

Lecture Notes

September 11, 2018

Software engineering is not [just] programming

Programmers Program	Software Engineers Build Software Products
Usually well-defined problem	Often ill-defined problem
Alone or Small Team	Teams of Teams
Hundreds/thousands LOC	Tens-thousands-→Millions LOC
Submitted for grading or for personal use	Runs in production
Fresh codebase	Existing codebase
Little if any 3rd party code	Use anything that saves time
Stop and Restart to Change	Continual Availability
Low cost of errors	High cost of errors that reach production

The "[Knightmare: A DevOps Cautionary Tale](#)" assigned reading presents a great example of the potentially high cost (>\$400 million!) of errors that reach production. The article blames devops human error and weak manual workflow procedures, but also mentions an underlying software bug that should have been found and removed - what was it?

How to find bugs: review/inspection, static analysis and dynamic analysis - and user bug reports, but the goal is to find bugs *before* deploying to users

Review (aka *inspection*) is what *humans* can do solely by reading the code and the documentation

Static analysis is what an *automated tool* can do just by examining and manipulating the source code and executables, but without actual execution

Review and static analysis apply to a range of inputs, possibly "all" inputs - or perhaps only all valid inputs
→ What is an "invalid input"?

Dynamic analysis is what an *automated tool* can do by executing the code with selected input(s), so
choose wisely


This includes various kinds of testing, monitoring and measurement, debugging, etc.

Symbolic execution is in-between static and dynamic analysis, emulates executing the code with symbolic inputs representing "all" (or at least some >1) inputs

Focus initially on static analysis bug finders. You do **not** need to understand how they work internally in order to use static analysis tools.

However, if you would like to understand how they work internally, take [Prof. Ray's](#) section of 4115 PLT, which will focus on program analysis. This will not be covered in [Prof. Edwards'](#) section of 4115 PLT, which will focus on functional programming languages. Or wait until spring and take 6156 Topics in Software Engineering.

The most interesting properties we would like a hypothetical static analysis tool to check are **undecidable**, e.g., halting problem

	<pre>CHECK-HALT(program) { if (program halts) infinite loop else halt }</pre>
---	---

But static analysis can look for “patterns” in the code, certain rules to *always* do (style guidelines) and/or “antipatterns”, certain rules to *never* do (code smells, security vulnerabilities)

Sometimes lexical (text), but usually works with an internal representation of program (structure)

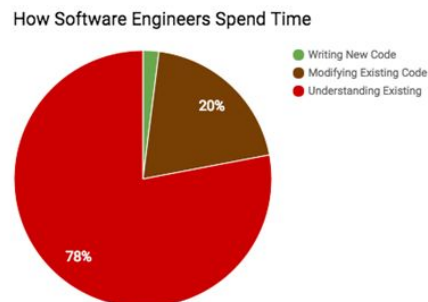
The simplest kind of static analysis is style checking

Most programming languages have a publicly defined "coding convention", some languages have multiple mutually-incompatible conventions

- Oracle has a style guide for [Secure Coding Guidelines for Java SE](#)
- Google has style guides for everything: [Google Style Guides](#)

The main purpose of coding conventions is to make it easier for **other** developers to understand **your** code. If previous developers didn't use good coding conventions, how would you understand **their** code? (see [How To Write Unmaintainable Code](#) and [The International Obfuscated C Code Contest](#))

Code is read far more often than it is written



As quickly as possible, read the following Java code and tell me which tests pass - neither, first, second, both

```
public class TestBadCode {  
    public int calculateFoo(int x, int y, boolean increment){  
        if(increment)  
            x++;  
            x *=2;  
        x += y;  
        return x;  
    }  
  
    @Test  
    public void test() {  
        assertEquals(calculateFoo(3, 5, true), 13);  
        assertEquals(calculateFoo(3, 5, false), 8);  
    }  
}
```

This coding style violation concerns indentation - which may mask a bug

Coding conventions govern how and when to use comments and whitespace, proper naming of variables and functions, code grouping and organization, sometimes file/folder structure

- Patterns to be used
- Anti-patterns to be avoided
- Makes errors more obvious - unfamiliar patterns jump out of the code when you look at it, even if does not match a common anti-pattern there may be some problem
- (Not same as [design patterns](#) and [anti-patterns](#))

What is wrong with this code?

```
switch(value) {  
  case 1:  
    doSomething();  
  
  case 2:  
    doSomethingElse();  
    break;  
  
  default:  
    doDefaultThing();  
}
```


Imagine a style guide that says something like *"All switch statement cases must end with break, throw, return, or a comment indicating a fall-through."*

```
switch(value) {  
    case 1:  
        doSomething();  
        //falls through  
  
    case 2:  
        doSomethingElse();  
        break;  
  
    default:  
        doDefaultThing();  
}
```

There are numerous "style checkers" (or "[linters](#)") that try to enforce coding conventions. These may run as standalone tools or as plugins to code editor, IDE, build tools - or as part of a more general bug finder

For example, [Checkstyle](#) for Java can be configured to almost any coding standard, numerous plugins available for development environments and build processes

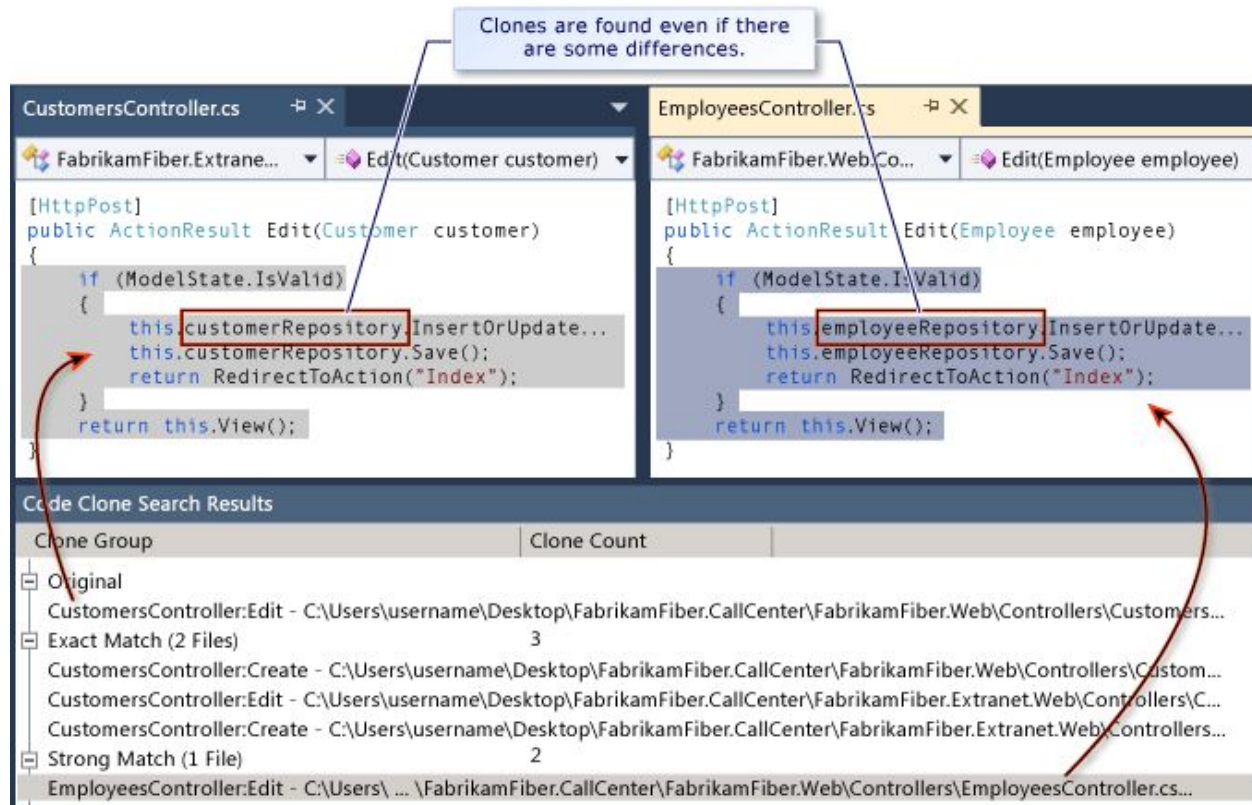
Some static analyzers can detect "[code smells](#)" = conforms to style guidelines but badly written, refers to symptoms in the source code that possibly indicates deeper problem

A code smell does not necessarily indicate a bug, yet, but tends to make changes more difficult and more likely to lead to bugs

"[Technical debt](#)" = cost of additional rework later on caused by choosing a fast and easy solution now instead of a better solution that would take longer

Analogous to monetary debt - if technical debt not repaid quickly, accumulates "interest" that adds up substantially over time

Duplicated code ("[code clones](#)") - identical or very similar code exists in more than one location in the codebase, typically arises via copy/paste

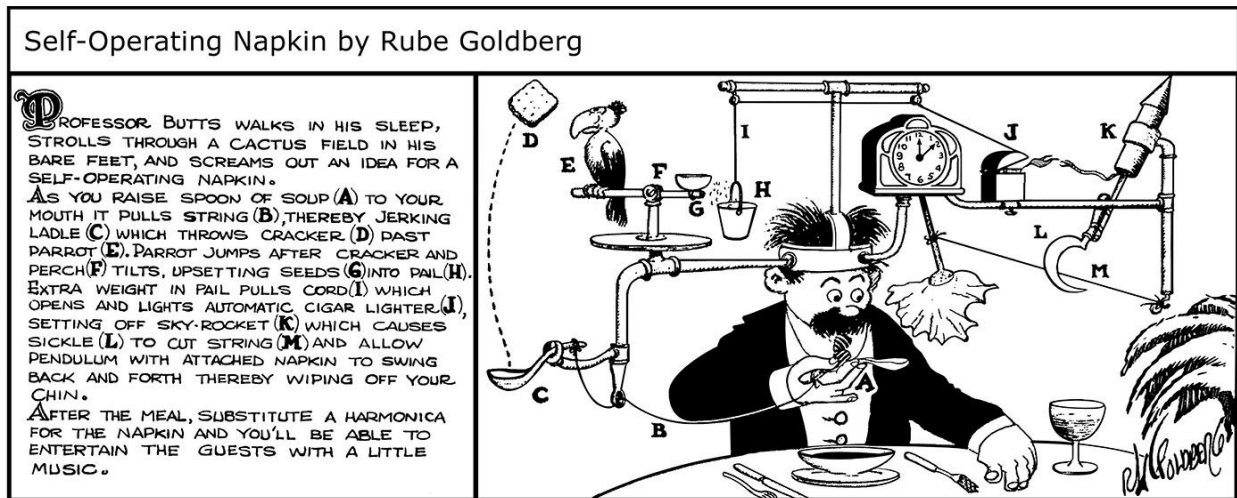


"[Simions](#)" are another kind of duplicated code in the same codebase - usually written independently and may not look similar, nor use similar naming scheme, but have the same or similar *functionality*. Maintaining these duplicates often requires redundant effort

Example of a "dispensable" code smell - absence would (usually) make code cleaner and easier to understand

Contrived complexity - forced usage of overly complicated design where simpler design would suffice

This makes code harder to understand and maintain



Artwork Copyright © Rube Goldberg Inc. All Rights Reserved. RUBE GOLDBERG ® is a registered trademark of Rube Goldberg Inc.

What does this code do?

<pre>int x, y, z; z = 0; while (x>0) { z += y; x -= 1; } while (x<0) { z -= y; x += 1; }</pre>	
--	--

Example simions

```
int x, y, z;  
z = 0;  
while (x>0) {  
    z += y;  
    x -= 1;  
}  
while (x<0) {  
    z -= y;  
    x += 1;  
}
```

```
int x, y, z;  
z = x*y;
```

Contrived complexity is not same as [cyclomatic complexity](#), which measures number of branches/loops

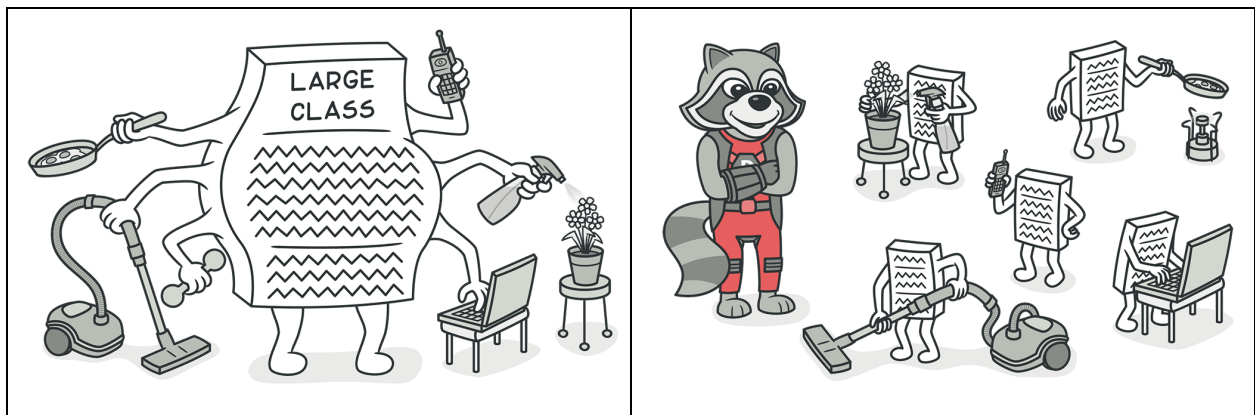
Cyclomatic complexity corresponds to the number of test cases required for branch coverage - but higher cyclomatic complexity does not necessarily make code harder to understand or maintain

Both of these examples have cyclomatic complexity 5

<pre>String getWeight(int i) { if (i <= 0) { return "no weight"; } if (i < 10) { return "light"; } if (i < 20) { return "medium"; } if (i < 30) { return "heavy"; } return "very heavy"; }</pre>	<pre>int sumOfNonPrimes(int limit) { int sum = 0; OUTER: for (int i = 0; i < limit; ++i) { if (i <= 2) { continue; } for (int j = 2; j < i; ++j) { if (i % j == 0) { continue OUTER; } } sum += i; } return sum; }</pre>
--	---

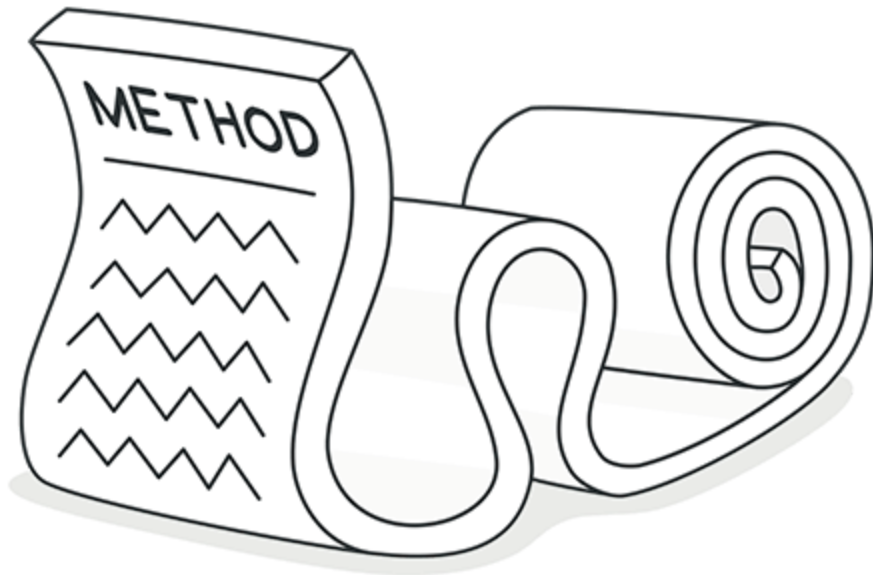
Nesting depth = maximal number of control structures (loops, if) that are nested inside each other, probably correlates more closely with understanding and maintenance challenges

Large class (god object) - a "bloater" class that has grown too large



Bloater = code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, accumulate over time as program evolves

Long methods are “bloaters” - too many lines of code covering too much functionality

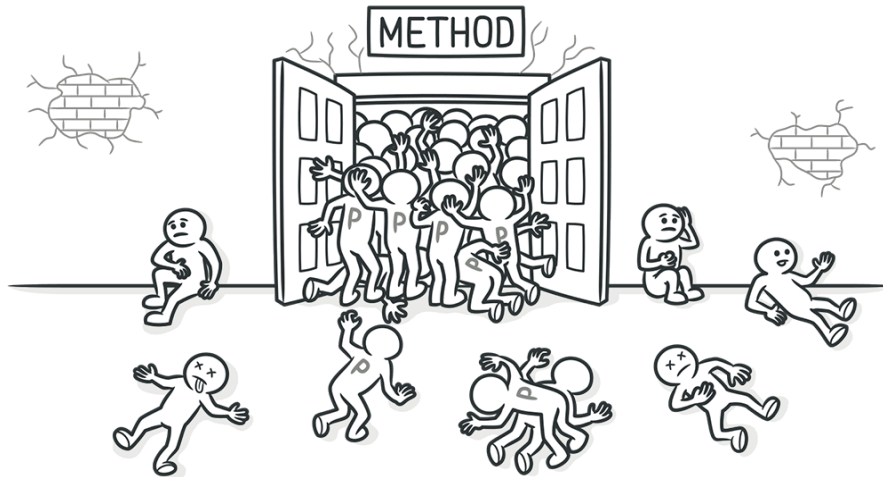


How long is too long?

Code smell detectors often impose an arbitrary threshold, or allow the user to configure

One good rule of thumb is if entire method will not fit in the code window of your IDE, or if you otherwise have to scroll to read all of it, it's too long

Too many parameters - a method with long "bloater" list of parameters. Hard to read, complicates calling and testing the method

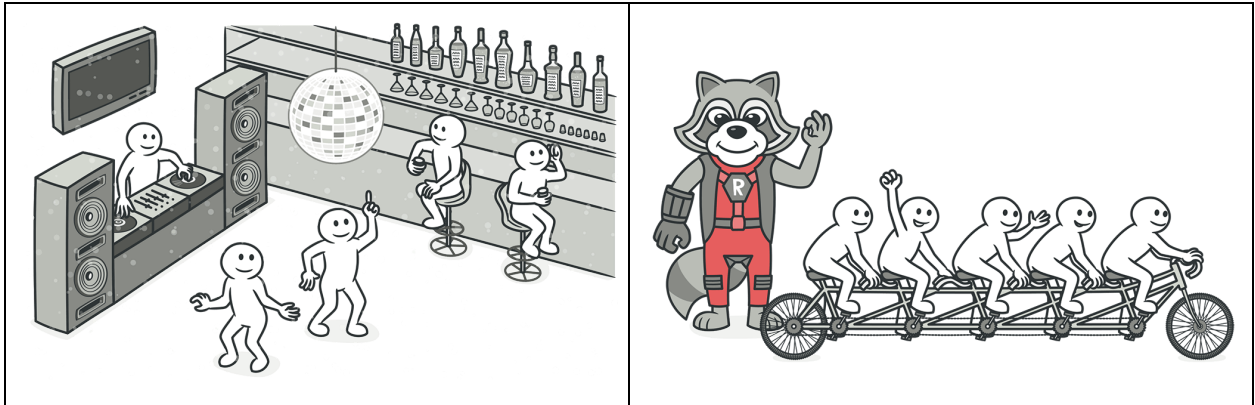


Typically means the method is trying to do too many things, break into multiple methods with logically related subsets of the parameters

How many is too many? If your IDE has a parameter window, then no more than will fit without scrolling

But do not reduce parameters if it will result in introducing additional dependencies between classes - if removing one code smell will introduce another, consider which is worse

Data clump - A related kind of "bloater" that occurs when the *same* long group of variables is passed around together in *multiple* parameter lists

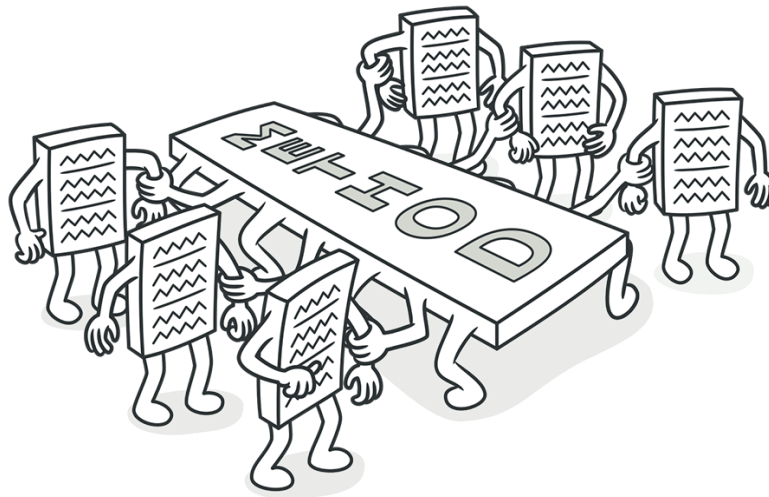


If these variables will always be fields of the same object, then simply pass that object

If not, can still merge the diverse variables together into a new "parameter object"

Feature envy - a "coupler" class that uses public methods and fields of another class excessively, perhaps more than it uses its own

Inappropriate intimacy - a "coupler" class that has dependencies on what should be *internal* fields and methods (implementation details) of another class



Coupling is the opposite of information hiding (or encapsulation), means changes in one code unit can force a ripple effect of changes to other code units

If a code unit cannot be tested in complete isolation, without another code unit, they are too tightly coupled

```
public class Phone {  
    private final String unformattedNumber;  
    public Phone(String unformattedNumber) {  
        this.unformattedNumber = unformattedNumber;  
    }  
    public String getAreaCode() {  
        return unformattedNumber.substring(0,3);  
    }  
    public String getPrefix() {  
        return unformattedNumber.substring(3,6);  
    }  
    public String getNumber() {  
        return unformattedNumber.substring(6,10);  
    }  
}
```

```
public class Customer...  
    private Phone mobilePhone;  
    public String getMobilePhoneNumber() {  
        return "(" +  
            mobilePhone.getAreaCode() + ")" +  
            mobilePhone.getPrefix() + "-" +  
            mobilePhone.getNumber();  
    }  
}
```

```
public class Phone {
    private final String unformattedNumber;
    public Phone(String unformattedNumber) {
        this.unformattedNumber = unformattedNumber;
    }
    private String getAreaCode() {
        return unformattedNumber.substring(0,3);
    }
    private String getPrefix() {
        return unformattedNumber.substring(3,6);
    }
    private String getNumber() {
        return unformattedNumber.substring(6,10);
    }
    public String toFormattedString() {
        return "(" + getAreaCode() + ") " + getPrefix() + "-" +
getNumber();
    }
}

public class Customer...
    private Phone mobilePhone;
    public String getMobilePhoneNumber() {
        return mobilePhone.toFormattedString();
    }
}
```

Message chains - another example of coupling, when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object ...

The client needs to know the detailed navigation structure of a graph of objects to construct the message chain

If anything along the path changes, the client needs to change too

Message Chain

```
salary = database.get_company(company_name).  
           get_manager(manager_name).  
           get_team_member(employee_name).  
           salary
```

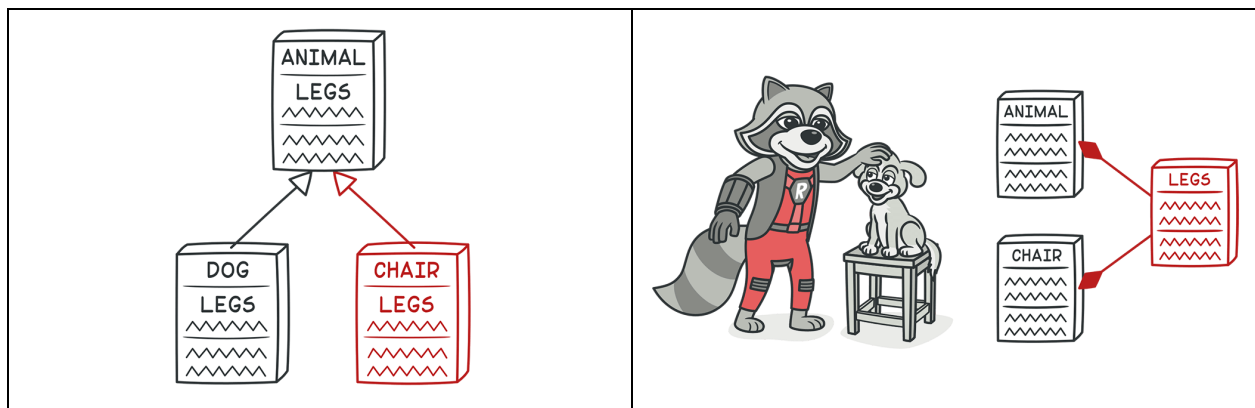
Better

```
salary =  
database.get_employee_by_name(employee_name).  
salary
```

Refused bequest - an "object-orientation abuser" class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class

Inheritance created between classes only by the desire to reuse some of the code in a superclass, but the superclass and subclass are completely different

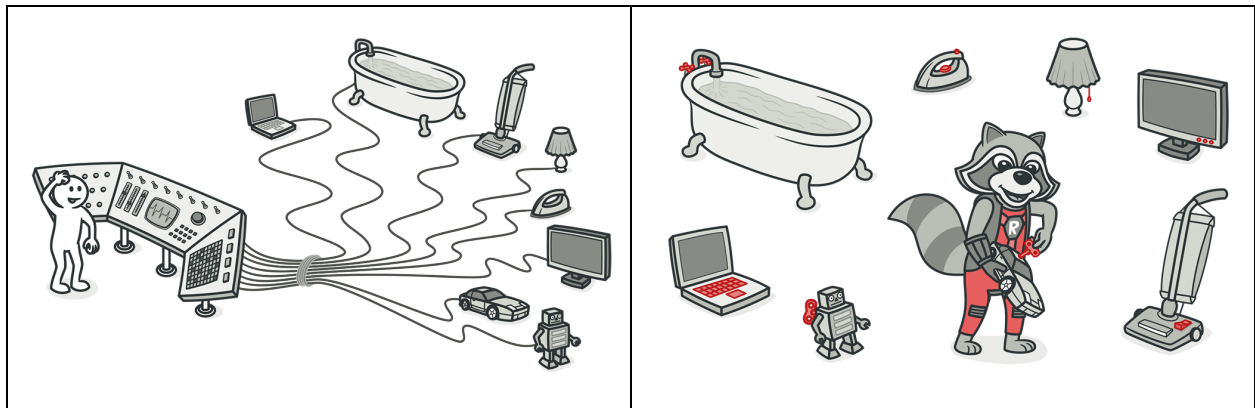
Liskov substitution principle - an object of any child class should be a suitable substitute for objects of the parent class in any methods that normally operate on objects of the parent class ("is a" relationship)



Replace inheritance with delegation

"Type checking" switch statements - complex switch operator or sequence of if statements that consider the category of parameter object (based on, e.g., the values of certain fields in the object)

This is another "object-orientation abuser", should define the types as classes and use polymorphism



Most other uses of switch statement should also be refactored, e.g., cases based on current application state could use [strategy design pattern](#)

Dispensable unused code - sometimes previously used (dead code), sometimes created "just in case" for future (speculative generality)



Don't just comment out dead code, delete it - if it turns out you need it again, it's still in the version control repository

One common maxim of agile processes is YAGNI = "you ain't gonna need it", don't write code until you need it for to implement a feature, part of the cross-cutting infrastructure, or to support testing

Reducing code size means less code to understand and maintain

These are some of the code smells detected by general bug finder tools like [PMD](#) and [Spotbugs](#), which also find many other kinds of bugs.

[Spotbugs Bug Descriptions](#) is a long list

Other tools only detect a few code smells, e.g., [JDeodorant](#) (**Feature Envy, Type Checking, Long Method, God Class and Duplicated Code**)

Read for Thursday: [A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World](#)

First individual/pair homework assignment: [Practice with Github and submitting assignments on Canvas](#) due before class on next Tuesday, to make sure you can use github and post assignment submissions on canvas.