

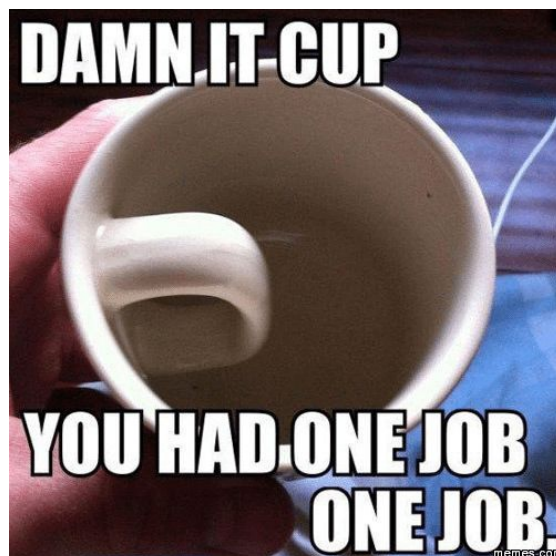
Lecture Notes
October 20, 2020

The assessment will be discussed briefly on *Thursday*

Last Thursday's lecture used [Java](#) examples to show the SOLID software design principles. Here are some short articles that show examples of the SOLID principles using [C++](#), [Javascript](#), and [Python](#).

The SOLID principles illustrated with pictures (taken from the Python article):

Single Responsibility Principle (SRP): Every entity or class should have a single responsibility, and all its tasks or services should be focused on that responsibility



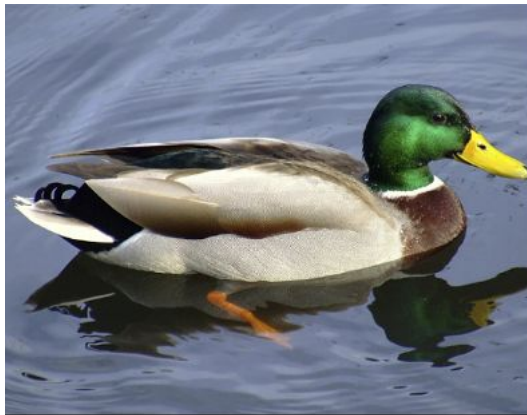
Open-Closed Principle (OCP): Code units should be “open for extension and closed for modification”. This allows change without modifying any existing code (except for fixing bugs in the existing code)



Open Closed Principle

You don't need to rewire your MoBo to plug in "Mr Happy"

Liskov Substitution Principle (LSP): Extends OCP to require that subtypes must be substitutable for their base types (polymorphism). If class *A* is a subtype of class *B*, then we should be able to replace *B* with *A* without disrupting the behavior of our program



If it looks like a duck and quacks like a duck but it needs batteries, you probably have the wrong abstraction.

Interface Segregation Principle (ISP): Large interfaces should be split into smaller ones. Clients should not be forced to depend upon interfaces they do not use



Interface Segregation Principle

If IRequireFood, I want to Eat(Food food) not,
LightCandelabra() or LayoutCutlery(CutleryLayout preferredLayout)

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules, which makes it hard to reuse the high-level modules. Instead both should depend on abstractions.



Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

Static analysis bug finders like SpotBugs look for *patterns* in the code. They can detect some deviations from the SOLID principles as well as other kinds of design and coding flaws known as “[code smells](#)”. The smelly code may conform to style guidelines but is badly written.

A code smell is a surface indication that usually corresponds to a deeper problem in the system. It does not necessarily indicate a bug, yet, but tends to make changes more difficult and more likely to lead to bugs (as does bad coding style)

“[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

I've previously mentioned one example of code smells: duplicate or redundant code. This violates the Don't Repeat Yourself design principle

“[Code clones](#)” are nearly identical code that exists in more than one location in the codebase, typically arising via copy/paste. Some refactoring tools (e.g., [JDeodorant](#)) automatically detect a group of code clones and help developers replace each instance of the clone with a call to a single code unit.

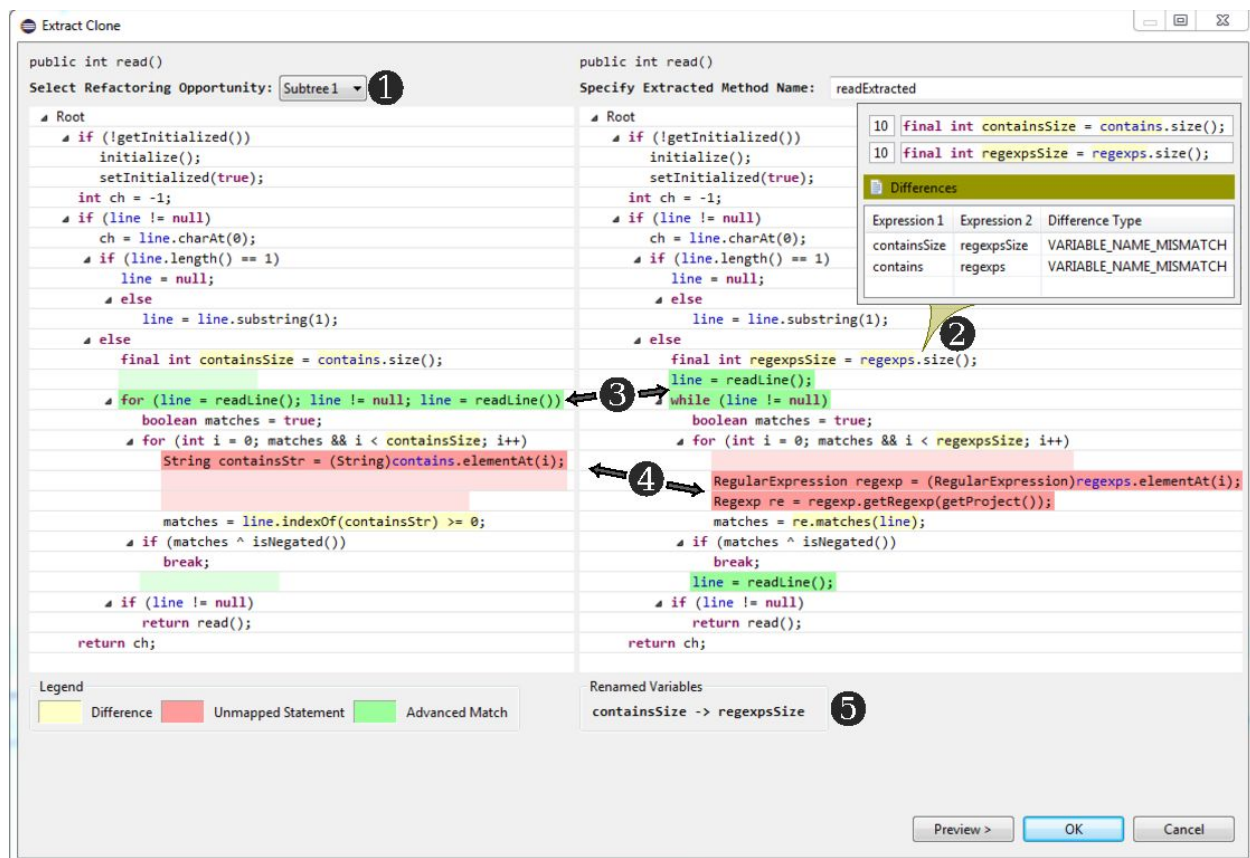



Figure 2: Clone pair visualization and refactorability analysis.

“Simions” are another kind of duplicated code in the same codebase - usually written independently and do not look similar, but have the same or similar *functionality*.

Sometimes called semantic or behavioral clones to distinguish from the syntactic or structural clones above.

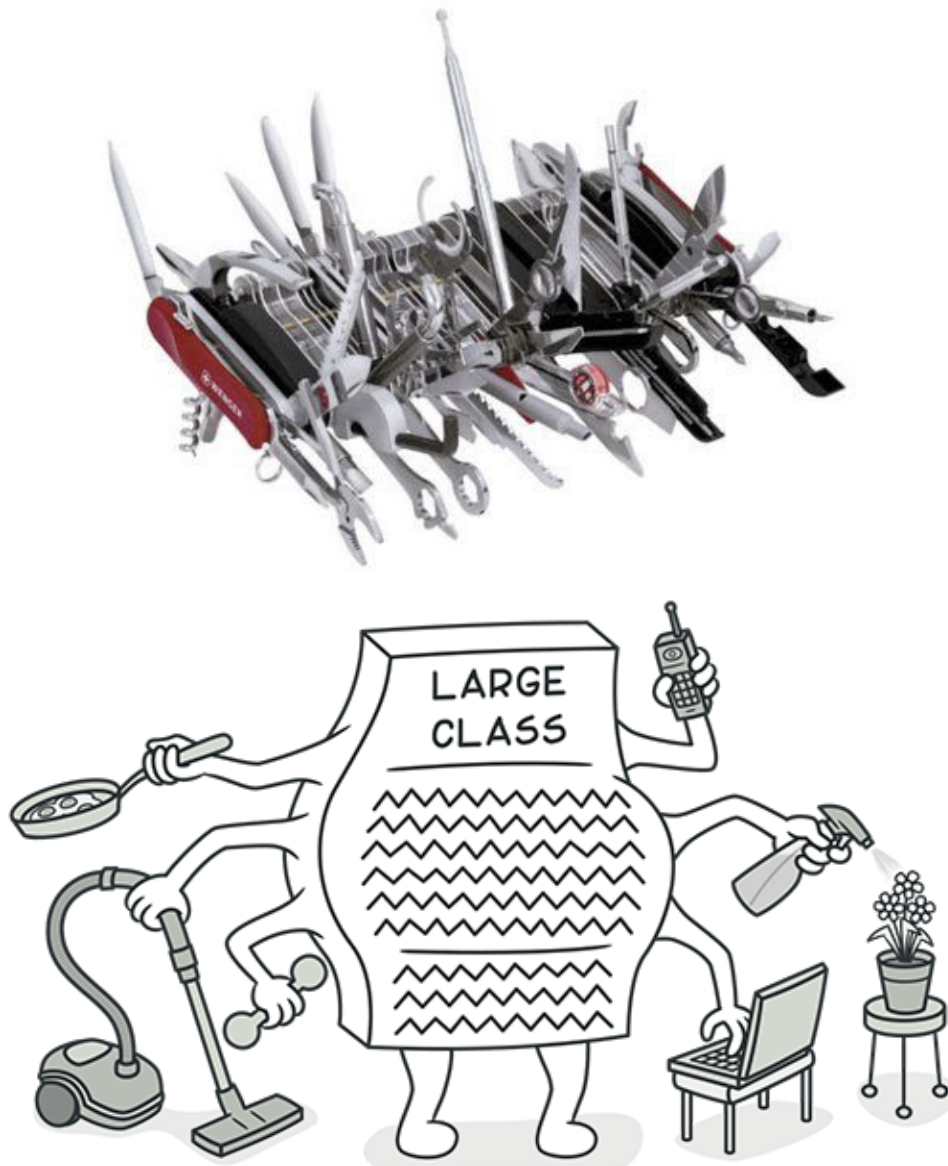
<pre>int x, y, z; z = 0; while (x>0) { z += y; x -= 1; } while (x<0) { z -= y; x += 1; }</pre>	<pre>int x, y, z; z = x*y;</pre>
--	----------------------------------

Impossible to detect automatically in the general case, but [tools](#) can identify in many practical cases.

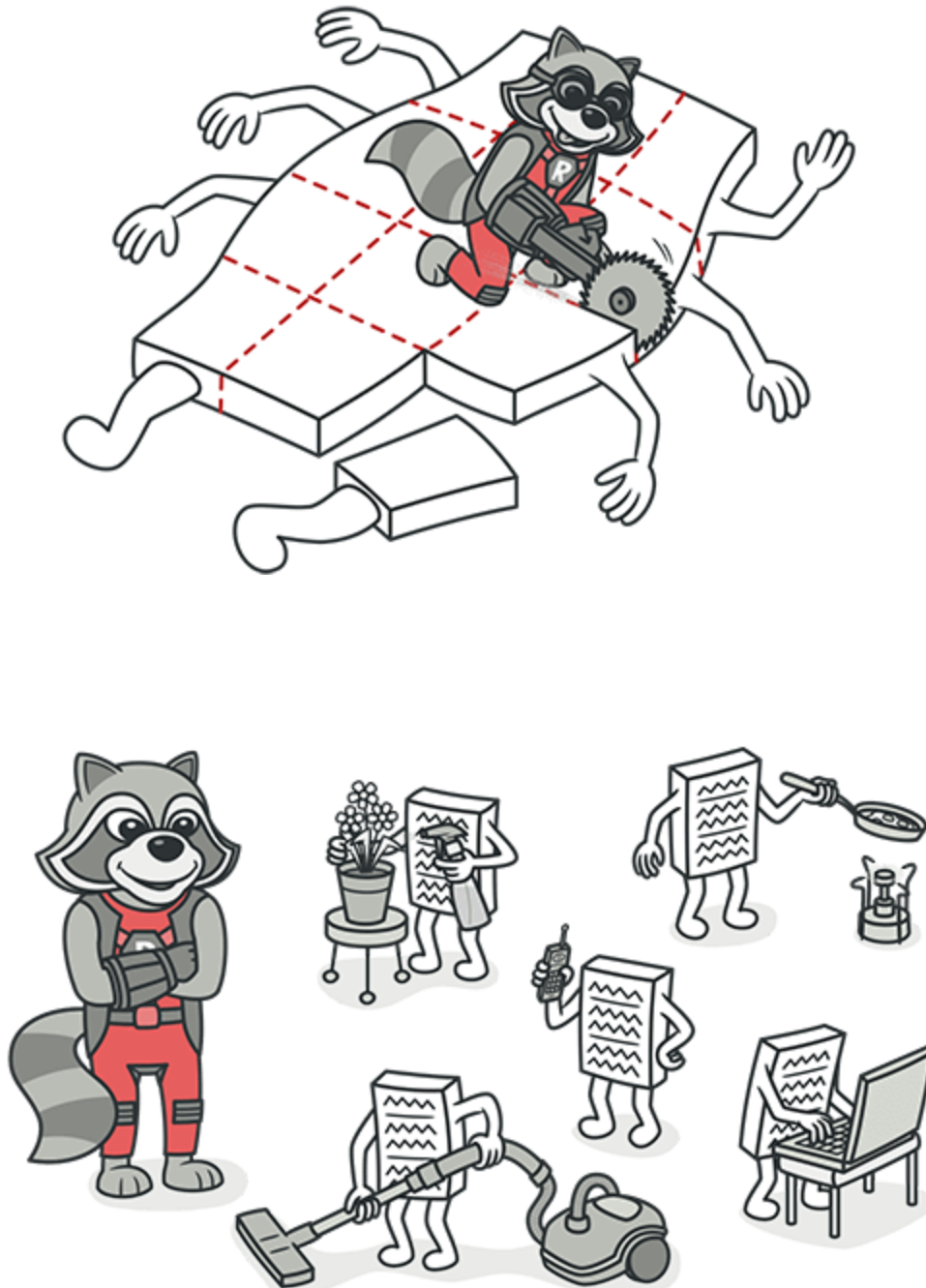
	<pre>CHECK-HALT(program) { if (program halts) infinite loop else halt }</pre>
---	---

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

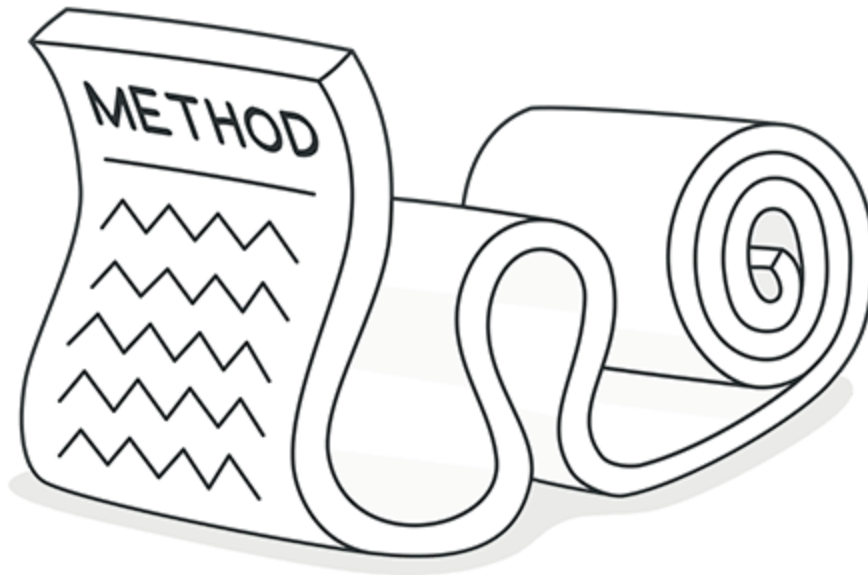
Bloater class - a class that has grown too large, too many fields and methods. Violates the Single Responsibility Principle



Code smell detectors impose an arbitrary threshold on the number of methods, or allow the user to configure. They may not be smart enough to figure out the different responsibilities, but the developers should be



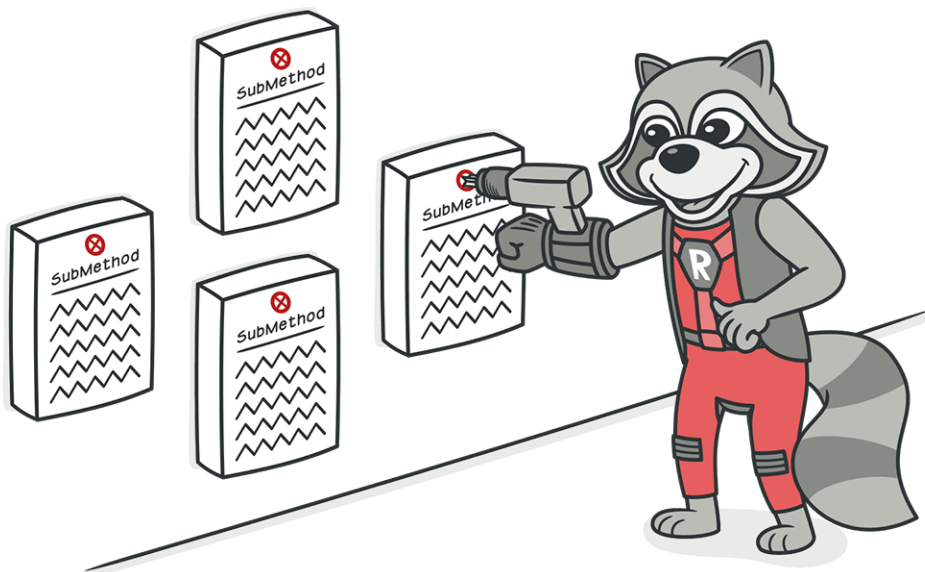
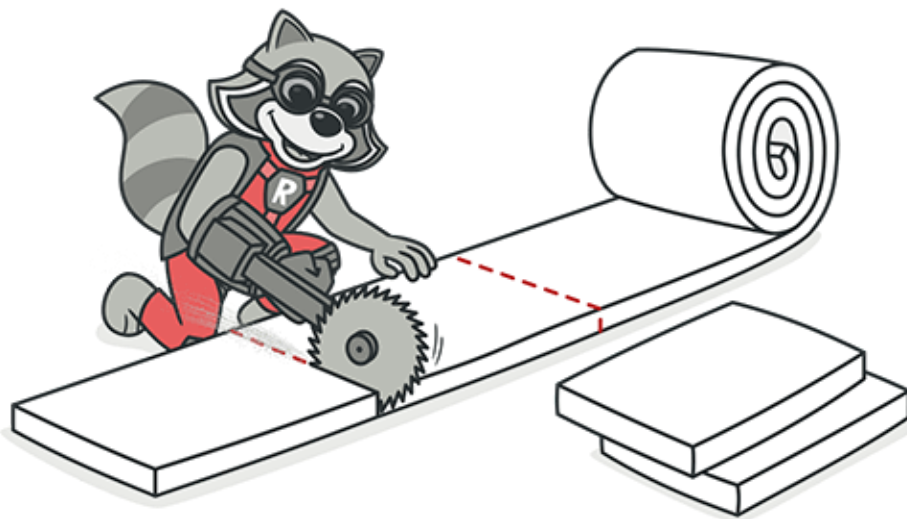
Bloater method - too many lines of code, which is likely to cover too much functionality



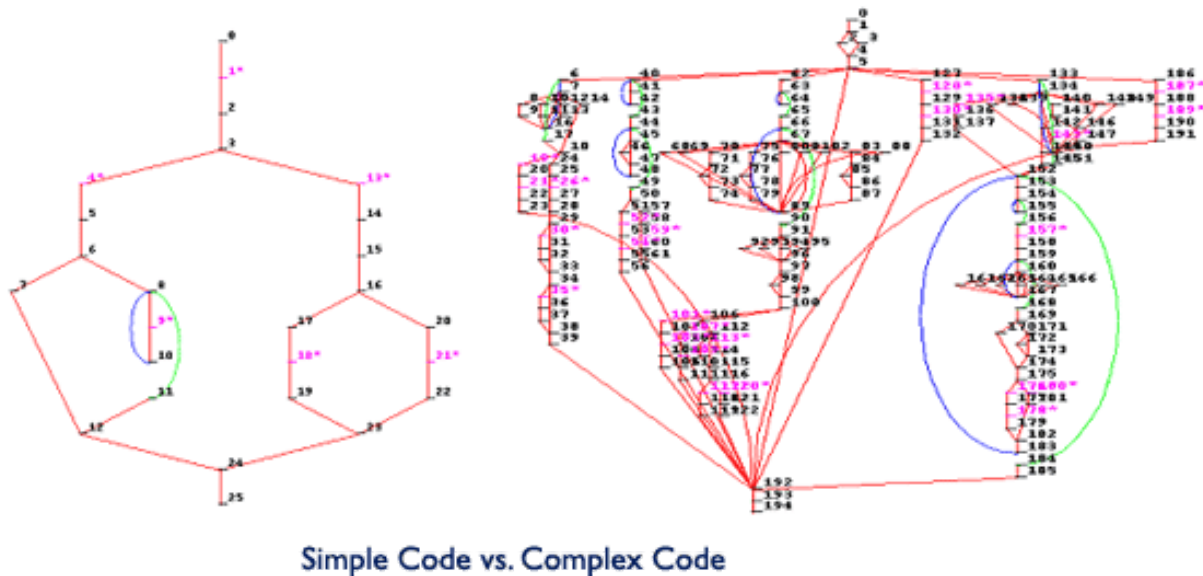
How long is too long? Code smell detectors impose an arbitrary threshold, e.g., 10 lines, 40 lines, or allow the user to configure

If the 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 probably too long

If you feel the need to comment on something inside a method, put this code in a new method. Even a single line can and should be split off into a separate method, if it requires explanation separate from the enclosing method



Another kind of bloater method: Too many paths



[Cyclomatic complexity](#) = number of conditions + 1. It corresponds to the number of test cases required for branch coverage. Higher cyclomatic complexity usually, but not always, makes code harder to understand and maintain

This code has cyclomatic complexity 14, far beyond the typical threshold - but no developer would consider this code complex. So a good code smell detector needs a more sophisticated complexity metric

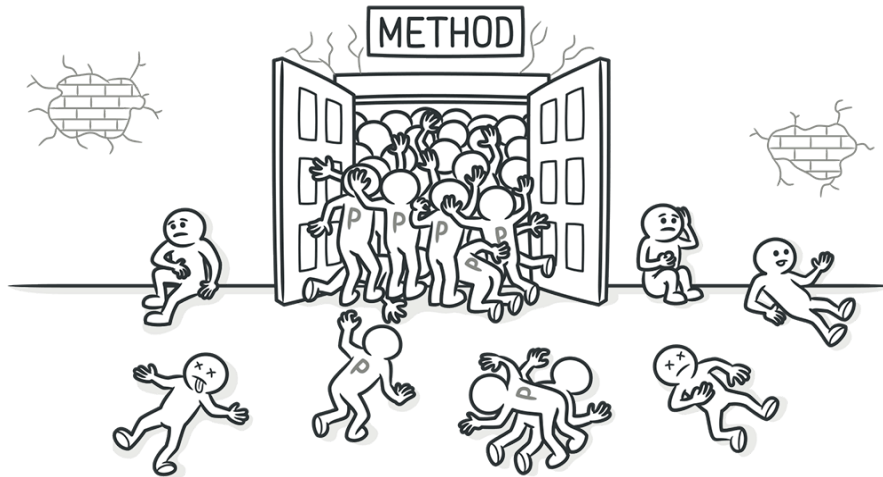
```
String getMonthName (int month) {  
    switch (month) {  
        case 0: return "January";  
        case 1: return "February";  
        case 2: return "March";  
        case 3: return "April";  
        case 4: return "May";  
        case 5: return "June";  
        case 6: return "July";  
        case 7: return "August";  
        case 8: return "September";  
        case 9: return "October";  
        case 10: return "November";  
        case 11: return "December";  
        default: throw new IllegalArgumentException();  
    }  
}
```

Both of these examples have cyclomatic complexity 5

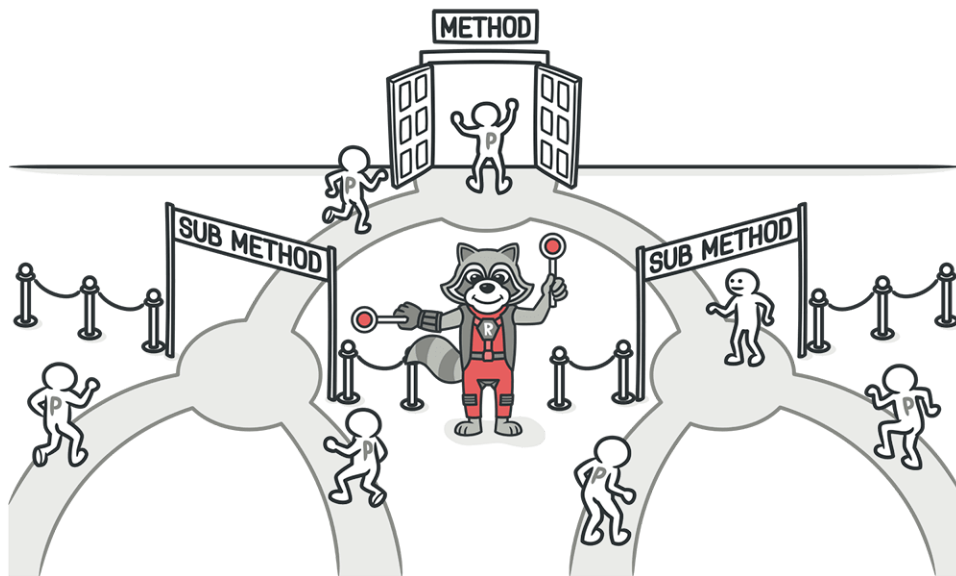
<pre>String getWeight(int i) { if (i <= 0) { return "no weight"; } if (i < 10) { return "light"; } if (i < 20) { return "medium"; } if (i < 30) { return "heavy"; } return "very heavy"; }</pre>	<pre>int sumOfNonPrimes (int limit) { int sum = 0; OUTER: for (int i = 0; i < limit; ++i) { if (i <= 2) { continue;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

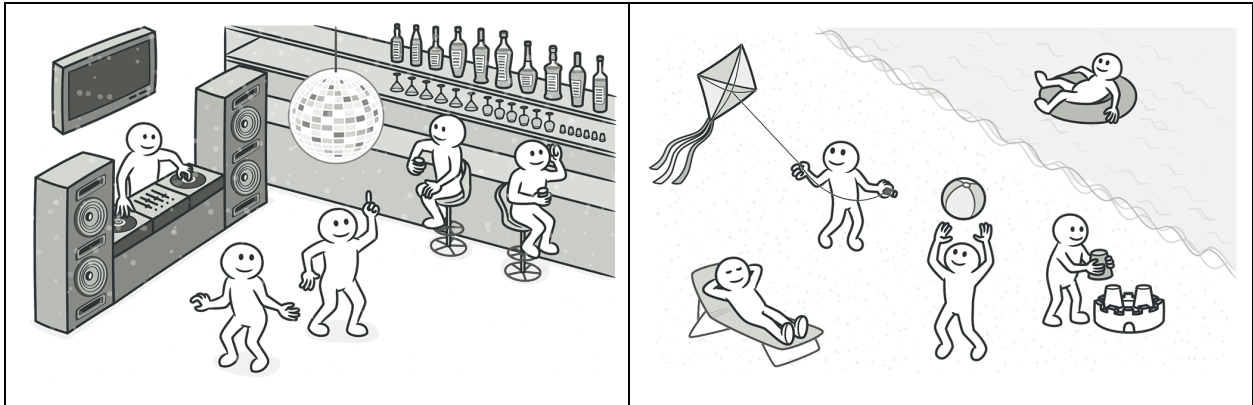
Bloater parameters - a method with a long list of parameters, e.g., more than three or four. Hard to read, complicates calling and testing the method



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

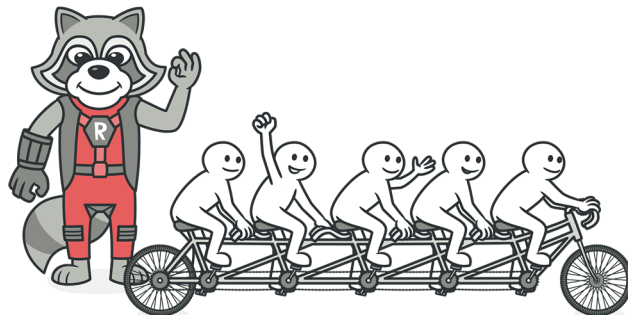


Data clump - A related kind of “bloater” that occurs when the *same* long group of variables is passed around together in *multiple* parameter lists



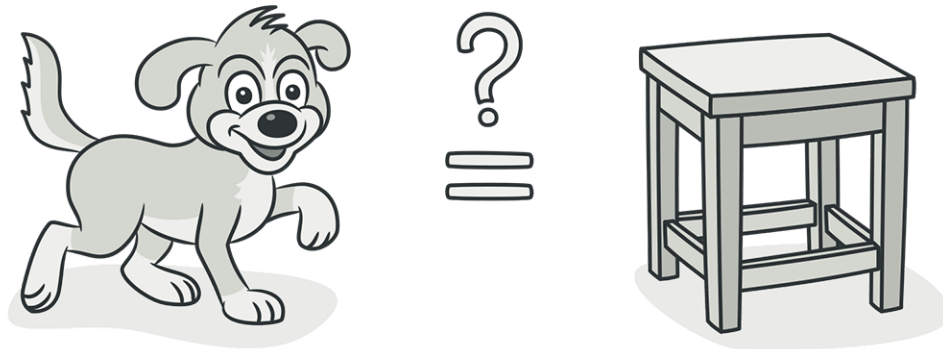
If these variables are always fields of the same object, then simply pass that object

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

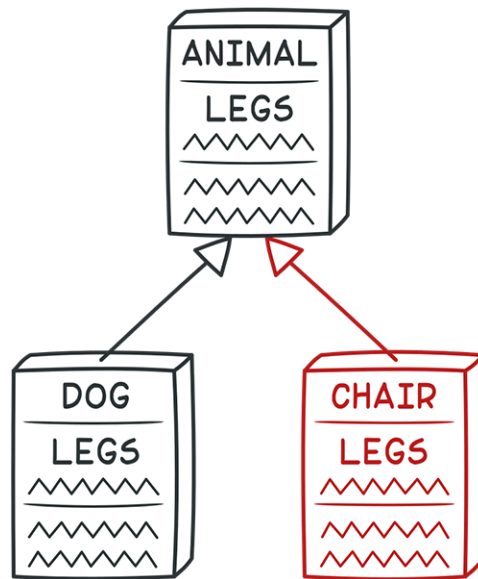


There are other code smells besides bloaters, such as “object-orientation abusers”

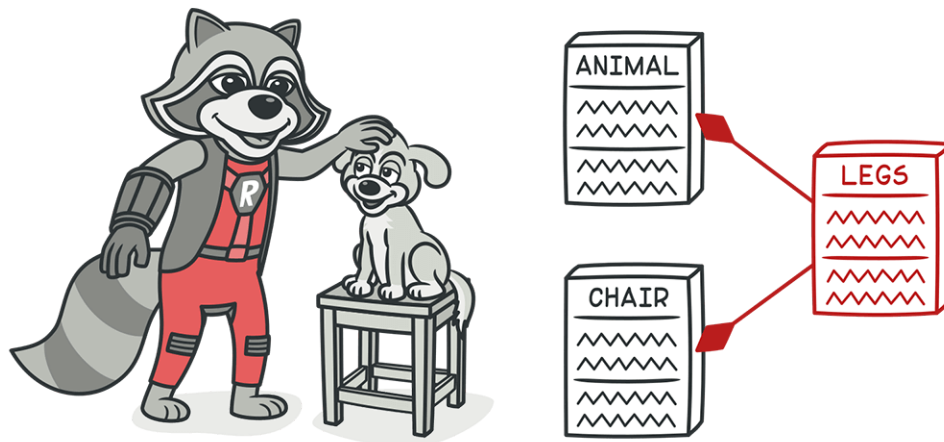
Refused bequest - Inheritance created between classes to reuse some of the code in the base class (avoiding copy/paste!). But other code in the base class is overridden in such a way that the contract of the base class is not honored by the derived class. Violates the Liskov Substitution Principle



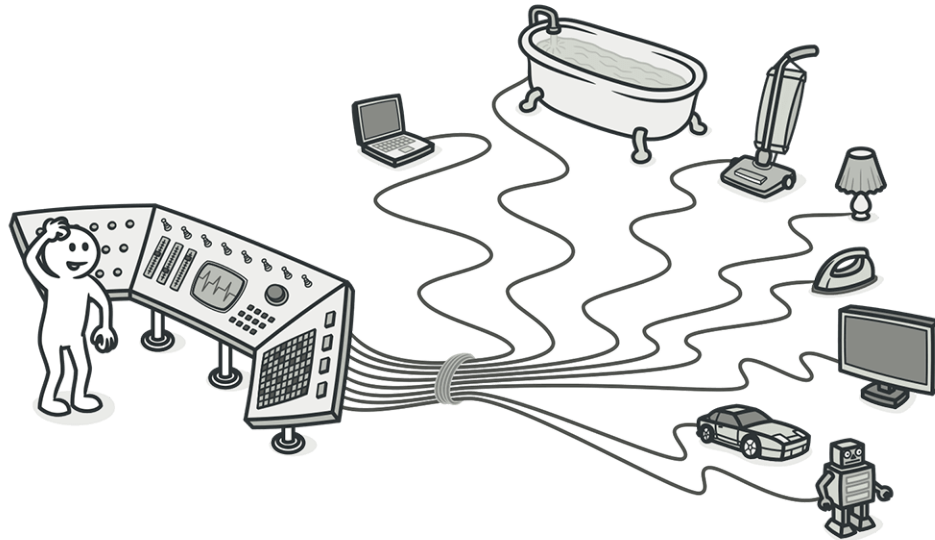
What is the relationship between the dog and the stool?
They both have four legs...



Instead replace inheritance with object composition = 'has-a' rather than 'is-a'



Another object-orientation abuser is “type checking” switch statements - complex ‘switch’ operators or sequences of ‘if’ statements that consider the category of the parameter object based on, e.g., the values of certain fields



Instead define the types as classes and use [polymorphism](#)

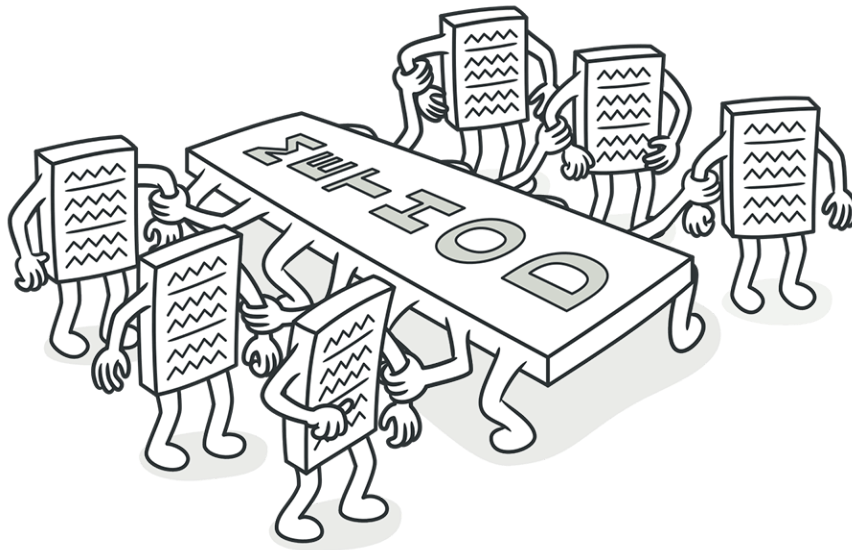


“Coupler” classes involve excessive coupling between the classes. *Coupling* is the opposite of *cohesion* (the goal of the Single Responsibility Principle) - means changes in one code unit force a ripple effect of changes to other code units



Feature envy - a “coupler” class that uses *public* methods and fields of another class excessively

Inappropriate intimacy - a “coupler” class that has dependencies on what should be *internal* fields and methods of another class



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

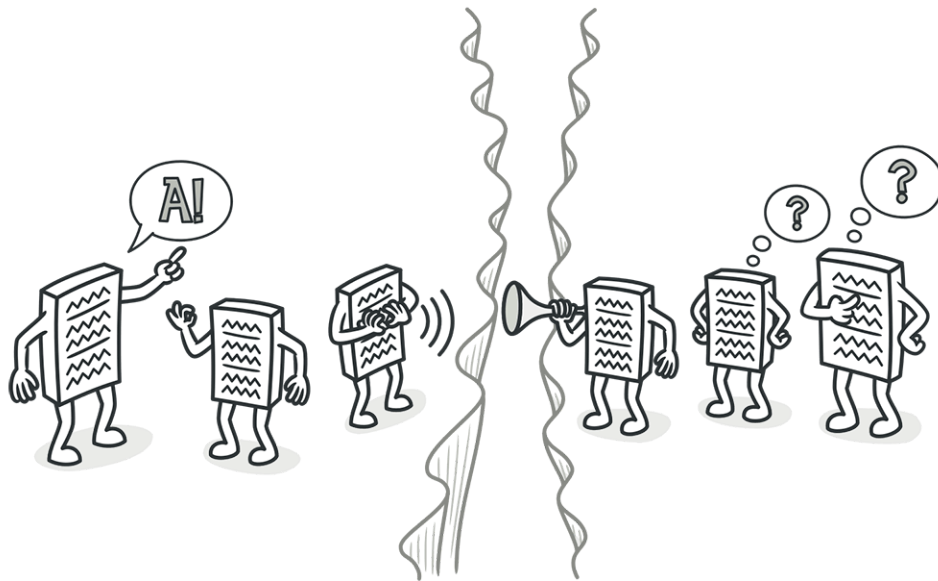
```
public class Phone {  
    private final String unformattedNumber;  
    public Phone(String unformattedNumber) {  
        this.unformattedNumber = unformattedNumber;  
    }  
    public String getAreaCode() {  
        return unformattedNumber.substring(0,3);  
    }  
    public String getPrefix()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

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


Message chain - another “coupler”, 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 object ...



A series of calls resembling `a->b()->c()->d()`

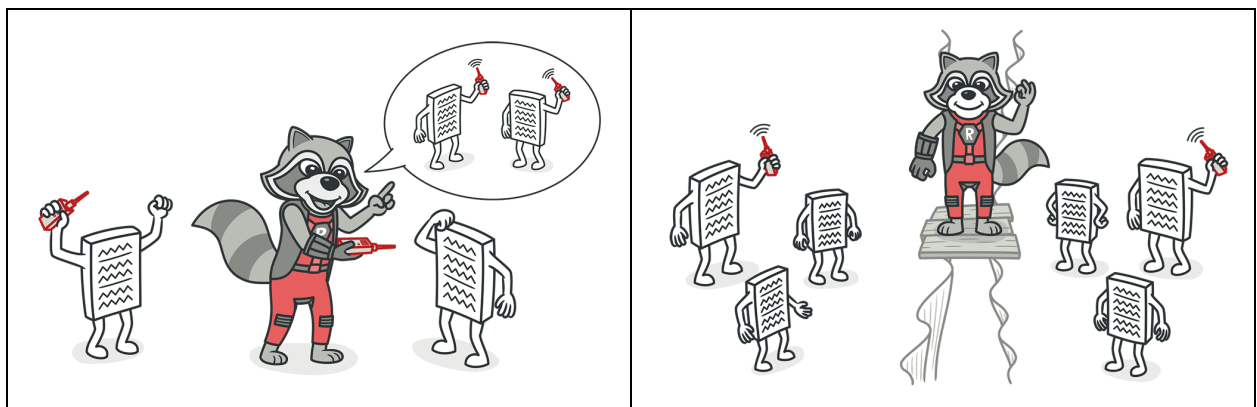
The client needs to know the detailed navigation structure of a graph of objects to construct the message chain. If anything along the path changes, the client needs to change too

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
```



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. Reducing code size (even in comments) means less code to understand and maintain

Many of the illustrations came from [here](#), which has several more examples of code smells and discusses the refactoring steps to remove them.

[SonarQube](#) is probably the best multi-language bug finder, it supports every language you've ever heard of and many you haven't

“Assignment T1: Preliminary Project Proposal” due
October 27

<https://courseworks2.columbia.edu/courses/104335/assignments/486922>

Each team proposes their own project, within constraints:

- Must impose authenticated login.
- Must be demoable online (e.g., zoom or discord).
- Must store some application data persistently (database or key-value store).
- Must use some publicly available API beyond those that “come with” the platform. It does not need to be a REST API.

Describe Minimal Viable Product (MVP) both in prose - a few paragraphs - and via 3-5 “user stories”.

< label >: As a < type of user >, I want < some goal > so that < some reason >.

My conditions of satisfaction are < list of common cases and special cases that must work >.

Describe acceptance testing plan that covers these cases.

List the tech you plan to use. If different members of the team plan to use different tools, please explain.

Two sample team projects from a previous offering of this course are linked below. All submitted assignments as well as the code are included in each repository. These projects had three iterations because the team project ran the entire duration of the course, there was no individual project.

Code Phoenix - https://github.com/s4chin/coms_4156

Space Panthers - <https://github.com/wixyFun/openSpace>