Manutenção e Evolução de Software

Marco Túlio Valente

mtov@dcc.ufmg.br

DCC - UFMG

Making Program Refactoring Safer

Gustavo Soares, Rohit Gheyi, Dalton Serey, Tiago Massoni (UFCG)

IEEE Software, 2010

Introduction

- Refactoring changes a software system in such a way that:
  - It doesn’t alter the code’s external behavior
  - But improves its internal structure

- Developers perform refactoring:
  - Either manually, which is error-prone and time consuming
  - Or with the help of IDEs (Eclipse, Netbeans, JBuilder, IntelliJ)

- **Problem**:
  - Each refactoring can contain several preconditions
  - Refactoring tools typically don’t implement all preconditions because formally establishing them is nontrivial
  - Therefore, <u>refactoring tools often allow wrong transformations</u> to be applied without any warning

Example

```
public class A {
    public int k(long i) {
      return 10;
    }
}
public class B extends A {
    public int k(int i) {
      return 20;
    }
    public int test() {
      return new A().k(2);
    }
}
```
(a)

```
public class A {
    public int k(long i) {
    return 10;
  }
  public int k(int i) {
    return 20;
  }
}
public class B extends A {
  public int test() {
    return new A().k(2);
  }
}
```
(b)

Figure 1. Pull-up method refactoring enables overloading: (a) source program and (b) target program. Eclipse doesn't detect a behavioral change in the test method.

More Examples (1)

- Moving class *B* to another package with refactoring factoring will produce a compilation error

```
package a;
class A {
  B b;
}

package a;
class B {}
```

- (only IDEA issues a warning that B will become inaccessible).

- Examples from: F. Steimann, A. Thies: From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. ECOOP 2009

More Examples (2)

- Moving class *B* to another package will change the meaning of the program:
  - Rather than the method *n* in *A* calling *m(String)* in *B* as before the change, *m(Object)* gets invoked instead

```
package a;
public class A {
  void n() { (new B()).m("abc"); }
}

package a;
public class B {
  public void m(Object o) {…}
  void m(String s) {…}
}
```

More Examples (3)

- Moving B to another package changes the behavior because:
  - *m(String)* in *A* changes its status from being overridden in *B* to not being overridden
  - so that calling *m(String)* on a receiver of static type *A* is no longer dispatched to the implementation in *B*.

```
package a;
public class A {
  void m(String s) {…}
  void n() { ((A) new B()).m("abc"); }
}
```

```
package a;
public class B extends A {
  void m(String s) {…}
}
```

In ECLIPSE and NETBEANS, this change of meaning goes unnoticed, IDEA notes that class *A* contains a reference to class *B*, but this is not indicative of the problem.

Current Practices

- The current practice to avoid behavioral changes in refactorings relies on solid tests
- However, test suites often don’t catch behavioral changes during transformations. The tools might refactor them as well (for instance, with the rename method)
- Clearly, this scenario is undesirable because the refactoring tool can change the test suite meaning.

SafeRefactor

- We describe and evaluate SafeRefactor, a tool for checking refactoring safety in sequential Java programs using Eclipse IDE.

- For each transformation, it analyzes the transformation and <u>generates a test suite for detecting behavioral changes</u>.

- SafeRefactor is an Eclipse plugin that:
  - Receives a source code and a refactoring to apply (input)
  - Reports whether it’s safe to apply the transformation (output).

SafeRefactor

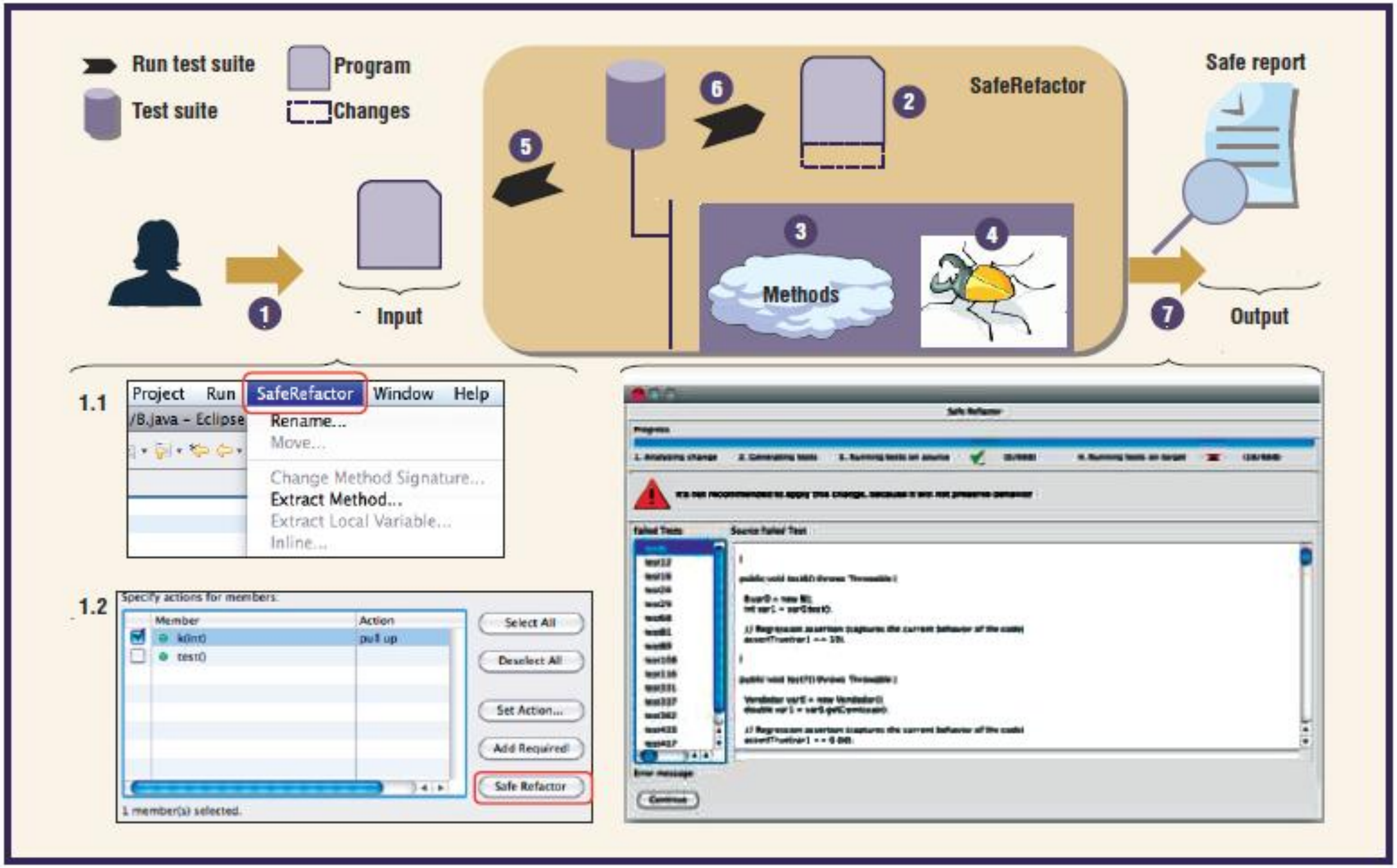

Figure 2. SafeRefactor. Step 1: The developer (1.1) selects a refactoring in the menu to apply and (1.2) clicks on the SafeRefactor button. Step 2: The tool generates the target program using the Eclipse refactoring API and then (step 3) identifies common methods in the source and target programs. Step 4: The tool generates unit tests using the modified Randoop and then (step 5) runs the test suite on the source program. Step 6: The tool runs the test suite on the target program and then (step 7) shows the report to developer. If it finds a behavior change, the developer can see some unit tests that fail.

SafeRefactor

1. First, the developer selects the refactoring to be applied on the source program (step 1.1) and then uses SafeRefactor (step 1.2) to check the refactoring safety (steps 2-7)

2. SafeRefactor generates a target program based on the desired transformation using the Eclipse refactoring API (step 2)

3. In step 3, static analysis automatically identifies <u>methods common in both source and target programs</u>

4. Step 4 <u>generates unit tests</u> for methods identified in step 3

5. In step 5, the plugin runs the generated test suite on the source program and on the target program (step 6)

SafeRefactor

6. If a test passes in one of the programs and fails in the other, the plugin detects a behavioral change and reports it to the user (step 7).

7. Otherwise, the developer can have more confidence that the transformation doesn’t introduce behavioral changes.

Step 3

- The static analysis goal (step 3) is to identify methods in common that have exactly the same modifier, return type, qualified name, parameter types, and exceptions thrown in source and target programs.

- For example, Figure 1 has *A.k(long)* and *B.test()* in common.

- After identifying a set of useful methods, the plugin uses Randoop to generate unit tests (step 4) for classes within a time limit.

Randoop

- From: http://code.google.com/p/randoop/

- Randoop is an automatic unit test generator for Java. It automatically creates unit tests for your classes, in JUnit format.

- Randoop has found serious errors in widely-deployed software, such as JDK 1.5

- A test suite that is generated by Randoop typically consists of:
  - A sequence of method calls that create and mutate objects with random values
  - Plus an assertion about the result of a final method call

Randoop

- Slides presented at ICSE 2007:

Technique input/output

- Input:
  - classes under test
  - time limit
  - set of contracts
    - Method contracts (e.g. "o.hashCode() throws no exception")
    - Object invariants (e.g. "o.equals(o) == true")
- Output: contract-violating test cases. Example:

no contracts violated up to last method call

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

fails when executed

Randoop

- This test case actually reveals two errors:
  - one in equals
  - and one in the *TreeSet(Collection)* constructor, which failed to throw *ClassCastException* as required by its specification.

Back to SafeRefactor

- The whole process to finish Figure 1 takes less than 8 seconds on a dual-processor 2.2-GHz Dell Vostro 1400 laptop
  - Generates 154 unit tests (151 of them failed).

- In this case, SafeRefactor reports that the refactoring shouldn't be applied.
- In other situations, SafeRefactor can report compilation errors that refactoring tools might introduce.

- If SafeRefactor doesn't find a behavior change or compilation error, it reports that the transformation is most likely sound.

Evaluation – First Experiment

- The first category consists of the refactorings proposed in the literature applied to Java applications using tools or manual steps.
- Developers consider all of them “behavior preserving.”
- We then used SafeRefactor to evaluate whether the transformation preserves the observable behavior.
- We used a command-line interface provided by SafeRefactor that receives three parameters:
  - Source and target program paths
  - And timeout to generate tests.

First Experiment

Table I

Summary of SafeRefactor evaluation in refactoring real applications

Subject	Program	KLOC	Refactoring	Tests	Error	Total time (sec)	Result
1	JHotDraw	23	Extract exception handler	2,245	273	148	Behavior change
2	CheckStylePlugin	20	Extract exception handler	5,864	0	235	-
3	JUnit	3	Infer generic type	1,127	0	99	-
4	Vpoker	4	Infer generic type	466	0	109	-
5	ANTLR	32	Infer generic type	-	-	2	Compilation error
6	Xtc	100	Infer generic type	-	-	4	Compilation error
7	TextEditor	15	Replace deprecated code	16,009	0	107	-

1st Experiment – Example of Failure

- Developers refactored JHotDraw to avoid code duplication with identical exception handlers in different parts of a system.

- Eight programmers working in pairs performed the change:
  - They extracted the code inside the *try*, *catch*, and *finally* blocks to methods in specific classes that handle exceptions.
  - They used refactoring tools, pair review, and unit tests to assure that the behavior was preserved.

1st Experiment – Example of Failure

- Some classes that implement *Serializable* were refactored.
- Developers changed the *clone* method and introduced the *handler* attribute to handle exceptions.
- However, they forgot to *serialize* this new attribute.
- Thus, when the method *clone* tried to serialize the object, an exception was thrown, meaning refactored method *clone* has a different behavior.

Second Experiment

- The second category includes nonbehavior-preserving transformations applied by refactoring tools.
- We used SafeRefactor to evaluate whether SafeRefactor detected the behavior changes.
- In the second category, SafeRefactor identified all but one behavior change that uses standard output

Second Experiment

Table 2

Summary of SafeRefactor evaluation in the catalog of defective refactorings

Subject	Refactoring	Bug description	Tests	Error	Result
8	Push down method	Incorrect handling of super accesses	488	0	-
9	Rename class	Renaming a class leads to undiagnosed shadowing	102	95	Behavior change
10	Rename variable	Renaming a local variable leads to shadowing by field	494	492	Behavior change
11	Rename method	Renaming a method leads to shadowing of statically imported method	93	91	Behavior change
12	Encapsulate field	No check for overriding problems	474	464	Behavior change
13	Extract method	Incorrect dataflow analysis	558	554	Behavior change
14	Push down method	Incorrect handling of super accesses	486	404	Behavior change
15	Push down method	Incorrect handling of field accesses	78	75	Behavior change
16	Push down method	Pushing down a method enables overloading	101	99	Behavior change
17	Move class	Move a class to another package disables overriding	101	99	Behavior change
18	Move class	Move a class to another package disables overloading	79	77	Behavior change
19	Change method signature	Increasing method visibility enables overriding	214	40	Behavior change
20	Change method signature	Increasing method visibility enables overloading	79	76	Behavior change
21	Change method signature	Increase method visibility enables overriding to another package	121	40	Behavior change
22	Pull up method	Pulling up a method enables overloading	101	99	Behavior change
23	Pull up method	Pulling up a method enables overriding	170	88	Behavior change
24	Pull up method	Pulling up a method to a class in another package enables overriding	167	163	Behavior change

Conclusions

- Here, we evaluated our technique with empirical studies.
- In the near future, we intend to create a plugin for other IDEs, such as Netbeans.
- In addition to using it in more real case studies, we plan to test refactoring tools to automatically find nonbehavior-preserving transformations

Discussão ?

Refactoring Sequential Java Code for Concurrency via Concurrent Libraries

Danny Dig, John Marrero, Michael Ernst (MIT)

ICSE 2009

Introduction

What is the problem that the paper solves?
Why is this problem important?

Motivation and Relevance

- The hardware industry’s shift to multicore processors demands that programmers exploit parallelism in their programs, if they want to reap the same performance benefits as in the past.

- **Problem (general definition)**:
  - Most desktop <u>programs were not designed to be concurrent</u>
  - So programmers have to refactor existing sequential programs for concurrency.

- **Key idea:**
  - It is easier to retrofit concurrency than to rewrite, and retrofitting is often possible.
  - (Wikipedia: Retrofitting refers to the addition of new technology or features to older systems.)

Context and Focus

- Java 5’s java.util.concurrent (j.u.c.) package supports writing concurrent programs.

- Its *Atomic** classes offer thread-safe, lock-free programming over single variables.
- Its thread-safe abstract data types (e.g., *ConcurrentHashMap*) are optimized for scalability

Particular Problem

- To benefit from Java's concurrent utilities, the Java programmer needs to refactor existing code.

- This is tedious because it requires changing many lines of code.
  - For example, the developers of six widely used open-source projects changed 1019 lines when converting to use *AtomicInteger* and *ConcurrentHashMap*.

- Second, manual refactoring is error-prone because the programmer can choose the wrong APIs
  - In the above-mentioned projects, the programmers four times mistakenly used *getAndIncrement* API methods instead of *incrementAndGet*, which can result in off-by-one values.

Particular Problem

- Third, manual refactoring is omission-prone because the programmer can miss opportunities to use the new, more efficient API methods.
  - In the same projects, programmers missed 41 such opportunities.

How was the problem solved?

Solution

- This paper presents our approach for incrementally retrofitting parallelism through a series of behavior preserving program transformations, namely refactorings.

- Our tool, CONCURRENCER, enables Java programmers to refactor their sequential programs to use j.u.c. utilities:
  - The programmer selects shared data and a target refactoring
  - And CONCURRENCER analyzes all accesses to the shared data and applies the transformation

Our Solution

- Currently, CONCURRENCER supports three refactorings:
  - CONVERT INT TO ATOMICINTEGER
  - CONVERT HASHMAP TO CONCURRENTHASHMAP
  - CONVERT RECURSION TO FORKJOINTASK.

- Although <u>these are not all the refactorings</u> that one needs for parallelization, the first two refactorings are among the most commonly used in practice, as evidenced by our study [3] of how open-source developers parallelized five projects.
  - [3] “How do programs become more concurrent? A story of program transformations” by Danny Dig, John Marrero, and Michael D. Ernst. In Proceedings of the 4th International Workshop on Multicore Software Engineering, 2011.
  - (wikipedia: recall is a measure of completeness => their solution fails on completeness; but it is sound)

Refactoring #1

- The first refactoring, CONVERT INT TO ATOMICINTEGER, enables a programmer to convert an *int* field to an *AtomicInteger*, a utility class that encapsulates an *int* value.

- The encapsulated field can be safely accessed from multiple threads, without requiring any synchronization code.

- Our refactoring replaces all field accesses with calls to AtomicInteger's thread-safe APIs.
- For example, it replaces expression *f = f + 3* with *f.addAndGet(3)* which executes atomically.

Refactoring #2

- The second refactoring, CONVERT HASHMAP TO CONCURRENTHASHMAP, enables a programmer to convert a *HashMap* field to *ConcurrentHashMap*, a thread-safe, highly scalable implementation for hash maps.

- Our refactoring replaces map updates with calls to the APIs provided by *ConcurrentHashMap*.
- For example, a common update operation is:
  - Check whether a map contains a certain key
  - if not present, create the value object
  - place the value in the map.
- CONCURRENCER replaces such an updating pattern with a call to *ConcurrentHashMap*'s *putIfAbsent* which atomically executes the update, without locking the entire map

Refactoring #3

- The third refactoring, CONVERT RECURSION TO FORKJOINTASK, enables a programmer to convert a sequential divide-and-conquer algorithm to a parallel algorithm.

- The parallel algorithm solves the subproblems in parallel using the *Fork-JoinTask* framework.

- Using the skeleton of the sequential algorithm, CONCURRENCER extracts the sequential computation into tasks that run in parallel and dispatches these tasks to the *ForkJoinTask* framework

Rationale

- Typically a user would first make a program thread-safe
  - i.e., the program has the same semantics as the sequential program even when executed under multiple threads
- And then make the program run concurrently under multiple threads.

- CONCURRENCER supports both kinds of refactorings.
  - The first two refactorings are “enabling transformations” that make a program thread-safe.
  - The third refactoring makes a sequential program run concurrently.

What are the main difficulties in solving the problem?

Main difficulties

- The transformations performed by these refactorings require matching certain code patterns which can span several non-adjacent program statements, and they require program analysis which uses data-flow information.
- Such transformations can not be safely executed by find-and-replace.

How good is the solution? What experiments are made to show the value of the solution?

Evaluation

- We used CONCURRENCER to refactor the same code that the open-source developers of 6 popular projects converted to *AtomicInteger* and *ConcurrentHashMap*.

- By comparing the manually vs. automatically refactored output, we found that CONCURRENCER applied all the transformations that the developers applied.

- Even more, CONCURRENCER avoided the errors which the open-source developer committed
- And CONCURRENCER identified and applied some transformations that the open-source developers omitted.

Evaluation

- We also used CONCURRENCER to parallelize 6 divide-and-conquer algorithms.
- The parallelized algorithms perform well and exhibit good speedup.

- These experiences show that CONCURRENCER is useful!