A Taxonomy for "Bad Code Smells"

Citation

If you wish to cite this taxonomy please use the following article

Mäntylä, M. V. and Lassenius, C. **"Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study"**. Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431. (**pdf**)

Taxonomy

The reason for creating the taxonomy is to provide better understanding of the smells and to recognize the relationships between smells. I feel that with a long flat list of the code smells it is easy to lose the overall picture (at least that happened to me when I first studied the list of code smells in (Fowler&Beck 2000)). Thus, here is a taxonomy of five groups.

Group name	Smells in group	Discussion
The Bloaters	-Long Method -Large Class -Primitive Obsession -Long Parameter List -DataClumps	Bloater smells represents something that has grown so large that it cannot be effectively handled. It seems likely that these smells grow a little bit at a time. Hopefully nobody designs, e.g., Long Methods. Primitive Obsession is actually more of a symptom that causes bloats than a bloat itself. The same holds for Data Clumps. When a Primitive Obsession exists, there are no small classes for small entities (e.g. phone numbers). Thus, the functionality is added to some other class, which increases the class and method size in the software. With Data Clumps there exists a set of primitives that always appear together (e.g. 3 integers for RGB colors). Since these data items are not encapsulated in a class this increases the sizes of methods and classes.
The Object-Orientation Abusers	-Switch Statements -Temporary Field -Refused Bequest -Alternative Classes with Different Interfaces	The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design. For example, a Switch Statement might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in object-oriented programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case in which a variable is in the class scope, when it should be in method scope. This violates the information hiding principle.
The Change Preventers	-Divergent Change -Shotgun Surgery -Parallel Inheritance Hierarchies	Change Preventers are smells is that hinder changing or further developing the software These smells violate the rule suggested by Fowler and Beck which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class. The Divergent Change smell means that we have a single class that needs to be modified by many different types of changes. With the Shotgun Surgery smell the situation is the opposite, we need to modify many classes when making a single change to a system (change several classes when changing database from one vendor to another) Parallel Inheritance Hierarchies, which means a duplicated class hierarchy, was originally placed in OO-abusers. One could also place it inside of The Dispensables since there is redundant logic that should be replaced.
The Dispensables	-Lazy class -Data class -Duplicate Code -Dead Code, -Speculative Generality	The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code. This group contains two types of smells (dispensable classes and dispensable code), but since they violate the same principle, we will look at them together. If a class is not doing enough it needs to be removed or its responsibility needs to be increased. This is the case with the Lazy class and the Data class smells. Code that is not used or is redundant needs to be removed. This is the case with Duplicate Code, Speculative Generality and Dead Code smells.
The Couplers	-Feature Envy -Inappropriate Intimacy -Message Chains -Middle Man	This group has four coupling-related smells. One design principle that has been around for decades is low coupling (Stevens et al. 1974) . This group has 3 smells that represent high coupling. Middle Man smell on the other hand represent a problem that might be created when trying to avoid high coupling with constant delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application. The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Message Chains is a smell where class A needs data from class D. To access this data, class A needs to retrieve object C from object B (A and B have a direct reference). When class A gets object C it then asks C to get object D. When class A finally has a reference to class D, A asks D for the data it needs. The problem here is that A becomes unnecessarily coupled to classes B, C, and D, when it only needs some piece of data from class D. The following example illustrates the message chain smell: `A.getB().getC().getD().getTheNeededData()` Of course, I could make an argument that these smells should belong to the Object-Orientation abusers group, but since they all focus strictly on coupling, I think it makes the taxonomy more understandable if they are introduced in a group of their own.