Manutenção e Evolução de Software

Marco Túlio Valente

mtov@dcc.ufmg.br

DCC - UFMG
Software Reengineering
Reengineering

- It may be hard to tell the difference between software “reengineering” and software “maintenance”.

- Most people would probably consider that “maintenance” is routine whereas “reengineering” is a drastic, major effort to recast a system.
Forward, reverse and reengineering

Figure 1.1: Forward, reverse and reengineering
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, refactoring tools often allow wrong transformations to be applied without any warning
Example

```java
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)

```java
public class A {
    public int k(long i) {
        return 10;
    }

    public int k(int i) {
        return 20;
    }

    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 will produce a compilation error

```java
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 \texttt{n} in A calling \texttt{m(String)} in B as before the change, \texttt{m(Object)} gets invoked instead

```java
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:
  - \textit{m(String)} in A changes its status from being overridden in B to not being overridden
  - so that calling \textit{m(String)} on a receiver of static type A is no longer dispatched to the implementation in B.

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

```java
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.
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.
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 programs were not designed to be concurrent
  - 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, CONCURRENCE, enables Java programmers to refactor their sequential programs to use j.u.c. utilities:
  - The programmer selects shared data and a target refactoring
  - And CONCURRENCE 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 these are not all the refactorings 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.
  2. (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 FORKJOIN TASK, 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-Join Task* 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 *ForkJoin Task* 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.

- CONCURRENCE 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!