

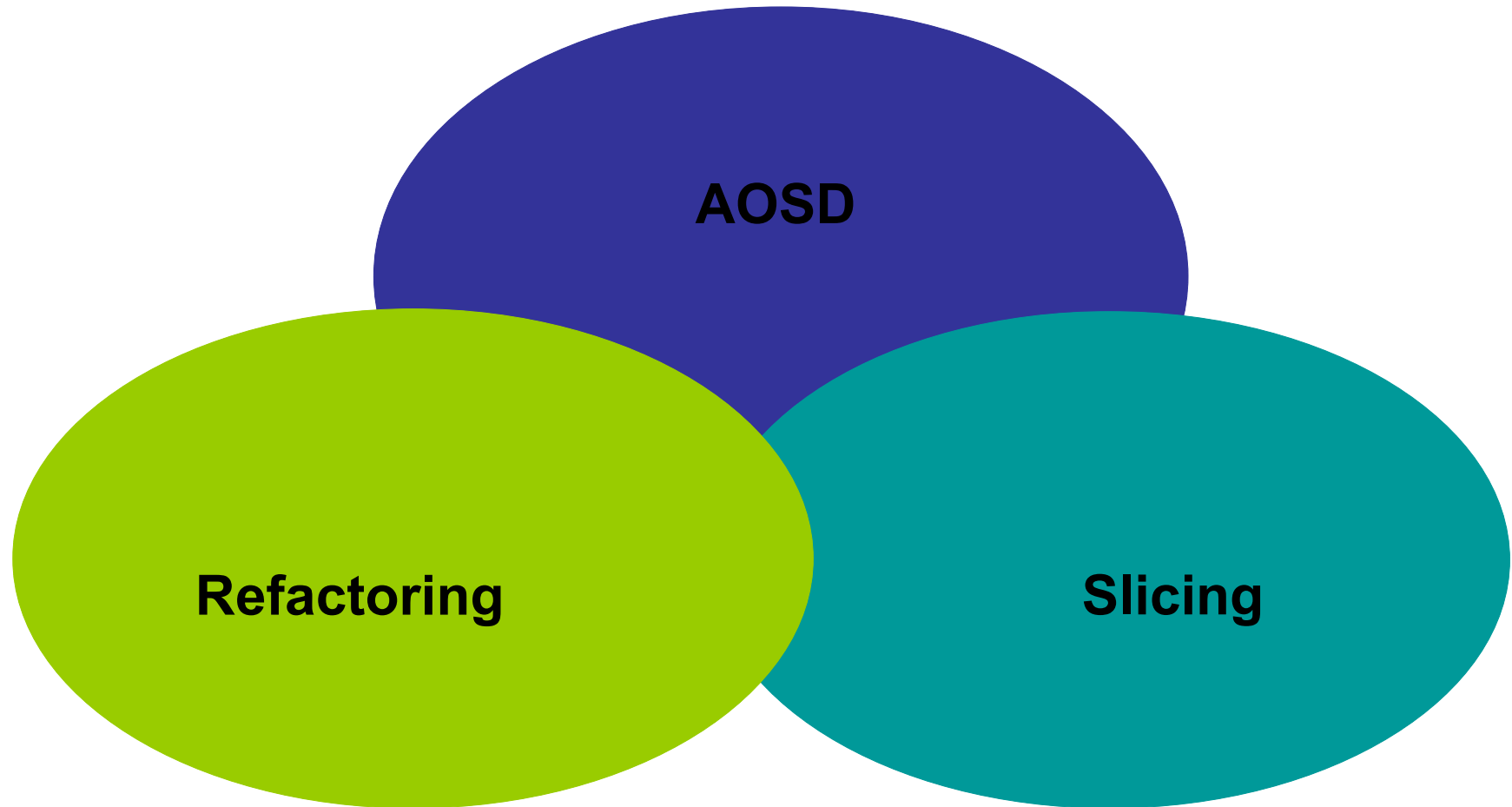
Untangling: A Slice Extraction Refactoring

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



The problem of **tangled** code

*“The tangled code is extremely difficult to maintain, since small changes to the functionality require mentally **untangling** and then **re-tangling** 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

```
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
            inword = false;
        else if (inword == false) {
            inword = true;
            nw = nw + 1;
        }
        i = i + 1;
    }
    lines = nl;
    words = nw;
    chars = nc;
}
```

Slice for variable *nl*

```
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)

Refactored Source

```
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

'in' is not modified

Original Source	Refactored Source
<pre>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 = nl; words = nw; }</pre>	<pre>public void count(int[] in) { int i=0; int c; int nw=0; int nc=0; boolean inword=false; while (i < in.length) { c = in[i]; nc = nc + 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; }</pre>

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**

Extract Slice as Method (5)

```
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;
}
```

Refactoring rejected:

Input statements in extracted code
(internal state of 'in' is modified)

Extract Fragments of a Slice

```
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;
}
```

Alternative:

Extract 'exclusive fragments' only
(by using Extract Method on each
fragment)

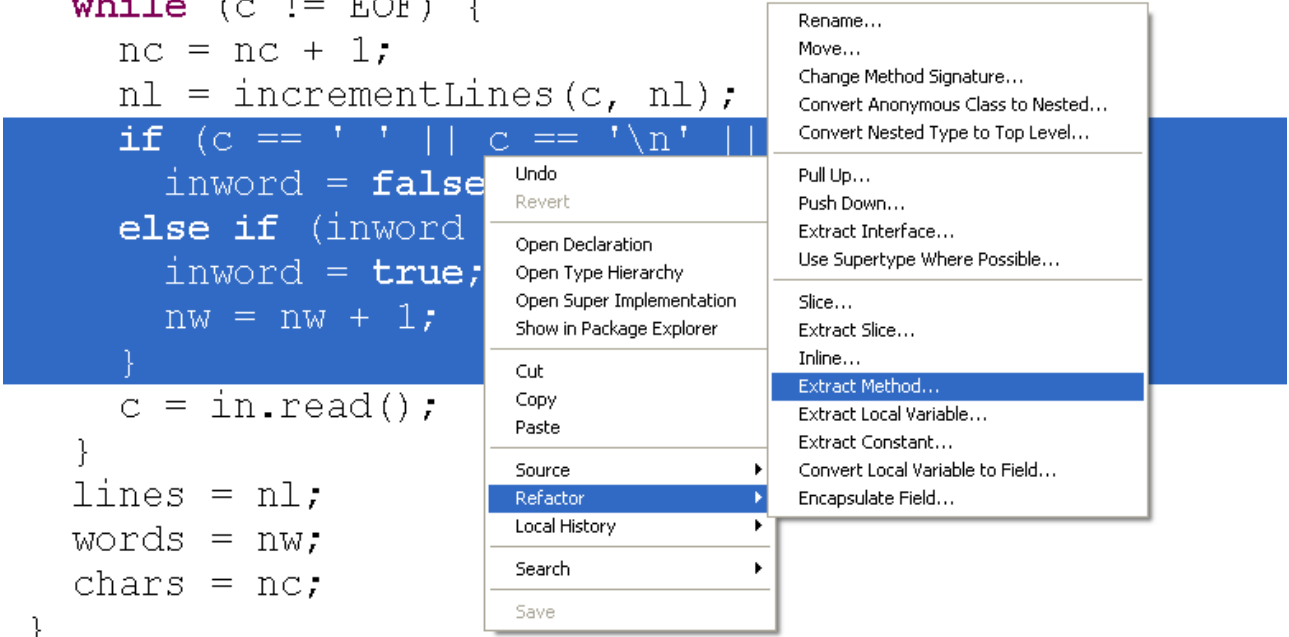
Extract Fragments of a Slice

```
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)

Extract Fragments: words

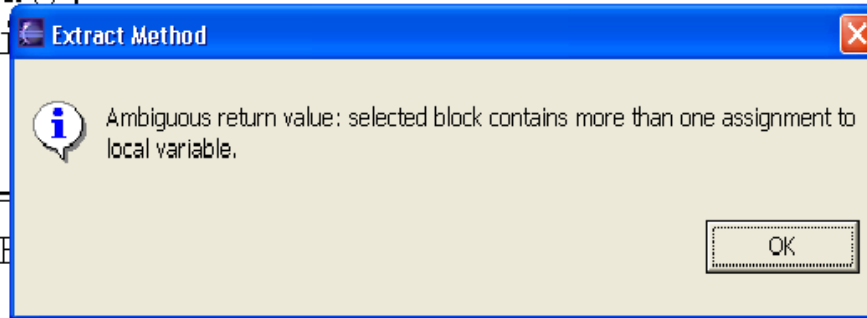
```
public void count(InputStream in) throws IOException {  
    int c = in.read();  
    int nl = initLines();  
    int nw=0;  
    int nc=0;  
    boolean inword=false;  
    while (c != EOF) {  
        nc = nc + 1;  
        nl = incrementLines(c, nl);  
        if (c == ' ' || c == '\n' ||  
            inword = false;  
        else if (inword  
            inword = true;  
            nw = nw + 1;  
        }  
        c = in.read();  
    }  
    lines = nl;  
    words = nw;  
    chars = nc;  
}
```



The image shows a code editor with a context menu open over a code snippet. The code snippet is highlighted in blue. The context menu is open, showing options like 'Extract Method...', 'Extract Local Variable...', and 'Extract Constant...'. The 'Extract Method...' option is highlighted in blue.

Extract Fragments: Problem of Local Variables

```
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;
}
```



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]

Extract Slice as Object

```
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 == '\n' || c == '\t')  
            inword = false;  
        else if (inword == false) {  
            inword = true;  
            result = result + 1;  
        }  
    }  
    void init() {  
        result=0;  
        inword=false;  
    }  
}
```

The diagram illustrates the extraction of code slices from the provided Java code. Three rectangular boxes are drawn around specific code segments: one around the `init();` call, one around the `processChar(c);` call inside the `while` loop, and one around the `processChar(c)` method definition. Arrows originate from these boxes: one points from the `init();` box to the `void init()` method definition, another points from the `processChar(c);` box to the `void processChar(int c)` method definition, and a third points from the `processChar(c)` box to the `void processChar(int c)` method definition. A fourth arrow points from the `init();` box to the `void init()` method definition.

Extract Slice as Object (2)

```
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;
}
```

Extract Slice as Object (3)

```
public void count(InputStream in) throws IOException {
    int c = in.read();
    CL clObj = new CL();
    CW cwObj = new CW();
    CC ccObj = new CC();
    clObj.init();
    cwObj.init();
    ccObj.init();
    while (c != EOF) {
        ccObj.processChar();
        clObj.processChar(c);
        cwObj.processChar(c);
        c = in.read();
    }
    lines = clObj.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

Extract Slice as Aspect (2) - Exposing Joinpoints

```
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

```
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 Aspect (4)

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

```
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 == '\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** (5) – untangled **lines count** aspect

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

**Repeated pointcuts: could be factored
out into an abstract aspect**

```
    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;  
    }  
}
```

Extract Slice as **Aspect** (6) – untangled base program

```
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

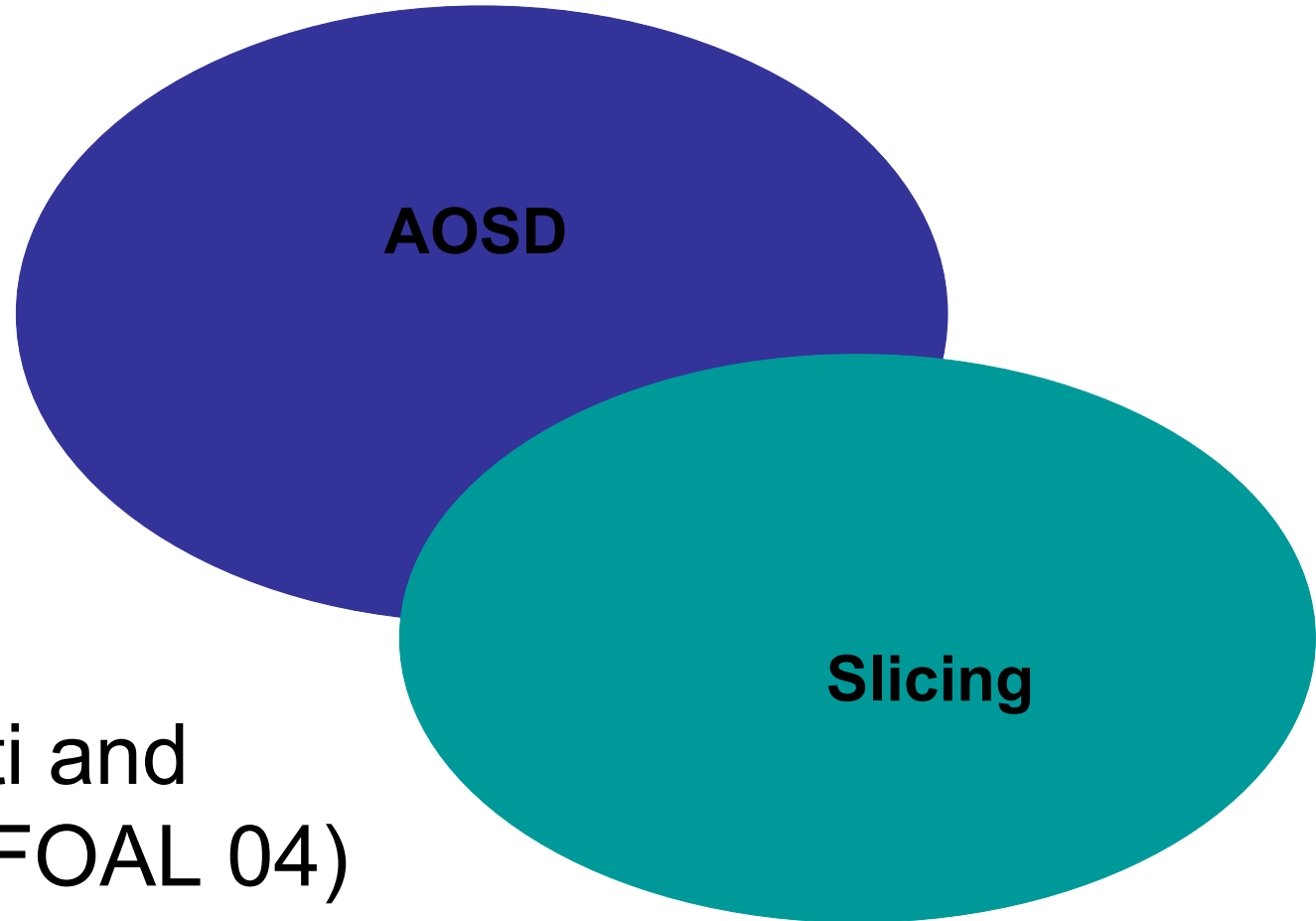


AOSD

Refactoring

- Hanenberg
- Zhao
- Monteiro

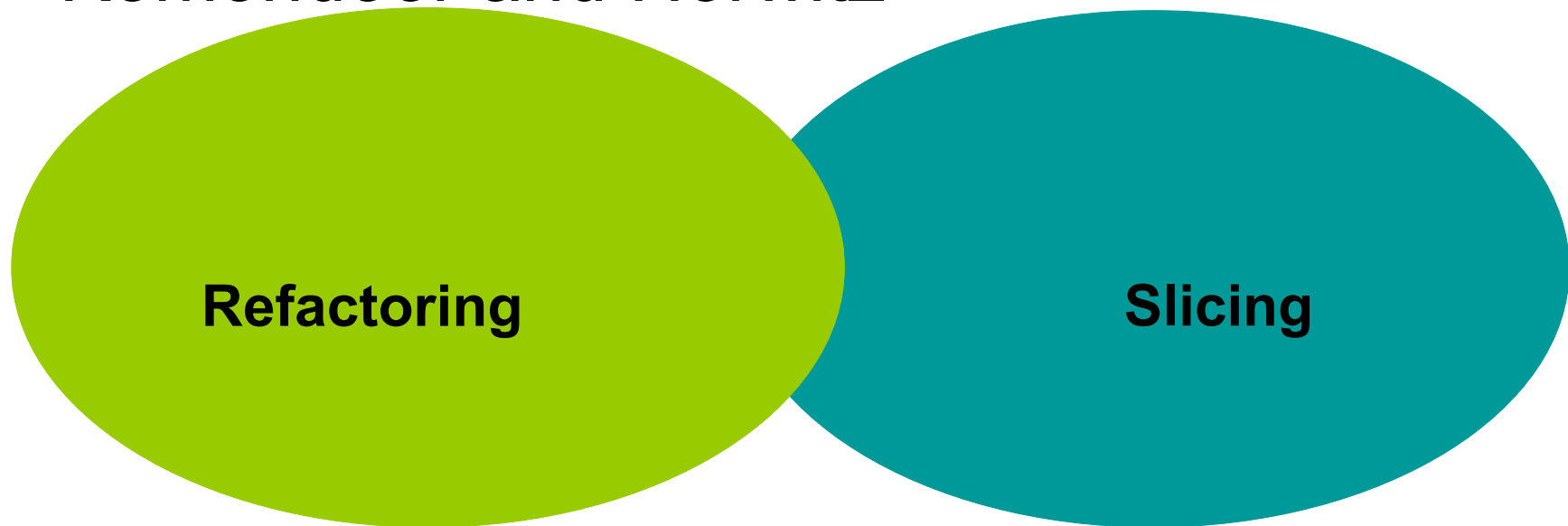
Related Work



- 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!

```
CodeDec.java x FlowCheckerPass.java
54     public void walkFlow(FlowCheckerPass w) {
55         if (getBody() != null) {
56             setupFlowWalker(w);
57             w.setLive(true);
58             w.process(getBody());
59
60             if (! getResultType().isVoid() && w.isLive()) {
61                 showError("missing return statement");
62             }
63         }
64     }
```

```

CodeDec.java | FlowCheckerPass.java X
32/** This pass signals errors for
33
34    <ul>
35        <li>used-before-assigned vars and blank-final fields</li>
36        <li>assigned-twice blank finals</li>
37        <li>constructors not filling in blank final fields</li>
38        <li>unreachable stmts</li>
39        <li>missing return stmts</li>
40        <li>various illegal try/catch stmts</li>
41    </ul>
42
43    <p> It makes exactly one side-effect: In anonymous classes, it
44    sets up the throws clause to include any thrown checked
45    exceptions, as required by JLS 15.9.5.1. </p>
46
47    <p> Other than that one side-effect, it only signals errors. </p>
48*/
49
50public final class FlowCheckerPass extends WalkerPass {

```

```

CodeDec.java | FlowCheckerPass.java | FlowCheckerAspect.java x
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     // -----
9     // INTRO to AndAndOpExpr
10
11     public void AndAndOpExpr.walkFlow(FlowCheckerPass w) {
12         // w.setArgs(w.getArgs());
13         w.processBoolean(getRand1());
14         FlowCheckerPass.Vars p1 = w.getVars();
15
16         w.setVars(p1.getTrue());
17         w.processBoolean(getRand2());
18         FlowCheckerPass.Vars p2 = w.getVars();
19
20         w.setVars(p2.getTrue(), p1.getFalse().join(p2.getFalse()));
21     }

```

```
CodeDec.java | FlowCheckerPass.java | FlowCheckerAspect.java | MissingReturnAspect.java X
1 package org.aspectj.compiler.base;
2
3 import org.aspectj.compiler.base.ast.CodeDec;
4
5 public aspect MissingReturnAspect {
6
7     pointcut codeDecWalkFlow(CodeDec node, FlowCheckerPass w) :
8         call(void CodeDec.walkFlow(FlowCheckerPass)) &&
9         target(node) &&
10        args(w);
11
12    after(CodeDec node, FlowCheckerPass w) : codeDecWalkFlow(node, w) {
13        if (node.getBody() != null) {
14            if (! node.getResultType().isVoid() && w.isLive()) {
15                node.showError("missing return statement");
16            }
17        }
18    }
19 }
```