *max_length* and $2 * max\_length$.

Figure 2 shows an example of a sequence of code changes. For the presentation purposes, we denote every distinct kind of code change with a different character and consider that $max\_length = 6$. For the actual mining, we set *max_length* to five minutes, and thus, transactions contained various number of items.

An important observation is that although code changes form an ordered sequence, a code change pattern is unordered because the corresponding high level code change may be performed in different orders. For example, a developer who performs a Rename Local Variable refactoring might first change the variable's declaration and then its ref-

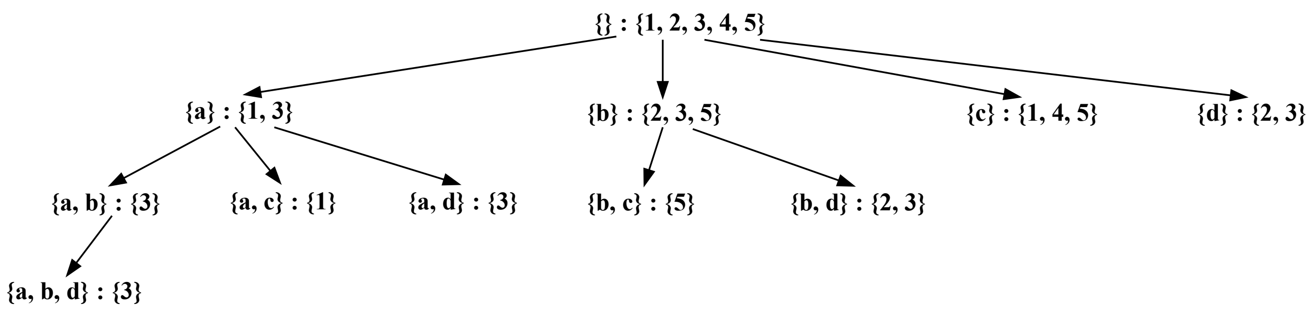

Figure 1: The IT-tree for the transaction database from Table 3.

erences or vice versa, or even intersperse changing the declaration and the references. Thus, the order of a transaction's items does not matter. Another observation is that a high level code change may contain several instances of the same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references of the renamed variable. Consequently, a transaction's items form a bag rather than a set. For example, the first transaction in Figure 2 contains three items a and two items c. Moreover, this transaction contains two instances of code change pattern $\{a, c\}$, which illustrates a scenario when two high level code changes happen within max_length time frame.

The major difference between our frequent code change patterns mining algorithm and the existing approaches to mining frequent itemsets is that our algorithm handles overlapping transactions and *itembags* rather than itemsets. To distinguish different *occurrences* of an item in the same transaction as well as the overlapped parts of two transactions, our algorithm assigns each item's occurrence a unique ID. The first line in Figure 2 shows the IDs assigned to the underlying items' occurrences. For example, the first transaction contains occurrences of item a with IDs $\{2, 6, 12\}$, the second transaction — $\{12, 16\}$, and the third — $\{16\}$. Note that although our algorithm handles itembags, we continue to use the notion of itemsets throughout our presentation, since the fact that our itemsets are actually itembags is accounted for by explicitly tracking each item's occurrences.

Tracking an item's occurrences. In order to track items' occurrences, a node in our variation of IT-tree is defined as follows:

$$
\begin{aligned}
&[item_1, item_2, ..., item_n] :\\
&\quad [tid_1 : [[occurrences_1], [occurrences_2], ..., [occurrences_n]],\\
&\quad\ tid_2 : [[occurrences_1], [occurrences_2], ..., [occurrences_n]],\\
&\quad\ ...,\\
&\quad\ tid_m : [[occurrences_1], [occurrences_2], ..., [occurrences_n]]]
\end{aligned}
$$

We use square brackets to denote ordered sets. The order of items in an itemset does not matter for a pattern, but it helps our algorithm to track occurrences of every item in each transaction that contains this itemset. Thus, we represent an n-itemset as an ordered set of items $[item_1, item_2, ..., item_n]$. For a given itemset, a node in an IT-tree contains an ordered set of *tid*s of transactions that contain this itemset. Ordering of transactions enables our algorithm to effectively handle overlapping parts of the neighboring transactions. For each transaction, the IT-tree node also tracks all occurrences for every item in the given itemset (in the above representation, $[occurrences_i]$ are all occurrences of $item_i$ in a particular transaction). Our algorithm also orders an item's occurrences to ensure the optimal result of our itemset frequency computation technique that we discuss below.

Similarly to CHARM [41], we compute our IT-tree by traversing the 1-itemsets from left to right and intersecting the tidset of a particular itemset with the tidsets of the not yet considered 1-itemsets to generate new IT-tree nodes. The major difference from the CHARM's approach is that our algorithm tracks items' occurrences, and thus, whenever a new item is added to an itemset, the item's occurrences are appended to the set of occurrences of every transaction in the corresponding IT-tree node. Table 4 shows several examples of itemsets and their corresponding IT-tree nodes for the sequence of code changes from Figure 2. Note that storing an item's occurrences in every IT-tree node that contains this item is not only redundant, but also prohibitively expensive. Instead, our algorithm stores occurrences of individual items and then just refers these occurrences from the containing IT-tree nodes. We inline the referred occurrences for the presentation purposes only.

Computing the frequency of an itemset. Due to overlapping transactions and multiple occurrences of an item in the same transaction, our algorithm can not compute the frequency of an itemset as the number of transactions that contain this itemset (as it is done in the existing frequent itemset mining techniques). Instead, we devised our own itemset frequency computation technique that accounts for the particularities of our mining problem. In a given transaction k:

$$tid_k : [[occurrences_1], [occurrences_2], ..., [occurrences_n]]$$

the frequency of the corresponding itemset is:

$$f_k = \min_{i=1..n} |[occurrences_i]| \tag{1}$$

That is, f_k is the number of occurrences of an itemset's item that appears the least number of times. The *overall* frequency of an itemset that is contained in m transactions is:

$$F = \sum_{k=1}^{m} f_k \tag{2}$$

If an item occurrence is shared between two neighboring transactions, k and l, the algorithm should count this oc-

Table 4: Examples of itemsets and their corresponding IT-tree nodes for the sequence of code changes from Figure 2.

Itemset	**IT-tree node**
$\{a\}$	$[a] : [1 : [[2, 6, 12]], 2 : [[12, 16]], 3 : [[16]]]$
$\{c\}$	$[c] : [1 : [[3, 5]], 2 : [[13, 15]], 3 : [[13, 15]]]$
$\{d\}$	$[d] : [1 : [[1, 10, 11]], 2 : [[10, 11]], 3 : [[20]]]$
$\{a, c\}$	$[a, c] : [1 : [[2, 6, 12], [3, 5]], 2 : [[12, 16], [13, 15]], 3 : [[16], [13, 15]]]$
$\{a, c, d\}$	$[a, c, d] : [1 : [[2, 6, 12], [3, 5], [1, 10, 11]], 2 : [[12, 16], [13, 15], [10, 11]], 3 : [[16], [13, 15], [20]]]$

currence only once, either as part of f_k or as part of f_l. In an ordered set of transactions, two transactions, k and l, are neighboring if and only if $|k - l| = 1$ and $|tid_k - tid_l| = 1$. That is, the neighboring transactions follow each other both in the ordered set and in the original code changes sequence. For example, in Figure 2 transactions of the ordered set $[1, 2]$ are neighboring, while transactions of the ordered set $[1, 3]$ are not neighboring.

Let's denote $[occurrences_i^k]$ the ordered set of occurrences of $item_i$ in a transaction k. Let's denote o_j an occurrence o with the index j in the ordered set of occurrences. To compute the frequency of an n-itemset that is contained in m transactions, our algorithm visits each pair of transactions k and $k+1$, where $1 \leq k < m$. First, our algorithm computes f_k using formula in (1). Then, if transactions k and $k + 1$ are neighboring, our algorithm visits every occurrence $o_j \in [occurrences_i^k]$, where $1 \leq i \leq n$. If $o \in [occurrences_i^{k+1}]$, then our algorithm checks whether the shared occurrence o should be removed from the transaction k or $k+1$. If $j \leq f_k$, then o is removed from the transaction $k + 1$. Otherwise, it is removed from the transaction k. Note that removing shared occurrences never affects the initially computed f_k. Finally, our algorithm computes the overall frequency using formula in (2).

For the best performance of our algorithm, we order an item's occurrences such that those that happened earlier in time appear earlier in the ordered set. Since occurrences' IDs are generated incrementally (see Figure 2), such ordering is easily achieved by sorting occurrences in ascending order of their IDs. Consequently, the occurrences that are shared between transactions k and $k+1$ are placed at the end of the ordered set of all occurrences for the transaction k. Hence, our algorithm computes the maximal possible frequency for the transaction k employing the shared occurrences only when needed, while the unused part of them is attributed to the subsequent transaction $k+1$, thus maximizing its frequency too. Going through each pair of transactions k and $k + 1$, $1 \leq k < m$, our algorithm propagates this maximization, thus computing the optimal overall frequency F.

Table 5 shows the frequency computation result for the itemset $\{a, c, d\}$. Occurrences removed by the algorithm are crossed out. The values in the fourth column, f'_k, are computed by applying the formula in (1) without removing the shared occurrences. The overall frequency of the itemset $\{a, c, d\}$ is $F = 4$.

Optimizing the algorithm. To optimize the mining algorithm, CHARM exploits several insights about the properties of itemsets and tidsets. Let's recall that the computation of immediate children of an IT-tree node of an itemset X involves traversing the not yet considered 1-itemsets from left to right. Let's assume that this traversal visits two 1-itemsets, Y and Z, in this order. If $t(X \cup Y) = t(X \cup Z)$, then there is no need to create separate children nodes corresponding to Y and Z. Instead, CHARM removes both Y and Z from the list of not yet considered 1-itemsets and adds a node $X \cup Y \cup Z$ as a child of the IT-tree node of the itemset X. Similarly, if $t(X \cup Y) \supset t(X \cup Z)$, then a child node $X \cup Y \cup Z$ is added to the IT-tree, but the 1-itemset Z is not removed from the yet not considered 1-itemsets, which means that CHARM would consider Z without Y while computing another immediate child, $X \cup Z$. To maximize the chances of encountering such scenarios, CHARM sorts the not yet considered 1-itemsets in ascending order of the number of transactions that contain them.

Table 5: The frequency computation result for the itemset $\{a, c, d\}$.

tid	**Occurences**	f_k	f'_k
1	[[2, 6, ~~12~~], [3, 5], [1, 10, ~~11~~]]	2	2
2	[[12, ~~16~~], [13, ~~15~~], [~~10~~, 11]]	1	2
3	[[16], [~~13~~, 15], [20]]	1	1

In our algorithm, we adapt the CHARM's optimization insights to the specifics of our data mining problem. We introduce the notion of a *frequency descriptor*, FD, a 3-tuple that captures all relevant information about the frequency of an itemset that is contained in m transactions:

$$FD =< F, \overset{\rightarrow}{f_k}, \overset{\rightarrow}{f'_k} >, \text{ where } k = 1..m.$$

For example, according to Table 5, the frequency descriptor of the itemset $\{a, c, d\}$ is $FD =< 4, (2, 1, 1), (2, 2, 1) >$. Two frequency descriptors are equivalent if they have exactly the same elements. A frequency descriptor FD_1 is *more powerful* than a frequency descriptor FD_2 if FD_1 and FD_2 are not equivalent and every element of FD_1 is equal to or greater than the corresponding element of FD_2. For example, $< 4, (2, 1, 1), (2, 2, 1) >$ is more powerful than $< 4, (2, 1, 1), (1, 2, 1) >$, but it is not more powerful than $< 4, (2, 1, 1), (1, 3, 1) >$.

To optimize the computation of the IT-tree, our algorithm first sorts the not yet considered 1-itemsets Y of a given itemset X by the number of transactions that contain $X \cup Y$. If two 1-itemsets, Y and Z, are such that $t(X \cup Y) = t(X \cup Z)$, then the algorithm arranges Y and Z in the lexicographical order of the frequency descriptors of $X \cup Y$ and $X \cup Z$.

Next, our algorithm computes the immediate children of the IT-tree node of the itemset X by traversing the sorted list of 1-itemsets. If two yet not considered 1-itemsets, Y and Z, are such that $t(X \cup Y) = t(X \cup Z)$ and the frequency descriptors of $X \cup Y$ and $X \cup Z$ are equivalent, our algorithm removes both Y and Z from the list of the not yet considered 1-itemsets and adds a node $X \cup Y \cup Z$ as a child of the IT-tree node of the itemset X. If Y and Z are such that $t(X \cup Y) = t(X \cup Z)$ or $t(X \cup Y) \supset t(X \cup Z)$, and the frequency descriptor of $X \cup Z$ is more powerful than the frequency descriptor of $X \cup Y$, our algorithm adds a child node $X \cup Y \cup Z$ to the IT-tree, but keeps the 1-itemset Z in the list of the not yet considered 1-itemsets.

Computing closed itemsets. The output of our algorithm contains only closed code change patterns, i.e., for each itemset X that represents a particular pattern, in our result, there is no itemset Y such that $Y \supset X$ and $F_Y = F_X$. To ensure the closeness of the result, our algorithm checks every newly generated itemset X against the already accumulated itemsets. Since the number of the accumulated itemsets might be very big, to speed up the checking process, we hash the accumulated itemsets similarly to CHARM, i.e., using the sum of the itemsets' *tids* as the key.

Unlike CHARM though, our algorithm differentiates between *partially subsumed* and *completely subsumed* itemsets as well as detects scenarios, in which the previously accumulated itemset needs to be replaced with the new one. Also, our algorithm compares the frequency descriptor of the newly generated itemset X, FD_X, with the frequency descriptors of the previously generated itemsets Y, FD_Y. If $Y \supset X$ and FD_Y is more powerful than FD_X, then the itemset X is completely subsumed — it is not added to the results and the algorithm stops computing the children of the corresponding IT-tree node. Otherwise, if $Y \supset X$ and the overall frequency of X, F_X, is equal to the overall frequency of Y, F_Y, then the itemset X is partially subsumed — it is not added to the results, but the algorithm proceeds computing the children of the corresponding IT-tree node. Finally, if $Y \subset X$ and $F_X = F_Y$, Y is removed from the results. Note that all the above decisions are valid only for X and Y that appear in the same transactions, i.e., $t(X) = t(Y)$.

Establishing frequency thresholds. To decide whether an itemset is frequent enough to be added to the results, our algorithm employs two thresholds. First, our algorithm checks whether an itemset's overall frequency is not less than the *absolute frequency threshold*, i.e., the frequency threshold that ignores all other characteristics of the itemset (e.g., its size or composition). Next, our algorithm checks whether the itemset's overall frequency multiplied by the itemset's size is not less than the *dynamic threshold*, i.e., the longer an itemset grows, the less frequent it can be in order to pass this threshold. For example, a dynamic threshold of 100 filters out 1-itemsets that are less frequent than 100, 2-itemsets that are less frequent than 50, and so on.

Putting it all together. Figure 3 shows a high level overview of our frequent code change patterns mining algorithm. The pseudocode of the figure contains all the basic functionality features of our algorithm discussed above.

4. EVALUATION

In our evaluation, we would like to answer these questions:

- **Q1**(scalability): Is our algorithm scalable enough to

```
input: codeChangesSequence
output: frequentPatterns

procedure: mine(codeChangesSequence) {
  frequentPatterns = ∅; emptySet = ∅;
  inputItems = getInputItems(codeChangesSequence);
  solve(emptySet, inputItems);
}

procedure: solve(currentItemset, remainingItems) {
  remainingSortedItems = sort(currentItemset, remainingItems);
  while (remainingSortedItems ≠ ∅) {
   nextItem = pollFirst(remainingSortedItems);
   newItemset = addItem(currentItemset, nextItem);
   if (getFrequency(newItemset) ≥ ABSOLUTE_FREQUENCY) {
    newRemainingItems = copy(remainingSortedItems);
    foreach (forwardItem ∈ remainingSortedItems) {
     freqCompResult = compare(freqDescr(newItemset),
                         freqDescr(addItem(newItemset, forwardItem));
     if (freqCompResult == EQUIVALENT ||
             freqCompResult == SECOND_MORE_POWERFUL) {
      newItemset = addItem(newItemset, forwardItem);
      removeItem(newRemainingItems, forwardItem);
      if (freqCompResult == EQUIVALENT) {
       removeItem(remainingSortedItems, forwardItem);
      }
     }
    }
    if (getFrequency(newItemset) * getSize(newItemset) ≥
                                        DYNAMIC_THRESHOLD) {
     subsumptionResult = checkSubsumption(newItemset);
     if (subsumptionResult ≠ FULLY_SUBSUMED) {
      if (subsumptionResult ≠ PARTIALLY_SUBSUMED) {
       frequentPatterns =
                       addItemset(frequentPatterns, newItemset);
      }
      solve(newItemSet, newRemainingItems);
     }
    }
   }
  }
}
```

Figure 3: Overview of our frequent code change patterns mining algorithm.

Number of participants	Programming Experience (years)
1	1 - 2
4	2 - 5
11	5 - 10
6	> 10

Table 6: Programming experience of the participants.

handle big amounts of data?

- **Q2**(effectiveness): Does our algorithm mine code change patterns that simplify identification of high level program transformations?
- **Q3**(usefulness): Can we detect interesting high level program transformations from the mined code change patterns?

To answer these questions, we applied our frequent code change patterns mining algorithm on a large corpus of real world data. In the following, we first describe how we collected the data and performed the evaluation of the algorithm. Then, we present our evaluation results.

4.1 Experimental Setup

We applied our algorithm on the data collected during our previous user study [26], which involved 23 participants: 10 professional programmers who worked on different projects in domains such as marketing, banking, business process management, and database management; and 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at Urbana-Champaign.

Table 6 shows the programming experience of our participants. Note that only 22 out of 23 participants filled the survey and specified their programming experience. In the course of our study, we collected code evolution data for 1,520 hours of code development with a mean distribution of 66 hours per developer and a standard deviation of 52.

The participants of our study installed the CodingTracker plug-in in their Eclipse IDEs. Throughout the study, CodingTracker recorded the detailed code evolution data ranging from individual code edits up to the high-level events like automated refactoring invocations. CodingTracker uploaded the collected data to our centralized repository using the existing infrastructure [36].

We first applied our AST node operations inference algorithm [26] on the collected raw data to represent code changes as *add*, *delete*, and *update* operations on the underlying AST. Next, we identified distinct kinds of code changes as combinations of the operation and the type of the affected AST node. For example, *add* `IfStatement`, *delete* `IfStatement`, and *add* `InfixExpression` are three different kinds of code changes. The instances of the identified code change kinds serve as input to our frequent code change patterns mining algorithm. That is, in our mining algorithm, a code change kind is an *item* and an instance of a code change kind is an item's *occurrence.*

For each mined code change pattern, our algorithm reports all occurrences of the pattern in the input sequence of

Table 7: Grouping of item kinds by their frequency. Column NK shows the number of item kinds in each group. Columns AFT and DT show the values of the absolute frequency threshold and dynamic threshold.

Frequency, F	NK	AFT	DT	Mining time
$10,000 \leq F$	23	30	10,000	15 minutes
$300 \leq F < 10,000$	81	30	300	5.2 hours
$5 \leq F < 300$	32	5	5	7.7 seconds

code changes. We use CodingTracker's replayer to manually investigate these occurrences. We replay the code changes of a particular occurrence to detect the corresponding high level program transformation. Since the mining result is huge, we order the mined patterns along three dimensions: by frequency of the pattern (F), by size of the pattern (S), and by $F * S$. Then, we output the top 1,000 patterns for each dimension and investigate them starting from the top of the list.

Recall that our algorithm uses two thresholds to decide whether a pattern is frequent: absolute frequency threshold and dynamic threshold. Both thresholds check whether a specific pattern's metrics (a pattern's frequency and a pattern's frequency multiplied by its size correspondingly) do not fall below a user-specified value. Providing different values for these thresholds, one can tune the output of our mining algorithm. We noticed that some items (i.e., AST node operations) are much more frequent than the others. Thus, picking a single pair of threshold values to analyze our data is impractical. If these values are too low, our algorithm's scalability would degrade, while the output would become disproportionately big. On the other hand, too high values would hinder the mining of patterns that involve less frequent items. Therefore, we divided the input items into three groups, applying different threshold values to each. Table 7 shows each group as well as the corresponding thresholds and the mining time. We performed all mining on a quad-core i7 2GHz machine with 8GB of RAM.

We observed that some AST node operations are too frequent to be considered at all. For example, adding and deleting `SimpleName` accompanies any code change that declares or references a program entity. Consequently, mining items that represent such AST node operations would only add noise to the detected code change patterns. Therefore, before applying our algorithm, we filtered out the noisy item kinds, thus reducing the total count of the considered item kinds from 162 to 138. According to Table 7, the total number of the considered item kinds is 136, which means that two item kinds were too infrequent to be part of any group.

4.2 Results

Table 8 summarizes performance statistics of our experiment. Our algorithm mined more than a million of item occurrences in less than six hours, and thus, **the answer to the first question is that our mining algorithm is sufficiently scalable to handle big amounts of data with the appropriate threshold values**.

The frequent patterns mined by our algorithm helped us identify ten kinds of program transformations. Table 9 shows

Table 8: Performance statistics of our experiment.

Item kinds	Transactions	Item occurrences	Total mining time
136	7,927	1,094,239	5.5 hours

Table 9: Identified kinds of program transformations. Column I shows the number of the investigated pattern occurrences. Column F shows the number of pattern occurrences that were fruitful.

Scope	Identified program transformation	I	F
Statement	Convert Element to List	5	2
Loop	Add a Loop Collector	3	1
	Wrap Loop with Timer	2	1
Method	Add Precondition Checks for a Parameter	5	1
Class	Add a New Enum Element	2	1
	Change and Propagate Field Type	3	1
	Change Field to ThreadLocal	2	1
	Copy Field Initializer	2	1
	Create and Initialize a New Field	4	1
	Move Interface Implementation to Inner Class	6	1

the identified kinds of program transformations grouped according to their scope. The last two columns of the table show the number of pattern occurrences that we investigated and the number of pattern occurrences that led to the discovery of the corresponding program transformations. Overall, 32% of pattern occurrences were fruitful. Hence, **our answer to the second question is that our algorithm is effective — it mines patterns that often lead to discovery of high level program transformations**.

In the following, we present the discovered transformation kinds in more details.

Convert Element to List. This is a statement-level transformation kind in which a developer converts a field, parameter, or a local variable of a certain type into a collection (e.g., list, set, array, etc.) of that type. Figure 4 shows an example of Convert Element to List transformation. In this and all subsequent examples of program transformations, we represent the changed parts of code as underlined text.

Add a Loop Collector. This is a transformation in which a developer introduces a new variable that collects or aggregates the data processed in a loop. Figure 5 shows an example of Add a Loop Collector transformation.

Wrap Loop with Timer. A developer applies this transformation to compute the execution time of a loop. The developer surrounds the loop with variables that hold the time before and after the loop execution and outputs the time difference. Figure 6 shows an example of Wrap Loop with Timer transformation.

Add Precondition Checks for a Parameter. This is a transformation in which a developer adds `null` precondition checks to all methods of a class that receive a parameter with the same semantics. Figure 7 shows an example of Add Precondition Checks for a Parameter transformation.

Add a New Enum Element. Adding a new element to `enum` triggers a ripple of changes such as adding new `switch` cases, `if-then-else` chains, and dealing with any duplicated code that uses the updated `enum`. Figure 8 shows an example of Add a New Enum Element transformation.

Change and Propagate Field Type. This is a transformation in which a developer changes the type of a field. As a result, the developer also has to update accordingly the type of some local variables as well as the return type of some methods. Figure 9 shows an example of Change and Propagate Field Type transformation.

Change Field to ThreadLocal. To improve thread safety of an application, a developer may decide to convert some fields to `ThreadLocal`. Besides changing the type and the initialization of the converted field, the developer also has to modify all field's accesses such that they use `get()` and `set()` of `ThreadLocal`. Figure 10 shows an example of Change Field to ThreadLocal transformation.

Copy Field Initializer. This is a transformation in which a developer copies the same initializer to several fields. Figure 11 shows an example of Copy Field Initializer transformation.

Create and Initialize a New Field. When a developer adds a new field, it has to be properly initialized alongside the already present fields in constructors and other initialization places (e.g., static initialization blocks). Figure 12 shows an example of Create and Initialize a New Field transformation.

Move Interface Implementation to Inner Class. This

Before

```
Lock myLock;
```

After

```
Set<Lock> myLock;
```

Figure 4: An example of Convert Element to List transformation.

Before

```
for (A a : collection) {
  ...
}
```

After

```
Set<A> collector = new HashSet<A>();
for (A a : collection) {
  ...
  collector.add(a);
}
```

Figure 5: An example of Add a Loop Collector transformation.

Before

```
for (...) {
  ...
}
```

After

```
long start = System.currentTimeMillis();
for (...) {
  ...
}
long end = System.currentTimeMillis();
output(end - start);
```

Figure 6: An example of Wrap Loop with Timer transformation.

Before

```
void m1(P1 p1, P2 p2) {
  ...
}
void m2(P1 p1, P3 p3) {
  ...
}
```

After

```
void m1(P1 p1, P2 p2) {
  if (p1 == null) return;
  ...
}
void m2(P1 p1, P3 p3) {
  if (p1 == null) return;
  ...
}
```

Figure 7: An example of Add Precondition Checks for a Parameter transformation.

Before

```
enum E {e1, ... , en};
switch (e) {
  case 1: ...
  ...
  case n: ...
}

if (m.isE1()) { ... }
...
if (m.isEn()) { ... }

P createP() {
  P p = new P();
  p.add(E.e1);
  ...
  p.add(E.en);
}
```

After

```
enum E {e1, ... , en, en+1};
switch (e) {
  case 1: ...
  ...
  case n: ...
  case n+1: ...
}

if (m.isE1()) { ... }
...
if (m.isEn()) { ... }
if (m.isEn+1()) { ... }

P createP() {
  P p = new P();
  p.add(E.e1);
  ...
  p.add(E.en);
  p.add(E.en+1);
}
```

Figure 8: An example of Add a New Enum Element transformation.

Before

```
A field;
public A method() {
  ...
  A localVar;
  ...
}
```

After

```
B field;
public B method() {
  ...
  B localVar;
  ...
}
```

Figure 9: An example of Change and Propagate Field Type transformation.

Before

```
class A {
  B f;
  void m() {
    int v = compute(f);
    ...
  }
}
```

After

```
class A {
  ThreadLocal<B> f;
  void m() {
    int v = compute(f.get());
    ...
  }
}
```

Figure 10: An example of Change Field to ThreadLocal transformation.

Before

```
class C {
  F field1;
  F field2;
  ...
}
```

After

```
class C {
  F field1 = new F() { ... };
  F field2 = new F() { ... };
  ...
}
```

Figure 11: An example of Copy Field Initializer transformation.

Before

```
class C {
  private A a;
  public C() {
    a = new A();
    ...
  }
  ...
}
```

After

```
class C {
  private A a;
  private B b;
  public C() {
    a = new A();
    b = new B();
    ...
  }
  ...
}
```

Figure 12: An example of Create and Initialize a New Field transformation.

Before

```
class C implements I1, I2{
  public void implementorI1() {
    ...
  }
  ...
}
```

After

```
class C implements I2{
  class InnerC implements I1{
    public void implementorI1() {
      ...
    }
  }
  ...
}
```

Figure 13: An example of Move Interface Implementation to Inner Class transformation.

is a transformation that describes a scenario in which a developer moves the implementation of an interface from a class to its newly created inner class. Figure 13 shows an example of Move Interface Implementation to Inner Class transformation.

The mined code change patterns helped us identify ten kinds of interesting program transformations whose scopes range from individual statements to whole classes. Thus, **our answer to the third question is that our algorithm is useful**.

5. THREATS TO VALIDITY

In our experiment, we used the output of the AST node operations inference algorithm [26] to prepare the input to our frequent code change patterns mining algorithm. Consequently, imprecisions in the inferred AST node operations could negatively affect our mining results. Note, however, that our approach to mining frequent code change patterns is independent of the way its input is produced.

We investigated the mined patterns manually, and thus, might have missed to identify some of their corresponding high level program transformations. Also, we investigated only a fraction of the mining results. However, our experiment did not aim at discovering all program transformations performed by our participants. Instead, our goal was to show that our algorithm mines patterns that effectively point to high level program transformations, and we believe that discovering several such transformations in a reasonable amount of time (identifying and documenting these transformations took the second author a couple of days) supports this claim.

In our study, we collected code evolution data from developers who use Eclipse for Java programming. Consequently, the identified high level program transformations might be specific to *similar* programming environments and languages. Nevertheless, our approach to identifying such transformations is orthogonal to the way developers make their code changes.

Our dataset is not publicly available due the nondisclosure agreement with our participants.

6. RELATED WORK

6.1 Mining Frequent Itemsets

The major challenge in mining frequent itemsets is to develop scalable algorithms that can effectively handle large transaction databases. Agrawal et al. [1] observed that an n-itemset is frequent only if all its subsets are also frequent. Their mining algorithm, Apriori, leverages this property by using frequent n-itemsets to generate $(n+1)$-itemset candidates. Apriori checks the newly generated candidates against the transaction database to establish those of them that are frequent. The algorithm starts with detecting frequent 1-itemsets directly from the transaction database and proceeds iteratively until no more frequent itemsets can be found.

A candidate generation approach repeatedly scans the mined transaction database. Also, at each iteration, it produces candidate itemsets, a significant fraction of which is infrequent. To address these limitations, Han et al. [18] suggested to mine frequent itemsets without candidate generation. Their approach operates on transaction databases in the horizontal data format. First, they scan the database to detect frequent 1-itemsets. Their algorithm uses these 1-itemsets to construct a frequent-pattern tree (FP-tee), an extended prefix-tree structure. Then, the algorithm expands the initial FP-tree by growing pattern fragments in a recursive fashion.

Zaki [40] proposed a different approach to mining frequent itemsets without candidate generation. His algorithm, Eclat, explores the vertical data format, which explicitly stores transactions' identifiers (tidsets) for every itemset. Eclat computes $(n + 1)$-itemsets from n-itemsets by interesting their tidsets. The algorithm collects the initial set of frequent 1-itemset by scanning the transaction database.

Subsequently, Zaki [41] developed CHARM, a more advanced algorithm for mining data in the vertical data format. The algorithm is based on the same idea of intersecting itemsets' tidsets to produce new itemsets, but it specifies the search problem using the notion of itemset-tidset tree (IT-tree). Also, CHARM introduces several optimizations, including the search for closed itemsets.

All the approaches above operate on a database with disjoint transactions, each containing a set of items. On the contrary, our algorithm handles *overlapping* transactions and *itembags* rather than itemsets, which are the two major challenges specific to frequent code change pattern mining from *continuous* sequence of code changes.

6.2 Mining Source Code

Source code mining research has a long history. Here, we present several representative examples.

Michail [25] applied data mining techniques to detect how a library is reused in different applications. The mined library reuse patterns, represented as association rules, facilitate the reuse of the library components by developers.

Li et al. [22] employed frequent itemset mining to extract programming rules from the source code of an application. They also showed that source code fragments that violate the extracted rules are likely to be buggy.

Holmes et al. [19] matched the structural context of the edited source code against a code repository to present a developer with the examples demonstrating the relevant API usage. Similarly, Bruch et al. [4] proposed to improve the IDE's code completion systems by making them learn from code repositories.

Hovemeyer et al. [20] developed FindBugs, a tool that detects a variety of bug patterns in an application by statically matching bug pattern descriptions against the underlying source code. More recently, Lin et al. [24] proposed an approach to search for a specific kind of bug patterns — violations of check-then-act idioms.

All these approaches mine the application's source code, while our algorithm mines code changes, and thus, it can identify new patterns *on-the-fly*.

Another direction of research is mining source code change patterns from the Version Control System (VCS) history of an application. Ying et al. [38] and Zimmermann et al. [43] apply data mining techniques on the application's revision history to detect software artifacts (e.g., methods, classes, etc.) that are usually changed together. The mined association rules predict what other source code locations a developer needs to consider while performing a particular change. Uddin et al. [35] proposed to mine VCS histories of client applications to study how their use of APIs evolves over time, which is helpful both to developers and users of the libraries' APIs. Canfora et al. [5,6] and Thummalapenta et al. [33] used VCS snapshots to study and track the evolution of different software entities such as source lines, bugs, and clones. In the domain of software testing, Zaidman et al. [39] mined software repositories to explore how production and test code co-evolve.

Mining VCS snapshots of an application is exposed to the limited nature of VCS data. In our previous study [26], we showed that data stored in VCS is *imprecise*, *incomplete*, and makes it *impossible* to perform analysis that involves the time dimension inside a single VCS snapshot. Also, similarly to other source code mining techniques, these approaches are not suitable for identifying code change patterns on-the-fly.

6.3 Automated Inference of Refactorings

Early work by Demeyer et al. [9] inferred refactorings by comparing two different versions of source code using heuristics based only on low-level software metrics — method size, class size, and inheritance levels. Kim et al. [21] used a function similarity algorithm to detect methods that have been renamed. More recent refactoring inference approaches detect refactorings depending on how well they match a set of *characteristic properties* that are constructed from the differences between two consecutive versions of an application. Dig et al. [10] employed references of program entities like instantiation, method calls, and type imports as its set of characteristic properties. Weißgerber and Diehl [37] used characteristic properties based on names, signature analysis, and clone detection. Prete et al. [31] developed Ref-Finder, a tool that can infer the widest variety of refactorings to date — up to 63 of the 72 refactorings cataloged by Fowler [14]. Their set of characteristic properties involved accesses, calls, inherited fields, etc.

All these approaches infer refactorings from VCS snapshots, and thus, suffer from the limitations of VCS data. Also, such approaches can not be applied while the corresponding code changes are in progress.

Recently, Ge et al. [15] and Foster et al. [13] proposed tools that continuously monitor code changes to detect and complete manual refactorings in *real-time*. Although this direction of research is very promising, the proposed tools are limited to a single kind of program transformations — refactorings, and detect a small subset of already known refactorings. On the contrary, our algorithm is not restricted to any specific kind of program transformations and is designed to detect previously *unknown* code change patterns.

Wit et al. [8] performed live monitoring of the clipboard to detect clones. Their tool tracked the detected clones, offering several resolution strategies whenever the clones were edited inconsistently. Our approach of detecting similar changes to different parts of the code is complementary to detecting different changes to the similar parts of the code.

7. CONCLUSIONS

Although mining frequent code change patterns has a long research history, we are the first to present an algorithm that mines such patterns from a *continuous* sequence of code changes rather than from the static source code. Our algorithm effectively handles *overlapping* transactions that contain multiple instances of the same item kind — the major challenge that distinguishes our approach from the existing frequent itemset mining techniques. Since our algorithm mines code changes continuously, it can be applied either *on-the-fly* or on a previously recorded sequence of code changes.

To evaluate our algorithm, we used 1,520 hours of real world code changes that we collected from 23 developers. We showed that our mining algorithm is scalable, effective, and useful. Analyzing some of the mining results, we identified ten popular kinds of high level program transformations.

8. ACKNOWLEDGMENTS

We would like to thank Professor Jiawei Han for his guidance in the field of data mining.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36:81–94, November 1993.

[3] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *FSE*, 2009.

[5] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. How long does a bug survive? an empirical study. In *WCRE*, 2011.

[6] G. Canfora, L. Cerulo, and M. D. Penta. Tracking your changes: A language-independent approach. In *IEEE Software*, 2009.

[7] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[8] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, 2009.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, 2000.

[10] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[11] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2:17–23, 2000.

[12] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[13] S. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In *ICSE*, 2012.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *ICSE*, 2012.

[16] T. Guimaraes. Managing application program maintenance expenditures. *Commun. ACM*, 26:739–746, October 1983.

[17] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15, 2007.

[18] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[19] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32, 2006.

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[21] S. Kim, K. Pan, and J. W. Jr. When functions change their names: Automatic detection of origin relationships. In *WCRE*, 2005.

[22] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, 2005.

[23] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21:466–471, June 1978.

[24] Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In *To appear in ICST*, 2013.

[25] A. Michail. Data mining library reuse patterns in user-selected applications. In *ASE*, 1999.

[26] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP*, 2012.

[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 *ICSE*, 2012.

[28] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *ICSE*, 2012.

[29] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.

[30] H. T. T. Petteri Sevon and P. Onkamo. Gene mapping by pattern discovery. In *Data Mining in Bioinformatics*, 2005.

[31] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, 2010.

[32] P. Sevon, H. Toivonen, and V. Ollikainen. TreeDT: Tree pattern mining for gene mapping. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3, 2006.

[33] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. In *Empirical Software Engineering*, 2010.

[34] H. Toivonen, P. Onkamo, P. Hintsanen, E. Terzi, and P. Sevon. Data mining for gene mapping. In *Next Generation of Data Mining Applications*, 2005.

[35] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal api usage patterns. In *ASE*, 2011.

[36] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, 2012.

[37] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, 2006.

[38] A. 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, 2004.

[39] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *ICST*, 2008.

[40] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[41] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *SDM*, 2002.

[42] M. V. Zelkowitz. Perspectives in software engineering. *ACM Comput. Surv.*, 10:197–216, June 1978.

[43] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31, 2005.