Untangling: A Slice Extraction Refactoring

Ran Ettinger and Mathieu Verbaere
Programming Tools Group
Computing Laboratory
University of Oxford

Thanks to our supervisor Oege de Moor

Supported by an Eclipse Innovation Grant from IBM
Refactoring to Aspects by Slicing

AOSD

Refactoring

Slicing
The problem of tangled code

“The tangled code is extremely difficult to maintain, since small changes to the functionality require mentally untangling and then retangling it.” [Kiczales et al., 1997]

- Aspects allow structuring code in a non-tangled way
  - Concerns can be maintained independently
  - Concerns become (un)pluggable

- Refactoring tools should help in untangling
  - Moving existing tangled code to aspects
Program Slicing

• “When debugging unfamiliar programs programmers use program pieces called slices which are sets of statements related by their flow of data. The statements in a slice are not necessarily textually contiguous, but may be scattered through a program” [Weiser, 1982]

• Demo 1: slicing
A Slicing Example

Original program

```java
public void count(int[] in) {
    int i=0;
    int c;
    int nl=0;

    int nw=0;
    int nc=0;
    boolean inword=false;

    while (i < in.length) {
        c = in[i];
        nc = nc + 1;
        if (c == '\n')
            nl = nl + 1;
        if (c == ' ' || c == '\n' || c == '
' || c == false;
            inword = false;
        } else if (inword == false) {
            inword = true;
            nw = nw + 1;
        }
        i = i + 1;
    }

    lines = nl;
    words = nw;
    chars = nc;
}
```

Slice for variable `nl`

```java
public void count(int[] in) {
    int i=0;
    int c;
    int nl=0;

    while (i < in.length) {
        c = in[i];
        if (c == '\n')
            nl = nl + 1;
        i = i + 1;
    }
```
Slicing for Untangling

• Weiser’s hypothesis:
  Programmers mentally construct slices when debugging

• Our hypothesis:
  Programmers mentally construct slices when untangling

• Our suggestion:
  Support untangling through a slice extraction refactoring
  • Why refactoring? Our focus: readability and reusability
Untangling: Three Approaches

• Procedural
  – Extract Slice as Method

• Object-Oriented
  – Extract Slice as Object

• Aspect-Oriented
  – Extract Slice as Aspect
Extract Slice as **Method**

- Suggested in the past: Lakhotia, Maruyama
- A generalisation of standard refactorings such as
  - *Extract Method*
  - *Decompose Conditional*
  - *Separate Query from Modifier*
  - *Replace Temp with Query*
- **Nate**: our implementation in Eclipse
  - Current prototype: slicing a small subset of Java
  - Goal: a full Java slicer

- Demo 2: refactoring
Extract Slice as **Method** (2)

```java
private int countLines(int[] in) {
    int i = 0;
    int c;
    int nl = 0;
    while (i < in.length) {
        c = in[i];
        if (c == '\n')
            nl = nl + 1;
        i = i + 1;
    }
    return nl;
}
```
Extract Slice as Method (3)

‘exclusively extracted’ fragments

`public void count(int[] in) {
    int i=0;
    int c;
    int nl=0;
    int nw=0;
    int nc=0;
    boolean inword=false;
    while (i < in.length) {
        c = in[i];
        nc = nc + 1;
        if (c == '\n')
            nl = nl + 1;
        if (c == ' ' || c == '\n')
            inword = false;
        else if (inword == false) {
            inword = true;
            nw = nw + 1;
        }
        i = i + 1;
    }
    lines = countLines(in);
    words = nw;
    chars = nc;
}`
Extract Slice as Method (4)

• Pros
  – Reusability
  – Readability

• Cons
  – Duplicated code
  – Runtime overhead
    • Untangled version may be slower
  – Restrictive (low applicability)
    • E.g. side effects in duplicated statements

• Demo 3: rejected refactoring
public void count(InputStream in) throws IOException {
    int c = in.read();
    int nl=0;
    int nw=0;
    int nc=0;
    boolean inword=false;
    while (c != EOF) {
        nc = nc + 1;
        if (c == '\n')
            nl = nl + 1;
        if (c == ' ' || c == '\n' || c == '\t')
            inword = false;
        else if (inword == false) {
            inword = true;
            nw = nw + 1;
        }
        c = in.read();
    }
    lines = nl;
    words = nw;
    chars = nc;
}
Extract Fragments of a Slice

```java
public void count(InputStream in) throws IOException {
    int c = in.read();
    int nl=0;
    int nw=0;
    int nc=0;
    boolean inword=false;
    while (c != EOF) {
        nc = nc + 1;
        if (c == '\n')
            nl = nl + 1;
        if (c == ' ' || c == '\n' || c == '\t')
            inword = false;
        else if (inword == false) {
            inword = true;
            nw = nw + 1;
        } else
            c = in.read();
    }
    lines = nl;
    words = nw;
    chars = nc;
}
```

Alternative:
Extract ‘exclusive fragments’ only (by using Extract Method on each fragment)
public void count(InputStream in) throws IOException {
    int c = in.read();
    int nl = initLines();
    int nw=0;
    int nc=0;
    boolean inword=false;
    while (c != EOF) {
        nc = nc + 1;
        nl = incrementLines(c, nl);
        if (c == ' ' || c == '\n' || c == '\t')
            inword = false;
        else if (inword == false) {
            inword = true;
            nw = nw + 1;
        }
        c = in.read();
    }
    lines = nl;
    words = nw;
    chars = nc;
}

Alternative:
Extract ‘exclusive fragments’ only
(by using Extract Method on each fragment)
public void count(InputStream in) throws IOException {
    int c = in.read();
    int nl = initLines();
    int nw=0;
    int nc=0;
    boolean inword=false;
    while (c != EOF) {
        nc = nc + 1;
        nl = incrementLines(c, nl);
        if (c == ' ' || c == '
')
            inword = false
        else if (inword)
            inword = true;
        nw = nw + 1;
    }
    c = in.read();
}
public void count(InputStream in) throws IOException {
    int c = in.read();
    int nl = initLines;
    int nw=0;
    int nc=0;
    boolean inword=
    while (c != EOF) {
        nc = nc + 1;
        nl = incrementLines(c, nl);
        if (c == ' ' || c == '\n' || c == '\t')
            inword = false;
        else if (inword == false) {
            inword = true;
            nw = nw + 1;
        }
        c = in.read();
    }
    lines = nl;
Solving the Problem of Local Variables

• Turn locals to fields
  – Increase memory footprint (per object)
  – Problems with recursive methods

• Turn slice into a local object
  – We call this approach: **Extract Slice as Object**
  – A variation on the **Replace Method with Method Object** refactoring:

  “You have a long method that uses local variables in such a way that you cannot apply **Extract Method**. Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.” [Fowler, 2000]
class CW {
    int result;
    boolean inword;
    int countWords(InputStream in) throws IOException {
        int c = in.read();
        init();
        while (c != EOF) {
            processChar(c);
            c = in.read();
        }
        return result;
    }
    void processChar(int c) {
        if (c == ' ' || c == '
' || c == '	')
            inword = false;
        else if (inword == false) {
            inword = true;
            result = result + 1;
        }
    }
    void init() {
        result=0;
        inword=false;
    }
}
public void count(InputStream in) throws IOException {
    int c = in.read();
    int nl=0;
    CW cwObj = new CW();
    cwObj.init();
    int nc=0;
    boolean inword=false;
    while (c != EOF) {
        nc = nc + 1;
        if (c == '\n')
            nl = nl + 1;
        cwObj.processChar(c);
        c = in.read();
    }
    lines = nl;
    words = cwObj.result;
    chars = nc;
}
public void count(InputStream in) throws IOException {
    int c = in.read();
    CL c1Obj = new CL();
    CW cwObj = new CW();
    CC ccObj = new CC();
    c1Obj.init();
    cwObj.init();
    ccObj.init();
    while (c != EOF) {
        ccObj.processChar();
        c1Obj.processChar(c);
        cwObj.processChar(c);
        c = in.read();
    }
    lines = c1Obj.result;
    words = cwObj.result;
    chars = ccObj.result;
}
Extract Slice as Object (4)

• Mechanics:
  – Identify exclusively extracted fragments
  – Use Extract Slice as Method (without insertion of a call to that new method)
  – Replace the new method with a method object
    • Turn local variables into fields
  – Apply Extract Method to each exclusive fragment
  – Update original method
    • Create local instance of the method object
    • Replace exclusive fragments with method calls
Extract Slice as **Object** (5)

- **Pros**
  - Generally applicable
    - E.g. no objection for side effects in extracted code
  - Reduced runtime overhead

- **Cons**
  - Readability
    - Produced code is fragmented
  - Cumbersome user interaction
    - Selecting multiple method names (a method for each extracted fragment)
  - Duplicated code
  - Reusability
    - Extracted concerns are not pluggable
Extract Slice as Aspect

• Similar to Extract Slice as Object but:
  – Each exclusive fragment is extracted as an advice
    • Fragments must be extractable
    • Preparation steps (standard refactorings) may be needed
boolean inword;
public void count(InputStream in) throws IOException {
    lines=0;
    words=0;
    chars=0;
    inword=false;
    int c = in.read();
    while (c != EOF) {
        processChar(c);
        c = in.read();
    }
}

void processChar(int c) {
    chars = chars + 1;
    if (c == '\n')
        lines = lines + 1;
    if (c == ' ' || c == '\n' || c == '\t')
        inword = false;
    else if (inword == false) {
        inword = true;
        words = words + 1;
    }
}
Extract Slice as **Aspect (3)** – untangled **character count** aspect

‘exclusively extracted’ fragments

```java
static aspect CC {
    int WordCount.chars;

    before(WordCount wc) :
        execution(void WordCount.count(InputStream)) &&
        this(wc) {
        wc.chars = 0;
    }

    before(WordCount wc) :
        execution(void WordCount.processChar(int)) &&
        this(wc) {
        wc.chars = wc.chars + 1;
    }
```
Extract Slice as \textit{Aspect} (4)

Notice that the code for computing the number of characters is absent

\begin{verbatim}
class CountLines {
  public void count(InputStream in) throws IOException {
    lines = 0;
    words = 0;
    inword = false;
    int c = in.read();
    while (c != EOF) {
      processChar(c);
      c = in.read();
    }
  }

  void processChar(int c) {
    if (c == newline)
      lines = lines + 1;
    if (c == ' ' || c == newline || c == 'tab')
      inword = false;
    else if (inword == false) {
      inword = true;
      words = words + 1;
    }
  }
}
\end{verbatim}
Extract Slice as **Aspect** (5) – untangled **lines count** aspect

```java
static aspect LC {
    int WordCount.lines;

    before(WordCount wc) :
        execution(void WordCount.count(InputStream)) &&
        this(wc) {
            wc.lines = 0;
        }

    before(WordCount wc, int c) :
        execution(void WordCount.processChar(int)) &&
        this(wc) && args(c) {
            if (c == '\n')
                wc.lines = wc.lines + 1;
        }
}
```

Repeated pointcuts: could be factored out into an abstract aspect
Extract Slice as **Aspect** (6) – untangled base program

```java
public void count(InputStream in) throws IOException {
    int c = in.read();
    while (c != EOF) {
        processChar(c);
        c = in.read();
    }
}

void processChar(int c) {
}
```
Extract Slice as Aspect (7)

- Mechanics:
  - Identify exclusively extracted fragments
    - If joinpoints are missing, create them with *Extract Method*
    - If local variables are used turn them into fields or apply *Turn Method into Method Object* first
  - Introduce a new aspect
  - Apply *Extract Introduction* [Hanenberg, 2003] to move field declarations to the new aspect
  - Apply *Extract Advice* [Hanenberg, 2003] to each exclusive fragment
  - Apply *Extract Slice as Aspect* to the complement
    - To make the extracted concern reusable
Extract Slice as *Aspect* (8)

**Pros**
- Smooth user interaction
  - no need to name each fragment (advices are anonymous)
- No duplicated code
- Reusability
- The extracted concern becomes (un)pluggable
  - It is completely removed from the base program

**Cons**
- Exposed joinpoints
  - May require a preparation step
- Readability
  - Inlining may help
Related Work

AOSD

- Hanenberg
- Zhao
- Monteiro

Refactoring
Related Work

- Zhao
- Balzarotti and Monga (FOAL 04)
Related Work

• Lakhotia and Deprez
  – A transformation called *Tuck*
• Maruyama
• Komondoor and Horwitz
Current/Future Directions

• **Nate**: slicing-based refactoring in Eclipse
  – Aspect aware
  – Slice negotiation

• Case studies

• Aspect interference

• Nate and the **Concern Manipulation Environment (CME)**
Thanks!
public void walkFlow(FlowCheckerPass w) {
    if (getBody() != null) {
        setupFlowWalker(w);
        w.setLive(true);
        w.process(getBody());

        if (!getResultType().isVoid() && w.isLive()) {
            showError("missing return statement");
        }
    }
}
/** This pass signals errors for

<ul>
    <li>used-before-assigned vars and blank-final fields</li>
    <li>assigned-twice blank finals</li>
    <li>constructors not filling in blank final fields</li>
    <li>unreachable stmts</li>
    <li>missing return stmts</li>
    <li>various illegal try/catch stmts</li>
</ul>

<p>It makes exactly one side-effect: In anonymous classes, it sets up the throws clause to include any thrown checked exceptions, as required by JLS 15.9.5.1.</p>

<p>Other than that one side-effect, it only signals errors.</p>

*/

public final class FlowCheckerPass extends WalkerPass {

1 package org.aspectj.compiler.base;
2
3 import org.aspectj.compiler.base.ast.*;
4 import org.aspectj.compiler.crosscuts.ast. *
5
6 public aspect FlowCheckerAspect {
7    // ----------------------------------------
8    // INTRO to AndAndOpExpr
9    
10       public void AndAndOpExpr.walkFlow(FlowCheckerPass w) {
11           // w.setArgs(w.getArgs())
12           w.processBoolean(getRand1());
13           FlowCheckerPass.Vars p1 = w.getVars();
14
15           w.setVars(p1.getTrue());
16           w.processBoolean(getRand2());
17           FlowCheckerPass.Vars p2 = w.getVars();
18
19           w.setVars(p2.getTrue(), p1.getFalse()).join(p2.getFalse());
20       }
21   }
package org.aspectj.compiler.base;

import org.aspectj.compiler.base.ast.CodeDec;

public aspect MissingReturnAspect {

    pointcut codeDecWalkFlow(CodeDec node, FlowCheckerPass w):
        call(void CodeDec.walkFlow(FlowCheckerPass)) &&
        target(node) &&
        args(w);

    after(CodeDec node, FlowCheckerPass w) : codeDecWalkFlow(node, w) {
        if (node.getBody() != null) {
            if (! node.getResultType().isVoid() && w.isLive()) {
                node.showError("missing return statement");
            }
        }
    }
}