Automated Composition of Refactorings

Implementing and evaluating a search-based Extract and Move Method refactoring

Erlend Kristiansen, 2014

UNIVERSITY
OF OSLO

Refactoring, as defined in the literature

> *Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. [Fow99, p. 53]*

An alternative definition of refactoring

Definition

A *refactoring* is a transformation done to a program without altering its external behavior.

Primitive and composite refactorings

Definition

A *primitive refactoring* is a refactoring that cannot be expressed in terms of other refactorings.

Definition

A *composite refactoring* is a refactoring that can be expressed in terms of two or more other refactorings.

Motivation

Bad

```
class C {
  A a; B b; X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
}
class X {
  Y y;
}
class Y {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

Bad

```
class C {
  A a; B b; X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
}
class X {
  Y y;
}
class Y {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

Good

```
class C {
  A a; B b; X x;
  void method() {
    x.fooBar();
  }
}
class X {
  Y y;
  void fooBar() {
    y.foo();
    y.bar();
  }
}
class Y {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

- Get rid of long navigation paths.
- Move operations closer to the data they manipulate.
- Reduce coupling.
- Increase maintainability.

The primitive refactorings

The Extract Method refactoring

Extract a fragment of code into a new method.

```
class C {
  A a; B b; X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
}
```

The Extract Method refactoring

Extract a fragment of code into a new method.

```
class C {
  A a; B b; X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
}
```

```
class C {
  A a; B b; X x;
  void method() {
    fooBar();
  }
  void fooBar() {
    x.y.foo();
    x.y.bar();
  }
}
```

The Move Method refactoring

Move a method from one class to another.

```
class C {
  A a; B b; X x;
  void method() {
    fooBar();
  }
  void fooBar() {
    x.y.foo();
    x.y.bar();
  }
}
class X {
  Y y;
}
```

The Move Method refactoring

Move a method from one class to another.

```
class C {
  A a; B b; X x;
  void method() {
    fooBar();
  }
  void fooBar() {
    x.y.foo();
    x.y.bar();
  }
}
class X {
  Y y;
}
```

```
class C {
  A a; B b; X x;
  void method() {
    x.fooBar();
  }
}
class X {
  Y y;
  void fooBar() {
    y.foo();
    y.bar();
  }
}
```

The Extract and Move Method refactoring

Before

```
class C {
  A a; B b; X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
}
class X {
  Y y;
}
class Y {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

Before

```
class C {
  A a; B b; X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
}
class X {
  Y y;
}
class Y {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

After

```
class C {
  A a; B b; X x;
  void method() {
    x.fooBar();
  }
}
class X {
  Y y;
  void fooBar() {
    y.foo();
    y.bar();
  }
}
class Y {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

- Composed of *Extract Method* and *Move Method*.
- Conceptually, one “atomic” operation.
- Implemented as an Eclipse plugin.
  - The primitive refactorings are supplied by the Eclipse JDT.
  - The composition work had to be done by us.
  - Not seamless (find the extracted method, move target etc.).

Research questions

Main research question:

> *Is it possible to automate the analysis and execution of the Extract and Move Method refactoring, and do so for all of the code of a larger project?*

Secondary questions:

- Can we do this efficiently?
- Can we perform changes safely?
- Can we improve the quality of source code?
- How can the automation of the refactoring be helpful?

Automating the refactoring

For any given method: We want to find the best candidate for the *Extract and Move Method* refactoring, if any exist.

```
void method() {
  statement_1;
  statement_2;
  statement_3;
  statement_4;
  statement_5;
}
```

?

```
class C {
  A a; B b; boolean bool;
  void method(int val) {
    if (bool) {
      a.foo();
      a = new A();
      a.bar();
    }
    a.foo();
    a.bar();
    switch (val) {
    case 1:
      b.a.foo();
      b.a.bar();
      break;
    default:
      a.foo();
    }
  }
}
```

move target

text selection

```
class C {
  A a; B b; boolean bool;
  void method(int val) {
    if (bool) {
      a.foo();
      a = new A();
      a.bar();
    }
    a.foo();
    a.bar();
    switch (val) {
    case 1:
      b.a.foo();
      b.a.bar();
      break;
    default:
      a.foo();
    }
  }
}
```

move target

text selection

A **candidate** consists of a *text selection* and a *move target*.

A **valid text selection** is a text selection that contains all of one or more consecutive program statements. It is the input to the *Extract Method* refactoring.

A **move target** is a variable (local or field), whose type is the destination class in the *Move Method* refactoring.

Searching

Usually, search-based refactoring is based on metrics.

- Refactor a lot.
- Choose the best candidate based on measurements.

Our refactoring is based on heuristics.

- Up-front analysis.
- A set of assumptions defining what is considered the best candidate.
- No need to actually perform changes (before deciding).
- Search through all valid selections to find the best candidate.

Choosing a refactoring candidate

- Search through all selections to find the possible candidates.
- Find the best move target for all the candidates.
- Choose the best among the possible candidates.
- Based on the lengths of the navigation paths and the occurrence counts.

Demonstration

Case studies

Case studies performed on the `org.eclipse.jdt.ui` and `no.uio.ifi.refaktor` projects. The resulting code was analyzed with SonarQube.

The Eclipse JDT UI project:

- Over 300,000 lines of code.
- 2,552 methods out of 27,667 methods chosen to be refactored.
- Approx. 100 minutes.

The case studies are inconclusive

- Measurements show some deterioration regarding coupling.
- All improvement not measured, only strict coupling between classes.
- Examples exist where coupling is improved.
- More examples exist where dependencies are introduced.

Demonstration continued

Conclusions

- Automation is possible.
- Efficient enough for some kinds of use.
- Difficult not to break source code.
- Code is not improved in most cases.
- Not particularly useful in its current state.

Future work

- Complete analysis.
- Make refactoring safer.
- Improve heuristics to avoid introducing new dependencies.

References

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Reading, MA: Addison-Wesley, 1999.