Code smell (ii)

Example code smells

- Duplicated code
- Long method
- Large class
- Long parameter list
- Message chain
- Feature envy
- Switch statements
- Data class
- Speculative generality
- Temporary field
- Refused bequest

Message chain

- Long list of method calls:
  customer.getAddress().getState()
  window.getBoundingbox().getOrigin().getX()

- How to change this?

Message chain

- Long list of method calls:
  customer.getAddress().getState()
  window.getBoundingbox().getOrigin().getX()

- Replace with shorter calls:
  customer.getState()
  window.leftBoundary()

Data class

- Class has no methods except for getter and setters
- What to do:
  - Look for missing methods and move them to the class
  - Merge with another class

Switch statement

- (Long) if-else
- Switch case case case

- How to change?

Library example

```
class Book : Element …
class Collection : Element …


int computeWords(Element e) {
    if (!e.hasChildren()) { // e instanceof Book
        return ((Book)e).getBookWords();
    } else {
        return ((Collection)e).getTotalWords();
}
```

Library example

```
int computeWords(Element e) {
    if (!e.hasChildren()) { // e instanceof Book
        return ((Book)e).getBookWords();
    } else {
        return ((Collection)e).getTotalWords();
```

- Replace with a new method:

```
int computeWord(Element e) {
     return e.getWord();
}
```

Speculative generality

- What are the examples?

Speculative generality

- Interfaces/abstract classes that are implemented only one class
- Unused parameters

Temporary field

- Instance variable is only used during part of the lifetime of an object
- For example, it is only used while the object is initialized

- Move variable into another object (perhaps a new class)

Refused bequest

- A is a subclass of B
- A
  - Overrides inherited methods of B
  - Does not use some variables of B
  - Does not use some methods of B

Refused bequest

- A is a subclass of B
- A
  - Overrides inherited methods of B
  - Does not use some variables of B
  - Does not use some methods of B
- Give A and B a common superclass and move common code into it

Other smells

- Non-localized plans
- Too many bugs
- Too hard to understand
- Too hard to change

Summary

- Code smells are code pieces with potentially bad design
- Fairly subjective
  - Fowler: “You will have to develop your own sense of how many instance variables are too many instance variables and how many lines of code in a method are too many lines.”

Refactoring

Definitions

- Changing/improving the code structure w/o changing the program semantics

Key principles in refactoring

- Where to refactor
  - Code smell
- Refactor to what
  - Is it worthwhile to refactor?

- How to refactor?
  - What to change? (don’t miss anything!)
  - What are the steps? (keep each step as small as possible!)
  - Testing after every step of change

- Use automated refactoring tool whenever you can

Example 0

- What if the name of a method is not clear?

- Why should we make this change?
  - Make sure the name reflect the content → easy for future maintenance

What to change?

- Declaration
- Caller
- Subclasses and superclasses

How to change?

- Check if the method is inherited from super class
- Create a new method, declare it, copy the code
- Let the old method calls the new method
  - If the old method is used in many places
- Replace the old method every place it is called
- Remove the old method

Example 1

- What if the callee needs more information from the caller?

- Why do we need this? Any alternative?
  - When new functionality is added to a function, we need …
  - Alternatives: try to derive this information from other parameters or member fields
  - Be careful --- too many parameters make the code smell

What needs to be changed?

Steps to perform

- Check superclasses and subclasses
- Make copy of old method, add parameter
- Change body of old method so that it calls new one
- Find all references to the old method and change them to refer to the new
- Test should run after each change
- Remove old method

Example 2

- What if the parameter list is too long?
  ➔ Create parameter object

When is this change worthwhile?

- Many methods have same parameters
- The parameter list is *very* long

- The danger is creating too many data classes (code smell)

Introduce Parameter Object (1)

- Make a new class for the group of parameters
- Use Add Parameter for the new class
  - Use a new object for the parameter in all the callers
- For each of the original parameters: ...

Introduce Parameter Object (2)

- For each of the original parameters:
  - Modify caller to store parameter in the new object and omit parameter from call
  - Modify method body to omit original parameter and to use the value stored in the new parameter
  - If method body calls another method with parameter object, use existing parameter object instead of making a new one

```
class Account …
    double getFlowBetween(Date start, Date end) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            Date date = each.getDate();
            if (date.equals(start) || date.equals(end) ||
                (date.after(start) && date.before(end))) {
                result += each.getValue();
            }
        }
        return result;
    }
```

```
class DateRange {
    DateRange (Date start, Date end) {
        _start = start;
        _end = end;
    }
    Date getStart() {
        return _start;
    }
    Date getEnd() {
        return _end;
    }
    private final Date _start;
    private final Date _end;
}
```

```
class Account …
    double getFlowBetween(Date start, Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            Date date = each.getDate();
            if (date.equals(start) || date.equals(end) ||
                (date.after(start) && date.before(end))) {
                result += each.getValue();
            }
        }
        return result;
    }
```

Changing callers (1)

```
double flow = anAccount.getFlowBetween(startDate, endDate);
```

```
double flow = anAccount.getFlowBetween(startDate, endDate, new
  DateRange(null, null))
```

Changing callers (2)

```
double flow = anAccount.getFlowBetween(startDate, endDate, new
  DateRange(null, null))
```

```
double flow = anAccount.getFlowBetween(endDate, new
  DateRange(startDate, null))
```

```
class Account …
    double getFlowBetween(Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            Date date = each.getDate();
            if (date.equals(range.getStart()) || date.equals(end) ||
                (date.after(range.getStart()) && date.before(end))) {
                result += each.getValue();
            }
        }
        return result;
    }
```

```
class Account …
    double getFlowBetween(DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            Date date = each.getDate();
            if (date.equals(range.getStart()) ||
                date.equals(range.getEnd()) ||
                (date.after(range.getStart()) &&
                    date.before(range.getEnd()))) {
                result += each.getValue();
            }
        }
        return result;
    }
```

Changing callers (3)

```
double flow = anAccount.getFlowBetween(endDate, new
  DateRange(startDate, null))
```

```
double flow = anAccount.getFlowBetween(new DateRange(startDate,
  endDate))
```

Introduce Parameter Object

After introducing a parameter object, look to see if code should be moved to its methods

```
class DateRange …
  boolean includes (Date arg) {
      return (arg.equals(_start) || arg.equals(_end) ||        (arg.after(_start) &&
      arg.before(_end)));
  }
```

```
class Account …
    double getFlowBetween(DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (range.includes(each.getDate())) {
                result += each.getValue();
            }
        }
        return result;
    }
```

Lessons

- Refactorings should be small
  - Test cases
  - Version control
- Check after each step to make sure you didn’t make a mistake
- One refactoring leads to another
- Major change requires many refactorings

More OO refactoring

```
Class Person{
   private:
     string First;
     string Last;
     string Address;
}
```

```
Class Female: public Person{
   public:
     void printName() {
          cout << “Ms. ”<<First<<“ “<<Last;
     }
     void printAddress(){
          cout << “Ms. ”<<First<<“ “
                    <<Last<<endl<<Address;
     }
}
```

```
Class Male: public Person{
   public:
     void printName() {
          cout << “Mr. ”<<First<<“ “<<Last;
     }
     void printAddress(){
          cout << “Ms. ”<<First<<“ “
                    <<Last<<endl<<Address;
     }
}
```

Example 4 pull up method

- What if there is code duplication across two classes?

- Why is it worthwhile?
- What to do?
- What are the steps?

Example 5 push down methods

- When does that happen?
- What to do?

```
class JobItem ...
public JobItem (int unitPrice, int
quantity, boolean isLabor,
Employee employee) {
  _unitPrice = unitPrice;
  _quantity = quantity;
  _isLabor = isLabor;
  _employee = employee;
}
public int getTotalPrice() {
  return getUnitPrice() * _quantity;
}
public int getUnitPrice(){
  return (_isLabor) ?
_   employee.getRate():
    _unitPrice;
}
```

```
public int getQuantity(){
  return _quantity;
}
public Employee getEmployee() {
  return _employee;
}
private int _unitPrice;
private int _quantity;
private Employee _employee;
private boolean _isLabor;
```

```
class Employee...
public Employee (int rate) {
  _rate = rate;
}
public int getRate() {
  return _rate;
}
private int _rate;
```

```
class JobItem ...
public JobItem (int quantity) {
   _quantity = quantity;
}
public int getTotalPrice() {
  return getUnitPrice() * _quantity;
}
public int getUnitPrice();//virtual
```

```
public int getQuantity(){
  return _quantity;
}
private int _quantity;
```

```
JobItem j1=new JobItem(XX,2,True,Bob);
JotItem j2=new JobItem(10,3,False,XX);
```

```
Class NLaborItem: JobItem

private int _unitPrice;
Public NLaborItem(int q, int up){…};
Public int getUnitPrice(){
   return  _unitPrice:
}
```

```
Class LaborItem: JobItem

private Employee _employee;
public Employee getEmployee() {
  return _employee;
}
Public LaborItem(int q,Employee e){…};
Public int getUnitPrice(){
   return  _employee.getRate():
}
```

Example 6: extract sub-class

- Extract sub-class vs. extract class
  - When to use what?
- What to do?
- What steps?

```
class Employee...
public Employee (String name, String id, int annualCost)
{
  _name = name;
  _id = id;
  _annualCost = annualCost;
}
public int getAnnualCost() {
  return _annualCost;
}
public String getId(){
  return _id;
}
public String getName() {
  return _name;
}
private String _name;
private int _annualCost;
private String _id;
```

```
public class Department...
public Department (String name) {
  _name = name;
}
public int getTotalAnnualCost(){
  Enumeration e = getStaff();
  int result = 0;
  while (e.hasMoreElements()) {
    Employee each = (Employee) e.nextElement();
    result += each.getAnnualCost();
  }
  return result;
}
public int getHeadCount() {
  return _staff.size();
}
public Enumeration getStaff() {
  return _staff.elements();
}
public void addStaff(Employee arg) {
  _staff.addElement(arg);
}
public String getName() {
  return _name;
}
private String _name;
private Vector _staff = new Vector();
```

Example 7: extract super-class

- When to do?
- What to do?
- Steps?

Change design XOR functionality

- Separate changing behavior from refactoring
  - Changing behavior requires new tests
  - Refactoring must pass all tests
- Only refactor when you need to
  - Before you change behavior
  - After you change behavior
  - To understand

Some refactorings

- Composing methods
  - Extract method
  - Inline method
  - Inline temporary variable
  - Introduce explaining variable
  - Split temporary variable
  - Replace method with method object
  - …

More refactorings

- Moving features between objects
- Organizing data
- Simplifying conditional expressions
- Making method calls simpler
- Generalization
- …

Automated refactoring support

- Deciding where to refactor
  - Tools for measuring cohesion, size, etc.
  - Tools for measuring code duplication/cloning
- Performing the change
  - Refactoring Browser for Smalltalk, first
  - Over a dozen of tools for Java
  - Eclipse