Refactoring: To the Rubicon… and Beyond!

Roly Perera
Ergnosis Ltd
Third Floor, 14 King Square
Bristol, BS2 8JJ, UK
+44 (0)117 924 8915
roly.perera@ergnosis.com

ABSTRACT
We demonstrate a new approach to refactoring which involves the decomposition of familiar high-level refactorings such as *Extract method* into their components. By understanding all refactorings as the introduction or elimination of degrees of freedom we show how a large proportion of programming edits are in fact micro-refactorings, and gain an insight into how tools that support these micro-refactorings could have a dramatic impact on developer productivity.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement – Restructuring, reverse engineering, and reengineering.

General Terms
Design, Languages

Keywords
Refactoring

1. BACKGROUND
A significant portion of a software developer's time is spent *refactoring*: preparing for the insertion of new functionality, and consolidating existing functionality, without changing the current behaviour of the system. Without this ongoing maintenance effort, entropy rapidly takes hold and delivering further features or bug-fixes becomes difficult.

Interest in software refactoring and tools for assisting with this activity has been growing steadily over the last decade, thanks to the influential efforts of Roberts and Brant[3], Opdyke[2], Fowler[1] and others. However the refactorings discussed to date, such as *Extract method*, are in desperate need of decomposition into more primitive, but more widely applicable refactorings, such as *Push code into method*. By identifying a kernel of micro-refactoring primitives we gain new insights into the opportunities for tools to change the way developers work.

2. A MICRO-REFACTORING KERNEL
The following Java examples show how a macro-refactoring like *Extract method* can be decomposed into its parts. Refactoring in a language like Java is a hard problem, thanks to the ubiquitous side-effect. Although most refactoring tools for such languages are in a convenient state of denial, requiring the user to be on the lookout for unintended changes in behaviour due to the reordering, duplication or elimination of side-effects, we believe this simply represents the relative immaturity of refactoring tools compared to other transformation tools that need to preserve behaviour, such as optimizing compilers. For the purposes of our current demonstration we will reluctantly join the denial camp.

Our first example is based on Fowler's *Hide delegate* refactoring [1]. As a manual activity, this is bread and butter to any experienced OO programmer. Yet no tool supports this important refactoring, not because it is hard to implement, but because it cannot be applied as a single transformation without the explicit selection of an actor for each role: client, server and delegate. If however *Hide delegate* is broken down into constituent operations, each of which can be applied directly without requiring complicated decisions to be made in advance, then the developer can achieve *Hide delegate* without having to specify all arguments up front. In effect, she *composes* an instance of *Hide delegate* by stepwise interaction with her source code, obtaining confidence-building feedback at each step.

The following code is adapted from Fowler, p. 158-9:

```
class Person {
    private Department _dept;

    public Department getDepartment () {
        return _dept;
    }

    public void setDepartment (Department dept) {
        _dept = dept;
    }
}

class Department {
    private Person _manager;
}
    public Department (Person manager) {
        _manager = manager;
    }

    public Person get Manager () {
        return _manager;
    }
}
```

To obtain a person's manager, the following client code is used:

```
manager = john.getDepartment().getManager()
```

If client code becomes riddled with traversals of the path from person to manager, then the case for centralizing this traversal in one place becomes fairly strong. The first step is a natural one: select the code which navigates the path:

```
manager = john.getDepartment().getManager()
```


OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

and apply *Extract method*. This creates a new method, in the client class, which is static as it uses no members of its host class:

```
class Client {
    public static Manager (final Person person) {
        return person.getDepartment().getManager();
    }

    public void fireJohn () {
        final Person john = ...;
        final Manager manager;
        manager = getManager(john);
        // ... tell John's manager the news
    }
}
```

At this point the developer may notice that *other* clients use a similar query and decide that it therefore more properly belongs on the `Person` class itself. She can achieve this by simply selecting the argument whose type is to become the host class of the query:

```
manager = getManager(john)
```

and applying *Push method into parameter type*. The client-side transformation is intuitive:

```
manager = john.getManager()
```

and the `getManager()` method is now where it belongs (and as one would expect, no longer static):

```
class Person {
    // ...

    public Manager getManager () {
        return getDepartment().getManager();
    }
}
```

The benefits of decomposing *Hide delegate* and similar refactorings in this way are significant. The user does not need to have memorized a large repertoire of macroscopic refactorings such as *Hide Delegate*. Nor need she hold a complicated conversation with her tool before the activity starts. Instead she can decide how to proceed at each step, perhaps even exploring an entirely different refactoring which only suggests itself halfway through the process. Finally, she has a new primitive at her disposal, *Push method into parameter type*, which can be used in a variety of common situations, not just on methods which have been freshly extracted as part of *Hide delegate*.

Things get even more interesting when we carry out a similar decomposition of well-known "primitives" such as *Extract method*. A set of even more primitive and general operations emerges. Again we take an example from Fowler, this time *Extract method* (p. 114):

```
void printOwing {
    Enumeration e = _orders.elements();
    double outstanding;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

The developer wishes to extract the calculation of `outstanding` (shown selected above) to a new method so that it can be used elsewhere:

```
void printOwing {
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding () {
    Enumeration e = _orders.elements();
    double outstanding;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Now imagine the developer wishes to generalize the `getOutstanding()` method further, so that it is not coupled to the current instance, and furthermore works with any enumeration of orders. His first step is to *Make method static* (roughly the inverse of *Push method into parameter type*):

```
static double getOutstanding (final Vector orders) {
    Enumeration e = orders.elements();
    double outstanding;
    // ...
}
```

Finally, he selects the call which obtains the elements of the vector:

```
    Enumeration e = orders.elements();
```

and applies *Push code out of method*, effectively replacing the vector parameter by an enumeration:

```
static double getOutstanding (final Enumeration e) {
    double outstanding;
    // ...
}
```

and forcing each call site to wrap its vector argument in a query for its elements:

```
void printOwing {
    double outstanding =
        getOutstanding(_orders.elements());
    printDetails(outstanding);
}
```

Rather than having to re-inline the entire method and start again, the developer was simply able to inline that *part* of the method which he didn't want to be shared. We hope to demonstrate that a tool based on these principles gives developers power editing features with no loss of control.

3. REFERENCES

[1] Fowler, M. *Refactoring*
Addison-Wesley, Reading, MA, 1999.

[2] Opdyke, W. *Refactoring Object-Oriented Framework*
PhD. Thesis, University of Urbana-Champaign, 1992.

[3] Roberts, D., Brant, J., and Johnson, R. *A Refactoring Tool for Smalltalk*
Theory and Practice of Object Systems archive, 3, 4 (1997), 253-263