Refactoring

Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Refactoring

M. Fowler, Refactoring: Improving the Design of Existing Code, 2000.

- ***with contributions by K. Beck, J. Brant, W. Opdyke, and D. Roberts***

A structured, disciplined method to rewrite or restructure existing code without changing its external behavior

- **applying small transformation steps**
- **combined with re-executing tests each step**

Minor Transformations

Refactoring is done by applying a series of standardized basic transformations

Each is usually a small change in a program's source code that either

- **preserves the behavior of the software**
- **does not modify its satisfaction of functional requirements**

Sample Refactorings

Refactoring	Description
Extract Method	**Transform a long method into a shorter one by factoring out a portion into a private helper method.**
Extract Constant	**Replace a literal constant with a constant variable.**
Introduce Explaining Variable • **specialization of Extract Local Variable**	**Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.**
Replace Constructor Call with Factory Method	**In Java, for example, replace using the new operator and constructor call with invoking a helper method that creates the object (hiding the details).**

Code Smells

Refactoring was inspired by the problem of “code smells”

- **a method may be very long**
- **a method may be a near duplicate of another nearby method**

Transform the code into a new form that behaves the same as before but that no longer "smells”

- **for a long module, one or more smaller modules may be extracted**
- **for duplicate routines, replace the duplication with one shared routine**

Failure to refactor can result in growing technical debt.

Benefits of Refactoring

Maintainability

- **code is easy to read**
- **intent of the author is easy to grasp**

Extensibility

- **easier to extend the capabilities of the application if it uses recognizable design patterns**
- **easier to modify if it provides flexibility where none before may have existed**

A Prerequisite for Refactoring

A solid test set of regression tests to verify that the refactoring has not broken any functionality

Automated

Related to continuous integration practice

The Refactoring Process

An iterative cycle

Make a single small behavior-preserving transformation

- **about 100 named refactorings**

Re-execute regression tests to ensure correctness (behavior remains the same)

Make another small transformation…

If at any point a test fails, the last small change is undone and repeated in a different way.

Alphabetical List of Refactorings

http://refactoring.com/catalog/index.html

Add Parameter
Change Bidirectional Association to Unidirectional
Change Reference to Value
Change Unidirectional Association to Bidirectional
Change Value to Reference
Collapse Hierarchy
Consolidate Conditional Expression
Consolidate Duplicate Conditional Fragments
Convert Dynamic to Static Construction
Convert Static to Dynamic Construction

Decompose Conditional
Duplicate Observed Data
Eliminate Inter-Entity Bean Communication
Encapsulate Collection
Encapsulate Downcast
Encapsulate Field
Extract Class
Extract Interface
Extract Method
Extract Package
Extract Subclass
Extract Superclass
Form Template Method
Hide Delegate
Hide Method

Hide presentation tier-specific details from the business tier
Inline Class
Inline Method
Inline Temp
Introduce A Controller
Introduce Assertion
Introduce Business Delegate
Introduce Explaining Variable
Introduce Foreign Method
Introduce Local Extension
Introduce Null Object
Introduce Parameter Object
Introduce Synchronizer Token

Localize Disparate Logic
Merge Session Beans
Move Business Logic to Session
Move Class
Move Field
Move Method
Parameterize Method
Preserve Whole Object
Pull Up Constructor Body
Pull Up Field
Pull Up Method
Push Down Field
Push Down Method

Reduce Scope of Variable
Refactor Architecture by Tiers
Remove Assignments to Parameters
Remove Control Flag
Remove Double Negative
Remove Middle Man
Remove Parameter
Remove Setting Method
Rename Method
Replace Array with Object
Replace Assignment with Initialization
Replace Conditional with Polymorphism
Replace Conditional with Visitor

Replace Constructor with Factory Method
Replace Data Value with Object
Replace Delegation with Inheritance
Replace Error Code with Exception
Replace Exception with Test
Replace Inheritance with Delegation
Replace Iteration with Recursion
Replace Magic Number with Symbolic Constant
Replace Method with Method Object
Replace Nested Conditional with Guard Clauses
Replace Parameter with Explicit Methods
Replace Parameter with Method
Replace Record with Data Class
Replace Recursion with Iteration
Replace Static Variable with Parameter

Replace Subclass with Fields
Replace Temp with Query
Replace Type Code with Class
Replace Type Code with State/Strategy
Replace Type Code with Subclasses
Reverse Conditional
Self Encapsulate Field
Separate Data Access Code
Separate Query from Modifier
Split Loop
Split Temporary Variable
Substitute Algorithm
Use a Connection Pool
Wrap entities with session

Refactoring
Consolidate Conditional Expression

You have a sequence of conditional tests with the same result.

Combine them into a single conditional expression and extract it.

```
if (seniority < 2) return 0;
if (monthsDisabled > 12) return 0;
if (isPartTime) return 0;

if (isNotEligibleForDisability()) return 0;
```

For more information see page 240 of Refactoring.

Refactoring
Decompose Conditional

You have a complicated conditional statement.

Extract methods from the condition, then part, and else part.

```
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);
```

For more information see page 238 of Refactoring.

Refactoring
Encapsulate Field

There is a public field.

Make it private and provide accessors.

```
public String _name;

private String _name;
public String getName() {return _name;}
public void setName(String arg) {name = arg;}
```

For more information see page 206 of Refactoring.

Refactoring
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}
```

For more information see page 124 of Refactoring.

Another Refactoring Example
Introduce Explaining Variable

```
// good method name, but the logic of the body is not clear

boolean isLeapYear( int year )
{
        return      ( ( ( year % 400 ) == 0 ) ||
                    ( ( ( year % 4 ) == 0 ) &&
                    ( ( year % 100 ) != 0 ) ) );
}
```

Leap year is defined to be a year divisible by 4

- **unless it is divisible by 100**
- **but is if it is divisible by 400**

```
boolean isLeapYear( int year )
{
        boolean isFourthYear = ( ( year % 4 ) == 0 );
        boolean isHundrethYear = ( ( year % 100 ) == 0);
        boolean is4HundrethYear = ( ( year % 400 ) == 0);
        return (
                is4HundrethYear
                || ( isFourthYear && ! isHundrethYear ) );
}
```

Refactoring
Replace Error Code with Exception

A method returns a special code to indicate an error.

Throw an exception instead.

```
int withdraw(int amount) {
        if (amount > balance)
        return -1;
        else {
        balance -= amount;
        return 0;}}

void withdraw(int amount) throws BalanceException {
        if (amount > balance) throw new BalanceException();
        balance -= amount;}
```

For more information see page 310 of Refactoring.

Refactoring
Replace Magic Number with Symbolic Constant

You have a literal number with a particular meaning.

Create a constant, name it after the meaning, and replace the number with it.

```
double potentialEnergy(double mass, double height) {
        return mass * height * 9.81;}

double potentialEnergy(double mass, double height) {
        return mass * GRAVITATIONAL_CONSTANT
                * height;}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

For more information see page 204 of Refactoring.

Refactoring
Replace Parameter with Explicit Methods

You have a method that runs different code depending on the values of an enumerated parameter.

Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
        if (name.equals("height")) {
                height = value;
                return;}
        if (name.equals("width")) {
                width = value;
                return;}
        Assert.shouldNeverReachHere();}
```

```
void setHeight(int arg) {
        height = arg;}
void setWidth (int arg) {
        width = arg;}
```

For more information see page 285 of Refactoring.

Refactoring
Split Loop

You have a loop that is doing two things.

Duplicate the loop.

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

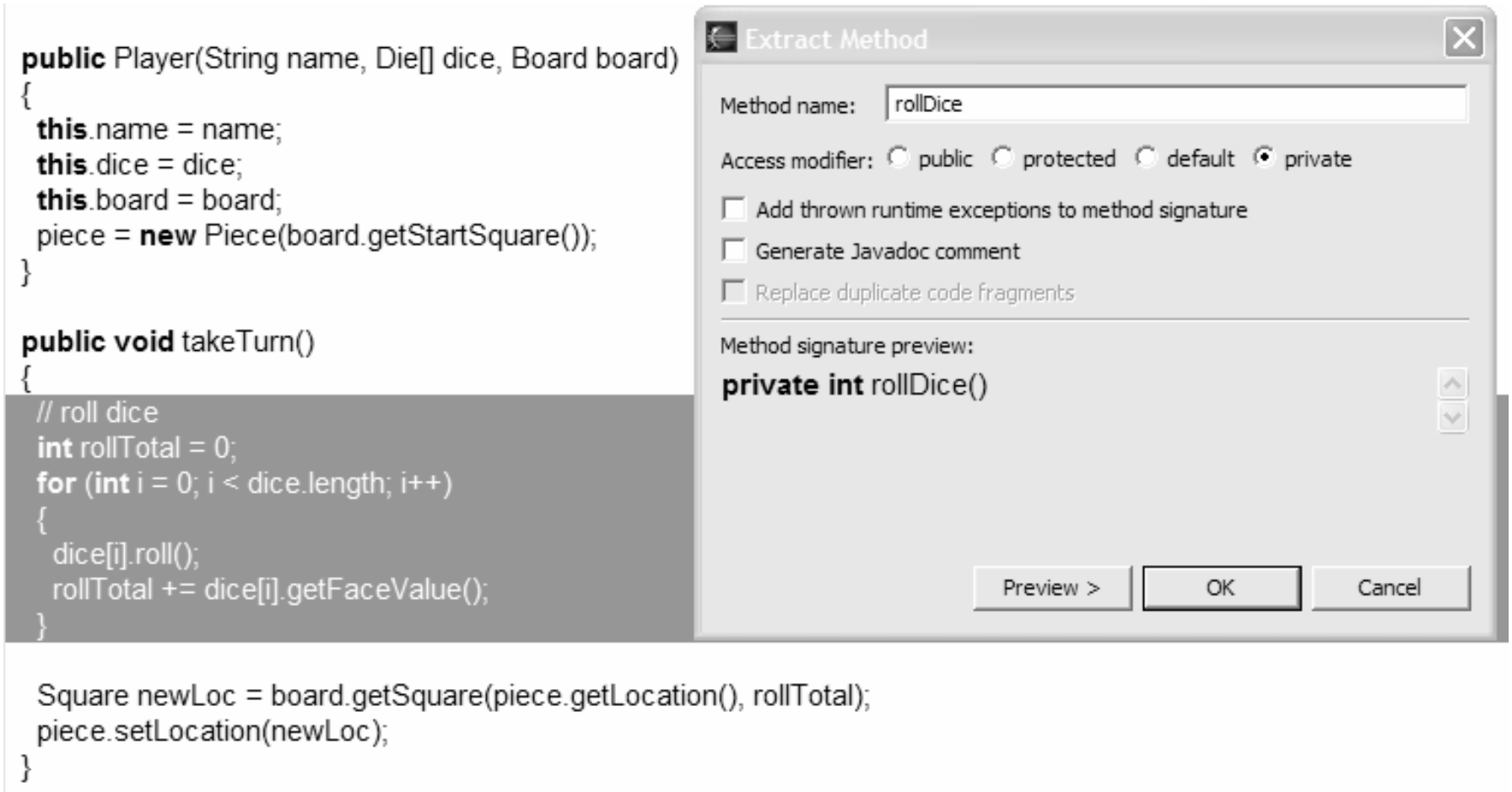
```
void printValues() {
        double totalSalary = 0;
        for (int i = 0; i < people.length; i++) {
                totalSalary += people[i].salary;
        }

        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);
}
```

Motivation for Split Loop

You often see loops that are doing two different things at once, because they can do that with one pass through a loop. Most programmers would feel uncomfortable with this refactoring as it forces you to execute the loop twice - which is double the work.

But like so many optimizations, doing two different things in one loop is less clear than doing them separately. It also causes problems for further refactoring as it introduces temps that get in the way of further refactorings. So while refactoring, don't be afraid to get rid of the loop. When you optimize, if the loop is slow that will show up and it would be right to slam the loops back together at that point. You may be surprised at how often the loop isn't a bottleneck, or how the later refactorings open up another, more powerful, optimization.

Automated Refactoring, Eclipse IDE



After Automated Refactoring

```
public void takeTurn()
{
  int rollTotal = rollDice();

  Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
  piece.setLocation(newLoc);
}

private int rollDice()
{
  // roll dice
  int rollTotal = 0;
  for (int i = 0; i < dice.length; i++)
  {
    dice[i].roll();
    rollTotal += dice[i].getFaceValue();
  }
  return rollTotal;
}
```

History of Refactoring

W.F. Opdyke and R.E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), September 1990.

W.G. Griswold, "Program Restructuring as an Aid to Software Maintenance," Ph.D. thesis, University of Washington, July 1991.

W.F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois at Urbana-Champaign, June 1992.

Questions and Answers