On Preserving the Behavior in Software Refactoring: A Systematic Literature Review

1. Running Example

In this section, we demonstrate an example that will be used throughout the paper. The example in Listing 1 and 2 illustrates unexpected behavioral changes performed by Eclipse. The original and resulting program are shown with a diff view in which green is for inserts and red is for deletions.

Consider class `Employee` and its subclass `Salesman` as illustrated in Listing 1. Class `Salesman` declares methods `setSSN`, `getSSN`, `getFullName`, `getSalary`, `getSomething`, `toString`, `yearlySalary`, `yearlySalaryIncrease`, `displayYearlySalaryIncrease`, `test1`, and `test2`. Running the method `getSomething` in Listing 1 yields 1000.33. Suppose we want to apply the *PullUpMethod* refactoring using Eclipse to move method `getSomething` from `Salesman` to `Employee`. This method contains a reference to `Employee.getSalary` using the `super` access. The use of Eclipse will produce the program presented in Listing 2. Although method `getSomething` is moved to the superclass and `super` is updated to `this`, a behavioral change was introduced (i.e., method `test1` in the target program (Listing 2) yields 2000.90 instead of 1000.33).

2. Examples of the Behavior Preservation Approaches

To summarize the behavior preservation approach topics covered by the primary studies, we derived the keywords of each study from its title. To get a high-level picture of the covered topics, we generated a word cloud of paper titles as depicted in Figure 1.

2.1. Refactoring Formalisms and Techniques

A detailed overview of refactoring operations and their overlapped between strategies is depicted in Tables 1 and 2.

2.1.0.1. Graph Transformation. We summarize the formal properties by showing the correspondence between refactoring and graph transformation as shown in Table 3.

Listing 1: Original Program

```
public class Employee {
    public double yearlySalary;
    public String getName()
    {   return "John";}
    public double getSalary()
    {   return 1000.33;}
    public double yearlySalary() {
        return yearlySalary=
        (getSalary() * 12);
    }
}



public class Salesman extends Employee
    {public String ssn;
    public void setSSN(String setSNN)
    {   ssn=setSNN;}
    public String getSSN()
    {   return ssn;}
    public String getFullName()
    {   return "John Smith";}
    public double getSalary()
    {   return 2000.99; }
+   public double getSomething() {
+       return super.getSalary();
+   }
    public String toString() {
        return "Employee[Salary="
        +getSalary()+"]";
    }
    public double yearlySalary() {
        double yearlySalary;
        yearlySalary=(getSalary()*12);
        return yearlySalary;
    }
    public double yearlySalaryIncrease()
    {    double yearlySalaryIncrease;
        yearlySalaryIncrease =
        (((yearlySalary()*
        (0.1)) + yearlySalary()));
        return yearlySalaryIncrease;
    }
    public void
    displayYearlySalaryIncrease() {
        System.out.printf
        ("Yearly Salary Increase
        is"+yearlySalaryIncrease());
    }
    public double test1() {
        return getSomething();
    }
    public String test2() {
        return getName();
    }
}
public class TestRefactoring {
    public static void main
    (String[] args) {
    Salesman s=new Salesman();
    System.out.println(s.toString());
+   System.out.println(s.test1());
    System.out.println(s.test2());
    }
}
```

Listing 2: Resulting Program

```
public class Employee {
    public double yearlySalary;
    public String getName()
    {   return "John";}
    public double getSalary()
    {   return 1000.33;}
    public double yearlySalary() {
        return yearlySalary=
        (getSalary() * 12);
    }
+   public double getSomething() {
+       return this.getSalary();
+   }
}

public class Salesman extends Employee
    {public String ssn;
    public void setSSN(String setSNN)
    {   ssn=setSNN;}
    public String getSSN()
    {   return ssn;}
    public String getFullName()
    {   return "John Smith";    }
    public double getSalary()
    {   return 2000.99;}
-   public double getSomething() {
-       return super.getSalary();
-   }
    public String toString() {
        return "Employee[Salary="
        +getSalary()+"]";
    }
    public double yearlySalary() {
        double yearlySalary;
        yearlySalary=(getSalary()*12);
        return yearlySalary;
    }
    public double yearlySalaryIncrease()
    {    double yearlySalaryIncrease;
        yearlySalaryIncrease =
        (((yearlySalary()*
        (0.1)) + yearlySalary()));
        return yearlySalaryIncrease;
    }
    public void
    displayYearlySalaryIncrease() {
        System.out.printf
        ("Yearly Salary Increase
        is"+yearlySalaryIncrease());
    }
    public double test1() {
        return getSomething();
    }
    public String test2() {
        return getName();
    }
}
public class TestRefactoring {
    public static void main
    (String[] args) {
    Salesman s=new Salesman();
    System.out.println(s.toString());
+   System.out.println(s.test1());
    System.out.println(s.test2());
    }
}
```

Table 1: Behavior Preservation Approaches and its Strategies in Related Work.

Study	Year	Approach	Strategy	Refactorings
Roberts et al. [1]	1997	Refactoring Safety Tool	Precondition Checking	Add Variable
				Rename Variable
				Remove Variable
				Push Down Variable into Subclass(es)
				Pull Up Variable from Subclass(es)
				Create Accessors for a Variable
				Change all Variable refs to Accessors Calls
				Create New Class
				Rename Class
				Remove Class
				Add Method
				Rename Method
				Remove Method
				Push Down Method into Subclass(es)
				Pull Up Method from Subclass(es)
				Add Parameter to Method
				Move Method across Object Boundary
				Extract Code as Method
Mens et al. [2]	2003	Graph Transformation	Graph Rewriting Rules & Expressions	Encapsulate Field
				Pull Up Method
Tip et al. [3] [4]	2003,2011	Type Constraints	Constraint Rules	Extract Interface
				Pull Up Method
				Pull Up Field
				Push Down Methods
				Push Down Field
				Extract Subclass
				Generalize Type
Garrido and Meseguer [5]	2006	Formal Specification & Verification	Rewriting Logic	Push Down Method
				Pull Up Field
				Rename Temporary
Straeten et al. [6]	2007	Model Transformation	Description Logic	Move States into Orthogonal Composite State
				Flatten States
				Add Subclass
Massoni et al. [7]	2008	Model Transformation	Laws of Programming	Introduce Generalization
				Introduce Signature
				Introduce Subsignature
				Introduce Relation
				Remove Optional Relation
				Remove Scalar Relation
				Split Relation
Soares et al. [8] [9] [10][11]	2009,2010,2011	Refactoring Safety Tool	Test Suite Generation	Rename Class
				Rename Field
				Rename Local Variable
				Rename Intertype Declaration
				Rename Variable
				Rename Method
				Encapsulate Field
				Extract Method
				Extract Class
				Push Down Method
				Move Class
				Change Method Signature
				Pull Up Method
				Extract Exception Handler
				Infer Generic Type
				Replace Deprecated Code
				Inline Method
Ubayashi et al. [12]	2008	Contract-based Verification	Contract Writing Language	Change Abstract Class to Interface
				Extract Feature into Aspect
				Extract Fragment into Advice
				Extract Inner Class to Standalone
				Inline Class within Aspect
				Inline Interface within Aspect
				Move Field from Class to Inter-type
				Move Method from Class to Inter-type
				Replace Implements with Declare Parents
				Split Abstract Class into Aspect and Interface
				Extend Marker Interface with Signature
				Generalize Target Type with Marker Interface
				Introduce Aspect Protection
				Replace Inter-type Field with Aspect Map
				Inter-type Method with AspectMethod
				Tidy Up Internal Aspect Structure
				Extract Superaspect
				Pull Up Advice
				Pull Up Declare Parents
				Pull Up Inter-type Declaration
				Pull Up Marker Interface
				Pull Up Pointcut
Schäfer et al. [13]	2008	Naming Binding Preservation	Invariant-based	Rename
Tsantalis and Chatzigeorgiou [14]	2009	Precondition Examination	Precondition Checking	Move Method
Schäfer and Moor [15]	2010	Specification-based Refactoring	Dependency Preservation	Convert Anonymous To Nested
			Language Extension	Extract Class
			Microrefactorings	Extract Constant
				Extract Temp
				Inline Constant
				Inline Temp
				Introduce Factory
				Introduce Indirection
				Introduce Parameter
				Introduce Parameter Object
				Move Inner to Toplevel
				Move Instance Method
				Move Members
				Promote Temp to Field
				Pull Up
				Push Down
				Self-Encapsulate Filed
Tsantalis and Chatzigeorgiou [16]	2010	Refactoring Safety Tool	Precondition Checking	

Table 2: continued from previous page.

Study	Year	Approach	Strategy	Refactorings
Overbey and Johnson [17]	2011	Differential Precondition Checking	Preservation Analysis Algorithm	Rename Move Introduce USE Change Function Signature Introduce Implicit None Add Empty Subprogram Safe Delete Pull Up Method Copy Up Method Extract Local Variable Add Local Variable Introduce Block Insert Assignment Move Expression Extract Function Add Empty Function Populate Function Replace Expression
Soares et al. [18], Mongiovi et al. [19]	2011,2017	Overly Strong Preconditions Identification	Differential Testing Disabling Preconditions	Add Method Remove Method Change Method Body Change Method Modifier Add Field Remove Field Change Field Modifier Change Field Initializer Change Static Field Initializer Rename Class Rename Method Rename Field Rename Intertype Declaration Push Down Method Pull Up Method Inline Method Pull Up Field
Jonge and Visser [20]	2012	Name Binding Preservation	Invariant-based	Rename
Noguera et al. [21]	2012	Refactoring Safety Tool	Annotation-aware	Not Mentioned
Thies and Bodden [22]	2012	Refactoring Safety Tool	Reflective Calls	Rename Field Rename Method Rename Type Rename Package Move Type Change Method Signature
Soares et al. [23]	2013	Refactoring Safety Tool Commit Message Analysis Manual Analysis	Test Suite Generation Keywords-based Search Source Code Comparison	Pull Up Method Replace Code with Method Call Move Operation to Listener Extract Method Remove Unused Variable Change Instance Access to Static Remove Immutable Object Copy Replace Direct Access with Getter Replace Instance with isInstance Add Parameter Remove Parameter Replace Field with Method Decrease Method Visibility Replace Direct Access with Setter Inline Temp Move Method Consolidate Duplicate Code Fragment Rename Constant Rename Local Variable Replace Generic Cast with classCast Replace Generic Cast with isInstance Replace Method with Method Object Change Statement Order Swap Access Method Remove Duplicate Assignment Consolidate Conditional Expression Introduce Explaining Variable Remove Assignment to Parameters Rename Class Increase Method Visibility Rename Method Rename Field Replace if with Switch Replace Equivalent Method Call Introduce Null Object Replace Magic Number with Constant
Soares et al. [8], Mongiovi et al. [24]	2009,2014	Refactoring Safety Tool	Change Impact Analysis	Pull Up Method Rename Method Move Method Push Down Field Push Down Method Add Parameter Encapsulate Field Rename Field Rename Type
Najaf et al. [25]	2016	Annealing & Introduce Subtyping	UML-B Refactoring Rules	Not Mentioned
Horpácsi et al. [26]	2017	Decomposition & Schemes	Strategic Term Rewriting Rules	Wrap (Change) Expression Extract to Function Extract to Variable Outer Variable Variable to Function Parameter Rename Function
Chen et al. [27]	2018	Refactoring Safety Tool	Test Suite Generation	Extract Method Pull Up Method Move Method Extract Super Class Move Type to New File Extract & Move Method Extract & Pull Up Method
Insa et al. [28]	2018	Refactoring Safety Tool	Test Suite Generation	Not Mentioned

Figure 1: Word Cloud of Paper Titles.

Table 3: Formal Properties of Graph Transformation (Extracted from Mens and Tourwe [29])

Refactoring	**Graph Transformation**
software artifact	graph
refactoring	graph production
composite refactoring	composition of graph productions
refactoring application	graph transformation
refactoring precondition	application precondition
refactoring postcondition	application postcondition

2.1.0.2. Type Constraints. Because this technique is developed only for generalization-related refactorings, we use *Pull Up Attribute* and *Pull Up Method* refactorings to demonstrate this technique. By referring to the example in Section 1, we notice that certain preconditions need to be checked before performing refactoring, as follows:

- Methods `setSSN` , `getSSN`, and field `ssn` can be pulled up from class `Salesman` into class `Employee` without affecting program behavior.
- Method `yearlySalary` cannot be pulled up into class `Employee` because class `Employee` has a method with same signature defined.
- If method `toString` pulled up into superclass, there is no compilation error introduced but the program is behaviorally changed. This is because the call `s.toString()` dispatches to a different method implementation of the method `toString()`.
- Method `displayYearlySalaryIncrease` cannot be pulled up without pulling up `yearlySalaryIncrease()` because `yearlySalaryIncrease()` is not declared in class `Employee`.

2.1.0.3. Formal Specification and Verification. Consider the formal specification of *Pull Up Attribute* defined in [5]. By applying this refactoring operation on field `ssn` (Listing 1) to move the field to the class `Employee`, the following preconditions must hold in order for transformation to be carried out successfully.

- There is a class named `Employee`.
- Class `Employee` has at least one subclass.
- Class `Employee` does not define the field `ssn`.
- Subclass of `Employee` defines the field `ssn`.

These preconditions are checked by `preconditionsPullUpFieldHold` operation and applied by operation `applyPullUpField` in the formal specification listed in [5].

2.1.0.4. Model Transformation .

Model Refactoring and Model Refinement An example of this approach is illustrated briefly in Figure 2. Class `Employee` (version 1.0) is behaviorally refined into a subclass `Salesman` (version 1.1) using the inheritance consistency relationship (i.e., the behavior of a subclass should specialize the behavior of a superclass). Suppose that the subclass `Salesman` evolves into a new version (version 1.2) by either adding new functionality or removing existing functionality. The evolved version of class `Salesman` should still be behaviorally consistent with the class `Employee`. For the purpose of simplifying the design of the class, suppose that `Salesman` (version 1.2) is refactored into a new version (version 1.3) without affecting the existing behavior. The refactored version of `Salesman` should still be behaviorally preserved along with the original class `Employee`. To help guarantee behavior preservation of this model, the model should be behaviorally consistent when performing a refinement to expect that the evolved model be behaviorally consistent as well.

Model-Driven Refactoring In Figure 3 (a) and (b), we explain this technique, as follows:

- In this refactoring, we use the following primitive model transformations from the catalog in [7]: introduce subsignature, remove relation, and introduce relation. To apply the corresponding strategies, we introduce two subsignatures `Savings Account` and `Checking Account` with the `Bank Account` supersignature. We then remove the two original relations `(has)` and `(consist of)`, and introduce a relation `(has)` with these two subsignatures.
- This refactoring consists of introducing signature, removing relation and introducing relation primitive transformations. We first restructure the relationship between `Bank Account` and `Transaction` by removing `(credit`

to) and (debit from) relations, and adding a relation (posts). Since two types of transaction can be made (i.e., withdrawal or deposit), we add a new signature `Transaction Kind` and introduce a new relation between `Transaction` and `Transaction Kind`.

As stated in [7], applying strategies that are in accordance with laws of programming helps ensure behavior preservation as these laws provide a formal basis for program refactoring.

2.1.0.5. Differential Precondition Checking. By way of illustration, Overbey and Johnson [17] show the differences between the traditional precondition checking and the differential checking for *Pull Up Method* refactoring. For the traditional version, the method needs to be moved from subclass to its superclass, replacing all occurrences of superclass with `this`. Using preservation rule for the differential version, however, this refactoring is composed of two smaller refactoring operations: (1) *Copy Up Method* to move a method to its superclass and replace all occurrences of the superclass with `this` and (2) *Delete Overriding Duplicate* to delete the original method from the subclass using the preservation rule in [17]. The process of applying the transformation is illustrated in Figure 4.

2.1.0.6. Decomposition and Schemes. To demonstrate this technique, we use a recurring example (i.e., video rental) that is taken from Fowler's book. We discuss how a complex refactoring transformation is decomposed into simple yet behavior-preserving refactoring steps. The example requires several primitive refactorings in order to remove the long `statement` method :

- To safely split up `statement()`, the first step is to find a complex piece of code and use *Extract Method* refactoring.
- To avoid name conflict, the *Rename Field* refactoring is used to rename the variable `each` to `aRental` in the extracted code fragment.
- Since the extracted piece of code uses some information from other class, *Move Method* refactoring is used to move `amountFor()` to class `Rental`.
- To avoid name conflict, the method moved in the previous step is renamed to `getCharge()` using *Rename Method* refactoring.
- A temporary variable `thisAmount` is used to hold the result of the expression. To eliminate this temporary variables, *Replace Temp with Query* refactoring is used to prevent other parameters from being passed around when they don't have to be.
- The `frequent rental points` part of code is extracted , using *Extract Method* refactoring.
- Two temporary variables `this Amount` and `frequentRentalPoints` are replaced with `getTotalCharge` and `getTotalFrequentRenterPoints` query methods respectively by using *Replace Temp with Query*.

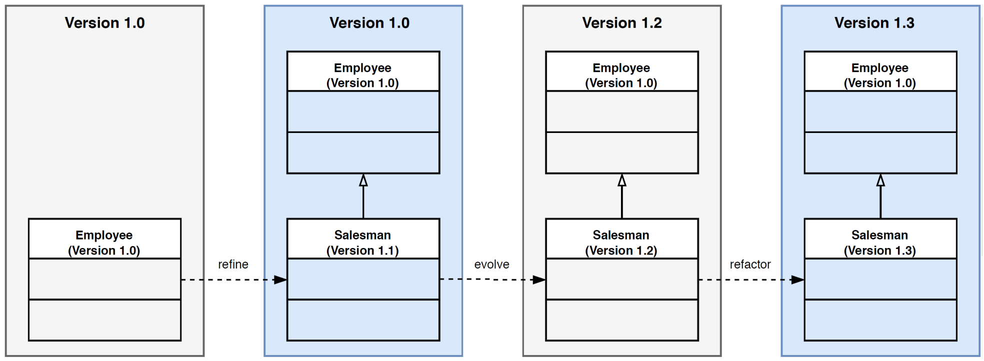

Figure 2: Model Refinement & Model Refactoring Formalism

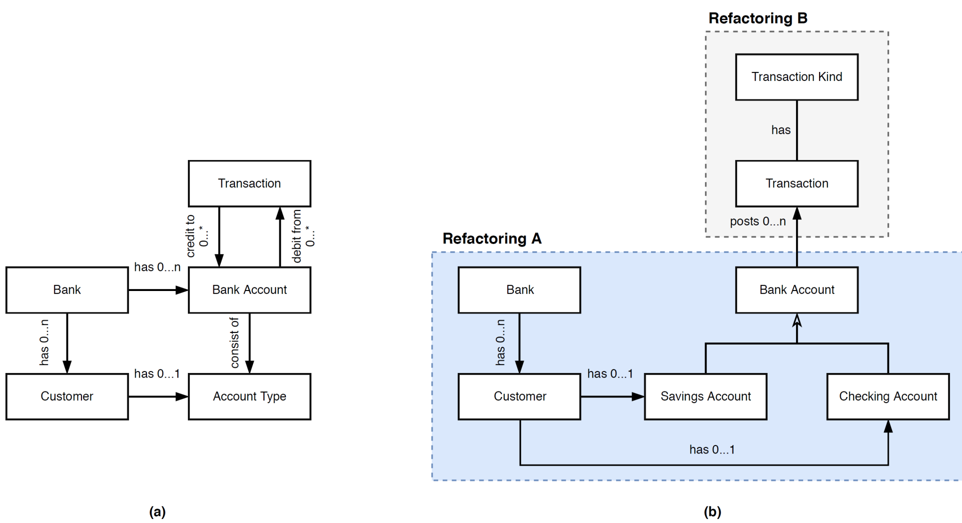

Figure 3: Bank Application Object Model

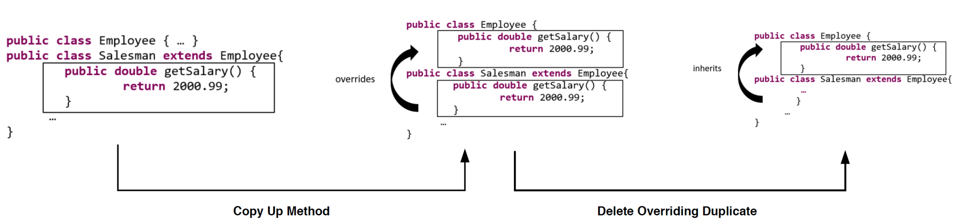

Figure 4: Differential Precondition Checking Process (Pull Up Method)

- Because Switch statement performs various actions depend on the attribute of another object, a State pattern is introduced using three refactoring operations. First, *Replace Type Code with State* refactoring is used to move type code behavior into the state pattern. Then, move switch statement to `price` class using *Move Method* refactoring. *Replace Conditional with Polymorphism* refactoring is lastly performed to eliminate switch statement.

The above sequence of primitive refactorings forms composite refactoring the safely help to eliminate complex data by proper controlling of the dependencies. The decomposition process of the complex refactorings reflects behavior-preserving transformation [26].

2.1.0.7. Overly Strong Precondition Identification. .

For an example of such an overly strong condition, reconsider Listing 1 in the running example illustrated in Section 1. Suppose we apply *Rename Method* refactoring to rename method `getFullName` to `getName`. If we apply this refactoring using Eclipse, we get the following warning message: *Problem in 'Salesman.java'. The reference to `getName` will be shadowed by a renamed declaration.* The resulting program is presented in Listing 3. After applying the transformation, the `test2` method outputs `John Smith` (Listing 3) instead of `John` (Listing 1). This transformation exposes a behavioral change after ignoring a warning message. Similarly, NetBeans applies the transformation and yields to the program in Listing 3.

By applying this refactoring using JRRT, however, the transformation preserves behavior. JRRT adds a `super` access to method `getName` inside `test2` to ensure that the resulting program correctly refactors the source program.

We notice that Eclipse rejects the transformation, and NetBeans and JRRT apply it with the conformance from SafeRefactor tool that it is behaviorally preserved. Thus, by comparing the results of Eclipse, JRRT, and NetBeans, it indicates that Eclipse has an overly strong condition because it rejects useful behavior preserving transformation.

Listing 3: Eclipse's target program after ignoring the warning message

```
public class Salesman extends Employee {
    public String ssn;
    public void setSSN(String setSNN) {
        ssn=setSNN;
    }
    public String getSSN() {
        return ssn;
    }
+   public String getName() {
+       return "John Smith";
+   }
    public double getSalary() {
        return 2000.99;
    }
    public double getSomething() {
        return super.getSalary();
    }
    public String toString() {
```

```
        return "Employee[Salary= " + getSalary()+"]";
    }
    public double yearlySalary() {
        double yearlySalary;
        yearlySalary = (getSalary() * 12);
        return yearlySalary;
    }
    public double yearlySalaryIncrease() {
        double yearlySalaryIncrease;
        yearlySalaryIncrease = (((yearlySalary() * (0.1))
        + yearlySalary()));
        return yearlySalaryIncrease;
    }
    public void displayYearlySalaryIncrease() {
        System.out.printf("Yearly Salary Increase is"+
        yearlySalaryIncrease());
    }
    public double test1() {
        return getSomething();
    }
+   public String test2() {
+       return getName();
+   }
}
```

In the following study that complements this work, Mongiovi et al. [19] propose a new technique called Disabling Preconditions (DP) to detect overly strong preconditions. The process starts with using JDolly as test inputs (Step 1). For each generated program, the refactoring engine is used to apply the transformations. In Step 2, authors collected the messages reported by the refactoring engine about the rejection of certain refactoring transformations. The next step is to manually inspect the code fragments and its related precondition for the purpose of disabling the execution of the precondition (i.e., DP technique). Step 5 involves reapplying the same transformation with a disabled precondition. After ensuring that the refactoring implementation applies the transformation and this transformation is behaviorally preserved according to SafeRefactorImpact, DP technique classifies a precondition as overly strong precondition.

2.1.0.8. Behavior Preservation Preconditions Examination. Tsantalis and Chatzigeorgiou [14] propose a methodology to preserve the behavior of the code by examining a set of preconditions when applying *Move Method* refactoring. These preconditions should be satisfied in order to avoid behavioral changes. Tsantalis and Chatzigeorgiou [14] formally define a set of auxiliary functions that describe behavior preservation preconditions as follows:

- A class should not inherit a method having a matching signature with the moved method. This action will lead the inherited method to override causing behavioral changes of the target class and its derived one. The moved method needs to be renamed to resolve the issue.

- When moving a method, the method should not override an inherited method. The original method should be kept as delegate to the moved method.

- When moving a method, the method should have a valid reference to its target class. The moved method can have a reference via its parameters or fields in the original class.

- When moving a method, the method should not be synchronized. Moving the synchronized method might cause concurrency issues to the original class's objects.

2.2. Automated Analyses

2.2.0.1. Refactoring Safety Tools.

SafeRefactor As a concrete example of how SafeRefactor detects behavioral change, reconsider refactoring performed in the running example of Section 1. Suppose we apply the *PullUpMethod* refactoring to move `getSomething()` from class `Salesman` to `Employee`. This method contains a reference to `Employee.getSaraly()` using `super` keyword. As depicted in Figure 5, SafeRefactor prevents this transformation because a behavioral change will introduced (i.e., method `test1` yields 2000.90 instead of 1000.33). Figure 5 also shows that 264 out of 326 units tests fail as the target program does not have the same behavior of the source program. Listing 4 presents one of the test cases (i.e., `test48()`) generated by SafeRefactor tool.

Listing 4: Test suite of the program presented in Listing 1

```
public void test48() throws Throwable {
   if (debug)
   System.out.printf("%nRandoopTest0.test48");
   Salesman var0 = new Salesman();
   double var1 = var0.test1();
   double var2 = var0.test1();
   java.lang.String var3 = var0.test2();
   java.lang.String var4 = var0.test2();
   java.lang.String var5 = var0.toString();
   java.lang.String var6 = var0.test2();
   var0.setSSN("Employee[MonthlySalary= 2000.99]");
   double var9 = var0.getSomething();
   double var10 = var0.yearlySalaryIncrease();
   java.lang.String var11 = var0.getSSN();
   java.lang.String var12 = var0.getFullName();
   double var13 = var0.yearlySalary();

   // Regression assertion (captures the current
   behavior of the code)
   assertTrue(var1 == 1000.33d);

   // Regression assertion (captures the current
   behavior of the code)
   assertTrue(var2 == 1000.33d);

   // Regression assertion (captures the current
   behavior of the code)
   assertTrue("'" + var3 + "' != '" + "John"+ "'",
   var3.equals("John"));
```

```
    // Regression assertion (captures the current
    behavior of the code)
    assertTrue("'" + var4 + "' != '" + "John"+ "'",
    var4.equals("John"));

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue("'" + var5 + "' != '" + "Employee
    [Salary= 2000.99]"+ "'", var5.equals("Employee
    [Salary= 2000.99]"));

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue("'" + var6 + "' != '" + "John"+ "'",
    var6.equals("John"));

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue(var9 == 1000.33d);

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue(var10 == 26413.068d);

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue("'" + var11 + "' != '" + "Employee
    [Salary= 2000.99]"+ "'", var11.equals("Employee
    [Salary=    2000.99]"));

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue("'" + var12 + "' != '" + "John Smith"+ "'",
    var12.equals("John Smith"));

    // Regression assertion (captures the current
    behavior of the code)
    assertTrue(var13 == 24011.88d);

  }
```

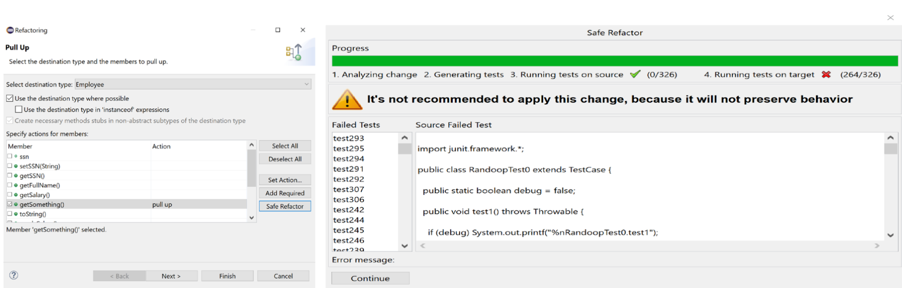

Figure 5: SafeRefactor Tool Output

SafeRefactorImpact The SafeRefactor tool has been extended, and includes AspectJ support [11], uses change impact analyzer called SAFIRA, and generates a test suite only for the methods impacted by the transformation [24]. SafeRefactor was renamed SafeRefactorImpact in [24]. This tool works by: (1) comparing the original and modified programs to identify entities (methods) impacted by the change, (2) performing a change impact analysis technique for the impacted methods in both program versions identifying methods that can be behaviourally changed after the transformation, (3) generating a test suite for the common methods identified in the previous step, (4) executing the test suite before and after the transformation, and (5) evaluating the results of the transformation to determine whether the transformation is behavior preserving. The main difference between SafeRefactor and SafeRefactorImpact tool is provided in Table 4.

Mongiovi et al. compare these tools in [24] with respect to several criteria: program correctness, performance, number of methods considered for test generation, change coverage, and relevant tests generated. Their findings show that the extended tool generates better results.

An example of a method that is not impacted by the change was already presented in Listing 4. All methods , except the methods `test1` and `getSalary`, do not expose any behavioral change since they are not relevant to test the aforementioned transformation. However, running irrelevant test cases in a large program can be time consuming. SafeRefactorImpact allows users to test only methods impacted by the transformation.

Table 4: Refactoring Safety Tools Comparison.

Tool	**SafeRefactor**	**SafeRefactorImpact**
Technology	OOP	OOP & AOP
Methods Detected	common methods	methods impacted by transformation
Test Cases Generated	relevant & non-relevant test cases	relevant test cases

Refactoring Browser In order to preserve the behavior of the program, each refactoring is associated with a reused set of preconditions that must be checked by the compilation framework in VisualWorks. For instance, to successfully implement *Add Method* refactoring, the method name should not conflict with a method defined in the class.

2.2.0.2. Commit Message Analysis. One of the approaches to analyze refactoring activity on software repositories is by analyzing commit messages. Ratzinger et al. propose this simple and fast approach to detect refactoring activity between a pair of program versions to determine whether a transformation is behavior preserving. They identified refactorings based on a set of keywords existing in the commit message. In particular, they focus on the following terms in their search approach: *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename, and move.*

Few commit messages containing some of these terms (i.e., *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename, and move*) are extracted from the Hadoop[1] project, as illustrated in the following comments:

> "*1. HADOOP-9805. Refactor RawLocalFileSystem rename for improved testability. Contributed by Jean-Pierre Matsumoto.*"

> "*2. HDFS-7743. Code cleanup of BlockInfo and rename BlockInfo to BlockInfoContiguous. Contributed by Jing Zhao.*"

References

[1] D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Theory and Practice of Object systems 3 (4) (1997) 253–263.

[2] T. Mens, N. Van Eetvelde, D. Janssens, S. Demeyer, Formalising refactorings with graph transformations, 2003, p. 69.

[3] F. Tip, A. Kiezun, D. Bäumer, Refactoring for generalization using type constraints, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 13–26. doi:10.1145/949305.949308.
URL http://doi.acm.org/10.1145/949305.949308

[4] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, B. De Sutter, Refactoring using type constraints, ACM Transactions on Programming Languages and Systems (TOPLAS) 33 (3) (2011) 1–47.

[5] A. Garrido, J. Meseguer, Formal specification and verification of java refactorings, in: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 165–174. doi:10.1109/SCAM.2006.16.

[6] R. Van Der Straeten, V. Jonckers, T. Mens, A formal approach to model refactoring and model refinement, Software & Systems Modeling 6 (2) (2007) 139–162. doi:10.1007/s10270-006-0025-9.
URL https://doi.org/10.1007/s10270-006-0025-9

[7] T. Massoni, R. Gheyi, P. Borba, Formal model-driven program refactoring, in: Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering, FASE'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 362–376.
URL http://dl.acm.org/citation.cfm?id=1792838.1792873

[8] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, M. Cornélio, Saferefactor-tool for checking refactoring safety.

[9] G. Soares, R. Gheyi, T. Massoni, M. Cornélio, D. Cavalcanti, Generating unit tests for checking refactoring safety, in: Brazilian Symposium on Programming Languages, 2009, pp. 159–172.

[10] G. Soares, R. Gheyi, D. Serey, T. Massoni, Making program refactoring safer, IEEE Software 27 (4) (2010) 52–57. doi:10.1109/MS.2010.63.

[1] https://github.com/apache/hadoop

[11] G. Soares, D. Cavalcanti, R. Gheyi, Making aspect-oriented refactoring safer, in: Proceedings of the 15th Brazilian Symposium on Programming Languages, SBLP, Vol. 11, 2011, pp. 91–105.

[12] N. Ubayashi, J. Piao, S. Shinotsuka, T. Tamai, Contract-based verification for aspect-oriented refactoring, in: 2008 1st International Conference on Software Testing, Verification, and Validation, IEEE, 2008, pp. 180–189.

[13] M. Schäfer, T. Ekman, O. De Moor, Sound and extensible renaming for java, in: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, 2008, pp. 277–294.

[14] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367. `doi:10.1109/TSE.2009.1`.

[15] M. Schäfer, O. De Moor, Specifying and implementing refactorings, in: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2010, pp. 286–301.

[16] N. Tsantalis, A. Chatzigeorgiou, Identification of refactoring opportunities introducing polymorphism, Journal of Systems and Software 83 (3) (2010) 391–404.

[17] J. L. Overbey, R. E. Johnson, Differential precondition checking: A lightweight, reusable analysis for refactoring tools, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 303–312. `doi:10.1109/ASE.2011.6100067`.

[18] G. Soares, M. Mongiovi, R. Gheyi, Identifying overly strong conditions in refactoring implementations, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 173–182. `doi:10.1109/ICSM.2011.6080784`.

[19] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, L. Teixeira, Detecting overly strong preconditions in refactoring engines, IEEE Transactions on Software Engineering 44 (5) (2018) 429–452. `doi:10.1109/TSE.2017.2693982`.

[20] M. De Jonge, E. Visser, A language generic solution for name binding preservation in refactorings, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, 2012, pp. 1–8.

[21] C. Noguera, A. Kellens, C. De Roover, V. Jonckers, Refactoring in the presence of annotations, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 337–346.

[22] A. Thies, E. Bodden, Refaflex: Safer refactorings for reflective java programs, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 1–11.

[23] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, Journal of Systems and Software 86 (4) (2013) 1006 – 1022, sI : Software Engineering in Brazil: Retrospective and Prospective Views. `doi:https://doi.org/10.1016/j.jss.2012.10.040`.
URL `http://www.sciencedirect.com/science/article/pii/S016412121200297X`

[24] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, P. Borba, Making refactoring safer through impact analysis, Sci. Comput. Program. 93 (2014) 39–64. `doi:10.1016/j.scico.2013.11.001`.
URL `http://dx.doi.org/10.1016/j.scico.2013.11.001`

[25] M. Najafi, H. Haghighi, T. Z. Nasab, A set of refactoring rules for uml-b specifications, Computing and Informatics 35 (2) (2016) 411–440.

[26] D. Horpácsi, J. Köszegi, Z. Horváth, Trustworthy refactoring via decomposition and schemes: A complex case study, in: VPT@ETAPS, 2017.

[27] Z. Chen, H.-F. Guo, M. Song, Improving regression test efficiency with an awareness of refactoring changes, Information and Software Technology 103 (2018) 174–187.

[28] D. Insa, S. Pérez, J. Silva, S. Tamarit, Behaviour preservation across code versions in erlang, Scientific Programming 2018.

[29] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139. doi:10.1109/TSE.2004.1265817.