



3

MEASURED SMELLS

The smells in this chapter are similar. They're dead easy to detect. They're objective (once you decide on a way to count and a maximum acceptable score). They're odious.

And, they're common.

You can think of these smells as being caught by a software metric. Each metric tends to catch different aspects of why code isn't as good as it could be. Some metrics measure variants of code length; others try to measure the connections between methods or objects; others measure a distance from an ideal.

Most metrics seem to correlate with length, so I tend to worry about size first (usually noticeable as a Large Class or Long Method). But if a metric is easy to compute, I'll use it as an indicator that some section of code deserves a closer look.

Metrics are indicators, not absolutes. It's very easy to get into the trap of *making numbers* without addressing the total complexity. So don't refactor just for a better number; make sure it really improves your code.

Smells Covered

- Comments
- Long Method
- Large Class
- Long Parameter List





Comments

Symptoms

- Comment symbols (`//` or `/*`) appear in the code. (Some IDEs help by color coding different types of comments.)

Causes

Comments may be present for the best of reasons: The author realizes that something isn't as clear as it could be and adds a comment.

Some comments are particularly helpful:

- Those that tell why something is done a particular way (or why it wasn't)
- Those that cite algorithms that are not obvious (where a simpler algorithm won't do)

Other comments can be reflected just as well in the code itself. For example, the goal of a routine can often be communicated as well through the routine's name as it can through a comment.

What to Do

- When a comment explains a block of code, you can often use *Extract Method* to pull the block out into a separate method. The comment will often suggest a name for the new method.
- When a comment explains what a method does (better than the method's name!), use *Rename Method* using the comment as the basis of the new name.
- When a comment explains preconditions, consider using *Introduce Assertion* to replace the comment with code.

Payoff

Improves communication. May expose duplication.

Contraindications

Don't delete comments that are pulling their own weight.



EXERCISE 3 Comments.

Consider this code. (Online at www.xp123.com/rwb)

Matcher.java

```
public class Matcher {
    public Matcher() {}
    public boolean match(int[] expected, int[] actual,
        int clipLimit, int delta)
    {

        // Clip "too-large" values
        for (int i = 0; i < actual.length; i++)
            if (actual[i] > clipLimit)
                actual[i] = clipLimit;

        // Check for length differences
        if (actual.length != expected.length)
            return false;

        // Check that each entry within expected +/- delta
        for (int i = 0; i < actual.length; i++)
            if (Math.abs(expected[i] - actual[i]) > delta)
                return false;

        return true;
    }
}
```

MatcherTest.java

```
import junit.framework.TestCase;

public class MatcherTest extends TestCase {
    public MatcherTest(String name) {super(name);}

    public void testMatch() {
        Matcher matcher = new Matcher();

        int[] expected = new int[] {10, 50, 30, 98};
        int clipLimit = 100;
        int delta = 5;

        int[] actual = new int[] {12, 55, 25, 110};

        assertTrue(matcher.match(expected, actual, clipLimit, delta));

        actual = new int[] {10, 60, 30, 98};
        assertTrue(!matcher.match(expected, actual, clipLimit, delta));
    }
}
```



EXERCISE 3 Comments. (Continued)

```
    actual = new int[] {10, 50, 30};  
    assertTrue(!matcher.match(expected, actual, clipLimit, delta));  
  }  
}
```

- A. Use *Extract Method* to make the comments in `match()` redundant.**

- B. Can everything important about the code be communicated using the code alone? Or do comments have a place?**

- C. Find some code you wrote recently. Odds are good that you commented it. Can you eliminate the need for some of those comments by making the code reflect your intentions more directly?**

■ See Appendix A for solutions.

Long Method

Symptoms

- Large number of lines. (I'm immediately suspicious of any method with more than 5 to 10 lines.)

Causes

I think of it as the *Columbo syndrome*. Columbo was the detective who always had “just one more thing.” A method starts down a path and, rather than break the flow or identify the helper classes, the author adds one more thing. Code is often easier to write than it is to read, so there's a temptation to write blocks that are too big.



What to Do

- Use *Extract Method* to break up the method into smaller pieces. Look for comments or white space delineating interesting blocks. You want to extract methods that are semantically meaningful, not just introduce a function call every seven lines.
- You may find other refactorings (those that clean up straight-line code, conditionals, and variable usage) helpful before you even begin splitting up the method.

Payoff

Improves communication. May expose duplication. Often helps new classes and abstractions emerge.

Discussion

- People are sometimes worried about the performance hit from increasing the number of method calls, but most of the time this is a nonissue. By getting the code as clean as possible before worrying about performance, you have the opportunity to gain big insights that can restructure systems and algorithms in a way that dramatically increases performance.
- Don't *game* the metrics; the goal of using *Extract Method* is to use it in a way that increases insight.

Contraindications

It may be that a somewhat longer method is just the best way to express something. (Like almost all smells, the length is a warning sign—not a guarantee—of a problem.)

EXERCISE 4 Long Method

Consider this code. (Online at www.xp123.com/rwb)

Machine.java

```
public class Machine {
    String name;
    String location;
    String bin;
}
```

EXERCISE 4 Long Method (Continued)

```
public Machine(String name, String location) {
    this.name = name;
    this.location = location;
}

public String take() {
    String result = bin;
    bin = null;
    return result;
}

public String bin() {
    return bin;
}

public void put(String bin) {
    this.bin = bin;
}

public String name() {return name;}
}
```

Robot.java

```
public class Robot {
    Machine location;
    String bin;

    public Robot() {}

    public Machine location() {return location;}
    public void moveTo(Machine location) {this.location = location;}

    public void pick() {this.bin = location.take();}
    public String bin() {return bin;}

    public void release() {
        location.put(bin);
        bin = null;
    }
}
```

EXERCISE 4 Long Method (Continued)**RobotTest.java**

```
import junit.framework.*;

public class RobotTest extends TestCase{
    public RobotTest(String name) {super(name);}

    public void testRobot() {
        Machine sorter = new Machine("Sorter", "left");
        sorter.put("chips");
        Machine oven = new Machine("Oven", "middle");
        Robot robot = new Robot();

        assertEquals("chips", sorter.bin());
        assertNull(oven.bin());
        assertNull(robot.location());
        assertNull(robot.bin());

        robot.moveTo(sorter);
        robot.pick();
        robot.moveTo(oven);
        robot.release();

        assertNull(robot.bin());
        assertEquals(oven, robot.location());
        assertNull(sorter.bin());
        assertEquals("chips", oven.bin());
    }
}
```

Report.java

```
import java.util.*;
import java.io.*;

public class Report {
    public static void report(Writer out, List machines, Robot robot)
        throws IOException
    {
        out.write("FACTORY REPORT\n");

        Iterator line = machines.iterator();
        while (line.hasNext()) {
            Machine machine = (Machine) line.next();
            out.write("Machine " + machine.name());
        }
    }
}
```

EXERCISE 4 Long Method (Continued)

```
        if (machine.bin() != null)
            out.write(" bin=" + machine.bin());
        out.write("\n");
    }
    out.write("\n");

    out.write("Robot");
    if (robot.location() != null)
        out.write(" location=" + robot.location().name());

    if (robot.bin() != null)
        out.write(" bin=" + robot.bin());

    out.write("\n");

    out.write("=====\n");
}
}
```

ReportTest.java

```
import junit.framework.TestCase;

import java.util.ArrayList;
import java.io.PrintStream;
import java.io.StringWriter;
import java.io.IOException;

public class ReportTest extends TestCase {
    public ReportTest(String name) {super(name);}

    public void testReport() throws IOException {
        ArrayList line = new ArrayList();
        line.add(new Machine("mixer", "left"));

        Machine extruder = new Machine("extruder", "center");
        extruder.put("paste");
        line.add(extruder);

        Machine oven = new Machine("oven", "right");
        oven.put("chips");
        line.add(oven);

        Robot robot = new Robot();
        robot.moveTo(extruder);
        robot.pick();
    }
}
```

**EXERCISE 4** Long Method (Continued)

```
StringWriter out = new StringWriter();
Report.report(out, line, robot);

String expected =
    "FACTORY REPORT\n" +
    "Machine mixer\nMachine extruder\n" +
    "Machine oven bin=chips\n\n" +
    "Robot location=extruder bin=paste\n" +
    "=====\n";

assertEquals(expected, out.toString());
}
```

A. In `Report.java`, circle four blocks of code to show which functions you might extract in the process of refactoring this code.

B. Rewrite the `report()` method as four statements, as if you had done *Extract Method* for each block.

C. Does it make sense to extract a one-line method?

D. Long methods are trivially easy to spot, yet they seem to occur often in real code. Why?

■ See Appendix A for solutions.

Large Class**Symptoms**

- Large number of instance variables
- Large number of methods
- Large number of lines



Causes

Large classes get big a little bit at a time. The author keeps adding just one more capability to a class until eventually it grows too big. Sometimes the problem is a lack of insight into the parts that make up the whole class. In any case, the class represents too many responsibilities folded together.

What to Do

In general, you're trying to break up the class. If the class has Long Methods, address that smell first. To break up the class, three approaches are most common:

- *Extract Class*, if you can identify a new class that has part of this class's responsibilities
- *Extract Subclass*, if you can divide responsibilities between the class and a new subclass
- *Extract Interface*, if you can identify subsets of features that clients use

Sometimes, the class is big because it's a GUI class, and it represents not only the display component, but the model as well. In this case, you can use *Duplicate Observed Data* to help extract a domain class.

Payoff

Improves communication. May expose duplication.

EXERCISE 5 Large Class.

Consider this declaration from the Java libraries:

```
public class JTable extends JComponent
    implements Accessible, CellEditorListener,
        ListSelectionListener, Scrollable,
        TableColumnModelListener, TableModelListener
{
    // Constants
    public static final int AUTO_RESIZE_ALL_COLUMNS
    public static final int AUTO_RESIZE_LAST_COLUMN
    public static final int AUTO_RESIZE_NEXT_COLUMN
    public static final int AUTO_RESIZE_OFF
    public static final int AUTO_RESIZE_SUBSEQUENT_COLUMNS
```

EXERCISE 5 Large Class. (Continued)

```
// Constructors
public JTable()
public JTable(TableModel, TableColumnModel)
public JTable(TableModel, TableColumnModel, ListSelectionModel)
public JTable(int, int)
public JTable(Object[][], Object[][] )
public JTable(java.util.Vector, java.util.Vector)

// Methods
public void addColumn(TableColumn column)
public void addColumnSelectionInterval(int start, int finish)
public void addNotify()
public void addRowSelectionInterval(int start, int finish)
public void clearSelection()
public void columnAdded(TableColumnModelEvent event)
public void columnAtPoint(Point p)
public void columnMarginChanged(ChangeEvent event)
public void columnMoved(TableColumnModelEvent event)
public void columnRemoved(TableColumnModelEvent event)
public void columnSelectionChanged(ListSelectionEvent event)
public void convertColumnIndexToModel(int viewColumn)
public void convertColumnIndexToView(int modelColumn)
public void createDefaultColumnsFromModel()
public boolean editCellAt(int row, int column)
public boolean editCellAt(int row, int column, EventObject event)
public void editingCanceled(ChangeEvent event)
public void editingStopped(ChangeEvent event)
public AccessibleContext getAccessibleContext()
public boolean getAutoCreateColumnsFromModel()
public int getAutoResizeMode()
public TableCellEditor getCellEditor()
public TableCellEditor getCellEditor(int row, int column)
public Rectangle getCellRect(int row, int column, boolean includeSpacing)
public boolean getCellSelectionEnabled()
public TableColumn getColumn(Object object)
public Class getColumnClass(int column)
public int getColumnCount()
public TableColumnModel getColumnModel()
public String getColumnName(int column)
public Boolean getColumnSelectionAllowed()
public TableCellEditor getDefaultEditor(Class class)
public TableCellRenderer getDefaultRenderer(Class class)
public int getEditingColumn()
public int getEditingRow()
public Component getEditorComponent()
public Color getGridColor()
public Dimension getInterCellSpacing()
public TableModel getModel()
```

EXERCISE 5 Large Class. (Continued)

```
public Dimension getPreferredSize()
public int getRowCount()
public int getRowHeight()
public int getRowMargin()
public Boolean getRowSelectionAllowed()

public int getScrollableBlockIncrement(
Rectangle visible, int orientation, int direction)
public Boolean getScrollableTracksViewportHeight()
public Boolean getScrollableTracksViewportWidth()
public int getScrollableUnitIncrement(
Rectangle visible, int orientation, int direction)
public int getSelectedColumn()
public int getSelectedColumnCount()
public int[] getSelectedColumns()
public int getSelectedRow()
public int getSelectedRowCount()
public int[] getSelectedRows()
public Color getSelectionBackground()
public Color getSelectionForeground()
public ListSelectionModel getSelectionModel()
public Boolean getShowHorizontalLines()
public Boolean getShowVerticalLines()
public JTableHeader getTableHeader()
public String getToolTipText(MouseEvent event)
public TableUI getUI()
public String getUIClassID()
public Object getValueAt(int row, int column)
public Boolean isCellEditable(int row, int column)
public Boolean isSelected(int row, int column)
public Boolean isColumnSelected(int column)
public Boolean isEditing()
public boolean isManagingFocus()
public Boolean isRowSelected(int row)
public void moveColumn(int column, int newColumn)
public Component prepareEditor(TableCellEditor editor,
int row, int column)
public Component prepareRenderer(TableCellRenderer renderer,
int row, int column)
public void removeColumn(TableColumn column)
public void removeColumnSelectionInterval(int column1, int column2)
public void removeEditor()
public void removeRowSelectionInterval(int row1, int row2)
public void reshape(int x, int y, int width, int height)
public int rowAtPoint(Point point)
public void selectAll()
public void setAutoCreateColumnsFromModel(Boolean doAutoCreate)
public void setAutoResizeModel(int mode)
public void setCellEditor(TableCellEditor editor)
```

EXERCISE 5 Large Class. (Continued)

```
public void setCellSelectionEnabled(Boolean maySelect)
public void setColumnModel(TableColumnModel model)
public void setColumnSelectionAllowed(Boolean maySelect)
public void setColumnSelectionInterval(int column1, int column2)
public void setDefaultEditor(Class class, TableCellEditor editor)
public void setDefaultRenderer(Class class, TableCellRenderer renderer)
public void setEditingColumn(int column)
public void setEditingRow(int row)
public void setGridColor(Color color)
public void setInterCellSpacing(Dimension dim)
public void setModel(TableModel model)
public void setPreferredSizeScrollableViewportSize(Dimension dim)
public void setRowHeight(int height)
public void setRowMargin(int margin)
public void setRowSelectionAllowed(Boolean maySelect)
public void setSelectionBackground(Color background)
public void setSelectionForeground(Color foreground)
public void setSelectionMode(int mode)
public void setSelectionMode(ListSelectionMode model)
public void setShowGrid(Boolean showing)
public void setShowHorizontalLines(Boolean b)
public void setShowVerticalLines(Boolean b)
public void setTableHeader(JTableHeader header)
public void setUI(TableUI ui)
public void setValueAt(Object value, int row, int column)
public void sizeColumnsToFit(int resizingColumn)
public void tableChanged(TableModelEvent event)
public void updateUI()
public void valueChanged(ListSelectionEvent)

// plus 22 protected variables

// plus 10 protected methods

// plus 97 methods inherited from JComponent (and fields etc.)

// plus about 35 methods from Container

// plus about 85 methods from Component

// plus 11 methods from Object
}
```

A. Why does this class have so many methods?

EXERCISE 5 Large Class. (Continued)

B. Go through the methods listed and categorize them into 5 to 10 major areas of responsibility.

C. In what ways could the library writers have eliminated some of these methods?

D. In Java, Object has 11 methods. In Smalltalk, it has more than 100. Why the difference? Talk to a Smalltalk person and find out why and whether or not this is a smell.

■ See Appendix A for solutions.

Long Parameter List

Symptoms

- A method has more than one or two parameters.

Causes

You might be trying to minimize coupling between objects. Instead of the called object being aware of relationships between classes, you let the caller locate everything; then the method concentrates on what it is being asked to do with the pieces.

Or a programmer generalizes the routine to deal with multiple variations by creating a general algorithm and a lot of *control* parameters.

What to Do

- If the parameter value can be obtained from another object this one already knows, *Replace Parameter with Method*.
- If the parameters come from a single object, try *Preserve Whole Object*.
- If the data is not from one logical object, you still might group them via *Introduce Parameter Object*.

Payoff

Improves communication. May expose duplication. Often reduces size.

Contraindications

- Sometimes, you *want* to avoid a dependency between two classes. For example, the caller may have the dependency, but you don't want to propagate it. Ensure that your changes don't upset this balance.
- Sometimes the parameters have no meaningful grouping—they don't go together.

Notes

This is one of those places where a smell doesn't guarantee a problem. You might smell a Long Parameter List but decide it's right for the situation at hand.

EXERCISE 6 Long Parameter List.

Consider these methods declared in the Java libraries:

From `java.swing.CellRendererPane`:

```
public void paintComponent(Graphics gr, Component renderer,  
    Container parent, int x, int y, int width, int height,  
    Boolean shouldValidate)
```

From `java.awt.Graphics`:

```
public Boolean drawImage(Image image,  
    int x1Dest, int y1Dest, int x2Dest, int y2Dest,  
    int x1Source, int y1Source, int x2Source, int y2Source,  
    Color color, ImageObserver obs)
```

From `java.swing.DefaultBoundedRangeModel`:

```
public void setRangeProperties(  
    int newValue, int newExtent,  
    int newMin, int newMax,  
    boolean isAdjusting)
```

From `java.swing.JOptionPane`:

```
public static int showConfirmDialog(Component parent, Object message, String title, int  
    optionType, int messageType, Icon icon)
```



EXERCISE 6 Long Parameter List. (Continued)

- A. For each declaration above, is there any cluster of parameters you might reasonably group into a new object?**

- B. Why might those signatures have so many parameters?**

- C. Look back at the JTable declaration (EXERCISE-5, earlier in the chapter). Do you see any clusters of parameters there?**

■ See Appendix A for solutions.

More Challenges

EXERCISE 7 Smells and Refactorings.

Consider these smells:

- A. Comments**
- B. Large Class**
- C. Long Method**
- D. Long Parameter List**

For each refactoring in the following list, write the letter for the smell(s) it might help cure:

- Duplicate Observed Data*
- Extract Class*
- Extract Interface*
- Extract Method*
- Extract Subclass*
- Introduce Assertion*
- Introduce Parameter Object*
- Preserve Whole Object*
- Rename Method*
- Replace Parameter with Method*

■ See Appendix A for solutions.





EXERCISE 8 Triggers.

Consider the smells described in this chapter (Comments, Large Class, Long Method, Long Parameter List).

A. Which of these do you find most often? Which do you create most often?

B. To stop children from sucking their thumbs, some parents put a bad-tasting or spicy solution on the child's thumb. This serves as a trigger that reminds the child not to do that. What triggers can you give yourself to help you recognize when you're just beginning to create one of these smells?

■ See Appendix A for solutions.

Conclusion

The smells in this chapter are the easiest to identify. They're not necessarily the easiest to fix.

There are other metrics that have been applied to software. Many of them are simply refinements of code length. Pay attention when things feel like they're getting too big.

There is not a one-to-one relationship between refactorings and smells; we'll run into the same refactorings again. For example, *Extract Method* is a tool that can fix many problems.

Finally, remember that a smell is an *indication* of a potential problem, not a *guarantee* of an actual problem. You will occasionally find *false positives*—things that smell to you, but are actually better than the alternatives. But most code has plenty of real smells that can keep you busy.







INTERLUDE 1 SMELLS AND REFACTORINGS

The smells (along with the refactorings commonly used to fix them) from Martin Fowler's book *Refactoring* are listed in Table I-1, and the refactorings are listed in Table I-2.

INTERLUDE I1.1

Tally. Put a tally mark by each refactoring for each time a smell references it.

INTERLUDE I1.2

Refactorings that Fix the Most Smells. Which refactorings fix the most smells?

■ See Appendix A for solution.

INTERLUDE I1.3

Refactorings Not Mentioned. Which refactorings aren't mentioned by any of the smells? Why not?

■ See Appendix A for solutions.

INTERLUDE I1.4

Other Smells. Does this list suggest any other smells you might want to be aware of?

■ See Appendix A for solutions.



TABLE I.1 Smells and Their Associated Refactorings (from Fowler's *Refactoring*, back cover)

SMELL	COMMON REFACTORINGS
Alternative Classes with Different Interfaces	<i>Rename Method, Move Method</i>
Comments	<i>Extract Method, Introduce Assertion</i>
Data Class	<i>Move Method, Encapsulate Field, Encapsulate Collection</i>
Data Clump	<i>Extract Class, Introduce Parameter Object, Preserve Whole Object</i>
Divergent Change	<i>Extract Class</i>
Duplicated Code	<i>Extract Method, Extract Class, Pull Up Method, Form Template Method</i>
Feature Envy	<i>Move Method, Move Field, Extract Method</i>
Inappropriate Intimacy	<i>Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate</i>
Incomplete Library Class	<i>Introduce Foreign Method, Introduce Local Extension</i>
Large Class	<i>Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object</i>
Lazy Class	<i>Inline Class, Collapse Hierarchy</i>
Long Method	<i>Extract Method, Replace Temp with Query, Replace Method with Method Object, Decompose Conditional</i>
Long Parameter List	<i>Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object</i>
Message Chains	<i>Hide Delegate</i>
Middle Man	<i>Remove Middle Man, Inline Method, Replace Delegation with Inheritance</i>
Parallel Inheritance Hierarchies	<i>Move Method, Move Field</i>
Primitive Obsession	<i>Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy</i>
Refused Bequest	<i>Replace Inheritance with Delegation</i>
Shotgun Surgery	<i>Move Method, Move Field, Inline Class</i>
Speculative Generality	<i>Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method</i>
Switch Statements	<i>Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object</i>
Temporary Field	<i>Extract Class, Introduce Null Object</i>

TABLE I.2 Refactorings (from Fowler, *Refactoring*, inside front cover)

<i>Add Parameter</i>	<i>Pull Up Constructor Body</i>
<i>Change Bidirectional Association to Unidirectional</i>	<i>Pull Up Field</i>
<i>Change Reference to Value</i>	<i>Pull Up Method</i>
<i>Change Unidirectional Association to Bidirectional</i>	<i>Push Down Field</i>
<i>Change Value to Reference</i>	<i>Push Down Method</i>
<i>Collapse Hierarchy</i>	<i>Remove Assignment to Parameters</i>
<i>Consolidate Conditional Expression</i>	<i>Remove Control Flag</i>
<i>Consolidate Duplicate Conditional Expression</i>	<i>Remove Middle Man</i>
<i>Convert Procedural Design to Objects</i>	<i>Remove Parameter</i>
<i>Decompose Conditional</i>	<i>Remove Setting Method</i>
<i>Duplicate Observed Data</i>	<i>Rename Method</i>
<i>Encapsulate Collection</i>	<i>Replace Array with Object</i>
<i>Encapsulate Downcast</i>	<i>Replace Conditional with Polymorphism</i>
<i>Encapsulate Field</i>	<i>Replace Constructor with Factory Method</i>
<i>Extract Class</i>	<i>Replace Data Value with Object</i>
<i>Extract Hierarchy</i>	<i>Replace Delegation with Inheritance</i>
<i>Extract Interface</i>	<i>Replace Error Code with Exception</i>
<i>Extract Method</i>	<i>Replace Exception with Test</i>
<i>Extract Subclass</i>	<i>Replace Inheritance with Delegation</i>
<i>Extract Superclass</i>	<i>Replace Magic Number with Symbolic Constant</i>
<i>Form Template Method</i>	<i>Replace Method with Method Object</i>
<i>Hide Delegate</i>	<i>Replace Nested Conditional with Guard Clause</i>
<i>Hide Method</i>	<i>Replace Parameter with Explicit Methods</i>
<i>Inline Class</i>	<i>Replace Parameter with Method</i>
<i>Inline Method</i>	<i>Replace Record with Data Class</i>
<i>Inline Temp</i>	<i>Replace Subclass with Fields</i>
<i>Introduce Assertion</i>	<i>Replace Temp with Query</i>
<i>Introduce Explaining Variable</i>	<i>Replace Type Code with Class</i>
<i>Introduce Foreign Method</i>	<i>Replace Type Code with State/Strategy</i>
<i>Introduce Local Extension</i>	<i>Replace Type Code with Subclasses</i>
<i>Introduce Null Object</i>	<i>Self Encapsulate Field</i>
<i>Introduce Parameter Object</i>	<i>Separate Domain from Presentation</i>
<i>Move Field</i>	<i>Split Temporary Variable</i>
<i>Move Method</i>	<i>Substitute Algorithm</i>
<i>Parameterize Method</i>	<i>Tease Apart Inheritance</i>
<i>Preserve Whole Object</i>	

