CS 683 Emerging Technologies: Embracing Change
Spring Semester, 2001
Doc 14 Refactoring Intro

Contents

Refactoring Intro ........................................................................................................... 2
The Broken Window ................................................................................................. 3
The Perfect Lawn .................................................................................................... 4
Familiarity verse Comfort ...................................................................................... 5
Refactoring .................................................................................................................. 6
Sample Refactoring: Extract Method ................................................................. 7
  Motivation ............................................................................................................... 7
  Mechanics ................................................................................................................. 8
  Example .................................................................................................................... 10
Simplifying Conditional Expressions ................................................................. 14
  Decompose Conditional ...................................................................................... 14
  Consolidate Conditional Expression ................................................................ 15
  Consolidate Duplicate Conditional Fragments .................................................... 16
  Remove Control Flag .............................................................................................. 18
  Replace Nested Conditional with Guard Clauses .................................................. 20
  Replace Conditional with Polymorphism .............................................................. 21
  Introduce Null Object .............................................................................................. 23
  Introduce Assertion ................................................................................................. 24

References


The Pragmatic Programmer, Hunt & Thomas, Addison Wesley Longman, 2000


Copyright©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Refactoring Intro

We have code that looks like:

at: anInteger put: anObject
(smallKey ~= largeKey)
  ifTrue:
    [(anInteger < smallKey)
      ifTrue: [self atLeftTree: anInteger put: anObject]
      ifFalse: [(smallKey = anInteger)
        ifTrue: [smallValue := anObject]
        ifFalse: [(anInteger < largeKey)
          ifTrue: [self atMiddleTree: anInteger put: anObject]
          ifFalse: [(largeKey = anInteger)
            ifTrue: [largeValue := anObject]
            ifFalse: [(largeKey < anInteger)
              ifTrue: [self atRightTree: anInteger put: anObject]]]]]]
    ifFalse:
      [self addNewKey: anInteger with: anObject].

Now what?
The Broken Window\textsuperscript{1}

In inner cities some buildings are:

- Beautiful and clean
- Graffiti filled, broken rotting hulks

Clean inhabited buildings can quickly become abandoned derelicts

The trigger mechanism is:

- A broken window

If one broken window is left unrepaired for a length of time

- Inhabitants get a sense of abandonment
- More windows break
- Graffiti appears
- Pipes break
- The damage goes beyond the owner's desire to fix

Don't live with Broken Widows in your code

\textsuperscript{1} Pragmatic Programmer, pp. 4-5
The Perfect Lawn

A visitor to an Irish castle asked the groundskeeper the secret of the beautiful lawn at the castle

The answer was:

• Just mow the lawn every third day for a hundred years

Spending a little time frequently

• Is much less work that big concentrated efforts
• Produces better results in the long run

So frequently spend time cleaning your code
Familiarity verse Comfort

Why don't more programmers/companies continually:

- Write unit tests
- Refactor
- Work on improving programming skills

Familiarity is always more powerful than comfort.

-- Virginia Satir
Refactoring

Refactoring is the modifying existing code without adding functionality

Changing existing code is dangerous

• Changes can break existing code

To avoid breaking code while refactoring:

• Need tests for the code
• Proceed in small steps
Sample Refactoring: 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*

**Motivation**

Short methods:

- Increase possible reuse
- Makes high level methods easier to read
- Makes easier to override methods

---

2 Refactoring Text, pp. 110-116
Mechanics

• Create a new method - the target method

  Name the target method after the intention of the method

  With short code only extract if the new method name is better than the code at revealing the code's intention

• Copy the extracted code from the source method into the target method

• Scan extracted code for references to local variables (temporary variables or parameters) of the source method

• If a temporary variable is used only in the extracted code declare it local in the target method

• If a parameter of the source method is used in the extracted code, pass the parameter to the target method
Mechanics - Continued

• See if the extracted code modifies any of the local variables of the source method

  If only one variable is modified, then try to return the modified value

  If more than one variable is modified, then the extracted code must be modified before it can be extracted

  Split Temporary Variables or Replace Temp with Query may help

• Compile when you have dealt with all the local variables

• Replace the extracted code in source code with a call to the target method

• Compile and test
Example
No Local Variables

Note I will use Fowler's convention of starting instance variables with "_" even though one can not do this is Squeak.

printOwing
| outstanding |

outstanding := 0.0.
Transcript
  show: '********************';
  cr;
  show: '***Customer Owes***';
  cr;
  show: '********************';
  cr.

outstanding := _orders inject: 0 into: [:sum :each | sum + each].

Transcript
  show: 'Name: ';
  show: _name;
  cr;
  show: 'Amount: ';
  show: outstanding;
  cr.

---

3 Example code is Squeak version of Fowler's Java example
Extracting the banner code we get:

printOwing
  | outstanding |

  outstanding := 0.0.
  self printBanner.

  outstanding := _orders inject: 0 into: [:sum :each | sum + each].

Transcript
  show: 'Name: '; show: _name; cr;
  show: 'Amount: '; show: outstanding; cr.

printBanner
  Transcript
    show: '**************************************************';
    cr;
    show: '***Customer Owes***';
    cr;
    show: '**************************************************';
    cr
Examples: Using Local Variables

We can extract printDetails: to get

```smalltalk
printOwing
    | outstanding |
    self printBanner.
    outstanding := ^_orders inject: 0 into: [:sum :each | sum + each].
    self printDetails: outstanding

printDetails: aNumber
    Transcript
    show: 'Name: '; show: _name;
    cr;
    show: 'Amount: '; show: aNumber;
    cr.
```

Then we can extract outstanding to get:

```smalltalk
printOwing
    self
    printBanner;
    printDetails: (self outstanding)

outstanding
    ^_orders inject: 0 into: [:sum :each | sum + each]
```

The text stops here, but the code could use more work
Using Add Parameter (275)

printBanner

Transcript
  show: '********************';
  cr;
  show: '***Customer Owes***';
  cr;
  show: '********************';
  cr

becomes:

printBannerOn: aStream
aStream
  show: '********************';
  cr;
  show: '***Customer Owes***';
  cr;
  show: '********************';
  cr

Similarly we do printDetails and printOwing

printOwingOn: aStream
  self
    printDetails: (self outstanding)
    on: aStream

Perhaps this should be called Replace Constant with Parameter
Simplifying Conditional Expressions
Decompose Conditional\textsuperscript{4}

You have a complicated conditional (if-then-else) statement

*Extract methods from the condition, then part and else parts*

**Example\textsuperscript{5}**

\begin{verbatim}
(date before: SummerStart) | (date after: SummerEnd)
  ifTrue: [ charge := quantity * _winterRate + _winterServiceCharge]
  ifFalse: [ charge := quantity + _summerRate]
\end{verbatim}

becomes

\begin{verbatim}
(self notSummer: date)
  ifTrue: [ charge := self winterCharge: quantity]
  ifFalse: [ charge := self summerCharge: quantity]
\end{verbatim}

or the more Smalltalk like:

\begin{verbatim}
charge := (self notSummer: date)
  ifTrue: [ self winterCharge: quantity]
  ifFalse: [ self summerCharge: quantity]
\end{verbatim}

Each method (notSummer, winterCharge, summerCharge) should be extracted and tested one at a time

\textsuperscript{4} Refactoring Text, pp. 238-239
\textsuperscript{5} Recall that "\_" indicates an instance variable
Consolidate Conditional Expression

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

**Example**

disabilityAmount
   _senority < 2 ifTrue: [^0].
   _monthDisabled > 12   ifTrue: [^0].
   self isPartTime   ifTrue: [^0].
   "compute the disabilty amount here"

becomes

disabilityAmount
   (_senority < 2) | (_monthDisabled > 12) | (self isPartTime) ifTrue: [^0].
   "compute the disabilty amount here"

becomes:

disabilityAmount
   self isNotEliableForDisability ifTrue: [^0].
   "compute the disabilty amount here"

---

Refactoring Text, pp. 240-242
Consolidate Duplicate Conditional Fragments

The same fragment of code is in all branches of a conditional expression

*Move it outside of the expression*

**Example**

```smalltalk
self isSpecialDeal
  ifTrue:
    [total := price * 0.95.
     self send]
  ifFalse:
    [total := price * 0.98.
     self send]

Consolidating we get:

```smalltalk
self isSpecialDeal
  ifTrue:[total := price * 0.95]
  ifFalse:[total := price * 0.98].
self send

A more Smalltalk like version:

```smalltalk
total := self isSpecialDeal
  ifTrue:[price * 0.95]
  ifFalse:[price * 0.98].
self send
```

---

7 Refactoring Text, pp. 243-244
Example continued

The text stops here, but the code could use more work

Use Introduce Explaining Variable (124) to improve readability

\[
\text{discountRate := self isSpecialDeal}
\]

\[
\begin{align*}
\text{ifTrue:} & \ [0.95] \\
\text{ifFalse:} & \ [0.98].
\end{align*}
\]

\[
\text{total := price * discountRate.}
\]

\[
\text{self send}
\]

Using Replace Temp with Query (120) we get:

\[
\text{total := price * self discountRate.}
\]

\[
\text{self send}
\]

Where we have

\[
\text{discountRate}
\]

\[
\begin{align*}
\text{^self isSpecialDeal} \\
\text{ifTrue:} & \ [0.95] \\
\text{ifFalse:} & \ [0.98]
\end{align*}
\]

In Java or C++ we could use Replace Magic Number with Symbolic Constant (204) on the 0.95 and 0.98 to improve readability

In Smalltalk we can use Introduce Explaining Variable (124) or Constant Method (Beck)
Remove Control Flag

You have a variable that is acting as a control flag for a series of boolean expressions

*Use a break or return instead*
**Example\(^9\)**

checkSecurity: people
| found |
found := false.
1 to: people size do:
[:index |
found ifFalse:
[((people at: index) = 'Don') ifTrue:
 [self sendAlert().
 found := true].
((people at: index) = 'John') ifTrue:
 [self sendAlert().
 found := true]]

Becomes:

checkSecurity: people
people containsMiscreant ifTrue:[self sendAlert()]

containsMiscreant
1 to: self size do:
[:index |
((self at: index) = 'Don') ifTrue: [^true].
((self at: index) = 'John') ifTrue: [^true]].
^false

In Squeak the latter becomes:

containsMiscreant
^self anySatisfy: [:each | (each = 'Don') | (each = 'John')]  

\(^9\) John and Don happen to be the first names of the authors of the Refactoring Browser. Both are excellent Smalltalk programmers. I do not know why Fowler uses those names as examples of miscreants :)
Replace Nested Conditional with Guard Clauses

A method has conditional behavior that does not make clear the normal path of execution

*Use guard clauses for all the special cases*

**Example**

```smalltalk
payAmount
| result |
_isDead
  ifTrue: [result := self deadAmount]
  ifFalse:
    [_isSeparated
      ifTrue:[result := self separatedAmount]
      ifFalse:
        [_isRetired
          ifTrue:[result := self retiredAmount]
          ifFalse:[result := self normalPayAmount]].
    ]
  ]
^ result
```

becomes

```smalltalk
payAmount
  _isDead ifTrue: [^self deadAmount].
  _isSeparated ifTrue:[^self separatedAmount].
  _isRetired ifTrue:[^self retiredAmount].
  ^self normalPayAmount.
```

---

10 Refactoring Text, pp. 250-254
Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object

Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract

---

Refactoring Text, pp. 255-259
Example

Employee>>payAmount
  _type = Employee engineer ifTrue:[^ self _monthlySalary].
  _type = Employee manager ifTrue:[^ self _monthlySalary * 2].
  _type = Employee instructor ifTrue:[^ self _monthlySalary/2].
  self error: 'Invalid Employee'

becomes:

• Create an EmployeeType class
• Create Engineer, Manager & Instructor subclasses of EmployeeType

Employee>>payAmount
  ^_type payAmount: self

EmployeeType>>payAmount: anEmployee
  self subclassResponsibility

Engineer>>payAmount: anEmployee
  ^anEmployee monthlySalary

Manager>>payAmount: anEmployee
  ^anEmployee monthlySalary * 2

Instructor>>payAmount: anEmployee
  ^anEmployee monthlySalary/ 2
Introduce Null Object\textsuperscript{12}

You have repeated checks for a null value

*Replace the null value with a null object*

**Example**

customer isNil
  ifTrue: [plan := BillingPlan basic]
  ifFalse: [plan := customer plan]

becomes:

- Create NullCustomer subclass of Customer with:

  NullCustomer>>plan
  ^BillingPlan basic

- Make sure that each customer variable has either a real customer or a NullCustomer

Now the code is:

plan := customer plan

- Often one makes a Null Object a singleton

\textsuperscript{12} Refactoring Text, pp. 260-266
Introduce Assertion\textsuperscript{13}

A section of code assumes something about the state of the program

*Make the assumption explicit with an assertion*

**Example**

```plaintext
getExpenseLimit
 "Should have either expense limit or a primary project"
 ^_expenseLimit isNil
   ifTrue:[_expenseLimit]
   ifFalse:[_primaryProject memberExpenseLimit]

Becomes:

getExpenseLimit
 self assert: [^_expenseLimit isNotNil | primaryProject isNotNil].
 ^_expenseLimit isNil
   ifTrue:[^_expenseLimit]
   ifFalse:[^primaryProject memberExpenseLimit]
```

Recall that $\_\_$ is used to indicate an instance variable
(_primaryProject)

Squeak does have an assert: method in Object

\textsuperscript{13} Refactoring Text, pp. 267-270