Slicing for Refactoring

Mathieu Verbaere
Programming Tools Group
Computing Laboratory
University of Oxford

VASTT/ASTReNet Slicing Day workshop
8th November 2004

Introducing **Refactoring**...

- Refactoring is a technique to gradually improve the design of an existing software by performing source code transformations *without changing its runtime behaviour*.

- Performed interactively within an IDE.

- More and more popular both in development (eXtreme Programming) and maintenance activities.

- Catalog of refactorings maintained by Martin Fowler.

- Current research mainly on mechanizing and proving refactorings.

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]

- Refactoring focuses on readability and reusability. Hence, many well-known refactorings deal with untangling code.

- How slicing can be used to automate these refactorings?

Split Loop [Fowler's catalog]

```
void printValues() {
     double averageAge = 0;
     double totalSalary = 0;
     for (int i = 0; i < people.length; i++) {
          averageAge += people[i].age;
          totalSalary += people[i].salary;
     }
     i = 0;
     averageAge = averageAge / people.length;
     System.out.println(averageAge);
     System.out.println(totalSalary);
}
```

```
void printValues() {
     double totalSalary = 0;
     for (int i = 0; i < people.length; i++) {
          totalSalary += people[i].salary;
     }
     double averageAge = 0;
     for (int i = 0; i < people.length; i++) {
          averageAge += people[i].age;
     }
     averageAge = averageAge / people.length;
     System.out.println(averageAge);
     System.out.println(totalSalary);
}
```

Fowler's mechanics: "

- **Copy the loop and remove the differing pieces from each loop.**
- Compile and test.
- **Reorganize the lines to group the loop with related code from outside the loop.**
- Compile and test.
- Consider applying ***Extract Method*** or ***Replace Temp With Query***. "

Split Loop - automation (1)

```
1       void printValues() {
2               double averageAge = 0;
3               double totalSalary = 0;
4               for (int i = 0; i < people.length; i++) {
5                       averageAge += people[i].age;
6                       totalSalary += people[i].salary;
7               }
8               averageAge = averageAge / people.length;
9               System.out.println(averageAge);
10              System.out.println(totalSalary);
11      }
```

Automation...

- The user selects the loop $L = \{4,5,6,7\}$
- The tool asks for the variable *v* whose computation in the loop needs to be dissociated.

$$v \in def(L) \cap live(8)$$

with live(8) = {averageAge, people.length, totalSalary}

Split Loop - automation (2)

```
1     void printValues() {
2         double averageAge = 0;
3         double totalSalary = 0;
4         for (int i = 0; i < people.length; i++) {
5              averageAge += people[i].age;
6              totalSalary += people[i].salary;
7      }
8         averageAge = averageAge / people.length;
9         System.out.println(averageAge);
10        System.out.println(totalSalary);
11    }
```

```
void printValues() {
     double averageAge = 0;
     for (int i = 0; i < people.length; i++) {
          averageAge += people[i].age;
     }
     double totalSalary = 0;
     for (int i = 0; i < people.length; i++) {
          totalSalary += people[i].salary;
     }
     averageAge = averageAge / people.length;
     System.out.println(averageAge);
     System.out.println(totalSalary);
}
```

Automation based on Lakhotia and Deprez's technique:

- Compute the slice of the variable of interest (*v* =`averageAge`) in the scope *{2,3,4,5,6,7}*.
- Compute the slice, named complement, of the remaining statements.
- Arrange correctly the slice and its complement in order not to break any dependency if possible.

Extract Method [Fowler's catalog]

```
void printOwing() {
     Enumeration e = orders.elements();
     double outstanding;

     printBanner();

     while (e.hasMoreElements()) {
          Order each = (Order)e.nextElement();
          outstanding += each.getAmount();
     }

     printDetails(outstanding);
}
```

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

double getOutstanding() {
     Enumeration e = orders.elements();
     double outstanding = 0.0;
     while (e.hasMoreElements()) {
          Order each = (Order)e.nextElement();
          outstanding += each.getAmount();
     }
     return outstanding;
}
```

- The declarations of the variables needed for the computation of **`outstanding`** are moved into the new method.
- Can we automate this?

Extract Method - decomposition

```
void printOwing() {
    Enumeration e = orders.elements();
    double outstanding;

    printBanner();

    while (e.hasMoreElements()) {
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

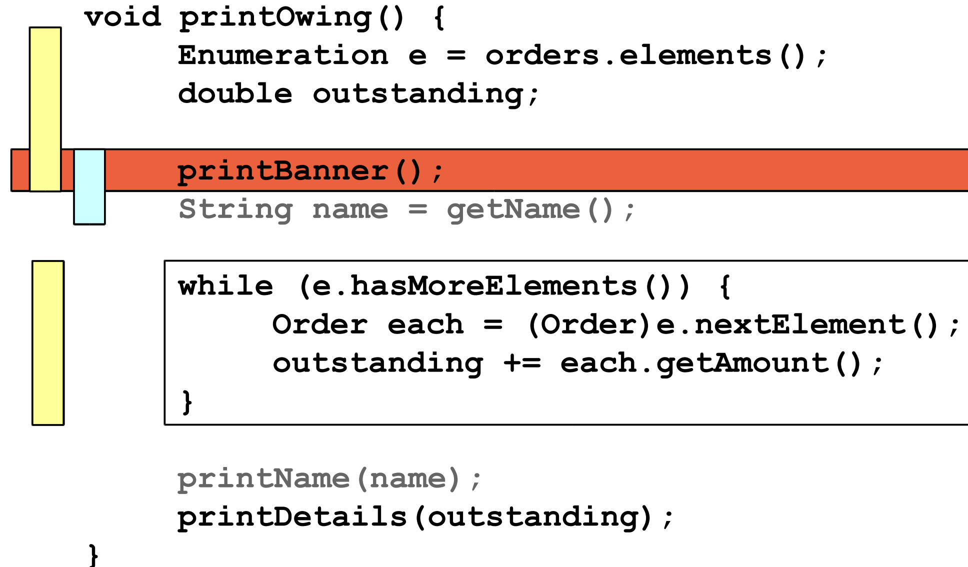
```
void printOwing() {
    printBanner();

    Enumeration e = orders.elements();
    double outstanding;
    while (e.hasMoreElements()) {
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

- Similar to **Split Loop**, except that we slice for all the variables relevant in the selected block.

$\Rightarrow$ **The untangling transformation** (based on slicing) **is a primitive used in the composition of refactorings.**

Conflicts when untangling

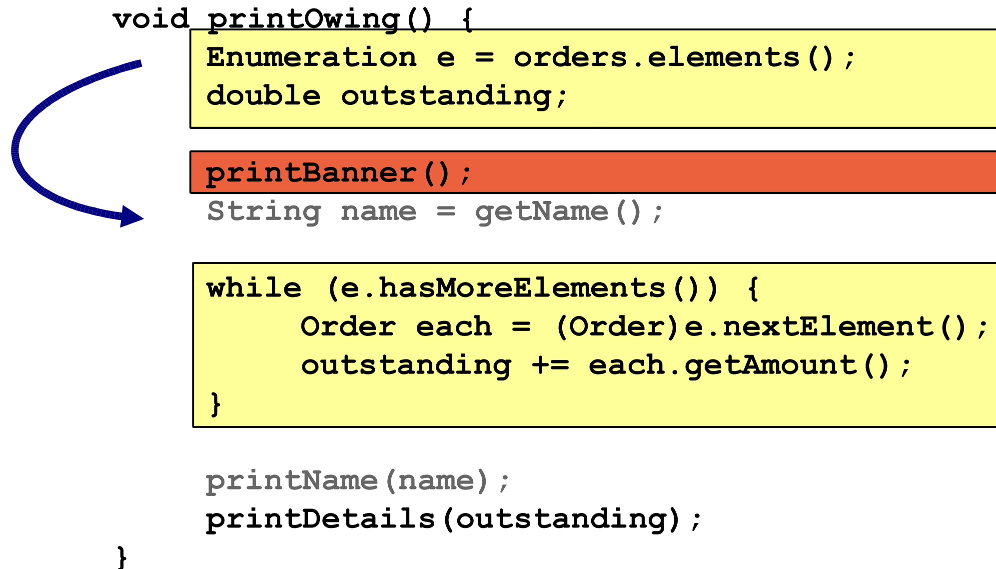
- Untangling is not always feasible because of conflicts: code duplication may change the behaviour of the program.



- If, for instance, our slicer is not precise enough, **`printBanner()`** may be both in the slice and the complement.
- **`PrintBanner()`** can not be duplicated $\Rightarrow$ conflict.

Optimizing untangling

- The transformation should not be rejected! The piece of code below the conflict can be extracted.



- We can even optimize the untangling by trying to push the two first statements below the conflict.
- How do we know we do not break the behaviour of the program? **By playing with forward and backward slicing...**

Non-behaviour-preserving transformations (1)

Extract Local Variable

(Eclipse 3.0 does not warn you on any change of behaviour)

```
int global = 0;

void updateGlobal() {
     global =  postIncrGlobal() - getGlobal();
}
int postIncrGlobal() {
     return global++;
}
int getGlobal() {
     return global;
}
```

≠

```
int global = 0;

void updateGlobal() {
     int tmp = getGlobal();
     global =  postIncrGlobal() - tmp;
}
int postIncrGlobal() {
     return global++;
}
int getGlobal() {
     return global;
}
```

Non-behaviour-preserving transformations (2)

Inline Variable

(Eclipse 3.0 does not warn you on any change of behaviour)

```
void oneEach() {
     int i = 0;
     int j = i++;
     int k = j;
     int l = j;
     System.out.print(i);
}
```

$\neq$

```
void oneEach() {
     int i = 0;
     int k = i++;
     int l = i++;
     System.out.print(i);
}
```

Directions

- Investigation of an Extensible Toolkit for Refactoring (funded by Microsoft Research) to:
  - allow composition of refactorings,
  - enable developers to author their own refactorings.
- A component for slicing appears as one of the most useful (but also one of the most complex) helper function of such a toolkit.