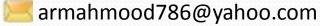
Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)





in pk.linkedin.com/in/armahmood

www.twitter.com/alphapeeler

www.facebook.com/alphapeeler

S abdulmahmood-sss S alphasecure

🥙 armahmood 786@hotmail.com

Bhttp://alphapeeler.sf.net/me

alphasecure@gmail.com

kttp://alphapeeler.sourceforge.net

t http://alphapeeler.tumblr.com

🕠 armahmood786@jabber.org

🤰 alphapeeler@aim.com

🛐 mahmood_cubix 🛭 👯 48660186

alphapeeler@icloud.com

B http://alphapeeler.sf.net/acms/

Refactoring-III

Bad Smells in Code

Inline Temp

- Problem: You have a temporary variable that's assigned a result of simple expression & nothing more.
- **Solution:** Replace the references to the variable with the expression itself.

```
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}

boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

Benefits

• You can marginally improve the readability of the program by getting rid of the unnecessary variable.

Replace Temp with Query

- **Problem:** You place the result of an expression in a local variable for later use in your code.
- **Solution:** Move the entire expression to a separate method and return the result from it. <u>Query the method instead of using a variable</u>. Incorporate the new method in other methods, if necessary.

```
double calculateTotal() {
  double basePrice = quantity *
        itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
}
```

```
double calculateTotal() {
  if (basePrice() > 1000) {
    return basePrice() * 0.95;
  }
  else {
    return basePrice() * 0.98;
  }
}
double basePrice() {
  return quantity * itemPrice;
}
```

Split Temporary Variable

- **Problem:** have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.
- Solution: Make a separate temporary variable for each assignment.

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);

final double perimeter = 2 * (height + width);
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```

Remove Assignments to Parameters

- **Problem:** Some value is assigned to a parameter inside method's body.
- **Solution:** Use a local variable instead of a parameter.

```
int discount(int inputVal, int quantity) {
  if (inputVal > 50) {
    inputVal -= 2;
 // ...
int discount(int inputVal, int quantity) {
  int result = inputVal;
  if (inputVal > 50) {
    result -= 2;
```

Bad Smells in Code

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements

- Parallel Interface Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Incomplete Library Class
- Data Class
- Refused Bequest

Few solutions to Bad Smells

Duplicated Code

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them. <u>solution</u>: perform EXTRACT METHOD and invoke the code from both places.

Long Method

The longer a procedure is the more difficult it is to understand.

<u>solution</u>: perform EXTRACT METHOD or Decompose Conditional or Replace Temp with Query.

Large class

When a class is trying to do too much, it often shows up as too many instance variables.

<u>solution</u>: perform EXTRACT CLASS or SUBCLASS

Long Parameter List

With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs *solution*: Use REPLACE PARAMETER with METHOD when you can get the data in one parameter by making a request of an object you already know about.

Shotgun Surgery

This situation occurs when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

<u>solution</u>: perform MOVE METHOD/FIELD or INLINE Class bring a whole bunch of behavior together.

Feature Envy

It is a method that seems more interested in a class other in the one that it is in.

solution: perform MOVE METHOD or EXTRACT METHOD on the jealous bit and get it home.

Switch Statements

They are generally scattered throughout a program. If you add or remove a clause in one switch, you often have to find and repair the others too.

<u>solution</u>: Use EXTRACT METHOD to extract the switch statement and then MOVE METHOD to get it into the class where the polymorphism is needed.

 Data Classes are classes, which only contain fields, getters and setters. Or even worse: only public fields.
 They are just dumb data holders.

fieldB

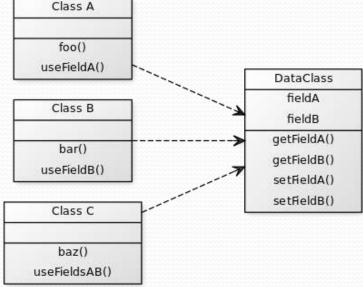
getFieldA() getFieldB() setFieldA()

setFieldB()

- Are they really so bad?
- When you do some quick prototyping and need to store and transfer data without too much coding effort, they are sometimes a good compromise.
- But when your project grows, you should fix this to accomplish one major goal of object orientation: <u>Low Coupling between classes</u>.

 But when your project grows, you should fix this to accomplish one major goal of object orientation: Low Coupling between classes. You want as few dependencies between your classes as possible. But currently everything that's done with your data, is

done in other classes:



• Everyone but the Data Class itself uses its data. This results in high coupling.

- What to do about it?
- To achieve a low coupling and a high cohesion you need to put the methods in the classes they're working on most. If you already introduced a Data Class you either need to move the methods closer to the fields or vice versa:

DataClass Class A fieldA foo() fieldB getFieldA() Class B getFieldB() setFieldA() bar() setFieldB() useFieldA() Class C useFieldB() useFieldsAB() baz()

 Moving the methods into the Data Class itself eliminates the dependencies.

• If no more class accesses the fields, you can even remove their getters and setters and leave them private:

DataClass
fieldA
fieldB
useFieldA()
useFieldB()
useFieldsAB()

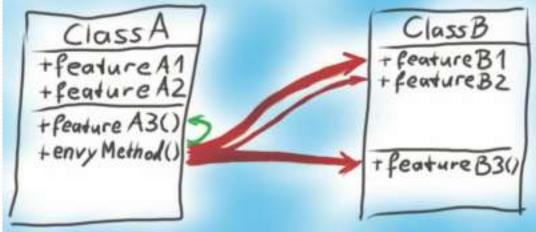
Now our *DataClass* no longer smells like a Data Class.
 We reduced the coupling and from this point we could change the ex-*DataClass* without effecting the other classes. Enjoy your great code

What is Feature Envy?

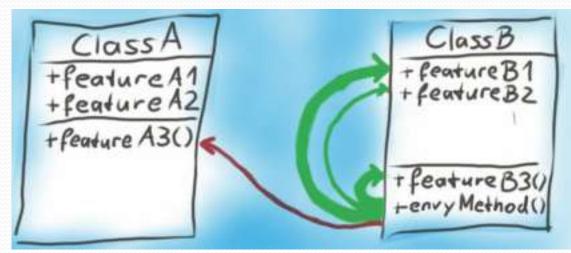
- Feature Envy is a Code Smell which occurs in methods.
 A method has Feature Envy on another class, if it uses more features (i.e. fields and methods) of another class than of its own.
- <u>Ideodorant</u> is an eclipse plugin, which promises finding feature envy and <u>visualizing</u> it. I wanted to use it for a simple example, but the plugin didn't work properly.
- https://marketplace.eclipse.org/content/jdeodorant
- https://www.youtube.com/watch?v=LtH8uFoepVo
- JDeodorant: Code Smell Visualization Demo

Feature Envy

Displaying the dependencies of a method. Thicker arrows mean more uses of the same feature.



After moving the envy method to the desired class, the green arrows total thickness exceed the others. We have no more Feature Envy.



Lazy Class

Signs and Symptoms

 Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted.

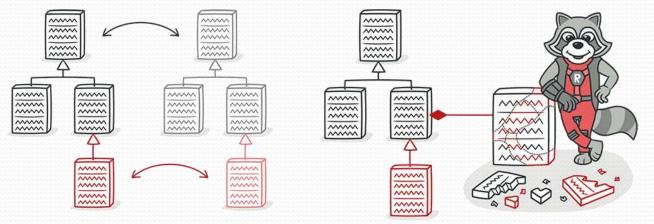


Lazy Class

- Reasons for the Problem
- Perhaps a class was designed to be fully functional but after some of the refactoring it has become ridiculously small.
- Or perhaps it was designed to support future development work that never got done.
- Treatment
- Components that are near-useless should be given the <u>Inline Class</u> treatment.
- For subclasses with few functions, try <u>Collapse</u> <u>Hierarchy.</u>

Parallel Inheritance Hierarchies

- Signs and Symptoms
- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.



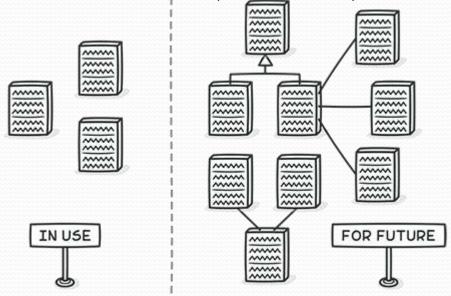
Treatment
Move Method and
Move Field.

- Reasons for the Problem
- All was well as long as the hierarchy stayed small. But with new classes being added, making changes has become harder and harder.

Speculative Generality

Signs and Symptoms

• There's an unused class, method, field or parameter.



- Reasons for the Problem
- code is created "just in case" to support anticipated future features that never get implemented. As a result, code becomes hard to understand and support.

Speculative Generality

Treatment

- Unused abstract classes -> <u>Collapse Hierarchy</u>.
- Unnecessary delegation of functionality to another class can be eliminated via <u>Inline Class</u>.
- Unused methods? -> <u>Inline Method</u> to get rid of them.
- Methods with unused parameters -> Remove Parameter.

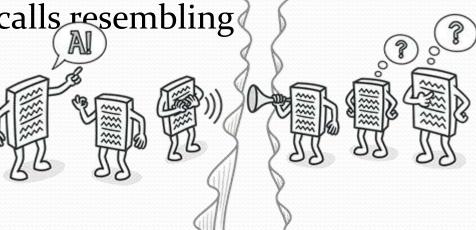
• Unused fields can be simply deleted.

Message Chains

Signs and Symptoms

In code you see a series of calls resembling

• a->b()->c()->d()



Reasons for the Problem

• A message chain occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.

Message Chains

- Treatment
- To delete a message chain, use <u>Hide Delegate</u>.
- Sometimes it's better to think of why the end object is being used. Perhaps it would make sense to use <u>Extract</u> <u>Method</u> for this functionality and move it to the beginning of the chain, by using <u>Move Method</u>.

Benefits of Refactoring

Refactoring is useful to any program that has at least one of the following shortcomings:

- Programs that are hard to read are hard to modify.
- Programs that have duplicate logic are hard to modify

Continued...

 Programs that require additional behavior that requires you to change running code are hard to modify.

 Programs with complex conditional logic are hard to modify.

Refactoring Risks

- Introducing a failure with refactoring can have serious consequences.
- When done on a system that is already in production, the consequences of introducing bugs without catching them are very severe.

Costs of Refactoring

<u>Language / Environment :</u>

Depends on how well the operations on the source code are supported. But in general the cost of applying the basic text modifications should be bearable.

Testing:

Relies heavily on testing after each small step and having a solid test suite of unit tests for the whole system substantially reduces the costs which would be implied by testing manually

Continued...

Documentation:

Costs of updating documentation of the project should not be underestimated as applying refactorings involves changes in interfaces, names, parameter lists and so on.

All the documentation concerning these issues must be updated to the current state of development.

Continued...

System:

The tests covering the system need an update as well as the interfaces and responsibilities change. These necessary changes can contribute to higher costs as tests are mostly very dependent on the implementation.

When to put Off Refactoring?

- Concerned code is neither able to compile or to run in a stable manner, it might be better to throw it away and rewrite the software from scratch.
- When a deadline is very close. Then it would take more time to do the refactoring than the deadline allows.