


Extract Slice Refactoring

Rani Ettinger, Programming Tools
Group, May 19th, 2003




Outline

- Extract Slice Refactoring:
 - Motivation
 - Example
 - Mechanics
 - Correctness issues:
 - Behaviour Preservation
 - Limitations and Preconditions
- 



Extract Slice


You have a method that does too much. Among others, it computes some value using a computation that deserves a method of its own.

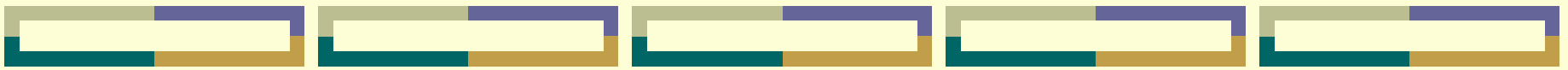
- *Extract the computation into a method whose name explains the purpose of the computation.*
- 



Extract Slice – Example (Before)


```
void sum_and_prod() {  
    int sum = 0;  
    prod = 1;  
    int i = 0;  
    while (i<array.length) {  
        sum += array[i];  
        prod *= array[i];  
        i++;  
    }  
    System.out.println(sum);  
    System.out.println(prod);  
}
```





Extract Slice – Example (After, Extracted Method)


```
int computeSum () {  
    int result = 0;  
    int i = 0;  
    while (i<array.length) {  
        result += array[i];  
        i++;  
    }  
    return result;  
}
```






Extract Slice – Example (After, Source Method)

```
void sum_and_prod() {  
    int sum = computeSum();  
    prod = 1;  
    i = 0;  
    while (i<array.length) {  
        prod *= array[i];  
        i++;  
    }  
    System.out.println(sum);  
    System.out.println(prod);  
}
```






Extract Slice - Motivation

- Decompose (de-fuse?) entangled computations
 - Enhance code reusability
 - Enhance code clarity
 - Avoid *Extract Method's Code Fragment* restriction
 - Introduce aspects
- 




Extract Slice - Mechanics

- In a method, say f , look for a variable, say v of type T , that is assigned with the result of a computation
 - ➔ *If the computation's statements are clearly consecutive, Extract Method may be used instead*
 - Create a new method, and name it after the intention of the computation
 - ➔ *For example, computeV - for the computation of v*
 - ➔ *If v is defined locally in f, define the target method to return a value of type T*
- 




Extract Slice – Mechanics (2)

- Identify all the statements that contribute to the computation of v , i.e. the backward slice with respect to the slicing criterion $\langle s, v \rangle$ where s uses the final value of v
 - *Copy the extracted code to the new target method*
 - *again, if v is defined locally in f , and thus redefined locally in computeV , add a `return v` statement at the end of the target method. Then you may substitute the name v in computeV with its local meaning: `result`*
- 




Extract Slice – Mechanics (3)

- Scan the extracted code for references to any variables that are parameters to f . These should be parameters to *computeV* also
 - *Check preconditions...*
- 




Extract Slice – Mechanics (4)

- *Look to see which of the extracted statements are no longer needed in f and delete those*
 - ➔ *Do not delete a statement if it defines or assigns to a variable that is still needed for other purposes in f , excluding v itself, which is computed now by the `computeV` method*
 - ➔ *Do not delete a statement if it controls the execution of other statements, like `if` and `while`, and the corresponding `break` and `continue` statements, if at least one of the controlled statements (`while` body, `then` branch or `else` branch) was not deleted*
 - ➔ *If deleting a statement will break the program (e.g. the only statement in the `then` branch of an `if` statement that also has an `else` branch), replace it with the empty statement*
- 




Extract Slice – Mechanics (5)

- *Replace the extracted code in f with a call to the target method*
 - ➔ *Finally, if v is defined locally in the source method, add the method call in the initialisation part of its definition, and you can now consider using the Replace Temp with Query refactoring, to replace uses of v with calls to the new method. Otherwise, the method call can be inserted anywhere between the start of the source method, and the first use of v , and you can now consider using the Separate Query from Modifier refactoring, to move the method call from f to all its call sites*
 - *Compile and test*
- 



Behaviour Preservation


- Opdyke's 7 properties
 - Properties 1-6: syntactic correctness
 - Property 7: semantic correctness:
“Semantically Equivalent
References and Operations” [Opdyke,
William F. “Refactoring Object-Oriented Frameworks.” Ph.D.
Dissertation, University of Illinois at Urbana-Champaign, 1992]
- 



Semantically Equivalent References and Operations


- Definition: “let the external interface to the program be via the function main. If the function main is called twice (once before and once after the refactoring) with the same set of inputs, the resulting set of output values must be the same”

[Opdyke, William F. “Refactoring Object-Oriented Frameworks.” Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1992]






Semantically Equivalent References and Operations(2)

- Explanation: “Imagine that a circle is drawn around the parts of a program affected by a refactoring. The behavior as viewed from outside the circle does not change. For some refactorings, the circle surrounds most or all of the program...the key idea is that the results (including side effects) of operations invoked and references made from outside the circle do not change, as viewed from outside the circle” [Opdyke, William F. “Refactoring Object-Oriented Frameworks.” Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1992]
- 



Limitations: Do Not Extract Return Statements


```
void sum_and_prod() {  
    int sum = 0, prod = 1, i = 0;  
    if (array.length == 0) return;  
    while (i < array.length) {  
        sum += array[i];  
        prod *= array[i];  
        i++;  
    }  
    System.out.println(sum);  
    System.out.println(prod);  
}
```





Limitations: Do Not Extract Global Modifications

```
void sum_and_prod() {  
    int sum = 0, prod = 1, i = 0;  
    array = initArray();  
    while (i < array.length) {  
        sum += array[i];  
        prod *= array[i];  
        i++;  
    }  
    System.out.println(sum);  
    System.out.println(prod);  
}
```






Limitations: Must Delete All Intermediate References

```
void sum_and_prod() {  
    int sum = 0, midSum, prod = 1, i = 0;  
    while (i < array.length) {  
        sum += array[i];  
        if (i == array.length/2) midSum = sum;  
        prod *= array[i];  
        i++;  
    }  
    System.out.println(sum);  
    System.out.println(midSum);  
    System.out.println(prod); }  


```





Limitations: Do Not Extract **Input** Statements


```
void sum_and_prod(InputStream in) {  
    int sum = 0, prod = 1, i = 0;  
    int size = System.in.readInt();  
    while (i < size) {  
        sum += array[i];  
        prod *= array[i];  
        i++;  
    }  
    System.out.println(sum);  
    System.out.println(prod);  
}
```





Limitations: Do Not Extract Output Statements

```
void sum_and_prod(OutputStream out) {  
    int sum = 0, prod = 1, i = 0;  
    while (i < array.length) {  
        sum += array[i];  
        out.print(sum);  
        prod *= array[i];  
        out.print(prod); i++;  
    }  
    System.out.println(sum);  
    System.out.println(prod);  
}
```





Summary of Limitations and Preconditions

- Target method name is valid and fresh
 - The extracted code must be clear of any:
 - input or output statements
 - return statements
 - Modifications of global-scoped variables (other than v itself, if it is global at all)
 - All references to intermediate values of v were extracted and deleted
- 