SWEN 6301 Software Construction

Module 7: Code Tuning and Refactoring

Ahmed Tamrawi

Code Tuning Strategies and Techniques

Outline

- Logic
- Loops
- Data Transformations
- Expressions

Logic

- Suppose you have a statement like

```
if ( 5 < x ) and ( x < 10 ) then ...
```

- Once you’ve determined that ***x* is not greater than 5**, you don’t need to perform the second half of the test.
- Some languages provide a form of expression evaluation known as **short-circuit evaluation**, which means that the compiler generates code that automatically *stops testing as soon as it knows the answer*.
- **If not, how to fix it?**

Logic

Stop Testing When You Know the Answer

If your language **doesn't** support short-circuit evaluation natively, you have to avoid using ***and*** **and *or*, adding logic instead**. With short-circuit evaluation, the code above changes to this:

```
if ( 5 < x ) and ( x < 10 ) then ...
```

⬇

```
if ( 5 < x ) then
   if ( x < 10 ) then ...
```

Logic

Any problem?

```
negativeInputFound = false;
for ( i = 0; i < count; i++ ) {
   if ( input[ i ] < 0 ) {
      negativeInputFound = true;
   }
}
```

Logic

Stop Testing When You Know the Answer

- The principle of not testing after you know the answer is a good one for many other kinds of cases as well.

  A search loop is a common case

- If you're scanning an array of input numbers for a negative value and you simply need to know whether a negative value is present, one approach is to check every value, setting a `negativeFound` variable when you find one.

Logic

Stop Testing When You Know the Answer

```
negativeInputFound = false;
for ( i = 0; i < count; i++ ) {
   if ( input[ i ] < 0 ) {
      negativeInputFound = true;
   }
}
```

Logic

Stop Testing When You Know the Answer

A better approach would be to stop scanning as soon as you find a negative value. Any of these approaches would solve the problem:

- Add a *break* statement after the *negativeInputFound = true* line.
- If your language doesn't have *break*, emulate a *break* with a *goto* that goes to the first statement after the loop.
- Change the *for* loop to a *while* loop, and check for *negativeInputFound* as well as for incrementing the loop counter past *count*.
- Change the *for* loop to a *while* loop, put a **sentinel value** in the first array element after the last value entry, and simply check for a negative value in the *while* test.
- After the loop terminates, see whether the position of the first found value is in the array or one past the end.

Logic

Order Tests by Frequency

- *Arrange tests so that the one* ***that's fastest and most likely to be true*** **is performed first**.
- It should be easy to drop through the normal case, and if there are inefficiencies, they should be in processing the uncommon cases. This principle applies to *case* statements and to chains of *if-then-else*s.

Logic

- Here's a Visual Basic *Select-Case* statement that responds to keyboard input in a word processor
- **Any problem?**

```
Select inputCharacter
   Case "+", "="
      ProcessMathSymbol( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

Logic

Order Tests by Frequency

- The cases in this *case* statement are ordered in something close to the ASCII sort order

Visual Basic Example of a Poorly Ordered Logical Test

```
Select inputCharacter
   Case "+", "="
      ProcessMathSymbol( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

Logic

Order Tests by Frequency

Here's the reordered *case* statement:

Visual Basic Example of a Well-Ordered Logical Test

```
Select inputCharacter
   Case "A" To "Z", "a" To "z"
      ProcessAlpha( inputCharacter )
   Case " "
      ProcessSpace( inputCharacter )
   Case ",", ".", ":", ";", "!", "?"
      ProcessPunctuation( inputCharacter )
   Case "0" To "9"
      ProcessDigit( inputCharacter )
   Case "+", "="
      ProcessMathSymbol( inputCharacter )
   Case Else
      ProcessError( inputCharacter )
End Select
```

Logic

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	**0.280**	**0.260**	**7%**

Note: Benchmarked with an input mix of 78 percent alphabetic characters, 17 percent spaces, and 5 percent punctuation symbols.

- The Microsoft Visual Basic results are as expected, but the Java and C# results are not as expected.
- Apparently that's because of the way *switch-case* statements are structured in C# and Java, the C# and Java code doesn't benefit from the optimization as the Visual Basic code does.
- This result underscores the importance of not following any optimization advice blindly—*specific compiler implementations will significantly affect the results*.

Logic

Order Tests by Frequency

You might assume that the code generated by the **Visual Basic** compiler for a set of ***if-then-elses*** that perform the same test as the *case* statement would be similar. Take a look at those results:

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	**1.36**	**1.00**	**26%**

Logic

Order Tests by Frequency

- For the same number of tests, the **Visual Basic** compiler takes about five times as long in the unoptimized case, four times in the optimized case, compared to their **switch-case versions**.
- This suggests that the **compiler** is generating different code for the ***case*** approach than for the ***if-then-else*** approach.

Logic

Compare Performance of Similar Logic Structures

- The test described above could be performed using either a *case* statement or *if-then-else*s.
- Depending on the environment, either approach might work better.
- Here is the data from the preceding two tables reformatted to present the "**code-tuned**" times comparing ***if-then-else*** and ***case*** performance:

Logic

Language	*case*	*if-then-else*	Time Savings
C#	0.260	0.330	-27%
Java	2.56	0.460	82%
Visual Basic	0.260	1.00	-258%

- In **Visual Basic**, *case* is dramatically superior to *if-then-else*, and in another, *if-then-else* is dramatically superior to *case*.
- In **C#**, the difference is relatively small. You might think that because C# and Java share similar syntax for *case* statements, their results would be similar, but in fact their results are opposite each other.
- This example clearly illustrates the difficulty of performing any sort of "**rule of thumb**" or "**logic**" to code tuning—there is simply no reliable substitute for *measuring* results.

Logic

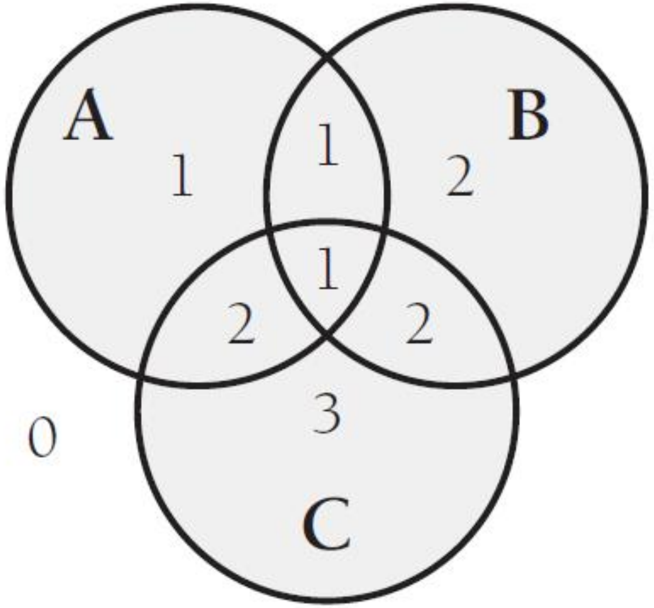
Substitute Table Lookups for Complicated Expressions

- In some circumstances, a table lookup might be quicker than traversing a complicated chain of logic.
- The point of a complicated chain is usually to categorize something and then to take an action based on its category.

Logic

Substitute Table Lookups for Complicated Expressions

As an abstract example, suppose you want to assign a category number to something based on which of three groups—Groups *A*, *B*, and *C*—it falls into:



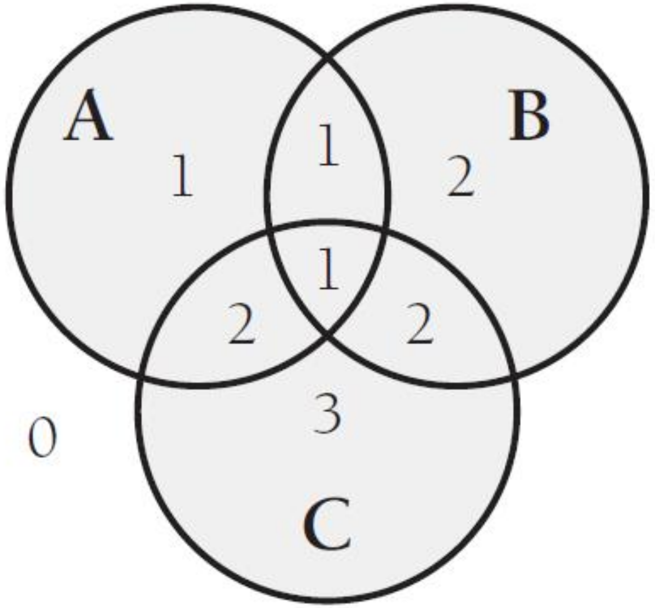
Logic

Substitute Table Lookups for Complicated Expressions

This complicated logic chain assigns the category numbers:

C++ Example of a Complicated Chain of Logic

```
if ( ( a && !c ) || ( a && b && c ) ) {
   category = 1;
}
else if ( ( b && !a ) || ( a && c && !b ) ) {
   category = 2;
}
else if ( c && !a && !b ) {
   category = 3;
}
else {
   category = 0;
}
```



Logic

Substitute Table Lookups for Complicated Expressions

You can replace this test with a more modifiable and higher-performance lookup table:

This table definition is somewhat difficult to understand. Any commenting you can do to make table definitions readable helps.

C++ Example of Using a Table Lookup to Replace Complicated Logic

```
// define categoryTable
static int categoryTable[ 2 ][ 2 ][ 2 ] = {
   // !b!c  !bc  b!c  bc
        0,   3,   2,   2,   //   !a
        1,   2,   1,   1    //    a
};
...
category = categoryTable[ a ][ b ][ c ];
```

Logic

Substitute Table Lookups for Complicated Expressions

Although the definition of the table **is hard to read**, if it’s well documented it won’t be any harder to read than the code for the complicated chain of logic was. If the definition changes, the table will be much easier to maintain than the earlier logic would have been.

Language	Straight Time	Code-Tuned Time	Time Savings
C++	5.04	3.39	33%
Visual Basic	5.21	2.60	50%

Logic

Use Lazy Evaluation

- If a program uses lazy evaluation, it avoids doing any work until the work is needed.
- For example, a program contains a table of 5000 values, generates the whole table at startup time, and then uses it as the program executes.
- If the program uses only a small percentage of the entries in the table, it might make more sense to compute them as they're needed rather than all at once.
- Once an entry is computed, it can still be stored for future reference (otherwise known as "cached").

Loops

- Because loops are executed many times, the **hot spots** in a program are often inside loops.
- The techniques in this section make the loop itself faster.

Loops

- **Any possible issues in terms of performance?**

```
for ( i = 0; i < count; i++ ) {
    if ( sumType == SUMTYPE_NET ) {
        netSum = netSum + amount[ i ];
    }
    else {
        grossSum = grossSum + amount[ i ];
    }
}
```

Loops – Unswitching

- Switching *refers to making a decision inside a loop every time it's executed*. If the decision doesn't change while the loop is executing, you can unswitch the loop by making the decision outside the loop.
- Usually this requires turning the loop inside out, putting loops inside the conditional rather than putting the conditional inside the loop.

C++ Example of a Switched Loop

```
for ( i = 0; i < count; i++ ) {
   if ( sumType == SUMTYPE_NET ) {
      netSum = netSum + amount[ i ];
   }
   else {
      grossSum = grossSum + amount[ i ];
   }
}
```

C++ Example of an Unswitched Loop

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount[ i ];
   }
}
```

Loops – Unswitching

- Good code?
- This code fragment violates several rules of good programming.
- Readability and maintenance are usually more important than execution speed or size, but the current topic is performance, and that implies a tradeoff with the other objectives

C++ Example of an Unswitched Loop

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount[ i ];
   }
}
```

Loops – Unswitching

This is good for about a 20 percent time savings:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**2.81**	**2.27**	**19%**
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

Loops – Unswitching

- Also, **the case is that the two loops have to be maintained in parallel**.
- If *count* changes to *clientCount*, you have to remember to change it in both places, which is an annoyance for you and a maintenance headache for anyone else who has to work with the code.

C++ Example of an Unswitched Loop

```
if ( sumType == SUMTYPE_NET ) {
   for ( i = 0; i < count; i++ ) {
      netSum = netSum + amount[ i ];
   }
}
else {
   for ( i = 0; i < count; i++ ) {
      grossSum = grossSum + amount[ i ];
   }
}
```

Loops – Jamming

- **Jamming**, or "**fusion**," is the result of **combining two loops** that operate on the same set of elements. The gain lies in cutting the loop overhead from two loops to one.
- Here's a candidate for loop jamming:

Visual Basic Example of Separate Loops That Could Be Jammed

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
Next
...
For i = 0 to employeeCount - 1
   employeeEarnings( i ) = 0
Next
```

Loops – Jamming

- When you jam loops, you find code in two loops that you ca combine into one.
- Usually, that means the loop counters have to be the same. In this example, both loops run ***from 0 to employeeCount - 1***, so you can **jam** them:

Visual Basic Example of a Jammed Loop

```
For i = 0 to employeeCount - 1
   employeeName( i ) = ""
   employeeEarnings( i ) = 0
Next
```

Loops – Jamming

- Here are the savings:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	3.68	2.65	28%
PHP	3.97	2.42	32%
Visual Basic	**3.75**	**3.56**	**4%**

Note: Benchmarked for the case in which *employeeCount* equals 100.

- As before, the results vary significantly among languages.

Loops – Unrolling

- **The goal of loop unrolling is to reduce the amount of loop iterations.**
- Although completely unrolling a loop is a fast solution and works well when you're dealing with a small number of elements, it's not practical when you have a large number of elements or when you don't know in advance how many elements you'll have.

Java Example of a Loop That Can Be Unrolled

```
i = 0;
while ( i < count ) {
   a[ i ] = i;
   i = i + 1;
}
```

Loops – Unrolling

- To unroll the loop partially, you handle two or more cases in each pass through the loop instead of one.
- This unrolling hurts readability but doesn't hurt the generality of the loop. Here's the loop unrolled once:

Java Example of a Loop That's Been Unrolled Once

```
i = 0;
while ( i < count - 1 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   i = i + 2;
}

if ( i == count - 1) {
   a[ count - 1 ] = count - 1;
}
```

These lines pick up the case that might fall through the cracks if the loop went by twos instead of by ones.

Loops – Unrolling

- The technique **replaced the original *a[i] = i* line with two lines, and *I* is incremented by *2* rather than by *1***. The extra code after the *while* loop is needed when *count* is odd and the loop has one iteration left after the loop terminates.

Java Example of a Loop That's Been Unrolled Once

```
i = 0;
while ( i < count - 1 ) {
   a[ i ] = i;
   a[ i + 1 ] = i + 1;
   i = i + 2;
}

if ( i == count - 1) {
   a[ count - 1 ] = count - 1;
}
```

These lines pick up the case that might fall through the cracks if the loop went by twos instead of by ones.

Loops – Unrolling

- A gain of 16 to 43 percent is respectable, although Python benchmark shows performance loss.
- The main hazard of loop unrolling is an off-by-one error in the code after the loop that picks up the last case.

Language	Straight Time	Code-Tuned Time	Time Savings
C++	1.75	1.15	34%
Java	**1.01**	**0.581**	**43%**
PHP	5.33	4.49	16%
Python	2.51	3.21	-27%

Note: Benchmarked for the case in which *count* equals 100.

Loops – Unrolling

- What if you unroll the loop even further, going for two or more unrollings? Do you get more benefit if you unroll a loop twice?

Java Example of a Loop That's Been Unrolled Twice

```java
i = 0;
while ( i < count - 2 ) {
   a[ i ] = i;
   a[ i + 1 ] = i+1;
   a[ i + 2 ] = i+2;
   i = i + 3;
}
if ( i <= count - 1 ) {
    a[ count - 1 ] = count - 1;
}
if ( i == count - 2 ) {
    a[ count -2 ] = count - 2;
}
```

Loops – Unrolling

Language	Straight Time	Single Unrolling Time	Time Savings
C++	1.75	1.15	34%
Java	**1.01**	**0.581**	**43%**
PHP	5.33	4.49	16%
Python	2.51	3.21	-27%

Note: Benchmarked for the case in which *count* equals 100.

- Here are the results of unrolling the loop the second time:

Language	Straight Time	Double Unrolled Time	Time Savings
C++	1.75	1.01	42%
Java	**1.01**	**0.581**	**43%**
PHP	5.33	3.70	31%
Python	2.51	2.79	-12%

Note: Benchmarked for the case in which count equals 100.

- The results indicate that **further loop unrolling can result in further time savings, but not necessarily** so, as the Java measurement shows.

Loops – Unrolling

- When you look at the previous code, you might not think it looks incredibly complicated, but when you see the performance gain, you can appreciate the tradeoff between performance and readability.

Loops – *Minimizing the Work Inside Loops*

- One key to writing effective loops is to minimize the work done inside a loop.
- If you can evaluate a statement or part of a statement outside a loop so that only the result is used inside the loop, do so.
- It’s good programming practice, and **in some cases it improves readability**.

Loops – *Minimizing the Work Inside Loops*

- Suppose you have a complicated pointer expression inside a loop:

C++ Example of a Complicated Pointer Expression Inside a Loop

```
for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * rates->discounts->factors->net;
}
```

Loops – *Minimizing the Work Inside Loops*

- In this case, assigning the complicated pointer expression to a well-named variable improves readability and often improves performance.

C++ Example of Simplifying a Complicated Pointer Expression

```
quantityDiscount = rates->discounts->factors->net;
for ( i = 0; i < rateCount; i++ ) {
   netRate[ i ] = baseRate[ i ] * quantityDiscount;
}
```

Loops – *Minimizing the Work Inside Loops*

- The extra variable, *quantityDiscount*, makes it clear that the *baseRate* array is being multiplied by a quantity-discount factor to compute the net rate.
- That wasn't at all clear from the original expression in the loop.
- Putting the complicated pointer expression into a variable outside the loop also saves the pointer accesses for each pass through the loop, resulting in the following savings:

Loops – *Sentinel Values*

Sentinel value

From Wikipedia, the free encyclopedia

Not to be confused with sentinel node.

In computer programming, a **sentinel value** (also referred to as a **flag value**, **trip value**, **rogue value**, **signal value**, or **dummy data**)[1] is a special value in the context of an algorithm which uses its presence as a condition of termination, typically in a loop or recursive algorithm.

Loops – *Sentinel Values*

- **Anything wrong?**

```
found = FALSE;
i = 0;
while ( ( !found ) && ( i < count ) ) {
    if ( item[ i ] == testValue ) {
        found = TRUE;
    }
    else {
        i++;
    }
}

if ( found ) {
    ...
```

Loops – *Sentinel Values*

- In this code, each iteration of the loop tests for *!found* and for *i < count*.
- The purpose of the *!found* test is to determine when the desired element has been found.
- The purpose of the *i < count* test is to avoid running past the end of the array. Inside the loop, each value of *item[]* is tested individually, so the loop really has three tests for each iteration.

C# Example of Compound Tests in a Search Loop

Here's the compound test.

```
found = FALSE;
i = 0;
while ( ( !found ) && ( i < count ) ) {
   if ( item[ i ] == testValue ) {
      found = TRUE;
   }
   else {
      i++;
   }
}

if ( found ) {
   ...
```

Loops – *Sentinel Values*

- In this kind of search loop, **you can combine the three tests so that you test only once per iteration by putting a “sentinel” at the end of the search range to stop the loop**.
- In this case, you can simply assign the value you’re looking for to the element just beyond the end of the search range. (Remember to leave space for that element when you declare the array.)
- You then check each element, and if you don’t find the element until you find the one you stuck at the end, you know that the value you’re looking for isn’t really there.

Loops – *Sentinel Values*

Remember to allow space for the sentinel value at the end of the array.

C# Example of Using a Sentinel Value to Speed Up a Loop

```
// set sentinel value, preserving the original value
initialValue = item[ count ];
item[ count ] = testValue;

i = 0;
while ( item[ i ] != testValue ) {
   i++;
}

// check if value was found
if ( i < count ) {
   ...
```

Loops – *Sentinel Values*

- When *item* is an array of **integers**, the savings can be dramatic:

Language	Straight Time	Code-Tuned Time	Time Savings
C#	**0.771**	**0.590**	**23%**
Java	1.63	0.912	44%
Visual Basic	1.34	0.470	65%

Note: Search is of a 100-element array of integers.

Loops – *Sentinel Values*

- The Visual Basic results are particularly dramatic, but all the results are good. *When the kind of array changes, however, the results also change*.
- When *item* is an array of single-precision floating-point numbers, the results are as follows:

Language	Straight Time	Code-Tuned Time	Time Savings
C#	**1.351**	**1.021**	**24%**
Java	1.923	1.282	33%
Visual Basic	1.752	1.011	42%

Note: Search is of a 100-element array of 4-byte floating-point numbers.

Loops

- The total number of loop executions?

```
for ( column = 0; column < 100; column++ ) {
    for ( row = 0; row < 5; row++ ) {
        sum = sum + table[ row ][ column ];
    }
}
```

Loops – *Putting the Busiest Loop on the Inside*

- When you have nested loops, think about which loop you want on the outside and which you want on the inside. Following is an example of a nested loop that can be improved:

Java Example of a Nested Loop That Can Be Improved

```java
for ( column = 0; column < 100; column++ ) {
   for ( row = 0; row < 5; row++ ) {
      sum = sum + table[ row ][ column ];
   }
}
```

Loops – *Putting the Busiest Loop on the Inside*

- The key to improving the loop is that the outer loop executes much more often than the inner loop.
- Each time the loop executes, it has to initialize the loop index, increment it on each pass through the loop, and check it after each pass.

Language	Straight Time	Code-Tuned Time	Time Savings
C++	4.75	3.19	33%
Java	**5.39**	**3.56**	**34%**
PHP	4.16	3.65	12%
Python	3.48	3.33	4%

Loops

- Any comments on the performance?
- How can we run it faster?

Visual Basic Example of Multiplying a Loop Index

```
For i = 0 to saleCount - 1
   commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

Loops – *Strength Reduction*

- Reducing strength means replacing an expensive operation such as multiplication with a cheaper operation such as addition.
- Sometimes you'll have an expression inside a loop that depends on multiplying the loop index by a factor.
- Addition is usually faster than multiplication, and if you can compute the same number by adding the amount on each iteration of the loop rather than by multiplying, the code will typically run faster.

Loops – *Strength Reduction*

Visual Basic Example of Multiplying a Loop Index

```
For i = 0 to saleCount - 1
   commission( i ) = (i + 1) * revenue * baseCommission * discount
Next
```

⬇

Visual Basic Example of Adding Rather Than Multiplying

```
incrementalCommission = revenue * baseCommission * discount
cumulativeCommission = incrementalCommission
For i = 0 to saleCount - 1
   commission( i ) = cumulativeCommission
   cumulativeCommission = cumulativeCommission + incrementalCommission
Next
```

Loops – *Strength Reduction*

- The key is that the original multiplication has to depend on the loop index. In this case, the loop index was the only part of the expression that varied, so the expression could be recoded more economically.

Language	Straight Time	Code-Tuned Time	Time Savings
C++	4.33	3.80	12%
Visual Basic	**3.54**	**1.80**	**49%**

Note: Benchmark performed with *saleCount* equals 20. All computed variables are floating point.

Data Transformations

- Changes in data types can be a powerful aid in reducing program size and improving execution speed.
- Data-structure design is outside the scope of this course, but modest changes in the implementation of a specific data type can also improve performance.
- Here are a few ways to tune your data types.

Data Transformations – *Integers over Floats*

- Integer addition and multiplication tend to be faster than floating point.
- Changing a loop index from a floating point to an integer, for example, can save time:

```
Dim x As Single
For x = 0 to 99
   a( x ) = 0
Next
```

```
Dim i As Integer
For i = 0 to 99
   a( i ) = 0
Next
```

Language	Straight Time	Code-Tuned Time	Time Savings
C++	2.80	0.801	71%
PHP	5.01	4.65	7%
Visual Basic	**6.84**	**0.280**	**96%**

Data Transformations

- How can we change and use this 2D array as 1D?

Java Example of a Standard, Two-Dimensional Array Initialization

```
for ( row = 0; row < numRows; row++ ) {
    for ( column = 0; column < numColumns; column++ ) {
        matrix[ row ][ column ] = 0;
    }
}
```

Data Transformations – *Fewer Array Dims*

- Multiple dimensions on arrays are expensive.
- If you can structure your data so that it's in a one-dimensional array rather than a two-dimensional or three-dimensional array, you might be able to save some time.
- Suppose you have initialization code like this:

Java Example of a Standard, Two-Dimensional Array Initialization

```
for ( row = 0; row < numRows; row++ ) {
   for ( column = 0; column < numColumns; column++ ) {
      matrix[ row ][ column ] = 0;
   }
}
```

Data Transformations – *Fewer Array Dims*

- When this code is run with 50 rows and 20 columns, it takes twice as long with a Java compiler as when the array is restructured so that it's one-dimensional.

Java Example of a One-Dimensional Representation of an Array

```
for ( entry = 0; entry < numRows * numColumns; entry++ ) {
   matrix[ entry ] = 0;
}
```

Data Transformations – *Fewer Array Dims*

- Here's a summary of the results, with the addition of comparable results in several other languages:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	8.75	7.82	11%
C#	3.28	2.99	9%
Java	**7.78**	**4.14**	**47%**
PHP	6.24	4.10	34%
Python	3.31	2.23	32%
Visual Basic	9.43	3.22	66%

Data Transformations – *Less Array Refs*

- In addition to minimizing accesses to doubly or triply dimensioned arrays, it’s often advantageous to minimize array accesses.
- A loop that repeatedly uses one element of an array is a goo candidate for the application of this technique.

C++ Example of Unnecessarily Referencing an Array Inside a Loop

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * discount[ discountType ];
   }
}
```

Data Transformations – *Less Array Refs*

- The reference to *discount[discountType]* doesn't change when *discountLevel* changes in the inner loop.
- Consequently, you can move it out of the inner loop so that you'll have only one array access per execution of the outer loop rather than one for each execution of the inner loop.

C++ Example of Moving an Array Reference Outside a Loop

```
for ( discountType = 0; discountType < typeCount; discountType++ ) {
   thisDiscount = discount[ discountType ];
   for ( discountLevel = 0; discountLevel < levelCount; discountLevel++ ) {
      rate[ discountLevel ] = rate[ discountLevel ] * thisDiscount;
   }
}
```

Data Transformations – *Less Array Refs*

- Results vary significantly from compiler to compiler.

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**32.1**	**34.5**	-7%
C#	18.3	17.0	7%
Visual Basic	23.2	18.4	20%

Note: Benchmark times were computed for the case in which *typeCount* equals 10 and *levelCount* equals 100.

Data Transformations – *Use Supplm Indexes*

- Using a supplementary index means adding related data that makes accessing a data type more efficient.
- You can add the related data to the main data type, or you can store it in a parallel structure

Data Transformations – *Use Supplm Indexes*

String-Length Index

- One example of using a supplementary index can be found in the different string-storage strategies.
- In C, strings are terminated by a byte that's set to 0.
  - To determine the length of a string in C, a program has to start at the beginning of the string and count each byte until it finds the byte that's set to 0.
- In Visual Basic string format, a length byte hidden at the beginning of each string indicates how long the string is.
  - To determine the length of a Visual Basic string, the program just looks at the length byte. Visual Basic length byte is an example of augmenting a data type with an index to make certain operations—like computing the length of a string—faster.

Data Transformations – *Use Supplm Indexes*

String-Length Index

- You can apply the idea of indexing for length to any variable-length data type.
- It’s often more efficient to keep track of the length of the structure rather than computing the length each time you need it.

Data Transformations – *Use Caching*

- Caching means saving a few values in such a way that you can retrieve the most commonly used values more easily than the less commonly used values.
- If a program randomly reads records from a disk, for example, a routine might use a cache to save the records read most frequently.
- When the routine receives a request for a record, it checks the cache to see whether it has the record. If it does, the record is returned directly from memory rather than from disk.

Data Transformations – *Use Caching*

- In addition to caching records on disk, you can apply caching in other areas.
- In a Microsoft Windows font-proofing program, the performance bottleneck was in retrieving the width of each character as it was displayed.
- Caching the most recently used character width roughly doubled the display speed

Data Transformations – *Use Caching*

- You can cache the results of time-consuming computations too—especially if the parameters to the calculation are simple.
- Suppose, for example, that you need to compute the length of the hypotenuse of a right triangle, given the lengths of the other two sides. The straightforward implementation:

Java Example of a Routine That's Conducive to Caching

```
double Hypotenuse(
   double sideA,
   double sideB
   ) {
   return Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
}
```

Data Transformations – *Use Caching*

- If you know that the same values tend to be requested repeatedly, you can cache values this way:

Java Example of Caching to Avoid an Expensive Computation

```java
private double cachedHypotenuse = 0;
private double cachedSideA = 0;
private double cachedSideB = 0;

public double Hypotenuse(
   double sideA,
   double sideB
   ) {

   // check to see if the triangle is already in the cache
   if ( ( sideA == cachedSideA ) && ( sideB == cachedSideB ) ) {
      return cachedHypotenuse;
   }

   // compute new hypotenuse and cache it
   cachedHypotenuse = Math.sqrt( ( sideA * sideA ) + ( sideB * sideB ) );
   cachedSideA = sideA;
   cachedSideB = sideB;

   return cachedHypotenuse;
}
```

Data Transformations – *Use Caching*

- The second version of the routine is more complicated than the first and takes up more space, so speed has to be at a premium to justify it. Many caching schemes cache more than one element, so they have even more overhead. Here's the speed difference:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	4.06	1.05	74%
Java	**2.54**	**1.40**	**45%**
Python	8.16	4.17	49%
Visual Basic	24.0	12.9	47%

Note: The results shown assume that the cache is hit twice for each time it's set.

Data Transformations – *Use Caching*

- The success of the cache depends on the relative costs of accessing a cached element, creating an uncached element, and saving a new element in the cache.
- Success also depends on how often the cached information is requested. In some cases, success might also depend on caching done by the hardware.
- Generally, the more it costs to generate a new element and the more times the same information is requested, the more valuable a cache is. The cheaper it is to access a cached element and save new elements in the cache, the more valuable a cache is.
- As with other optimization techniques, caching adds complexity and tends to be error-prone.

Expressions

- Much of the work in a program is done inside mathematical or logical expressions.
- Complicated expressions tend to be expensive, so this section looks at ways to make them cheaper.

Expressions – *Exploit Algebraic Identities*

- You can use algebraic identities to replace costly operations with cheaper ones.
- For example, the following expressions are logically equivalent:

```
not a and not b
not (a or b)
```

- If you choose the second expression instead of the first, you can save a *not* operation.
- Although the savings from avoiding a single *not* operation are probably inconsequential, the general principle is powerful.

Expressions – *Exploit Algebraic Identities*

- For example, a program on whether *sqrt(x) < sqrt(y).* Since *sqrt(x)* is less than *sqrt(y)* only when *x* is less than *y*, you can replace the first test with *x < y*.
- Given the cost of the *sqrt()* routine, you'd expect the savings to be dramatic, and they are. Here are the results:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	7.43	0.010	99.9%
Visual Basic	4.59	0.220	95%
Python	4.21	0.401	90%

Expressions – *Use Strength Reduction*

- Strength reduction means replacing an expensive operation with a cheaper one. Here are some possible substitutions:
  - Replace multiplication with addition.
  - Replace exponentiation with multiplication.
  - Replace floating-point numbers with fixed-point numbers or integers.
  - Replace double-precision floating points with single-precision numbers.
  - Replace integer multiplication-by-two and division-by-two with shift operations.

Expressions – *Use Strength Reduction*

- Suppose you have to evaluate a polynomial. If you're rusty on polynomials, they're the things that look like $Ax^2 + Bx + C$.
- The letters *A*, *B*, and *C* are coefficients, and *x* is a variable. General code to evaluate an *n*th-order polynomial looks like this:

Visual Basic Example of Evaluating a Polynomial

```
value = coefficient( 0 )
For power = 1 To order
   value = value + coefficient( power ) * x^power
Next
```

Expressions – *Use Strength Reduction*

- One solution would be to replace the exponentiation with a multiplication on each pass through the loop, which is analogous to the strength-reduction case a few sections ago in which a multiplication was replaced with an addition.

Visual Basic Example of a Reduced-Strength Method of Evaluating a Polynomial

```
value = coefficient( 0 )
powerOfX = x
For power = 1 to order
   value = value + coefficient( power ) * powerOfX
   powerOfX = powerOfX * x
Next
```

Expressions – *Use Strength Reduction*

- This produces a noticeable advantage if you’re working with second-order polynomials—that is, polynomials in which the highest-power term is squared—or higher-order polynomials:

Language	Straight Time	Code-Tuned Time	Time Savings
Python	3.24	2.60	20%
Visual Basic	**6.26**	**0.160**	**97%**

Expressions

- Compute the base-two logarithm of an integer, truncated to the nearest integer.

```
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / log( 2 ) );
}
```

- **Any suggestion to improve its performance?**

Expressions – *Initialize at Compile Time*

- If you're using a named **constant** or a magic number in a routine call and it's the only argument, that's a clue that you could **precompute** the number, put it into a constant, and avoid the routine call.
- The same principle applies to multiplications, divisions, additions, and other operations.
- For example, compute the base-two logarithm of an integer, truncated to the nearest integer. If the system doesn't have a log- base-two routine, a quick and easy approach:

C++ Example of a Log-Base-Two Routine Based on System Routines

```
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / log( 2 ) );
}
```

Expressions – *Initialize at Compile Time*

- This routine is very slow, and because the value of *log(2)* never changed, replace *log(2)* with its computed value, *0.69314718*, like this:

C++ Example of a Log-Base-Two Routine Based on a System Routine and a Constant

```
const double LOG2 = 0.69314718;
...
unsigned int Log2( unsigned int x ) {
   return (unsigned int) ( log( x ) / LOG2 );
}
```

Expressions – *Initialize at Compile Time*

- Since *log()* tends to be an expensive routine—much more expensive than type conversions or division—you'd expect that cutting the calls to the *log()* function by half would cut the time required for the routine by about half.

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**9.66**	**5.97**	**38%**
Java	17.0	12.3	28%
PHP	2.45	1.50	39%

Expressions – *Be Wary of System Routines*

- System routines are expensive and provide accuracy that's often wasted.
- Typical system math routines, <u>for example</u>, are designed to put an astronaut on the moon within ±2 feet of the target. If you don't need that degree of accuracy, you don't need to spend the time to compute it either.

Expressions – *Be Wary of System Routines*

- In the previous example, the *Log2()* routine returned an integer value but used a floating-point *log()* routine to compute it.
- That was problematic for an integer result, so write a series of integer tests that were perfectly accurate for calculating an integer log2.

C++ Example of a Log-Base-Two Routine Based on Integers

```
unsigned int Log2( unsigned int x ) {
   if ( x < 2 ) return 0 ;
   if ( x < 4 ) return 1 ;
   if ( x < 8 ) return 2 ;
   if ( x < 16 ) return 3 ;
   if ( x < 32 ) return 4 ;
   if ( x < 64 ) return 5 ;
   if ( x < 128 ) return 6 ;
   if ( x < 256 ) return 7 ;
   if ( x < 512 ) return 8 ;
   if ( x < 1024 ) return 9 ;
   ...
   if ( x < 2147483648 ) return 30;
   return 31 ;
}
```

Expressions – *Be Wary of System Routines*

- This routine uses integer operations, never converts to floating point, and blows the doors off both floating-point versions:

Language	Straight Time	Code-Tuned Time	Time Savings
C++	9.66	0.662	93%
Java	17.0	0.882	95%
PHP	2.45	3.45	-41%

Expressions – *Be Wary of System Routines*

- Another option is to take advantage of the fact that a right-shift operation is the same as dividing by two.
- The number of times you can divide a number by two and still have a nonzero value is the same as the log2 of that number.

C++ Example of an Alternative Log-Base-Two Routine Based on the Right-Shift Operator

```
unsigned int Log2( unsigned int x ) {
   unsigned int i = 0;
   while ( ( x = ( x >> 1 ) ) != 0 ) {
      i++;
   }
   return i ;
}
```

Expressions – *Be Wary of System Routines*

- To non-C++ programmers, this code is particularly hard to read. The complicated expression in the *while* condition is an example of a coding practice you should avoid unless you have a good reason to use it.
- This example highlights the value of not stopping after one successful optimization. The first optimization earned a respectable 30–40 percent savings but had nowhere near the impact of the second or third optimizations.

Expressions – *Precompute Results*

- A common low-level design decision is the choice of whether to compute results on the fly or compute them once, save them, and look them up as needed.
- If the results are used many times, it's often cheaper to compute them once and look them up the rest of the time.

Expressions – *Precompute Results*

- At the simplest level, you might compute part of an expression outside a loop rather than inside.
- At a more complicated level, you might compute a **lookup table** once when program execution begins, using it every time thereafter, or you might store results in a data file or embed them in a program.

Expressions – *Precompute Results*

- **Any performance improvement suggestion?**

```
double ComputePayments(
   int months,
   double interestRate
   ) {
   for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount < MAX_LOAN_AMOUNT;
      loanAmount++ ) {
      payment = loanAmount / (
         ( 1.0 - Math.pow( 1.0+(interestRate/12.0), - months ) ) /
         ( interestRate/12.0 )
         );
      ...
   }
}
```

Expressions – *Precompute Results*

Java Example of Precomputing the Second Complex Computation

```
double ComputePayments(
   int months,
   double interestRate
   ) {
   long loanAmount;
   double divisor = ( 1.0 - Math.pow( 1.0+(interestRate/12.0). - months ) ) /
      ( interestRate/12.0 );
   for ( long loanAmount = MIN_LOAN_AMOUNT; loanAmount <= MAX_LOAN_AMOUNT;
      loanAmount++ ) {
      payment = loanAmount / divisor;
      ...
   }
}
```

Expressions – *Precompute Results*

- This is similar to the techniques suggested earlier of putting array references and pointer dereferences outside a loop.
- The results for Java in this case are comparable to the results of using the precomputed table in the first optimization:

Language	Straight Time	Code-Tuned Time	Time Savings
Java	**7.43**	**0.24**	**97%**
Python	5.00	1.69	66%

Expressions – *Precompute Results*

- Optimizing a program by pre-computation can take several forms:
  - Computing results before the program executes, and wiring them into constants that are assigned at compile time
  - Computing results before the program executes, and hard-coding them into variables used at run time
  - Computing results before the program executes, and putting them into a file that's loaded at run time
  - Computing results once, at program startup, and then referencing them each time they're needed
  - Computing as much as possible before a loop begins, minimizing the work done inside the loop
  - Computing results the first time they're needed, and storing them so that you can retrieve them when they're needed again

Expressions – *Eliminate Common Subexpressions*

- If you find an expression that's repeated several times, assign it to a variable and refer to the variable rather than recomputing the expression in several places.
- The loan-calculation example has a common subexpression that you could eliminate. This is the original code:

Java Example of a Common Subexpression

```java
payment = loanAmount / (
      ( 1.0 - Math.pow( 1.0 + ( interestRate / 12.0 ), -months ) ) /
      ( interestRate / 12.0 )
   );
```

Expressions – *Eliminate Common Subexpressions*

- You can assign *interestRate/12.0* to a variable that is then referenced twice rather than computing the expression twice.
- If you have chosen the variable name well, this optimization can *improve the code's readability at the same time that it improves performance*.

Java Example of Eliminating a Common Subexpression

```
monthlyInterest = interestRate / 12.0;
payment = loanAmount / (
      ( 1.0 - Math.pow( 1.0 + monthlyInterest, -months ) ) /
      monthlyInterest
   );
```

Expressions – *Eliminate Common Subexpressions*

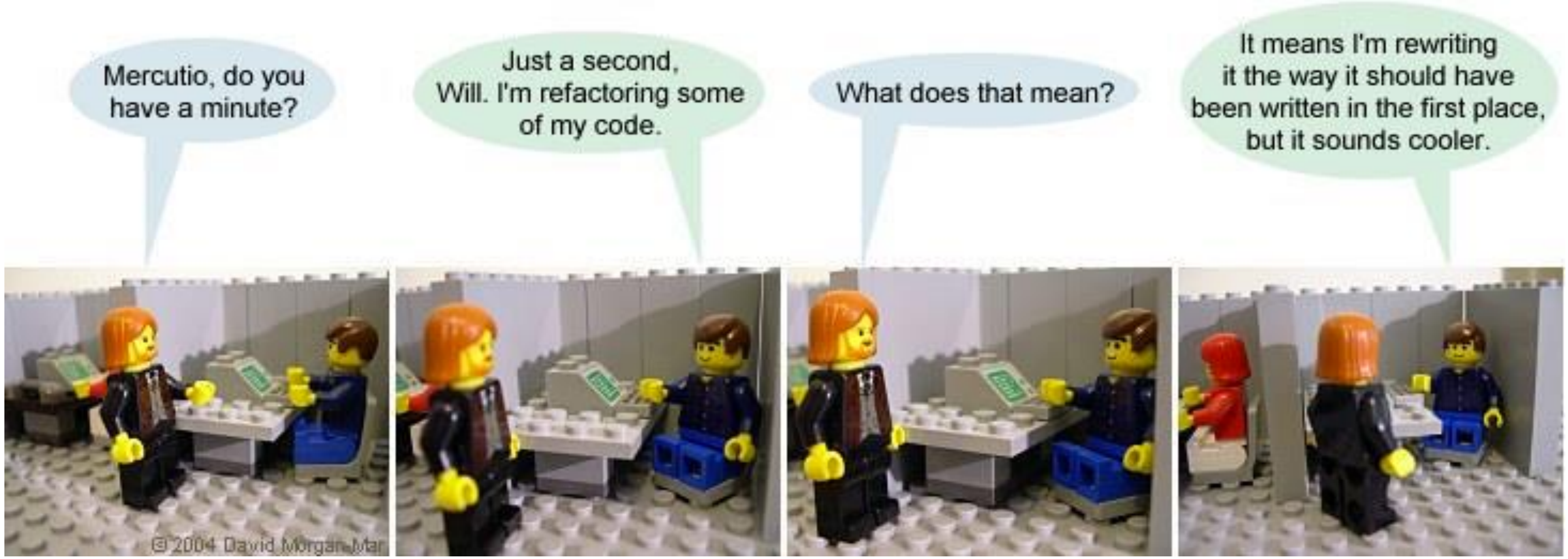
- The savings in this case don't seem impressive:

Language	Straight Time	Code-Tuned Time	Time Savings
Java	**2.94**	**2.83**	**4%**
Python	3.91	3.94	-1%

Code Refactoring

Refactoring

- Modifying software to improve its readability, maintainability, and extensibility *without changing what it actually does*.
  - External behavior does NOT change
  - Internal structure is improved



Refactoring

- It is a disciplined way to clean up code that minimizes the chances of introducing bugs.
- In essence when you refactor you are improving the design of the code after it has been written.
  - In software development, <u>we design first then we code</u>
  - Refactoring is the opposite of this practice: take a bad design, and rework it into well-designed code

Refactoring

- Each step is simple
  - *move* a field from one class to another,
  - *pull* some code out of a method to make into its own method, and
  - *push* some code up or down a hierarchy
- Yet the cumulative effect of these small changes can radically improve the design.

Composing Methods – *Extract Method*

- You have a code fragment that can be grouped together.
- Turn the fragment into a method whose name explains the purpose of the method.

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

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

→

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

Composing Methods – *Extract Method*

Motivation

- Method is too long or code that needs a comment to understand its purpose. Then turn that fragment of code into its own method.
- Prefer short, well-named methods for several reasons:
  - First, it increases the chances that other methods can use a method when the method is finely grained.
  - Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained.

Composing Methods – *Extract Method*

Mechanics

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).
- Copy the extracted code from the source method into the new target method.
- Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.

Composing Methods – *Extract Method*

Mechanics (cont.)

- See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
- Look to see whether any of these local-scope variables are modified by the extracted code.
  - If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned.

Composing Methods – *Extract Method*

Mechanics (cont.)

- Pass into the target method as parameters local-scope variables that are read from the extracted code.
- Replace the extracted code in the source method with a call to the target method

Composing Methods – *Extract Method*

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

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

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Composing Methods – *Extract Method*

- **Example: No Local Variables**
- Extract the code that prints the banner. Just cut, paste, and put a call:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

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

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

```
void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
}
```

Composing Methods – *Extract Method*

Example: Using Local Variables

- The problem is local variables: parameters passed into the original method and temporaries declared within the original method.
- The easiest case with local variables is when the variables are read but not changed.
  - In this case, can just pass them as parameters

Composing Methods – *Extract Method*

Example: Using Local Variables

- Extract the printing of details with a method with one parameter:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

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

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Composing Methods – *Extract Method*

Example: Using Local Variables

- extract the printing of details with a method with one parameter:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

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

    printDetails(outstanding);
}
```

```
void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Composing Methods – *Extract Method*

Example: Reassigning a Local Variable

- It's the assignment to local variables that becomes complicated. In this case we're only talking about temps.
- For temps that are assigned to, there are two cases:
  - The simpler case is that in which the variable is a temporary variable used only within the extracted code. When that happens, you can move the temp into the extracted code.
  - The other case is use of the variable outside the code. If the variable is not used after the code is extracted, you can make the change in just the extracted code.

Composing Methods – *Extract Method*

Example: Reassigning a Local Variable

- If it is used afterward, you need to make the extracted code return the changed value of the variable.

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

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

    printDetails(outstanding);
}
```

Composing Methods – *Extract Method*

Example: Reassigning a Local Variable

- If it is used afterward, you need to make the extracted code return the changed value of the variable.
- The enumeration variable is used only in the extracted code, so I can move it entirely within the new method.

```
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;
}
```

Composing Methods – Extract Method

Example: Reassigning a Local Variable

- Rename the returned value if required:

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

Composing Methods – *Extract Method*

Example: Reassigning a Local Variable

- If something more involved happens to the variable, have to pass in the previous value as a parameter.

```
void printOwing(double previousAmount) {

    Enumeration e =  orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

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

    printDetails(outstanding);
}
```

Composing Methods – *Extract Method*

Example: Reassigning a Local Variable

- In this case, the extraction would look like this:

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

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

Composing Methods – *Replace Temp with Query*

- You are using a temporary variable to hold the result of an expression.
- Extract the expression into a method.
- Replace all references to the temp with the expression. The new method can then be used in other methods.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

```
    if (basePrice() > 1000)
        return basePrice() * 0.95;
    else
        return basePrice() * 0.98;

  double basePrice() {
      return _quantity * _itemPrice;
  }
```

Composing Methods – *Replace Temp with Query*

Motivation

- The problem with temps is that they are temporary and local. Because they can be seen only in the context of the method in which they are used, temps tend to encourage longer methods, because that's the only way you can reach the temp.
- By replacing the temp with a query method, any method in the class can get at the information. That helps a lot in coming up with cleaner code for the class.

Composing Methods – *Replace Temp with Query*

Mechanics

- Look for a temporary variable that is assigned to once.
  - If a temp is set more than once consider Split Temporary Variable
- Declare the temp as final.
  - This will ensure that the temp is only assigned to once
- Extract the right-hand side of the assignment into a method.
  - Initially mark the method as private. You may find more use for it later, but you can easily relax the protection later.

Composing Methods – *Replace Temp with Query*

- **Example:** Start with a simple method

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Composing Methods – *Replace Temp with Query*

Example

- I'm inclined to replace both temps, one at a time.
- Although it's pretty clear in this case, I can test that they are assigned only to once by declaring them as final

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Composing Methods – *Replace Temp with Query*

Example

- Compiling will then alert me to any problems. I do this first, because if there is a problem, I shouldn't be doing this refactoring.
- I replace the temps one at a time. First I extract the right-hand side of the assignment:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;

    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}

private int basePrice() {
return _quantity * _itemPrice;
}
```

Composing Methods – *Replace Temp with Query*

Example

- First I replace the first reference to the temp:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Composing Methods – *Replace Temp with Query*

Example

- Do the next. Also remove the temp declaration:

```
double getPrice() {
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice() * discountFactor;
}
```

Composing Methods – *Replace Temp with Query*

Example

- With that gone, can extract discountFactor in a similar way:

```
double getPrice() {
    final double discountFactor = discountFactor();
    return basePrice() * discountFactor;
}

private double discountFactor() {
    if (basePrice() > 1000) return 0.95;
    else return 0.98;
}
```

Composing Methods – *Replace Temp with Query*

- **Example**
- See how it would have been difficult to extract discountFactor if I had not replaced basePrice with a query.
- The getPrice method ends up as follows:

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

➡

```
double getPrice() {
    return basePrice() * discountFactor();
}
```

Composing Methods – *Introduce Explaining Variable*

- You have a complicated expression. Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
 if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
      (browser.toUpperCase().indexOf("IE") > -1) &&
       wasInitialized() && resize > 0 )
{
      // do something
```

```
    final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") >
-1;
    final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  >
-1;
    final boolean wasResized  = resize > 0;

    if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
        // do something
    }
```

Composing Methods – *Introduce Explaining Variable*

Motivation

- Expressions can become very complex and hard to read.
- In such situations temporary variables can be helpful to break down the expression into something more manageable.

Composing Methods – *Introduce Explaining Variable*

Mechanics

- Declare a final temporary variable, and set it to the result of part of the complex expression.
- Replace the result part of the expression with the value of the temp.
  - If the result part of the expression is repeated, you can replace the repeats one at a time.
- Repeat for other parts of the expression.

Composing Methods – *Introduce Explaining Variable*

Example

- Start with a simple calculation:

```
double price() {
// price is base price - quantity discount + shipping
return _quantity * _itemPrice -
    Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
    Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Composing Methods – *Introduce Explaining Variable*

Example

- Simple: it may be, but can make it easier to follow.
- First I identify the base price as the quantity times the item price. I can turn that part of the calculation into a temp:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Composing Methods – *Introduce Explaining Variable*

Example

- Quantity times item price is also used later, so can substitute with the temp there as well:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(basePrice * 0.1, 100.0);
}
```

Composing Methods – *Introduce Explaining Variable*

Example

- Next I take the quantity discount:

```
   double price() {
      // price is base price - quantity discount + shipping
      final double basePrice = _quantity * _itemPrice;
      final double quantityDiscount = Math.max(0, _quantity - 500) *
_itemPrice * 0.05;
      return basePrice - quantityDiscount +
         Math.min(basePrice * 0.1, 100.0);
   }
```

Composing Methods – *Introduce Explaining Variable*

Example

- Finally, I finish with the shipping. As do that, can remove the comment, too, because now it doesn't say anything the code doesn’t say:

```
   double price() {
      final double basePrice = _quantity * _itemPrice;
      final double quantityDiscount = Math.max(0, _quantity - 500) *
_itemPrice * 0.05;
      final double shipping = Math.min(basePrice * 0.1, 100.0);
      return basePrice - quantityDiscount + shipping;
   }
```

Composing Methods – *Introduce Explaining Variable*

Example with Extract Method

- Start again:

```
double price() {
   // price is base price - quantity discount + shipping
   return _quantity * _itemPrice -
      Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
      Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Composing Methods – *Introduce Explaining Variable*

Example with Extract Method

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}

private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}

private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}

private double basePrice() {
    return _quantity * _itemPrice;
}
```

Composing Methods – *Introduce Explaining Variable*

Example with Extract Method

- When to use Introduce Explaining Variable? The answer is when Extract Method is more effort.
- If I'm in an algorithm with a lot of local variables, I may not be able to easily use Extract Method. In this case I use Introduce Explaining Variable to help me understand what is going on.
- As the logic becomes less tangled, I can always use Replace Temp with Query later. The temp also is valuable if I end up having to use Replace Method with Method Object.

Composing Methods – *Split Temporary Variable*

- You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.
- Make a separate temporary variable for each assignment.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

⬇

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

Composing Methods – *Split Temporary Variable*

Motivation

- Temporary variables are made for various uses. Some of these uses naturally lead to the temp's being assigned to several times.
- Loop variables change for each run around a loop (such as the i in for (int i=0; i<10; i++). Collecting temporary variables collect together some value that is built up during the method.
- Many other temporaries are used to hold the result of a long-winded bit of code for easy reference later.
- These kinds of variables should be set only once. Otherwise, its purpose will be confusing and it will be error-prone.

Composing Methods – *Split Temporary Variable*

Mechanics

- Change the name of a temp at its declaration and its first assignment.
  - If the later assignments are of the form i = i + some expression, that indicates that it is a collecting temporary variable, so don't split it. The operator for a collecting temporary variable usually is addition, string concatenation, writing to a stream, or adding to a collection.
- Declare the new temp as final.
- Change all references of the temp up to its second assignment.
- Declare the temp at its second assignment
- Repeat in stages, each stage renaming at the declaration, and changing references until the next assignment.

Composing Methods – *Remove Assignments to Parameters*

- The code assigns to a parameter.
- Use a temporary variable instead.

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

Composing Methods – *Remove Assignments to Parameters*

Motivation

- If you pass in an object named foo, in the parameter, assigning to the parameter means to change foo to refer to a different object.
- The reason don't like this comes down to lack of clarity and to confusion between pass by value and pass by reference

```
  void aMethod(Object foo) {
    foo.modifyInSomeWay();          // that's OK
    foo = anotherObject;            // trouble and despair will follow
you
```

Composing Methods – *Remove Assignments to Parameters*

Mechanics

- Create a temporary variable for the parameter.
- Replace all references to the parameter, made after the assignment, to the temporary variable.
- Change the assignment to assign to the temporary variable.

Composing Methods – *Remove Assignments to Parameters*

Example

- Start with the following simple routine:

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

Composing Methods – *Remove Assignments to Parameters*

Example

- Replacing with a temp leads to

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

Composing Methods – *Remove Assignments to Parameters*

Example

- You can enforce this convention with the final keyword:

```
    int discount (final int inputVal, final int quantity, final int
yearToDate) {
        int result = inputVal;
        if (inputVal > 50) result -= 2;
        if (quantity > 100) result -= 1;
        if (yearToDate > 10000) result -= 4;
        return result;
    }
```

Composing Methods – *Replace Method with Method Object*

- 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.

```
class Order...
  double price() {
  double primaryBasePrice;
      double secondaryBasePrice;
      double tertiaryBasePrice;
      // long computation;
      ...
  }
```

Composing Methods – *Replace Method with Method Object*

Motivation

- The difficulty in decomposing a method lies in local variables. If they are rampant, decomposition can be difficult.
- Using Replace Temp with Query helps to reduce this burden, but occasionally you may find you cannot break down a method that needs breaking.
- In this case you reach deep into the tool bag and get out your method object

Composing Methods – *Replace Method with Method Object*

Mechanics

- Create a new class, name it after the method.
- Give the new class a final field for the object that hosted the original method (the source object) and a field for each temporary variable and each parameter in the method.
- Give the new class a constructor that takes the source object and each parameter.
- Give the new class a method named "compute."

Composing Methods – *Replace Method with Method Object*

Mechanics (cont.)

- Copy the body of the original method into compute. Use the source object field for any invocations of methods on the original object.
- Replace the old method with one that creates the new object and calls compute.
- Because all the local variables are now fields, you can freely decompose the method without having to pass any parameters.

Composing Methods – *Replace Method with Method Object*

- **Example**
- A proper example of this requires a long chapter, so showing this refactoring for a method that doesn't need it.

```
Class Account
  int gamma (int inputVal, int quantity, int yearToDate) {
      int importantValue1 = (inputVal * quantity) + delta();
      int importantValue2 = (inputVal * yearToDate) + 100;
      if ((yearToDate - importantValue1) > 100)
          importantValue2 -= 20;
      int importantValue3 = importantValue2 * 7;
      // and so on.
      return importantValue3 - 2 * importantValue1;
  }
```

Composing Methods – *Replace Method with Method Object*

Example

- To turn this into a method object, I begin by declaring a new class. I provide a final field for the original object and a field for each parameter and temporary variable in the method.

```
class Gamma...
  private final Account _account;
  private int inputVal;
  private int quantity;
  private int yearToDate;
  private int importantValue1;
  private int importantValue2;
  private int importantValue3;
```

Composing Methods – *Replace Method with Method Object*

Example

- Add a constructor:

```
    Gamma (Account source, int inputValArg, int quantityArg, int
yearToDateArg) {
        _account = source;
        inputVal = inputValArg;
        quantity = quantityArg;
        yearToDate = yearToDateArg;
    }
```

Composing Methods – *Replace Method with Method Object*

Example

- Now can move the original method over; need to modify any calls of features of account to use the _account field

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    if ((yearToDate - importantValue1) > 100)
        importantValue2 -= 20;
    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}
```

Simplifying Conditional Expressions – *Decompose Conditional*

- You have a complicated conditional (if-then-else) statement.
- Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Simplifying Conditional Expressions – *Decompose Conditional*

Motivation

- As with any large block of code, you can make your intention clearer by decomposing it and replacing chunks of code with a method call named after the intention of that block of code.
- With conditions you can receive further benefit by doing this for the conditional part and each of the alternatives.
- This way you highlight the condition and make it clearly what you are branching on.
- You also highlight the reason for the branching.

Simplifying Conditional Expressions – *Decompose Conditional*

Mechanics

- Extract the condition into its own method.
- Extract the then part and the else part into their own methods.

Simplifying Conditional Expressions – *Decompose Conditional*

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

```
    if (notSummer(date))
        charge = winterCharge(quantity);
    else charge = summerCharge (quantity);

private boolean notSummer(Date date) {
    return date.before (SUMMER_START) || date.after(SUMMER_END);
}

private double summerCharge(int quantity) {
    return quantity * _summerRate;
}

private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

Simplifying Conditional Expressions – *Consolidate Conditional Expression*

- You have a sequence of conditional tests with the same result.
- Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // compute the disability amount
```

Simplifying Conditional Expressions – *Consolidate Conditional Expression*

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

Simplifying Conditional Expressions – *Consolidate Conditional Expression*

```java
public double getAdjustedCapital() {
  double result = 0.0;
  if (_capital > 0.0) {
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
  }
  return result;
}
```

```java
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

つづく