

Bad Smells

by Martin Fowler, Kent Beck, Leo Scott, Randy Ballew

Table of contents

1 Bad Smells in Code by Kent Beck and Martin Fowler.....	3
1.1 Duplicated Code.....	3
1.2 Long Method.....	3
1.3 Large Class.....	4
1.4 Long Parameter List.....	4
1.5 Divergent Change.....	4
1.6 Shotgun Surgery.....	5
1.7 Feature Envy.....	5
1.8 Data Clumps.....	5
1.9 Primitive Obsession.....	6
1.10 Switch Statements.....	6
1.11 Parallel Inheritance Hierarchies.....	6
1.12 Lazy Class.....	6
1.13 Speculative Generality.....	7
1.14 Temporary Field.....	7
1.15 Message Chains.....	7
1.16 Middle Man.....	7
1.17 Inappropriate Intimacy.....	8
1.18 Alternative Classes with Different Interfaces.....	8
1.19 Incomplete Library Class.....	8
1.20 Data Class.....	8
1.21 Refused Bequest.....	9

1.22 Comments..... 9

1. Bad Smells in Code by Kent Beck and Martin Fowler

This is a summary from Martin Fowler's classic Refactoring: Improving the Design of Existing Code. See the [refactoring web site](#).

Duplicated Code	Lazy Class
Long Method	Speculative Generality
Large Class	Temporary Field
Long Parameter List	Message Chains
Divergent Change	Middle Man
Shotgun Surgery	Inappropriate Intimacy
Feature Envy	Alternative Classes with Different Interfaces
Data Clumps	Incomplete Library Class
Primitive Obsession	Data Class
Switch Statements	Refused Bequest
Parallel Inheritance Hierarchies	Comments

1.1. Duplicated Code

If you see the same code in more than one place, **refactor**.

- Use [Extract Method](#) to deal with duplicated code inside a class.
- When you have the same code in sibling classes, use [Extract Method](#) in both classes then [Pull Up Method](#). If code is similar then use [Extract Method](#) on the similar part. You may be able to then use [Form Template Method](#). If the methods do the same thing with different algorithms, choose the clearer algorithm and use [Substitute Algorithm](#).
- If unrelated classes have duplicated code then use [Extract Class](#) from one and use the new class in the other.

1.2. Long Method

Object programs live best and longest with short methods. The payoffs of indirection - explanation, sharing and choosing - are supported by little methods. Everybody knows short is good.

- 99% of the time, just use [Extract Method](#) to shorten a method. Find a part that goes

together and make a new method.

- Parameters and temporary variables get in the way of extracting methods. Use [Replace Temp with Query](#) to remove the temps and [Introduce Parameter Object](#) or [Preserve Whole Object](#) to slim down the parameter lists.
- If you still have too many temps and parameters, it's time for the heavy artillery: [Replace Method with Method Object](#)
- Look for comments. A block of code with a comment should be replaced by a method named to match the comment. **Even a single line is worth extracting if it needs explanation.**
- Look for conditionals and loops. Use [Decompose Conditional](#). Extract each loop into its own method.

1.3. Large Class

- Class doing too much = too many instance variables = duplicated code. Use [Extract Class](#) to bundle like variables together. Use [Extract Subclass](#) when the methods associated with the variables extend the class function. Look for sets of variables that are only used by the class part time and extract classes or subclasses.
- Eliminate redundancy. Turn five 100 line methods into five 10 line methods with another ten 2 line methods extracted from the original.
- Determine how clients use the class and use [Extract Interface](#). This can give you ideas on how to break up the class.
- For a GUI class move data and behavior to a separate domain object. [Duplicate Observed Data](#) shows how to handle keeping the duplicated data in sync.

1.4. Long Parameter List

Old school, pass in everything as parameters. Was better than global data. With objects don't have to pass in everything, just enough so it can get to everything it needs. Good because long parameter lists hard to understand, they become inconsistent/hard to use and always changing because you need access to new data.

- Use [Replace Parameter with Method](#) when you can get the data from an object you already know about. Use [Preserve Whole Object](#) to replace a bunch of data from one object with the object itself. For data without a home use [Introduce Parameter Object](#).
- The exception to the rule occurs when you want to avoid a dependency on an object. In this case you will need to pass the data as parameters, but pay attention to the pain involved. When the parameter lists is too long or changes too often, rethink the dependency.

1.5. Divergent Change

Software is meant to be soft. When we make a change we want to jump to a single clear point in the system and do it. Divergent change occurs when a class is changed in different ways for a different reasons. "these three methods for database changes vs these four for a new loan type", two objects would be better than one. Each object only changed for one type of change. Often only clear after the fact. Identify everything that changes for a particular cause and use [Extract Class](#) to put them all together.

1.6. Shotgun Surgery

Opposite of divergent change. Every time you make a kind of change you have to make a lot of little changes to a lot of different classes. Hard to find, easy to miss an important change.

- Use [Move Method](#) and [Move Field](#) to put all the change sites into one class. If a good place to put them does not exist, create one.
- Use [Inline Class](#) to bring a bunch of behavior together. You will get a small amount of divergent change.
- Divergent change or shotgun surgery, you want a 1 to 1 link between common changes and classes.

1.7. Feature Envy

Objects exist to package data with the processes used on that data. Beware of methods that are more interested in data in another class than with its own. Half a dozen getters on another object to calculate some value, bad.

- When a method clearly wants to be elsewhere use [Move Method](#) to get it there. Sometimes only a part suffers envy. Use [Extract Method](#) to capture the jealous bit and [Move Method](#) to put it where it wants to be.
- Not all cases cut-and-dried. May use features of many classes. Which on should it live with? Put it with the data it uses most. Made easier if [Extract Method](#) is used to break the method up into parts that uses different data.
- Some patterns break the rule. Strategy, Visitor, Self Delegation. The point is still to group things that change together.

1.8. Data Clumps

Often you see the same three or four data items together in lots of places: fields in classes or parameters in method signatures. They really ought to be an object. A good test: think of deleting one of the data values. Do the others still make sense? If not then this is an object waiting to be born.

- When the clumps are fields, use [Extract Class](#).

- When the clumps are parameters, use [Introduce Parameter Object](#) or [Preserve Whole Object](#) to slim them down.

Don't worry about only using some of the fields. You come out ahead even if you are just replacing two fields with the object.

1.9. Primitive Obsession

People new to objects are reluctant to use small objects for small tasks, such as money classes that combine number and currency, ranges with an upper and lower, and special strings such as telephone numbers and ZIP codes.

- To move out of the primitive cave use [Replace Data Value with Object](#).
- If the data value is a type code (and does not affect behavior) use [Replace Type Code with Class](#).
- If the type code is driving conditionals use [Replace Type Codes with Subclasses](#) or [Replace Type Code with State/Strategy](#).

For primitive fields that go together use [Extract Class](#). For primitives in parameter lists use [Introduce Parameter Object](#). If you are picking primitives out of an array use [Replace Array with Object](#).

1.10. Switch Statements

Switch statements breed duplication. End up scattered through out a program. Have to find them for each change (always end up missing one). Polymorphism is the OO answer.

- For type codes, use [Extract Method](#) to extract the switch statement and then [Move Method](#) to get it into the class where polymorphism is needed. Pick either [Replace Type Code with Subclasses](#) or [Replace Type Code with State/Strategy](#). Then use [Replace Conditional with Polymorphism](#) after you setup the inheritance structure.
- If it is just a few cases in a single method, polymorphism is overkill. [Replace Parameter with Explicit Methods](#) is a good option. If one of your cases is null, try [Introduce Null Object](#).

1.11. Parallel Inheritance Hierarchies

A special case of shotgun surgery. Eliminate the duplication by insuring that instances of one hierarchy refer to instances of the other. Use [Move Method](#) and [Move Field](#) to make the hierarchy on the referring class disappear.

1.12. Lazy Class

Each class cost money and brain cells to maintain and understand. A class that isn't doing enough to pay for itself should be eliminated. Changes over time with refactoring. For subclasses that are not doing enough use [Collapse Hierarchy](#). For weak components use [Inline Class](#).

1.13. Speculative Generality

"Oh, I think we will need the ability to do this kind of thing someday." Hooks and special cases to handle things that are not required. It all costs. Just get rid of it.

- Abstract class not doing much? Use [Collapse Hierarchy](#) to get rid of it.
- Remove unnecessary delegation with [Inline Class](#).
- Subject methods with unused parameters to [Remove Parameter](#).
- Bring method named with odd abstract name down to earth with [Rename Method](#).
- When the only users of a method are test cases, remove the method and the tests.

1.14. Temporary Field

Instance variables that are only set sometimes are hard to understand - you expect an object to need all its variables.

- Give it a real home. Use [Extract Class](#) to put the orphan variable and all the code that concerns it in one place. Use [Introduce Null Object](#) when the variable is just around to deal with the null special case.
- Complicated algorithms breed temporary fields to avoid passing a huge parameter list. Use [Extract Class](#) on the variables and methods. This new object is called a method object.

1.15. Message Chains

`getA().getB().getC().getD().getE().doIt();` The client is coupled to the structure of the navigation.

- Somewhere in the chain use [Hide Delegate](#). If you do it everywhere you just end up with a lot of middlemen. Ask yourself how the resulting object is used. See if you can use [Extract Method](#) followed by [Move Method](#) to push the code down the chain to where the data is located.
- Some people see all chains as bad. Martin and Kent say some are ok.

1.16. Middle Man

Objects hide internal details. Encapsulation leads to delegation. Sometimes this goes to far.

- If you look at the interface of a class and find that half of the methods just delegate to another class it may be time to use [Remove Middle Man](#) and talk to the object that really knows what is going on.
- If it is only a few methods, use [Inline Class](#) to inline them to the caller.
- If there is additional behavior, use [Replace Delegation with Inheritance](#) to turn the middleman into a subclass of the real object.

1.17. Inappropriate Intimacy

Sometimes classes spend too much time messing with each others' private parts.

- Use [Move Method](#) and [Move Field](#) to separate the pieces.
- See if you can use [Change Bidirectional Association to Unidirectional](#).
- If they have common interests, use [Extract Class](#) to put the stuff in common in a safe place.
- You can use [Hide Delegate](#) to let another class act as a go-between.
- When subclasses know too much about their parents, use [Replace Inheritance with Delegation](#).

1.18. Alternative Classes with Different Interfaces

Use [Rename Method](#) on methods that do the same thing but have different signatures. Keep using [Move Method](#) to move behavior until protocols are the same. This may require redundant code that you correct with [Extract Superclass](#) later.

1.19. Incomplete Library Class

Objects == Reuse? Martin and Kent think reuse is overrated (just use). Still library classes important. Builder never guess right, but bad form to use [Move Method](#) to fix the library. What to do?

- For a library that is just missing a few methods, use [Introduce Foreign Method](#).
- For a whole load of extra behavior, use [Introduce Local Extension](#).

1.20. Data Class

Classes with just fields, getters, setters and nothing else.

- If there are public fields, use [Encapsulate Field](#) before anyone notices.
- Apply [Encapsulate Collection](#).
- For fields that should not be changed use [Remove Setting Method](#).
- Look at the clients using the getters/setters. Use [Move Method](#) to move the behavior to

the data class. Use [Extract Method](#) if you can not move the whole client method.

- After awhile you can start using [Hide Method](#) on the getters and setters.

1.21. Refused Bequest

You have a child that only wants part of what is coming from the parent.

- Traditional, hierarchy is wrong. Use [Push Down Method](#) and [Push Down Field](#) to push unused methods to a new sibling class. Parent only holds what is common. 9 times out of 10 Martin and Kent think think this smell too faint to worry about.
- On the other hand, not supporting the interface of the superclass is very bad. Gut it with [Replace Inheritance with Delegation](#).

1.22. Comments

Comments are not bad, but many times you look at thickly commented code and notice that the comments are there because the code is bad.

- First apply the refactorings already listed. Often you will find that the comments are now unneeded.
- If you still need a comment to explain a block of code, try [Extract Method](#). If it still needs a comment, use [renameMethod](#).
- If you need to state rules about a required state, use [Introduce Assertion](#).
- A good time for a comment is when you do not know what to do.
- A comment is a good place to say *why* you did something.