2018 12th International Symposium on Theoretical Aspects of Software Engineering

A taxonomy of code changes occurring within a statement or a signature

Chunhua Yang
School of Information
QiLu University of Technology (Shandong Academy of Sciences)
Jinan, China
jnych@126.com

E. James WhiteHead
Dept. of Computational Media
University of California, Santa Cruz
Santa Cruz, USA
ejw@soe.ucsc.edu

***Abstract*—We propose a taxonomy of code changes at a granularity finer than the statement level. It classifies changes that occur within a statement or signature. We firstly define the changes based on the proposed operations on the tree of the statement or signature. Then, we classify the changes according to action type, entity kind, element kind and pattern kind. Based on the current implementation, we applied it to four open Java projects. Through the case study, we validated that the taxonomy can classify changes in all the modified statements and signatures. And, we checked the proportions of change patterns. Furthermore, we demonstrated that it is easy to find out the rename-induced statement updates with the help of the taxonomy. As a result, this taxonomy can be used for further change understanding and change analysis.**

***Keywords*-source code change classification; taxonomy; software evolution**

I. Introduction

Software evolves for improving performance, fixing bugs, or meeting new requirements. Understanding how software changes, can help the development and maintenance process.

It has been proved by existing works that classification of fine–grained changes helps to understand and analyze software changes, e.g. reveal the frequency of different change types [1], uncover patterns of change types [2] [3], predict types of code changes [4], analyze the impact of changes [5], etc.

However, existing approaches to classifying fine–grained code changes generally limit the granularity to the statement-level. They focus on signature changes [2] [5], or inserting, deleting, or moving statements in classes or methods [6]. Changes in a statement are not further categorized.

For instance, Fig.1 shows a within-statement change. Analyzed by ChangeDistiller [6], a change of type `STATEMENT_UPDATE` will be extracted. This change type does not indicate which entities in the statement have changed and how they have changed. Such information are not only indispensable for understanding what happened in the statement, but also important for inferring the dependency and similarity relationship between the statement update and other changes. In the revision containing the statement update in Fig.1, the field `DEFAULT_NAME` has been moved from the interface `Container` to the class `Key`. If the changes in the statement in Fig.1 have been identified and classified, then we learn that the *scope* part of the field access expression `Container.DEFAULT_NAME` in the statement has been updated to `Key`. Then, we can infer the dependency relationship among the statement update and the field movement. And, we can infer the similarity relationship among the statement update in Fig.1 and the one in Fig.2. As a result, we can infer that these changes have accomplished a refactoring task.

Moreover, to properly understand the evolution of software, it is necessary to figure out what happened in the statement update. Currently, as no appropriate approaches for classifying changes in a statement, it is usually assumed that a statement update contributes to the functional modification of the parent method. However, according to [7], many statements are modified simply because they contain references to renamed entities. Such non-essential changes do not affect the functionality of the parent method. Including such changes will cause bias on the results.

Therefore, in order to help understand changes that occur in a statement, we present a classification of the within-statement changes. The major contributions of this paper include:

1) We propose a taxonomy of code changes at the granularity finer than the statement level. Such changes occur within a simple statement or a signature. For convenience, we call such changes *micro changes*. Note that although existing works have classified the changes in the signature of a declaration [2] [6], they did not further study the changes in the initial part of a field declaration or a variable declaration. Therefore, in addition to the changes within a normal statement, the taxonomy classifies changes happened in a signature.

We define micro changes based on the proposed edit operations on the tree of a statement or a signature, to answer two questions: I) In the tree, *which* elements of *which* entities have been changed? II) *How* they changed? Then, we classify the changes according to action type, entity kind, element kind and pattern kind. We propose six kinds of change patterns, in which pattern "$MovingAcrossElements$", "$EncapsulationLR$" and "$EncapsulationRL$" are not presented in any existing work.

2) Based on the current implementation, we have applied

978-1-5386-7305-8/18/$31.00 ©2018 IEEE
DOI 10.1109/TASE.2018.00020

```
337 337     public <T> ContainerBuilder alias(Class<T> type, String alias) {
338         -   return alias(type, Container.DEFAULT_NAME, alias);
    338     +   return alias(type, Key.DEFAULT_NAME, alias);
339 339     }
```

Figure 1. An example statement update. It belongs to a hunk that is part of the textual-differencing output of $ContainerBuilder.java$ in revision [6ab7e1f] of Guice. The red line 338 and green line 338 are the statement before and after update. Other context lines remain unchanged.

```
297 297     public <T> ContainerBuilder factory(Class<T> type, Scope scope) {
298         -   return factory(type, Container.DEFAULT_NAME, type, scope);
    298     +   return factory(type, Key.DEFAULT_NAME, type, scope);
299 299     }
```

Figure 2. Another statement update in revision [6ab7e1f].

the taxonomy to four open projects. Through the case study, we validated the proposed taxonomy. And, we provide a set of accessible example changes[1]. Also, we checked the most common change patterns and detected the rename–induced changes.

The remainder of the paper is organized as follows. In section 2, we introduce terms used in the paper. In section 3, we present the taxonomy. In section 4, we describe the current implementation. We present the case study in section 5. In section 6, we review the related work. We conclude the paper in Section 7.

II. Background

This section introduces terms that are used in this paper.

A. Statement and Signature

This paper uses the nodes of an abstract syntax tree (AST) as program entities. A program consists of type declarations, such as classes, annotations, or enums. A type declaration consists of a signature and a list of body declarations. For example, methods and fields are body declarations of a class. Body declarations other than fields consist of a signature and a list of member statements.

Structure statements such as `if` statement, `while` statement, `synchronized` statement contain a control part and member statements. For example, in the following if-statement, the condition `if(scope == null)` is its control part, while `return this.factory;` is its member statement. A member statement can be a simple statement or another structure statement.

```
if (scope == null) {
    return this.factory;
}
```

For a structure statement, changes to the member parts have been classified by ChangeDistiller, e.g. removing the else-part from an if-statement. Therefore, for a structure statement, we focus on the changes in the control part.

[1]https://github.com/jinanych/MicroChangeExamples

We define the control part of a structure statement as a *statement signature*. For example, `if(scope == null)` is the statement signature of the previous if-statement. In the rest of the paper, we use the term ***statement*** to represent a simple statement, and term ***signature*** to represent a declaration signature or a statement signature.

A statement or signature consists of *elements*. For example, an assignment statement `x=x+1;` consists of three elements, i.e. a *target* `x`, an *op* `=`, and a *value* `x+1`. Many statements or declarations contain keywords. For convenience, we treat keywords in an entity as special elements. For example, the keyword *if* is an special element *if* of an if-statement.

We use the entity types and element kinds defined in Java Parser[2]. An element usually is an expression. Some entities contain other kinds of elements, e.g. `Parameter`, `VariableDeclarator`, `TypeParameter`, etc.

B. Statement/signature AST (SAST for short)

A statement or signature corresponds to a subtree of the AST of the source code file. We define the AST of a statement or a signature as a *Statement/Signature AST* (*SAST* for short), to distinguish it from the AST of a source code file.

In a SAST, the root or each inner node represents an entity, while a leaf represents a token. The root is the statement entity or the signature entity. Each inner node is an entity that belongs to an element part of its parent.

Each node in SAST has a type and a value. For a leaf node, the type is $String$ and the value is the token. For the root or an inner node, the type is the entity kind, while the value is the textual representation of the entity.

Given a SAST node x, we use $t(x)$ to denote the type of x, $v(x)$ to denote the value of x, $p(x)$ to denote the parent of x, and $e(x)$ to denote the kind of the element to which the node belongs in its parent. Let r be the root of a SAST. We define the following auxiliary functions:

- $context(x)$ to return the sequence of the ancestor entities and elements.For example, the context of the blue node $FieldAccessExpr$ in Fig.3 is $ReturnStmt\_expr.MethodCallExpr\_arg$.
- $encapsulated(x, y)$ to check whether node x is encapsulated in node y. If the string function $contains(v(y), v(x))$ holds, then it returns $true$. For example, suppose the values of node x and y are "$injectors$" and "$this.injectors$" respectively, then $encapsulated(x, y)$ returns $true$.
- $tokens(x)$ to return all the tokens of node x.

In the graphical representation of a SAST, the type is placed on the left or right side of the root or an inner node, and the token is put below a leaf node. The inheritance line

[2]The version we used is available from https://github.com/javaparser/javaparser/tree/javaparser-1.0.6.

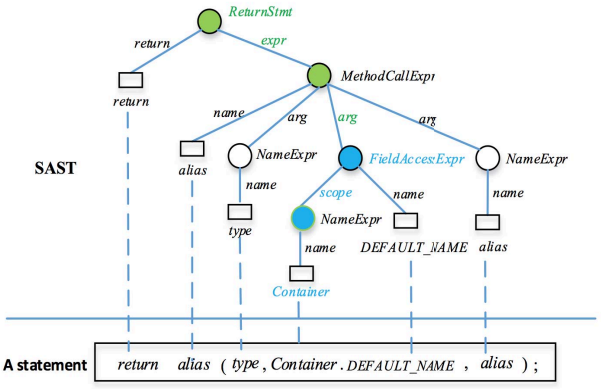

Figure 3. The SAST of the return statement before revision in Fig.1.

from the node to its parent is marked as the name of the element that the node belongs to in its parent.

An example SAST is shown in Fig.3. It is the SAST of the return-statement in the old version shown in Fig.1. In Fig.3, the value of the root or each inner node is omitted. For example, the blue node *FieldAccessExpr* belongs to the `arg` part of its parent *MethodCallExpr*. Its value is "$Container.DEFAULT\_NAME$".

C. The edit operations on SASTs

By comparing two versions of a SAST T_{old} and T_{new}, the changes in a statement or signature can be obtained. We use subscripts $_{old}$ and $_{new}$ to represent the values of the subscripted variable before revision and after revision. Usually, tree-edit operations are utilized to calculate the individual changes to an AST. In ChangeDistiller [8] and GumTree [9], the basic edit operations include *add*, *delete*, *move*, and *update*.

According to our observation, changes in a statement sometimes span different kinds of elements, for example, from the `scope` part to the `arg` part of a method invocation. Furthermore, we found some typical patterns, such as adding or removing casts, extending or limiting conditional expressions, parameter ordering change, etc. Therefore, we define the following basic editing operations on SAST.

- *Insert*: *INS(x, y, l)*; a new node x as one child of the element part l of node y;
- *Delete*: *DEL(x, l)*; delete node x from the element part l of its parent $p(x)$;
- *MoveWithinElement*: *MOV_Within_E(x, i, j, l)*; node x that was the ith child of the element part l of $p(x)$, becomes the jth child of the element part l of $p(x)$.
- *MoveAcrossElements*: *MOV_Across_E(x, l_1, l_2)*; node x that was a child in element part l_1 of $p(x)$, now becomes one child of the element part l_2 of $p(x)$.
- *Update*: *UPD(x, z, y, l)*; replace node x with node z, in which x was a child in element part l of node y and satisfies $!encapsulated(x,z) \wedge !encapsulated(z,x)$.
- *Encapsulate*:*Encap(x, z, y, l)*; replace node x with node z, in which x was a child in element part l of y, and satisfies $encapsulated(x,z)$.
- *Decapsulate*:*Decap(x, z, y, l)*; replace node x with node z, in which x was a child in element part l of y, and satisfies $encapsulated(z,x)$.

The *Insert* or *Delete* operation is like the typical one. However, to describe the movement within one element and that across elements, the typical *move* operation is split as two operations: *MoveWithinElement* and *MoveAcrossElements*, respectively. The typical *update* operation is split as three operations *Update*, *Encapsulate*, and *Decapsulate* according to whether there is an encapsulation relation between the node before update and the one after update.

D. Micro changes

Based on the basic operations on SAST, we define micro changes to describe the changes in SAST. Let n be the node in the SAST where a change took place, *EK* the set of entity kinds, *PK* the set of element kinds, and *PS*={*OrderingChange, MovingAcrossElements, EncapsulationLR, EncapsulationRL, NonCommonTokens, ContainingCommonTokens*} the set of change patterns. The entity kinds, element kinds, and change patterns will be explained in the section 3 in detail.

A micro change is a tuple <*context, action, entity, element, pattern, content*>, in which

- *context* represents the context of the parent of node n;
- *action*∈{*ADDITION, DELETION, CHANGE*} is the action type;
- *entity=t(p(n))*∈*TK* is the entity kind of the parent of node n;
- *element=e(n)*∈*EK* is the kind of the element to which the node belongs in its parent;
- *pattern* ∈*PS* is the change pattern *if action = CHANGE*;
- *content* is the change content, which takes the following value: *content = v(n_{new}), if action= ADDITION*;
  content = v(n_{old}), if action= DELETION;
  content =v(n_{old}) +"->" +v(n_{new}), if action=CHANGE.

Action *ADDITION* denotes the insert operation *INS(n_{new}, p(n_{new}), e(n_{new}))*. Action *DELETION* denotes the deletion operation *DEL(n_{old}, e(n_{old}))*. Action *CHANGE* denotes one of the following operations, each of which results in a kind of change pattern:

- *MOV_Within_E(n, i, j, e(n_{old}))* results in *OrderingChange*;
- *MOV_Across_E(n, e(n_{old}), e(n_{new}))*∧*e(n_{old})*≠*e(n_{new})*results in *MovingAcrossElements*;
- *Encap(n_{old}, n_{new}, p(n_{new}), e(n_{new}))* results in *EncapsulationLR*;
- *Decap(n_{old}, n_{new}, p(n_{new}), e(n_{new}))* results in *EncapsulationRL*;
- *UPD(n_{old}, n_{new}, p(n_{new}), e(n_{new}))* ∧ *tokens(n_{old})*∩ *tokens(n_{new})*=ϕ results in *NonCommonTokens*;

- *UPD(n_{old}, n_{new}, p(n_{new}), e(n_{new}))∧tokens(n_{old})∩ tokens(n_{new})≠ ϕ* results in *ContainingCommonTokens*.

Dimension $context$, $entity$ and $element$ describe ***where*** the micro change occurs. Dimension $action$ and $pattern$ describe ***how*** it happens, whereas $content$ describes ***what*** happened.

III. The taxonomy

The taxonomy of micro changes is listed in Table I.

A. Change types

For convenience, we use *change type* to represent $action$, $entity$ and $element$, which is denoted in the format "$action$ $entity\_element$", e.g. $CHANGE$ $FieldAccessExpr\_scope$.

(1) The entity kinds and the element kinds. We group entity kinds as four categories: *declaration*, *statement*, *expression*, and *other*.

Take *ClassOrInterfaceDeclaration* for illustration. Its elements include *annotation*, *modifier*, *name*, *typeparameter*, *isinterface*, *inherits*, and *inheritspart*, in which *annotation* is an annotation, *modifier* is a modifier, *typeparameter* is a type parameter, *name* is the name, *isinterface* represents the keyword *interface* or *class*, *inherits* stands for the keyword *extends* or *implements*, while *inheritspart* is a type that the declaration extends or implements.

Some elements e.g. *name* are common to many entity kinds. The element *variable* in *FieldDeclaration* acts like *name* but not the same. Several fields can be declared in one field declaration. Each variable in a *FieldDeclaration* is a *VariableDeclarator*. All the variables in a *FieldDeclaration* share the same type, modifiers, and annotations. Note that the element *isthis* in *ExplicitConstructorInvocationStmt* represents the keyword *this* or *super*.

For saving space, we use *LiteralExpr_value* to represents the element value of all literal expressions including *StringLiteralExpr*, *BooleanLiteralExpr*, *IntegerLiteralExpr*, etc.

(2) The action type. Changes types beginning with *C* means that the elements are indispensable and can only be changed. Example elements are *name*, *type*, *condition*, etc.

Change types beginning with $A/D/C$ means that the elements in the entity are optional and can be added, deleted, or changed. Typical example elements include *arg*, *parameter*, *scope*, *modifier*, *annotation*, and so on.

Change types beginning with A/D means that the elements in the entity are optional but can only be added or deleted. Examples elements are *arraycount* and *isvarargs*.

B. Change patterns

Pattern *OrderingChange* describes the ordinal change of elements. Usually, it appears in the parameters or arguments of entities such as *MethodDeclaration*, *MethodCallExpr*, *ObjectCreationExpr*, etc.

Pattern *MovingAcrossElements* describes the changes that move a node from an element part to another. A typical movement is between the *arg* and *scope* of a method call expression, such as the movement of `buffer` shown below.

```
getIcon(buffer)-> buffer.getIcon()
```

Pattern *EncapsulationLR* and *EncapsulationRL* describe the encapsulation pattern. The "LR" or "RL" in the pattern name indicates the direction. The following two examples illustrate the two patterns. The first one is an *EncapsulationLR* that encapsulates the *NameExpr* `interceptors` into the *FieldAccessExpr* `this.interceptors` as its *name*. This is what we usually call *adding this* pattern. The second one is an *EncapsulationRL* that decapsulates `builder` from the *scope* of *MethodCallExpr* `builder.getSource()`.

```
interceptors -> this.interceptors
builder.getSource()-> builder
```

Other types of micro changes are irregular. According to whether there are common tokens between the content before and after revision, we divide them into two kinds: *NonCommonTokens* and *ContainingCommonTokens*.

C. Denotation

Let *x* be the changed node. We call the parent of *x change entity*, and the element part of *x* in its parent *change element*. A micro change is expressed in the following format:

`action context.entity_element` "*content*".

For example, in Fig.3, the blue node *NameExpr* is the changed node. Therefore, the change entity is *FieldAccessExpr*, the change element is *scope*. Then, the micro change is expressed below. Its change pattern is *NonCommonTokens*.

`CHANGE ReturnStmt_expr.MethodCallExpr_arg.FieldAccessExpr_scope` "`Container->Key`".

D. Limitations

The taxonomy focuses on object-oriented programming languages, especially Java. It currently does not support systems written in procedural languages.

IV. Implementation

We have implemented an approach to extract micro changes from two versions of a Java source file. The outline of the approach is shown as follows:

1) We identify hunks based on the GNU Diff[3] algorithm.

2) We have invented a technology to match hunks with AST nodes.

By rewriting the *DumpVisitor* in Java Parser, we add the element name to each node and connect each token with a leaf. Through traversing the old AST, we get the deleted token set. Then, for each hunk, based on the range of removed lines, we retrieve all of its tokens from the deleted token set. Then, from these tokens, we extract a group of removed declarations and(/or) statements. Similarly, we get

[3]http://www.gnu.org/software/diffutils/

Table I

OVERVIEW OF CHANGE TYPES. EACH CHANGE TYPE IS EXPRESSED AS "$action\ entity\_element$". LABEL A STANDS FOR ACTION $ADDITION$, D FOR $DELETION$, AND C FOR $CHANGE$. THE BACKSLASH BETWEEN ACTIONS OR ELEMENTS MEANS or. FOR EXAMPLE, "$C\ ForeachStmt\_var/iterable$" INDICATES TWO CHANGE TYPES $CHANGE\ ForeachStmt\_var$ AND $CHANGE\ ForeachStmt\_iterable$. $*LiteralExpr\_value$ REPRESENTS THE $value$ PART OF ALL LITERAL EXPRESSIONS INCLUDING $StringLiteralExpr$, $BooleanLiteralExpr$, $IntegerLiteralExpr$, ETC.

Declaration	
A/D/C AnnotationDeclaration_annotation/modifier	C AnnotationDeclaration_name
A/D/C AnnotationMemberDeclaration_annotation/modifier/defaultvalue	A/D AnnotationMemberDeclaration_default
C AnnotationMemberDeclaration_name/type	A/D/C ClassOrInterfaceDeclaration_annotation/inherits/inheritpart
A/D/C ClassOrInterfaceDeclaration_modifier/typeparameter	C ClassOrInterfaceDeclaration_name/isinterface
A/D/C ConstructorDeclaration_annotation/modifier/parameter	A/D/C ConstructorDeclaration_throwspart/typeparameter
C ConstructorDeclaration_name	A/D/C EnumDeclaration_annotation/modifier/entries/inherits/inheritpart
C EnumDeclaration_name	A/D/C EnumConstantDeclaration_annotation/arg
C EnumConstantDeclaration_name	A/D/C FieldDeclaration_annotation/modifier/variable
C FieldDeclaration_type	A/D/C MethodDeclaration_annotation/modifier/typeparameter/parameter
A/D/C MethodDeclaration_throwspart/type	C MethodDeclaration_name
A/D MethodDeclaration_arraycount	C ImportDeclaration_name
A/D ImportDeclaration_isasterrisk	A/D InitializerDeclaration_isstatic
A/D/C InitializerDeclaration_annotation	C PackageDeclaration_name
Statement	
C AssertStmt_check	A/D/C AssertStmt_msg
A/D/C BreakStmt_id	C CatchClause_except
A/D/C ContinueStmt_id	C DoStmt_condition
A/D/C ExplicitConstructorInvocationStmt _arg/typearg	C ExplicitConstructorInvocationStmt_isthis
C ExpressionStmt_expr	A/D/C ForStmt_compare/init/update
C ForeachStmt_var/iterable	C IfStmt_condition
C LabeledStmt_label/stmt	A/D/C ReturnStmt_expr
C SwitchEntryStmt_label	C SwitchStmt_selector
C SynchronizedStmt_expr	C ThrowStmt_expr
C WhileStmt_condition	
Expression	
C ArrayAccessExpr_name/index	C ArrayCreationExpr_type
A/D ArrayCreationExpr_arraycount	A/D/C ArrayCreationExpr_initializer/dimension
A/D/C ArrayInitializerExpr_values	C AssignExpr_op/target/value
C BinaryExpr_op/left/right	C CastExpr_type/expr
C ClassExpr_type	C ConditionalExpr_condition/thenexpr/elseexpr
C EnclosedExpr_inner	C FieldAccessExpr_name/scope
A/D/C FieldAccessExpr_typearg	A/D FieldAccessExpr_arraycount
C InstanceOfExpr_expr/type	C MethodCallExpr_name
A/D/C MethodCallExpr_arg/scope/typearg	A/D/C ObjectCreationExpr_arg/scope/typearg
C ObjectCreationExpr_type	C MarkerAnnotationExpr_name
C MemberValuePair_name/value	C NormalAnnotationExpr_name
A/D/C NormalAnnotationExpr_pairs	C SingleMemberAnnotaionExpr_name/membervalue
C NameExpr_name	C QualifiedNameExpr_name/qualifier
C *LiteralExpr_value	A/D/C SuperExpr_classexpr
A/D/C ThisExpr_classexpr	C UnaryExpr_op/expr
A/D/C VariableDeclarationExpr_annotation/modifier/variable	C VariableDeclarationExpr_type
Other	
A/D/C Parameter_annotation/modifier	C Parameter_name/type
A/D Parameter_isvarargs	A/D/C TypeParameter_typebound
C TypeParameter_name	A/D/C VariableDeclarator_init
C VariableDeclarator_id	C VariableDeclaratorId_name
A/D VariableDeclaratorId_arraycount	C PrimitiveType_name
C ClassOrInterfaceType_name	A/D/C ClassOrInterfaceType_scope/typearg
A/D/C WildcardType_inherits/inheritpart	C ReferenceType_type
A/D ReferenceType_arraycount	

a group of added declarations or statements for each hunk by visiting the new AST.

3) Identify modified statements and declarations from hunks. For each hunk, we match the deleted statements with the added statements, the deleted declarations and the added declarations, respectively. The matching process is performed in ascending order of line numbers. After matching, each pair corresponds to a modified statement or declaration.

4) Extract micro changes. We have implemented a tree-differencing algorithm on a SAST pair to extract micro changes. Based on a top-down traversal strategy, the algorithm matches nodes in the old tree and the new tree according to their element names and values.

V. Case study

We have applied the current implementation to the four open projects: jEdit[4], eclipse JDT Core[5], Apache maven[6], and google-guice[7]. The aim is twofold:

1) To validate the taxonomy by extracting micro changes from all of the changed statements or signatures;

2) Through finding the most common pattern kinds and checking the rename–induced changes, to demonstrate that the taxonomy can help understand and analyze changes.

A. The Data Set

Using a MininGit tool [8], we extracted the change history of the selected projects from their source repository and stored it into a database. Information about the selected projects is shown in Table II. We filtered out revisions that do not contain modified code files, as well as those that only changed blank lines, code styles, or comments. There are a total of 17,406 revisions left.

Table II
Study projects, and their total number of revisions containing code changes in the time period.

Project	Period	Revisions
jEdit	1998-09-27∼2012-08-08	3,523
JDT Core	2001-06-05∼2013-10-16	8,783
Maven	2003-09-01∼2014-01-29	4,453
Guice	2006-08-22∼2013-12-11	647
Total		17,406

B. The validation

1) Micro changes extracted: By applying the implementation to all modified code files in the revision set, we extracted a total of 212,883 code hunks. From all the code hunks, we got 107,570 modified statements and declarations, of which 84,064 (78%) are statements, while 23,506 (22%) are declarations.

Table III
Number of hunks, modified statements/declarations, and micro changes extracted from each project.

Project	Code hunks	Modified Stmts/Decls			Micro changes
		Total	*Stmts*	*Decls*	
jEdit	50,279	21,953	16,585	5,368	28,070
JDTCore	110,640	61,281	51,170	10,111	78,462
Maven	42,275	18,313	12,664	5,649	25,945
Guice	9,689	6,023	3,645	2,378	8,030
Total	212,883	107,570	84,064	23,506	140,507

A total of 140,507 micro changes were extracted from all of the modified statements and declarations. And, from any modified statement or signature, at least a micro change has been extracted, which proves that the taxonomy can describe changes that occur in a statement or signature. Table III lists the numbers.

Except for several entity kinds and element kinds, all the types listed in the taxonomy had been identified. No micro changes were detected from entity *EnumConstantDeclaration*, *InitializerDeclaration*, *MarkerAnnotationExpr*, *NormalAnnotationExpr*, *MemberValuePair*, *LiteralExpr*, etc. And, no micro changes were detected from the *annotation* part of all declarations, the *arraycount* part of a *MethodDeclaration*, a *FieldAccessExpr* or a *Parameter*, the *typearg* part of a *FieldAccessExpr* or an *ObjectCreationExpr*, etc.

2) A package of examples: We made an accessible package of example micro changes for each change type and each pattern extracted. These changes are randomly selected. In addition to the change itself, each example includes the statement or signature that contains the change and the hunk where the statement or signature is located.

C. Finding the most common pattern kinds

We computed the number of micro changes of each pattern kind. Table IV lists the number of changes in the first three common pattern kinds, in which $P1$ stands for the number of changes of pattern *NonCommonTokens*, $P2$ the total number of changes of pattern *EncapsulationLR* and *EncapsulationRL*, and $P3$ the number of changes of pattern *ContainingCommonTokens*.

According to the table, on average, of all the changes that contain patterns, 63% have pattern *NonCommonTokens*, 25% have *Encapsulation*, 11% have *ContainingCommonTokens*, and the remaining 1% have the other patterns.

Through browsing the contents of changes of pattern *NonCommonTokens*, we found that many of them involve a single token, e.g. a reserved word and a user-defined name. For example, the contents of 260 micro changes are "`Constant->BooleanConstant`". Changes to modifiers are often "`public->final`".

[4]https://github.com/linzhp/jEdit-Clone
[5]https://github.com/eclipse/eclipse.jdt.core
[6]https://github.com/apache/maven/
[7]https://github.com/google/guice/
[8]MiniGit: https://github.com/SoftwareIntrospectionLab/MininGit

Table IV
THE FIRST THREE MOST COMMON PATTERNS IN EACH PROJECT. EACH CELL IS EXPRESSED AS $number(percent)$, IN WHICH $number$ MEANS THE NUMBER OF MICRO CHANGES.

Project	Total	P1	P2	P3
jEdit	20,435	14,267(70%)	2,939(14%)	2,778(14%)
JDT Core	62,952	36,545(58%)	20,504(33%)	5,115(8%)
Maven	16,048	11,442(71%)	2,031(13%)	2,096(13%)
Guice	5,825	3,757(64%)	561(10%)	1,396(24%)
Total	105,260	66,011(63%)	26,035(25%)	11,385(11%)

Table V
OF ALL THE REVISIONS CONTAINING CHANGES OF TYPE CHANGE METHODCALLEXPR_ARG, THOSE CAUSED BY RENAME–INDUCED CHANGES.

Project	Total	Rename–induced		
		Total	*Variable/Field*	*Parameter*
jEdit	240	99(41%)	53	55
JDT Core	407	229(56%)	153	116
Maven	267	114(43%)	51	74
Guice	74	54(73%)	34	39
Total	988	496(50%)	291	284

As for pattern *ContainingCommonTokens*, common tokens among the content before revision and that after revision are reserved words e.g. `new` and `this`, or user-defined tokens.

By searching the contents of micro changes of pattern *EncapsulationLR*, we found that many of them added keyword `this` to a *name* or an *expression*. For example, the contents of 435 changes are "`binding->this.binding`". Moreover, we found that many changes of pattern *EncapsulationRL* removed keyword `this`.

D. Checking the renamed–induced changes

With the help of the proposed taxonomy, we can identify renaming of a field, variable or parameter by finding changes of the following change types:

`CHANGE VariableDeclarator_id`,
`CHANGE VariableDeclarationExpr_variable`,
`CHANGE FieldDeclaration_variable`,
`CHANGE Parameter_name`,
`CHANGE ConstructorDeclaration_parameter`,
`CHANGE MethodDeclaration_parameter`,
or `CHANGE CatchClause_except`.

The first three types represent renaming changes to fields or variables. Others represent changes to parameters.

We computed rename-induced micro changes of type *CHANGE MethodCallExpr_arg*. Table V shows the result. According to the table, on average, in the half of the revisions that contain changes of type *CHANGE MethodCallExpr_arg*, the changes are caused by references to renamed entities (i.e., fields, variables, or parameters). In Guice, the percentage is as high as 73%. In addition, of all the rename-induced changes, half are caused by renamed variables or fields and the other half is caused by renamed parameters. This result corroborates the findings of a previous investigation by Kawrykow and Robillard [7], which showed the high ratio of non-essential changes in method updates.

E. Discussion

The taxonomy can be applied to the following scenarios.

Change dependency analysis. We have illustrated how to detect statement updates caused by referencing renamed entities. In addition to the rename–induced relationship, other kinds of dependencies between changes can be detected by combining the proposed taxonomy with a statement-level taxonomy, such as ChangeDistiller.

For example, for the revision containing the statement update in Fig.1, ChangeDistiller can extract an *ADDITIONAL_OBJECT_STATE* type of change for the field removal from `Container` and a *REMOVED_OBJECT_STATE* type of change for the field addition to `Key`. Using the proposed taxonomy, a *CHANGE FieldAccessExpr_scope* type of micro change will be detected from the statement update. Combining the information provided by ChangeDistiller and the taxonomy, we can detect the dependency among the statement update in Fig.1 and the field movement.

Similarly, we can detect the dependencies between the method removal/addition and the changes in the name of method invocations, the dependencies between the class removal/addition and the changes to the *type* parts of some entities, and so on.

Change similarity analysis. Patterns suggest similarities among changes. Existing work [10] has studied similar changes in method renaming. Based on the information provided by the proposed taxonomy, it is possible to analyze the similarities between changes in any element part of an entity, not just the name of a method.

Other possible scenarios include software evolution analysis, change impact analysis, etc.

F. Threats of validity

The micro changes used in the case study were extracted from the hunk set generated by the GNU diff. A limitation of textual differencing algorithms is that they do not always return syntax reasonable hunks. In addition, when a hunk contains more than one statement of the same type, the simple matching strategy we used to identify the modified statement or signature does not guarantee a perfect match. To a certain extent, this led to over-grinding changes, which may cause bias on the results of the case study.

VI. RELATED WORKS

Change classification. Fluri and Gall proposed the Change Distiller taxonomy [6], which classifies changes

according to tree edit operations. Each change type is assigned a significance level. They focus on changes in classes, methods, and fields, including signature-level and statement-level changes. However, changes to an annotation or an enum are not included in their taxonomy. Kim et al. [2] proposed a classification of the function signatures based on the impact on the data flow between caller and callee. However, they only focus on changes within the signature of methods. Sun et al. [5] classified code changes to analyze the impact of different change types. The classification focuses on changes to the signatures of classes, fields and methods. By contrast, our taxonomy focuses on changes that occur within a signature or statement, which is complementary to these taxonomies.

Textual Differencing. The well-known GNU diff [11] can return lines added or removed. LDiff [12] and LHdiff [13] improve the GNU diff to detect moved lines. TkDiff[9] and Kompare[10] can exhibit the within-line changes through highlighting the changed tokens in a line. The main issue with the textual-differencing approaches is that they only return textual lines or tokens.

Tree Differencing. Tree differencing approaches return fine-grained structural changes by comparing two versions of AST of the source code. The most famous algorithm is ChangeDistiller [8]. GumTree [9] and MTDIFF [14] improved ChangeDistiller by producing a shorter edit script. Changes returned by ChangeDistiller are classified as different types based on the taxonomy they proposed [6]. However, GumTree and MTDIFF return the edit scripts, with no high-level type information. Our implementation is based on a tree-differencing algorithm and the proposed taxonomy. Therefore, it returns classified changes.

VII. Conclusion

We defined micro changes to describe the changes that occur in a statement or in the signature of a declaration. And, we classified micro changes according to the action type, the entity kind, the element kind, and the proposed pattern kind. Through the case study, we validated that the taxonomy can classify changes in modified statements and signatures. And, we uncovered the high percentage of *Encapsulation* patterns. Also, we demonstrated that it is easy for detecting the rename-induced micro changes using the proposed taxonomy.

Future work includes applying the taxonomy to analyze the dependencies and similarities between changes.

Acknowledgment

The work is supported by the National Natural Science Foundation of China under Grant No.61375013 and No.61502259.

[9]https://sourceforge.net/projects/tkdiff/

[10]https://www.kde.org/applications/development/kompare/

References

[1] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," ACM SIGSOFT Software Engineering Notes, 2005, vol.30, no.4, pp.1-5.

[2] S. Kim and E. J. Whitehead, "Properties of signature change patterns," in Proceedings of the 22nd IEEE International Conference on .Software Maintenance(ICSM'06), 2006, pp. 4-13.

[3] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 463-466.

[4] E. Giger, M. Pinzger, and H. C. Gall, "Can we predict types of code changes? an empirical analysis," In 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pp. 217-226.

[5] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, (2010, July). "Change impact analysis based on a taxonomy of change types," In Proceedings of the 34th Annual Computer Software and Applications Conference (COMPSAC), 2010, pp. 373-382.

[6] B. Fluri and H. C. Gall,"Classifying change types for qualifying change couplings," In Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC 2006, pp. 35-45.

[7] D. Kawrykow, and M. P. Robillard, "Non-essential changes in version histories," In Proceedings of the 33rd International Conference on Software Engineering 2011, pp. 351-360.

[8] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," IEEE Transactions on software engineering, 2007, vol.33, no.11, pp.725-743.

[9] 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, 2014, pp. 313-324.

[10] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," In Proceedings of the 12th Working Conference on Mining Software Repositories, 2015, pp. 180-190.

[11] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," Communications of the ACM, 1977, vol.20, no.5, pp.350-353.

[12] G. Canfora, L. Cerulo and M. Di Penta, "Ldiff: An enhanced line differencing tool," In Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 595-598.

[13] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, " LHDiff: A language-independent hybrid approach for tracking source code lines," In 29th IEEE International Conference on Software Maintenance (ICSM 2013), 2013, pp. 230-239.

[14] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 660-671.