Refactoring

George Dinwiddie
iDIA Computing, LLC
http://idiacomputing.com
http://blog.gdinwiddie.com

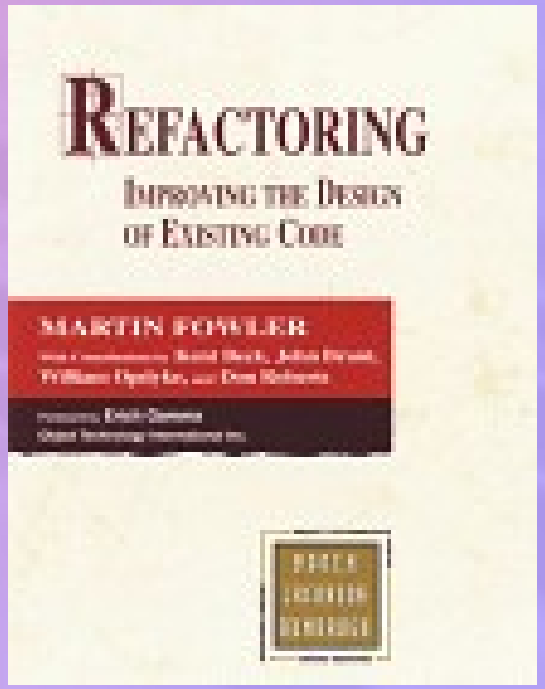
What is Refactoring?

“Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.”

– Martin Fowler, http://refactoring.com

What is Refactoring NOT?

- Fixing a bug.
- Rewriting a module.
- Adding a feature.
- Editing code.
- An excuse why your code doesn't work, yet.



What else have you heard people call refactoring that does not meet Martin Fowler's description?

Example 1: Obvious Duplication

(code from http://github.com/myabc/mono-olive/blob/3f5207fc0cdec631660e5dafb8a634950ebbd4bd/class/Microsoft.JScript.Compiler/Microsoft.JScript.Compiler/Tokenizer.cs)

```
case 'x'://2 hex digits
        ReadChar ();//next
        int r2 = 0;
        for (int i =0;i<2;i++) {
                ReadChar ();
                int d = PeekChar ();
                if (d >= '0' && d <= '9')
                        d -= '0';
                else if (d >= 'A' && d <= 'F')
                        d = d - 'A' + 10;
                else if (d >= 'a' && d <= 'f')
                        d = d - 'a' + 10;
                else
                        return CreateBad-
Token (builder.ToString(), Diagnostic-
Code.HexLiteralNoDigits);
                r2 = (r2 << 4) | d;
        }
        c = r2;
        break;
```

```
case 'u'://4 hex digits
        ReadChar ();//next
        int r4 = 0;
        for (int i = 0; i < 4; i++) {
                ReadChar ();
                int d = PeekChar ();
                if (d >= '0' && d <= '9')
                        d -= '0';
                else if (d >= 'A' && d <= 'F')
                        d = d - 'A' + 10;
                else if (d >= 'a' && d <= 'f')
                        d = d - 'a' + 10;
                else
                        return CreateBad-
Token (builder.ToString(), Diagnostic-
Code.HexLiteralNoDigits);
                r4 = (r4 << 4) | d;
        }
        c = r4;
        break;
```

Step 1.1: Introduce Exception

```
case 'x'://2 hex digits
        try {
                ReadChar ();//next
                int r2 = 0;
                for (int i =0;i<2;i++) {
                        ReadChar ();
                        int d = PeekChar ();
                        if (d >= '0' && d <= '9')
                                d -= '0';
                        else if (d >= 'A' && d <= 'F')
                                d = d - 'A' + 10;
                        else if (d >= 'a' && d <= 'f')
                                d = d - 'a' + 10;
                        else
                                throw new HexLiteralException();
                        r2 = (r2 << 4) | d;
                }
                c = r2;
        } catch (HexLiteralException e) {
                return CreateBadToken (builder.ToString(),
                                DiagnosticCode.HexLiteralNoDigits);
        }
        break;
```

Step 1.2: Extract Method

```
case 'x'://2 hex digits
        try {
                c=Read2HexDigits();
        } catch (HexLiteralException e) {
                return CreateBadToken(
                        builder.ToString(),
                        DiagnosticCode.
                        HexLiteralNoDigits);
        }
        break;
```

```
int Read2HexDigits() {
        ReadChar ();//next
        int r2 = 0;
        for (int i =0;i<2;i++) {
                ReadChar ();
                int d = PeekChar ();
                if (d >= '0' && d <= '9')
                        d -= '0';
                else if (d >= 'A' && d <= 'F')
                        d = d - 'A' + 10;
                else if (d >= 'a' && d <= 'f')
                        d = d - 'a' + 10;
                else
                  throw new
                        HexLiteralException();
                r2 = (r2 << 4) | d;
        }
        return r2;
}
```

Step 1.3: Parameterize Method

```
case 'x'://2 hex digits
        try {
                c=Read2HexDigits();
        } catch (HexLiteralException e) {
                return CreateBadToken(
                        builder.ToString(),
                        DiagnosticCode.
                        HexLiteralNoDigits);
        }
        break;


int Read2HexDigits() {
        return ReadHexDigits(2);
}
```

```
int ReadHexDigits(int numberOfDigits) {
        ReadChar ();//next
        int r2 = 0;
        for (int i =0;i<numberOfDigits;i++) {
                ReadChar ();
                int d = PeekChar ();
                if (d >= '0' && d <= '9')
                        d -= '0';
                else if (d >= 'A' && d <= 'F')
                        d = d - 'A' + 10;
                else if (d >= 'a' && d <= 'f')
                        d = d - 'a' + 10;
                else
                  throw new
                        HexLiteralException();
                r2 = (r2 << 4) | d;
        }
        return r2;
}
```

Step 1.4: Inline Method

```
case 'x'://2 hex digits
        try {
                c=ReadHexDigits(2);
        } catch (HexLiteralException e) {
                return CreateBadToken(
                        builder.ToString(),
                        DiagnosticCode.
                        HexLiteralNoDigits);
        }
        break;
```

Step 1.5: Repeat for other repetition

```
case 'u'://4 hex digits
        try {
                c=ReadHexDigits(4);
        } catch (HexLiteralException e) {
                return CreateBadToken(
                        builder.ToString(),
                        DiagnosticCode.
                        HexLiteralNoDigits);
        }
        break;
```

Step 1.6: Rename Variables

```
int ReadHexDigits(int numberOfDigits) {
        ReadChar ();//next
        int result = 0;
        for (int i =0;i<numberOfDigits;i++) {
                ReadChar ();
                int digit = PeekChar ();
                if (digit >= '0' && digit <= '9')
                        digit -= '0';
                else if (digit >= 'A' && digit <= 'F')
                        digit = digit - 'A' + 10;
                else if (digit >= 'a' && digit <= 'f')
                        digit = digit - 'a' + 10;
                else
                  throw new
                         HexLiteralException();
                result = (result << 4) | digit;
        }
        return result;
}
```

Example 2: Switch on Type Code

(from https://aspnet.svn.codeplex.com/svn/WebForms/Futures/GeneratedImage/GeneratedImage/GeneratedImage/ImageTransform.cs)

```
public override Image ProcessImage(Image img) {
    int scaledHeight = (int)(img.Height * (
                        (float)this.Width / (float)img.Width));
    int scaledWidth = (int)(img.Width * (
                        (float)this.Height / (float)img.Height));

    switch (Mode) {
        case ImageResizeMode.Fit:
            return FitImage(img, scaledHeight, scaledWidth);
        case ImageResizeMode.Crop:
            return CropImage(img, scaledHeight, scaledWidth);
        default:
            Debug.Fail("Should not reach this");
            return null;
    }
}
```

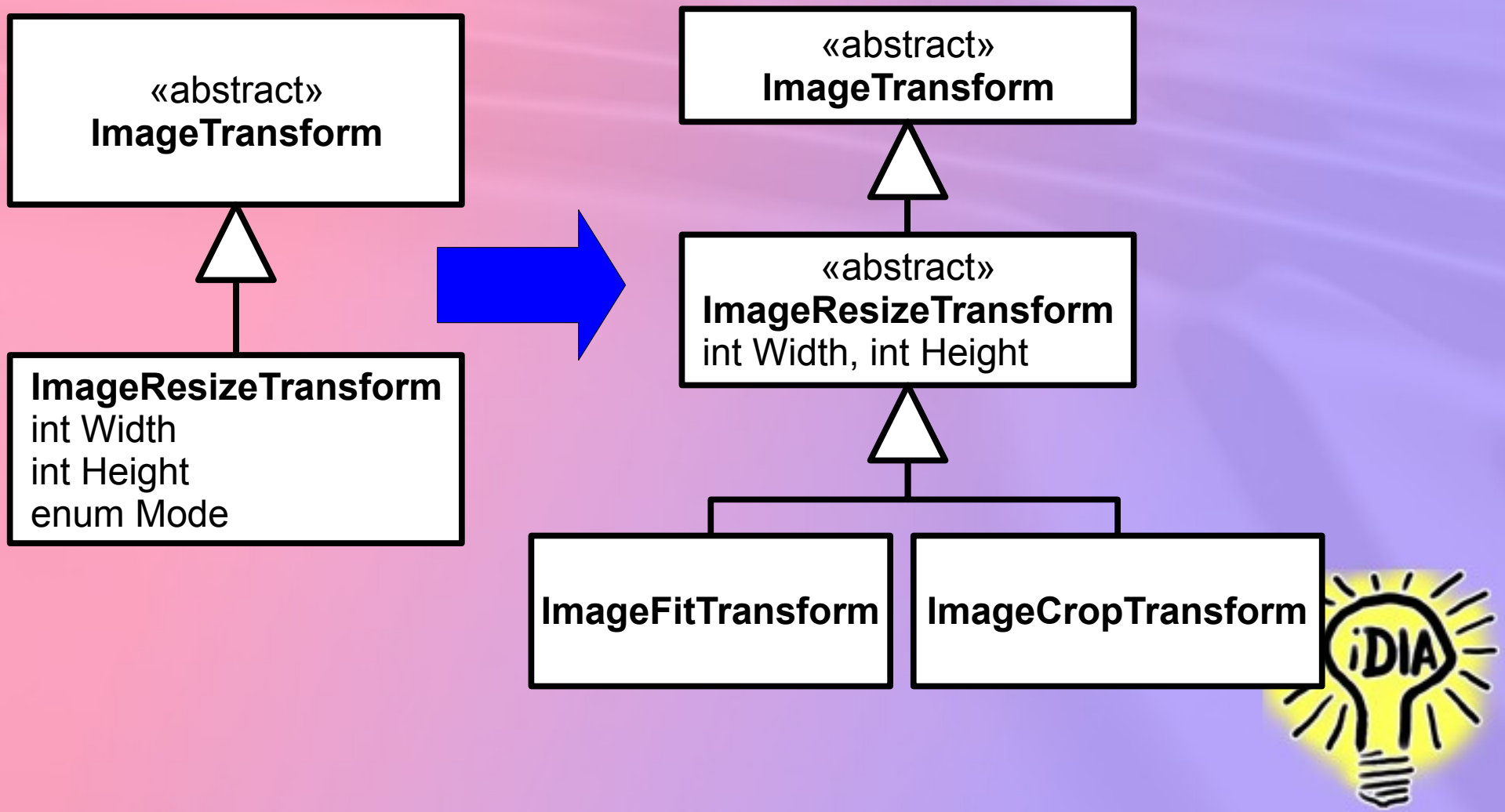
Example 2 continued: CropImage()

```
private Image CropImage(Image img,
                        int scaledHeight, int scaledWidth) {
    int resizeWidth = 0;
    int resizeHeight = 0;
    if (((float)this.Width / (float)img.Width >
                        this.Height / (float)img.Height)) {
        resizeWidth = this.Width;
        resizeHeight = scaledHeight;
    }
    else {
        resizeWidth = scaledWidth;
        resizeHeight = this.Height;
    }

    Bitmap newImage = new Bitmap(this.Width, this.Height);
    Graphics graphics = Graphics.FromImage(newImage);
    SetupGraphics(graphics);
    graphics.DrawImage(img, (this.Width - resizeWidth) / 2,
                            (this.Height - resizeHeight) / 2,
                            resizeWidth, resizeHeight);
    return newImage;
}
```

Example 2 continued: FitImage()

```
private Image FitImage(Image img,
                       int scaled_height, int scaled_width) {
    int resizeWidth = 0;
    int resizeHeight = 0;
    if (this.Height == 0) {
        resizeWidth = this.Width;
        resizeHeight = scaled_height;
    }
    else if (this.Width == 0) {
        resizeWidth = scaled_width;
        resizeHeight = this.Height;
    }
    else {
        if (((float)this.Width / (float)img.Width <
                                    this.Height / (float)img.Height)) {
            resizeWidth = this.Width;
            resizeHeight = scaled_height;
        }
        else {
            resizeWidth = scaled_width;
            resizeHeight = this.Height;
        }
    }

    Bitmap newimage = new Bitmap(resizeWidth, resizeHeight);
    Graphics gra = Graphics.FromImage(newimage);
    SetupGraphics(gra);
    gra.DrawImage(img, 0, 0, resizeWidth, resizeHeight);
    return newimage;
}
```

Replace Type Code with Subclasses



New Subclasses

```
public class ImageCropTransform : ImageResizeTransform {
        public override Image ProcessImage(Image img) {
            int scaledHeight = (int)(img.Height * (
                                (float)this.Width / (float)img.Width));
            int scaledWidth = (int)(img.Width * (
                                (float)this.Height / (float)img.Height));
            return CropImage(img, scaledHeight, scaledWidth);
        }       // move CropImage() to this subclass
}
public class ImageFitTransform : ImageResizeTransform {
        public override Image ProcessImage(Image img) {
            int scaledHeight = (int)(img.Height * (
                                (float)this.Width / (float)img.Width));
            int scaledWidth = (int)(img.Width * (
                                (float)this.Height / (float)img.Height));
            return FitImage(img, scaledHeight, scaledWidth);
        }       // move FitImage to this subclass
}
```

What other transformations would you make to these classes?

Why Refactor?

- Keep your work area clean for sustainable delivery (kitchen metaphor)
- Eliminate code smells
- Achieve simple design (Kent Beck)
    1. Passes all the tests
    2. Contains no duplication
    3. Expresses programmer's intent
    4. Contains no superfluous parts

Catalog of Code Smells

- Comments
- Long method
- Long parameter list
- Duplicated code
- Large class
- Speculative generality
- Primitive obsession
- Data class
- Data clumps
- Refused bequest
- Inappropriate intimacy
- Indecent exposure
- Lazy class
- Feature envy
- Divergent change
- Shotgun surgery
- Switch statements
- Temporary field
- Parallel inheritance hierarchies
- Speculative generalization
- Conditional complexity
- Dead code

http://wiki.java.net/bin/view/People/SmellsToRefactorings
http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm
http://xp123.com/rwb/ (William Wake's Refactoring Workbook)

(some of) Tim Ottinger's Rules for Variable and Class Naming

- Use intention revealing names
  - int d // elapsed time in days
  - int elapsedTimeInDays
  - int daysSinceCreation, int daysSinceModification
- Use noun and verb phrases
  - Classes and objects should have noun or noun phrase names.
  - Accessors *may* have noun names.
  - Other methods should have verb or verb phrase names

see http://objectmentor.com/resources/articles/Naming.pdf

Principles of Object Oriented Class Design (Uncle Bob Martin)

- **Open Closed Principle** – *A module should be open for extension but closed for modification. (Bertrand Meyer)*
- **Liskov Substitution Principle** – *Subclasses should be substitutable for their base classes. (Barbar Liskov)*
- **Dependency Inversion Principle** – *Depend upon abstractions. Do not depend upon concretions.*
- **Interface Segregation Principle** – *Many client-specific interfaces are better than one general-purpose interface.*
- **Single Responsibility Principle** – *There should never be more than one reason for a class to change.*

Commonly Reversible Refactorings

(see http://refactoring.com/catalog/)

- Extract method, class
- Extract subclass, Extract superclass
- Pull up method, field
- Replace delegation with inheritance
- Replace parameter with explicit methods
- Inline method, class
- Collapse hierarchy
- Push down method, field
- Replace inheritance with delegation
- Parameterize method

Refactorings for Expressiveness

- Rename variable, method, class
- Reduce scope of variable
- Remove assignments to parameters
- Remove control flag
- Remove double negative
- Replace array with object
- Replace magic number with symbolic constant

Refactoring for Robustness and Ease of Programming

- Introduce null object
- Replace type code with class
- Replace type code with subclasses
- Replace type code with state/strategy
- Replace nested conditional with guard clauses