6CS002

Bad Smells in Code

Dr. Kevan Buckley

Lecture Outcomes

- To be familiar with *Bad Smells*
  - *Indicators that your code is structured poorly*

- By the end of this session you should be able to start evaluating code with respect to bad smells.
  - You will be able to start Task 1 of your portfolio

References

- The material presented here is covered in:
  - Refactoring: Improving the Design of Existing Code by Martin Fowler.
    - Available from Safari and in the Harrison Library
    - Concentrate on pages 75 to 88 (http://www.laputan.org/pub/patterns/fowler/smells.pdf)
    - See http://sourcemaking.com/refactoring/bad-smells-in-code
- An alternative perspective is presented in:
  - Clean Code: A Handbook for Agile Software Craftsmanship by Robert C. Martin
    - Some useful additions, but a lot that is out of the scope of our work
- Try to use Fowler as the primary reference, but if there are things that you do not understand try reading the alternative book.

Bad Smells in Code

- *Smells* are the symptoms of bad code
- Fowler (1999) provides a catalogue of smells and refactorings
  - When you identify a smell, you select one of the suggested refactorings then carefully follow a step by step process to safely refactor the code.
- Code should be self-documenting
  - There should be no need for comments
  - If you can't understand it there are probably bad smells
- See the following for a summary
  - http://wiki.java.net/bin/view/People/SmellsToRefactorings
- See the following for Fowler's catalogue
  - http://www.refactoring.com/catalog/index.html

Bad Smells in Code

- Duplicate code
  - Results from "cut and paste" coding or poor design
  - Common code can be put into methods that can be called from many places.
  - Breaks the DRY principle
- Long method
  - The longer a method is, the more difficult it is to understand.
  - Short methods with good names improve readability.
  - Methods may be reusable elsewhere in the program
  - Poor cohesion breaks SRP
    - (you can apply these principals to structured programming)
- Large class
  - When a class is trying to do too much it can have too many instance variables and is susceptible to duplication and confusion.
    - e.g. a complex GUI application could be split into separate classes for the data and behaviour (see http://java.sun.com/blueprints/patterns/MVC.html)
  - Poor cohesion breaks SRP

Bad Smells in Code

- Long parameter list
  - Hard to understand (and remember) lots of parameters
  - Often better to pass a small number of objects
  - Indication that a method is trying to do many tasks
  - Poor cohesion breaks SRP
- Divergent Change
  - Multiple changes have been made to a class resulting in diverse responsibilities
    - Better split into a number of smaller classes
    - Changes need only be made to the relevant class
  - Breaks SRP
- Shotgun surgery
  - Opposite of divergent change
    - Changes have a small effect on lots of classes
  - Combine small classes into one larger one
    - All changes made in same place

Bad Smells in Code

- Feature envy
  - A method that is more interested in a class other than the one it is actually in.
  - The method should be moved to the class it operates on.
  - Poorly defined responsibilities
- Data clumps
  - Same data items occur together in lots of places.
  - Should be grouped together into a class.
  - Need to consolidate with small number of cohesive classes
- Primitive obsession
  - Programmers new to OO are reluctant (or do not consider) using classes for working on simple data items.
    - Even for a single numeric value a class can be useful –e.g.validation

Bad Smells in Code

- Switch statements
  - The same switch statement occurs in many places throughout the program
    - Including a new case requires the same change in several places
  - Need to make better use of polymorphism
- Parallel inheritance hierarchies
  - Changes to one class hierarchy always require similar changes to another.
    - Both classes embody different aspects of the same decision.
    - Eliminate one of the hierarchies by moving the features to the other.
  - Breaks the DRY principle
- Lazy class
  - After a lot of changes have been made to a program there might be classes that do not do much work
    - They can often be eliminated

Bad Smells in Code

- Speculative generality
  - Class is complicated by hooks for features that are not required – simplify
  - Too much planning for the future has been done.
  - Is the design really flexible to change?
- Temporary field
  - Some instance variables are only used in special cases
  - Should remove the instance variables and methods that operate on them to a new class.
  - Consider SRP. If inheritance is used consider LSP
- Message chains
  - There is a chain of method calls between objects to access some data.
    - Changes to intermediate objects can be problematic

Bad Smells in Code

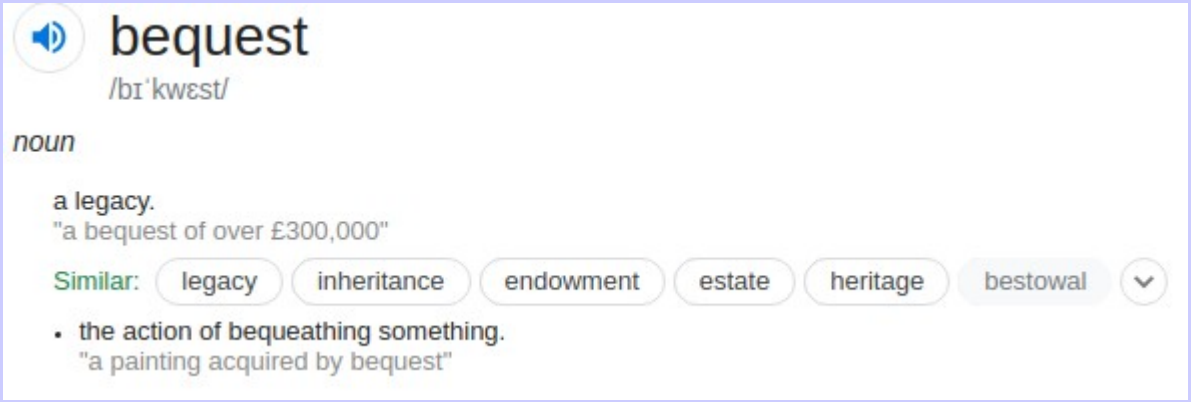
- Middle man
  - Sometimes a class delegates all its work to another class
    - The middleman can be eliminated
- Inappropriate intimacy
  - Occurs when classes spend a lot of time "delving into each others' private parts"
  - Can use another class to consolidate the common interests
- Alternative classes with different interfaces
  - There may be several methods that do the same thing that exist in different classes.
  - May eliminate code in a class and use another or use a common superclass.
  - Breaks the DRY principle

Bad Smells in Code

- Incomplete Library Class
  - Required if the same *general* functionality is used in multiple classes.
    - If the same code exists in several classes, form a library.
  - The general code in not really part of the responsibilities of any class.
  - If general code exists outside the library most suited to it, then the library is incomplete.
- Data Class
  - Just instance variables, constructors and accessor methods.
  - Need to identify where the class is being used and see if any responsibilities can be moved into the class.
  - Data classes are not always bad. They can be used to consolidate data clumps into cohesive units.

Bad Smells in Code

- Refused Bequest



  - A refused bequest exists when a child class does not want its parent's features
  - If no subclasses want the feature then it is a refused bequest

- Comments
  - If you need a lot of comments to explain code, the code could probably be improved

Comments

- Inappropriate information
  - e.g. change histories cause clutter
- Obsolete comments
  - A comment that has gotten old, incorrect, obsolete
- Redundant comments
  - e.g. i++ // increment I
  - e.g. javadoc comments on accessor methods
- Poorly written comments
  - If you need a comment, write it well
- Commented out code
  - Delete it

Martin pp. 286

Examples (1)

```
public static void main(String[] args) {
   System.out.println("Hello");
   System.out.println("Hello");
   System.out.println("Hello");
   System.out.println("Hello");
   System.out.println("Hello");
   System.out.println("Hello");

   int width = 2;
   int height = 3;
   int depth = 4;
   int baseArea = width * depth;
   int boxVolume = width * depth * height;
   System.out.printf("Base is %d square metres.\n", baseArea);
   System.out.printf("Box is %d cubic metres.\n", boxVolume);
}
```

Code Examples (2)

```
private int a=1;
private int b=2;
private int c=3;
private int d=4;

public void method1(int x){
  b = x * x + c * d;
  c = d * d + x;
}

public void method2(int x, int y){
  b = x * x + c * d;
  c = d * d + x;
  d = y;
}
```

Code Examples (3)

```
public class Class02 {
  public void calculate(int [][]a, int [][]b, int [][]c, int [][]d){
    for(int i=0;i<a.length;i++){
      for(int j=0;j<a[i].length;j++){
        c[i][j] = a[i][j] - b[i][j];
      }
    }
    for(int i=0;i<a.length;i++){
      for(int j=0;j<a[i].length;j++){
        d[i][j] = a[i][j] + b[i][j];
      }
    }
    System.out.println();
    for(int i=0;i<a.length;i++){
      for(int j=0;j<a[i].length;j++){
        System.out.printf("%3d",c[i][j]);
      }
      System.out.println();
    }
    System.out.println();
    for(int i=0;i<a.length;i++){
      for(int j=0;j<a[i].length;j++){
        System.out.printf("%3d",d[i][j]);
      }
      System.out.println();
    }
  }
```

```
public static void main(String[] args) {
  int [][]w = {{9,8,9},{4,5,6},{5,6,7}};
  int [][]x = {{1,2,3},{1,2,2},{2,1,1}};
  int [][]y = new int[3][3];
  int [][]z = new int[3][3];

  new Class02().calculate(w,x,y,z);

}
```

```
public class Class20 {
  class FirstName {
    String name;

    public FirstName(String name) {
      this.name = name;
    }

    public String getInitial() {
      return name.substring(0, 1);
    }

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

```
class LastName {
  String name;

  public LastName(String name) {
    this.name = name;
  }

  public String getInitial() {
    return n
  }

  public Str
    return n
  }
}
```

```
class FullName {
  FirstName fn;
  LastName ln;

  public FullName(FirstName fn, LastName ln)
    this.fn = fn;
    this.ln = ln;
  }

  public FirstName getFirstName() {
    return fn;
  }

  public LastName getLastName() {
    return ln;
  }
}
```

```
ublic void run() {
 FullName me = new FullName(new FirstName("Kevan"), new LastName("Buckley"));
 System.out.println("My name is " + me.getFirstName() + " "
     + me.getLastName());
 System.out.printf("My initials are %s %s\n", me.getFirstName().getInitial()
     .toUpperCase(), me.getLastName().getInitial().toUpperCase());
```

```java
public class Student {
  String id;
  String surname;
  String forename;
  String email;

  public Student(String id, String surname,
                 String forename, String email) {
    this.id = id;
    this.surname = surname;
    this.forename = forename;
    this.email = email;
  }

  public double getArea(Class30 a) {
    return a.width * a.height; // multiply width by height
  }

  public double applyVAT(double price){
    return price * 1.20;
  }
}
```

```java
public class Class30 {
  public double width;
  public double height;
}
```

```java
public class Class31 {
  public static void main(String[] args) {
    Student s = new Student("6060842", "Kevan",
                "Buckley", "K.A.Buckley@wlv.ac.uk");
    Class30 c30 = new Class30();
    c30.width = 2;
    c30.height = 3;
    double a = s.getArea(c30);
    System.out.println(a);
  }
```

```java
abstract class Shape {
  private static int count = 0
  private int id;
  protected double width;
  protected double height;
  protected double radius;

  public Shape(){
    id = count++;
  }

  public String toString() {
    return "Shape " + id + " has area
  }

  abstract public double getArea();

  protected int getId(){
    return id;
  }
}
```

```java
class Rectangle extends Shape {
  public  Rectangle(double width, double heigh
    this.width = width;
    this.height = height;
  }

  public double getArea() {
    return width * height;
  }
}
```

```java
class Circle extends Shape {
  public Circle(double radius){
    this.radius = radius;
  }

  public double getArea() {
    return Math.PI * radius * radius;
  }
}
```

```java
public class Class10 {
  public static void main(String[] args) {
    Shape s1 = new Rectangle(2, 3);
    Shape s2 = new Circle(1);
    System.out.println(s1);
    System.out.println(s2);
  }
}
```

Next

Refactoring

Summary and Things To Do

- You have just learnt about the bad smells documented in the Fowler book.
- You will receive access to source code for a large application:
  - Make a word processor document of the complete source code that you have been given. You can line number it with “cat -n”. In your word processor, tweak the font size and margins so that there is no wrap around.
  - Start annotating your printout with notes that explain what certain methods, variables etc. do and any occurrences of bad smells.
    - Complete the Bad Smells Checklist as you go.
  - You will need to submit your annotated printout as a portfolio task.

Start as soon as possible.