Thursday, November 7, 2013
Presentation by Nat Guy
11 commonly occurring "Code Smells"

Goal of evaluating code against code smells: To produce higher-quality, **maintainable** code that is superior to code that just "works" somehow but may be brittle or ugly. Can be done manually or in some cases with static checkers (search for your programming language and code smell detector).

Descriptions taken from
http://www.codinghorror.com/blog/2006/05/code-smells.html

YouTube demos mostly by Jason Gorman of Codemanship (http://www.codemanship.co.uk)
Most examples of how to fix smelly code from http://sourcemaking.com

A taxonomy of code smells (groups smells together by category) is at
http://www.soberit.hut.fi/mmantyla/badcodesmellstaxonomy.htm

Name	**Description**
Message chain	Watch out for long sequences of method calls or temporary variables to get routine data. Intermediaries are dependencies in disguise. http://www.youtube.com/watch?v=5EruE6OXYTU http://sourcemaking.com/refactoring/message-chains
Data clumps	If you always see the same data hanging around together, maybe it belongs together. Consider rolling the related data up into a larger class. http://www.youtube.com/watch?v=dytbm2coxNk http://sourcemaking.com/refactoring/data-clumps
Duplicate code	Duplicated code is the bane of software development. Stamp out duplication whenever possible. You should always be on the lookout for more subtle cases of near-duplication, too. Don't Repeat Yourself! http://en.wikipedia.org/wiki/Don%27t_repeat_yourself http://www.artima.com/intv/dry.html http://www.youtube.com/watch?v=n45-L8bp2cU http://sourcemaking.com/refactoring/duplicated-code
Nested ifs ("Conditional complexity" on codinghorror site)	Watch out for large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time. Consider alternative object-oriented approaches such as decorator, strategy, or state. Example code before and after refactoring http://blogs.agilefaqs.com/tag/conditional-complexity/
Long parameter lists	The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method, or use an object to combine the parameters. http://www.youtube.com/watch?v=szOkSrZGmHU http://sourcemaking.com/refactoring/long-parameter-list

Feature envy	Methods that make extensive use of another class may belong in another class. Consider moving this method to the class it is so envious of. http://www.youtube.com/watch?v=08odrj2pbCA http://sourcemaking.com/refactoring/feature-envy
Middleman classes	If a class is delegating all its work, why does it exist? Cut out the middleman. Beware classes that are merely wrappers over other classes or existing functionality in the framework. http://www.youtube.com/watch?v=jJfY0D0j5TU (by David Donahue) http://sourcemaking.com/refactoring/middle-man
Data classes	Avoid classes that passively store data. Classes should contain data and methods to operate on that data, too. http://www.youtube.com/watch?v=r6gy0qoR8AA http://sourcemaking.com/refactoring/data-class
Long method	All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot. Refactor long methods into smaller methods if you can. http://www.youtube.com/watch?v=U4hIpntxWYc http://sourcemaking.com/refactoring/long-method
Large class	Large classes, like long methods, are difficult to read, understand, and troubleshoot. Does the class contain too many responsibilities? Can the large class be restructured or broken into smaller classes? http://sourcemaking.com/refactoring/large-class
Uber-class (aka “god class” or “brain class”, not listed at codinghorror)	Some disagreements on whether these classes are “bad” and need to be split up into smaller classes. http://sourcemaking.com/antipatterns/the-blob