COMS W4156 Advanced Software Engineering (ASE)

October 18, 2022

Agenda

1. multi-client and persistent data API Testing demo
2. Finding Bugs by executing the code
3. Static Analysis

multi-client and persistent data API Testing demo: Alex Rupp-Coppi

demo repo

Agenda

1. multi-client and persistent data API demo
2. Finding Bugs by executing the code
3. Static Analysis

Bugs Are Everywhere

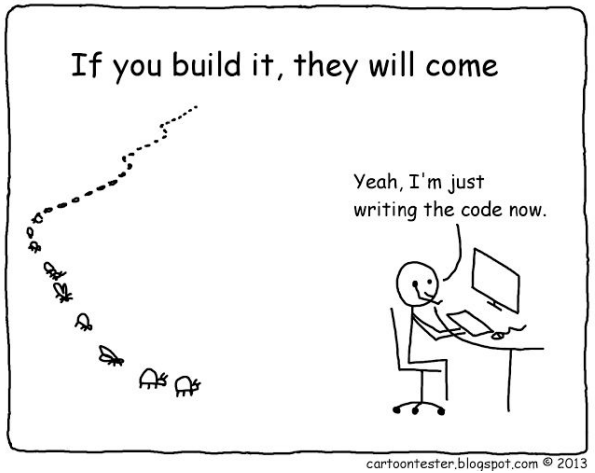
There is no shortage of bugs, every non-trivial program has bugs

Ideally find (and fix) them before your users do

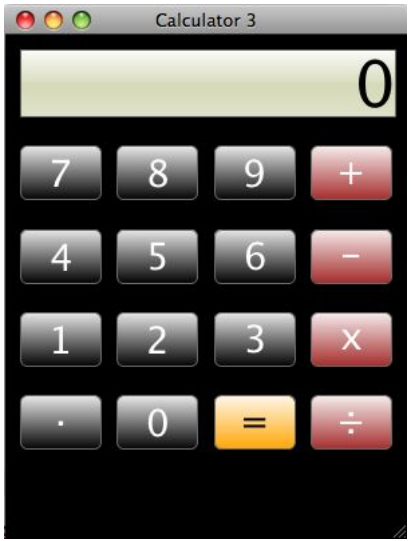
Testing (and Dynamic Analysis)

Static Analysis

~~Formal Methods~~



What Kinds of Bugs Can Be Found By Executing the Code



The tester (or test script) enters a number, presses +, enters another number, presses =

- Nothing happens
- Computes the wrong answer

Calculator works correctly for some period of time or some number of computations, and then does nothing

Calculator displays all 0's and won't do anything else

The calculator correctly adds, subtracts, multiplies and divides, but the tester found that if two operators are held down simultaneously, it appears to do square root

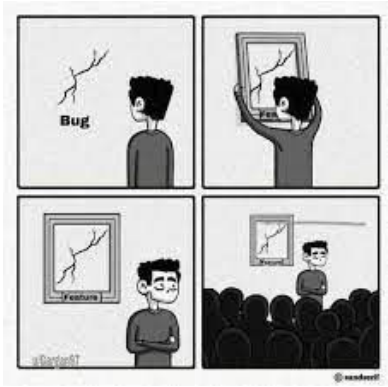
The calculator's buttons are too small, the = key is in an odd place, the display is hard to read, ...

Is This a Bug or a Feature?

The tester (or test script) enters a number, presses +, enters another number, presses =

- Nothing happens
- Computes the wrong answer

Probably a bug



Calculator works correctly for some period of time or some number of computations, and then does nothing

Bug - memory or other resource leak; Feature - trial period over, now you have to pay for it

Calculator displays all 0’s and won’t do anything else

Bug - code stuck in infinite loop; Feature - battery low

The calculator correctly adds, subtracts, multiplies and divides, but the tester found that if two operators are held down simultaneously, it appears to do square root

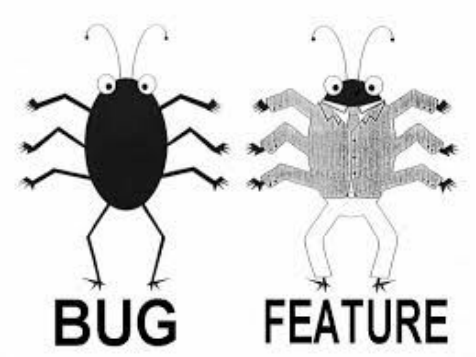
Undocumented feature

The calculator’s buttons are too small, the = key is in an odd place, the display is hard to read, ...

The user will consider these bugs even though the developers may not

What Kinds of Bugs Can Be Found By Executing the Code

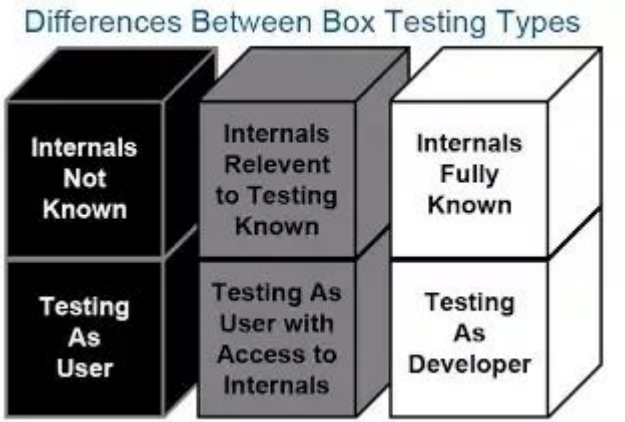
- Software doesn’t do something requirements say it should do
- Software does something requirements say it shouldn’t do
- Software does something that requirements don’t mention
- Software doesn’t do something that requirements don’t mention but should
- Software is difficult to understand, hard to use, slow, etc.

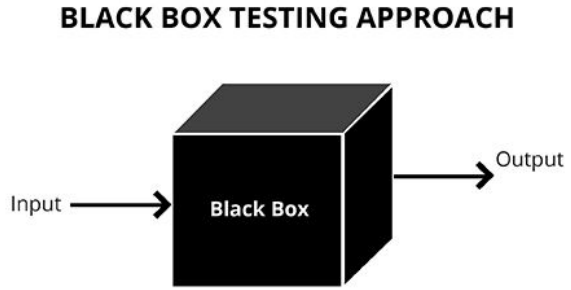


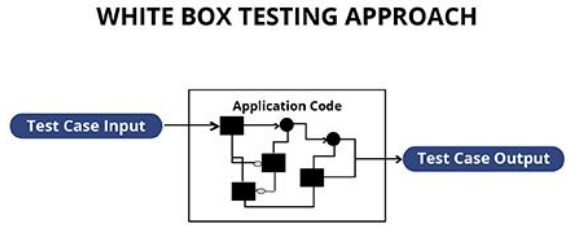
Test-to-Pass vs. Test-to-Fail

Initial testing makes sure the software minimally works with typical inputs (“smoke test”)

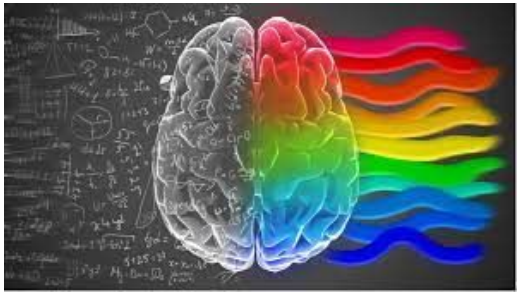
But most testing is trying to find bugs, which means testing with valid and invalid inputs designed to trigger “corner cases”







Blackbox matches Testing Intuition, so...



Why whitebox?

Coverage: If you never ran this part of the code, how can you have any confidence that it works? Helps with choosing inputs intended to reach specific parts of the code. Already discussed statement and branch coverage

Symbolic and Concolic Execution: “Run” the code with fully or partially symbolic inputs, e.g., Klee or Java Pathfinder, then use path conditions to determine concrete inputs necessary to force a particular execution path

Why greybox?

Some bugs are not immediately visible externally, e.g., in-memory side-effects, resource leaks, corrupted data

Check logs, databases, file system, network traffic

May be referred to as dynamic analysis or monitoring rather than testing, leverages binary/bytecode rewriting tools like valgrind and asm

Preventing Buggy Code from being Released to Production

Buggy code does not matter much if it never leaves the developer's machine

Buggy code committed to shared repositories will increase time and effort for *other* software engineers, but is unlikely to be humongously expensive if it never makes it to production

Testing and code review are usually the last barriers to preventing buggy code from leaving the developer's machine or, if it gets to shared repositories, preventing from being released into production

A lot of this course is about testing, but what can we do *before* testing? Or in addition to testing, since testing cannot avoid false negatives. Code review is human-intensive, what can we automate?

False Positives and False Negatives

Testing (and any dynamic analyses that run the code) finds bugs that *did* happen with a specific input, but it's infeasible to consider all possible inputs so misses many bugs - false negatives

Static analysis finds code that *might* lead to introducing bugs (and bugs that *might* happen with some input, covered next class) - false positives

Agenda

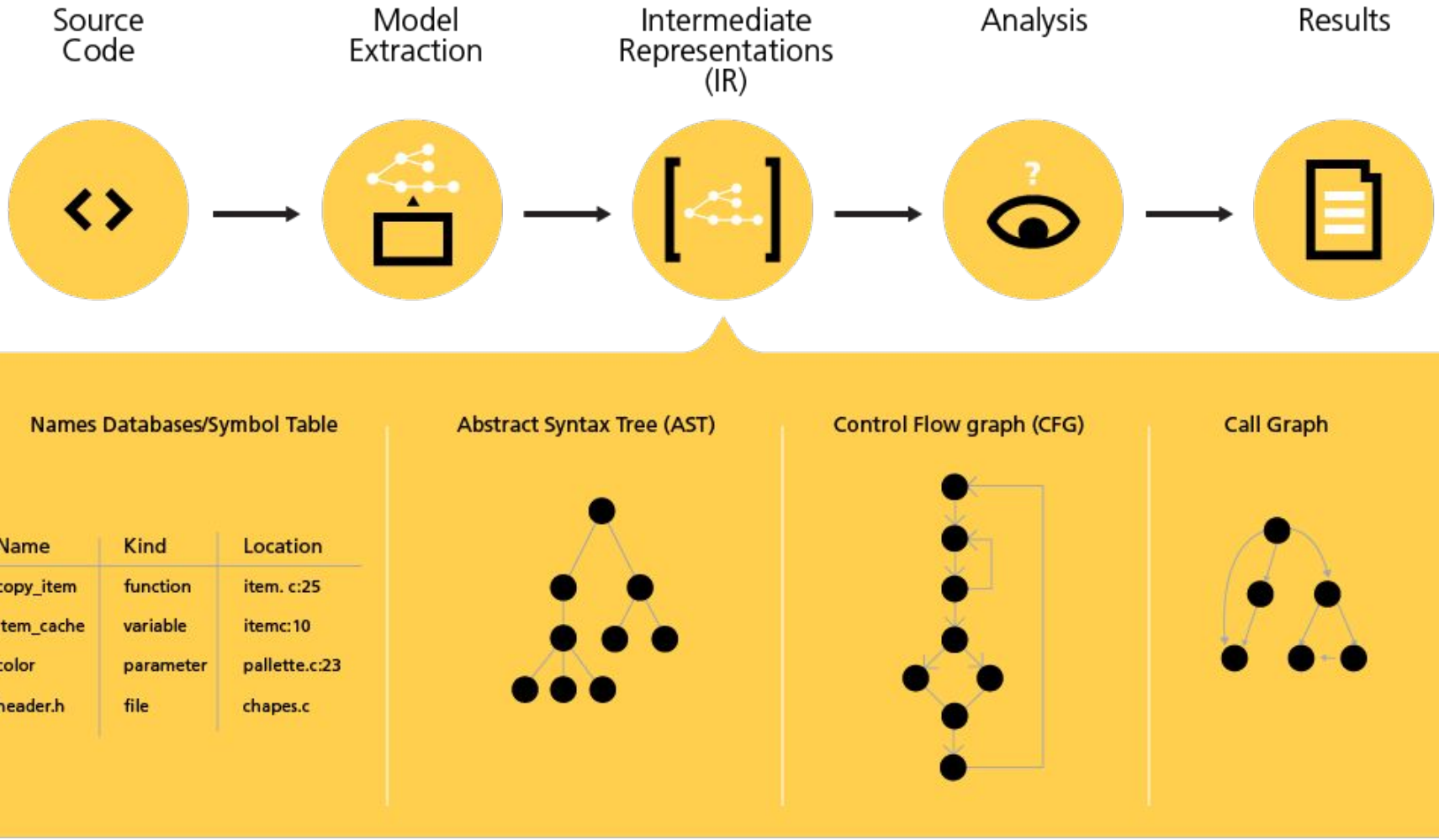
1. multi-client and persistent data API demo
2. Finding Bugs by executing the code
3. Static Analysis

Static Analysis

Any code analysis that does not actually execute the code is “static”

Includes style checkers: lack of compliance to coding standards does not necessarily mean there’s a bug, but does make it more likely for bugs to occur as multiple developers (mis-)understand and modify the code

But the term static analysis or static analyzer more commonly refers to “bad code” detectors and bug finders, often integrated in same tool like SpotBugs (for Java) and SonarQube (for C++, Java and many others)


Source Code
Model Extraction
Intermediate Representations (IR)
Analysis
Results
Names Databases/Symbol Table
Name Kind Location
copy_item function item. c:25
item_cache variable itemc:10
color parameter pallette.c:23
header.h file chapes.c
Abstract Syntax Tree (AST)
Control Flow graph (CFG)
Call Graph

Static Analysis Bug Finders

Generic bug finder tools do not need to know anything about
the intended functionality (features) and do not address coding style

Instead look for “patterns” in the code:

- Code smells may be compliant with style guidelines but are still “bad code”
- Resource leaks where resources (e.g., memory, locks, database connections) are not freed on every code path reachable from an allocation
- Code does not check/handle every possible error code or exception that can be returned by an API
- Using a “blacklisted” library, service, framework with known security vulnerabilities
- More…

Code Smells

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 multiple developers (or your future self) modify the code

Bloater Code Smells

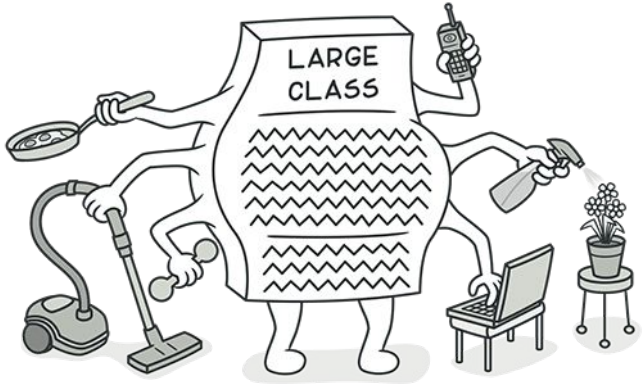
Some code smells are obvious and could be addressed by style checkers

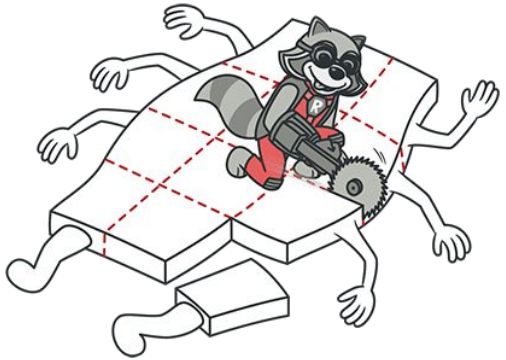
For example, “bloater” classes, methods, parameter lists that that have increased to such gargantuan proportions that they are hard to work with

Bloater smells do not usually crop up right away, they accumulate over time as the program evolves

Typically means the class or method is trying to do too many things

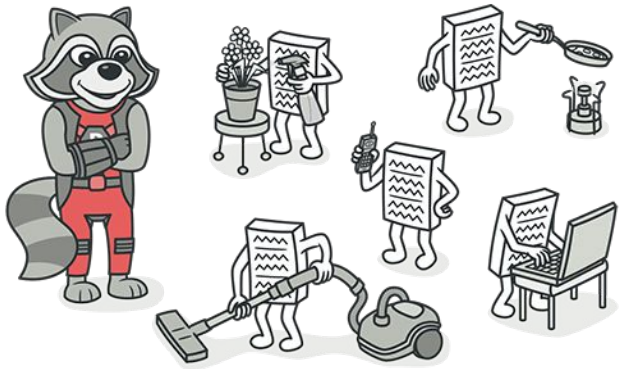
Bloater Class



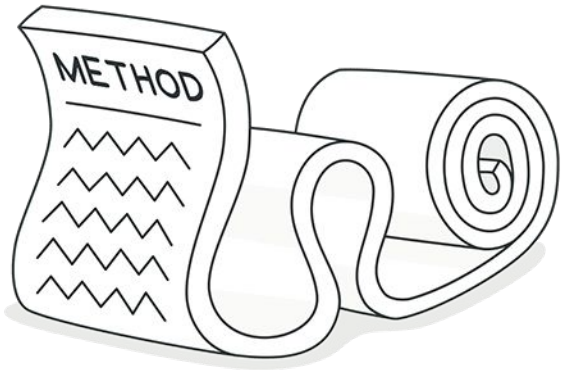


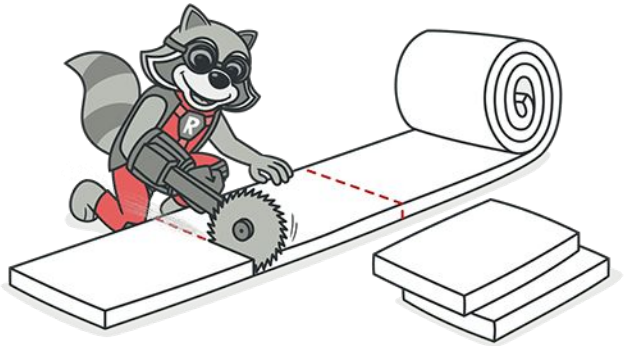
Code smell detectors impose an arbitrary threshold on the number of methods in a class, or allow the user to configure

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

Bloated Comments

Sometimes methods are too long because of embedded comments

If you feel the need to comment on something inside a method, put this code in a new method

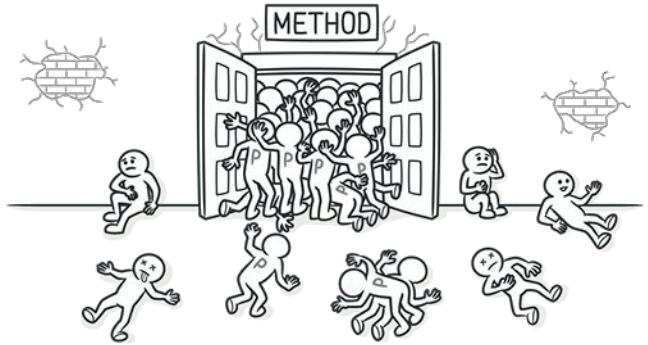
Even a single line can and should be split off into a separate method, if it requires explanation separate from the enclosing method

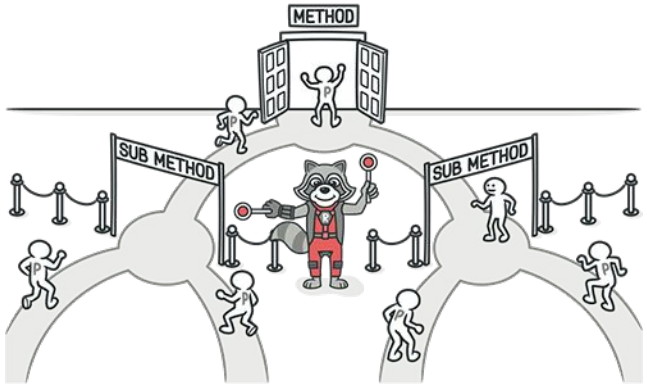
Bloater Parameter List

A method with a long list of parameters, e.g., more than three or four, is hard to read

Complicates calling and testing the method. Testing should consider input spaces individually for each input and in combination

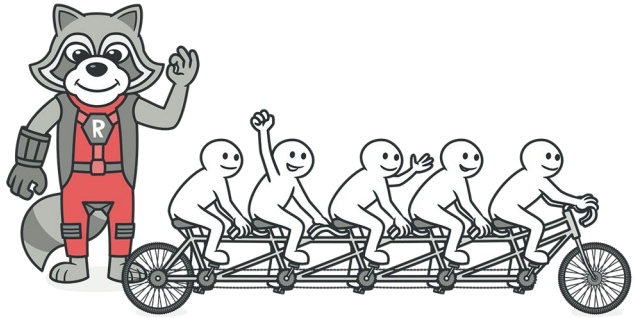
Break into multiple methods with logically related subsets of the parameters

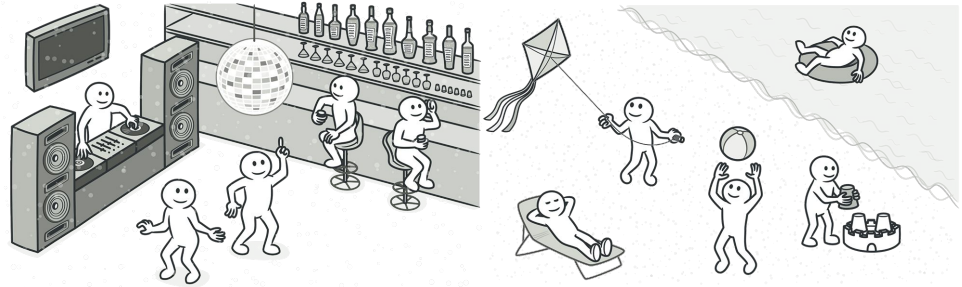




Data Clump

Special case of bloater parameters where the *same* long group of variables is passed around together in *multiple* parameter lists for multiple methods





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”

Coupler Classes

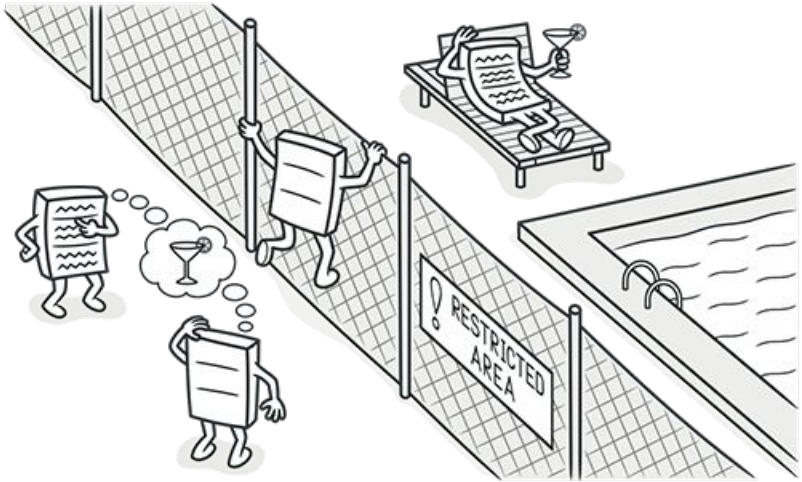
Some smells are less obvious, usually not addressed by coding style rules

“Coupler” classes involve excessive coupling between the classes

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

Feature Envy

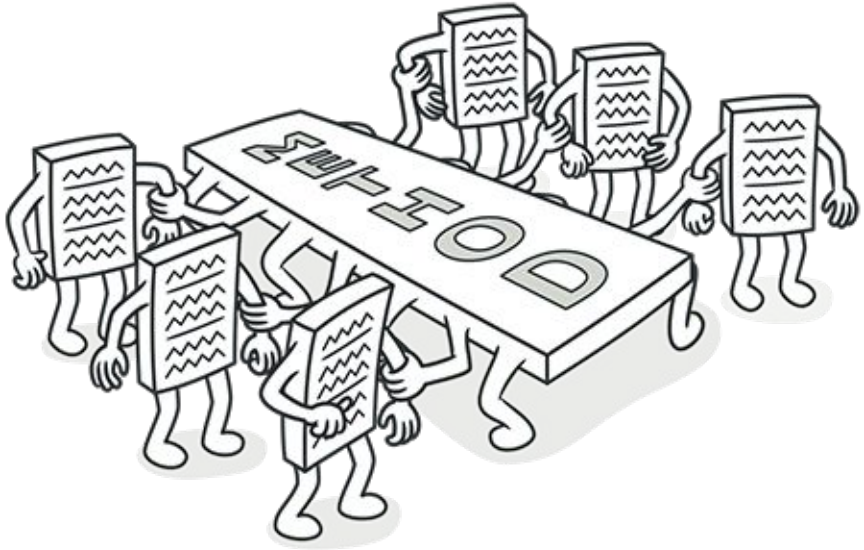
Feature envy - a “coupler” class that uses *public* methods and fields of another class excessively, perhaps more than it uses its own

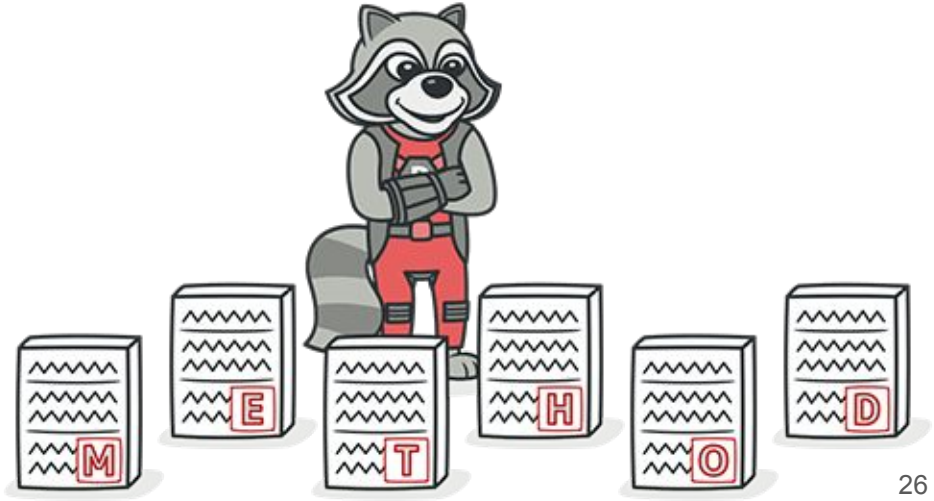


If a code unit cannot be tested in complete isolation, without another code unit, they are too tightly coupled

Inappropriate Intimacy

A “coupler” class that has dependencies on what should be *internal* fields and methods (implementation details) of another class





Example

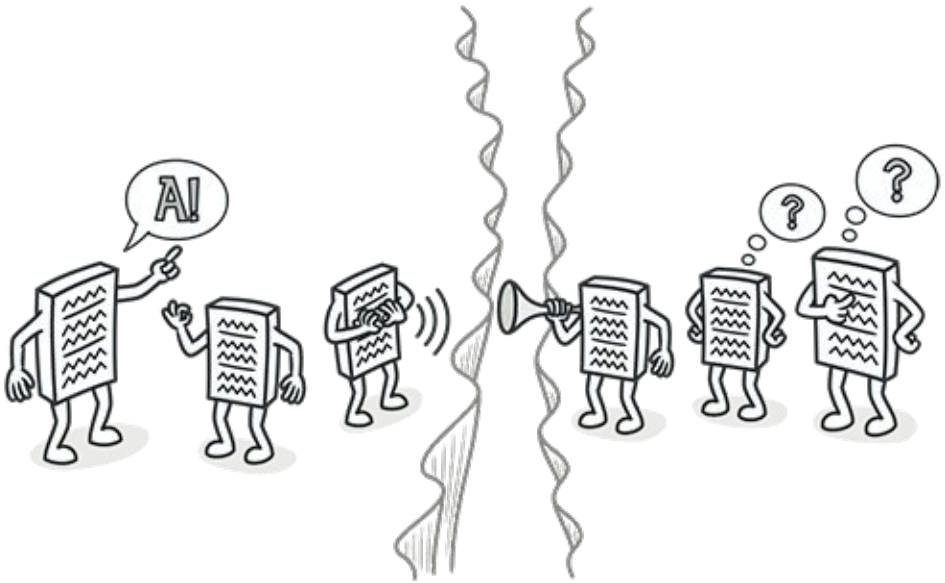
```
public class Phone {
  private final String unformattedNumber;
  public Phone(String unformattedNumber) {
     this.unformattedNumber = unformattedNumber;
   }
  public String getAreaCode() {
     return unformattedNumber.substring(0,3);
   }
  public String getPrefix() {
     return unformattedNumber.substring(3,6);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

Refactoring Message Chains

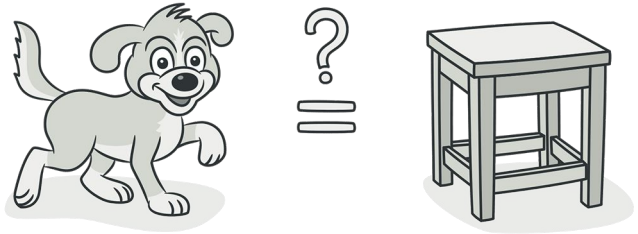
```
# Message Chain
salary =
database.get_company(company_name).
                get_manager(manager_name).
               get_team_member(employee_name).
                salary
```

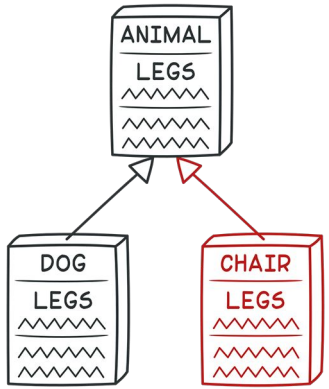
```
# Better
salary =
database.get_employee_by_name(employee_n
ame).salary
```

Object-Oriented 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

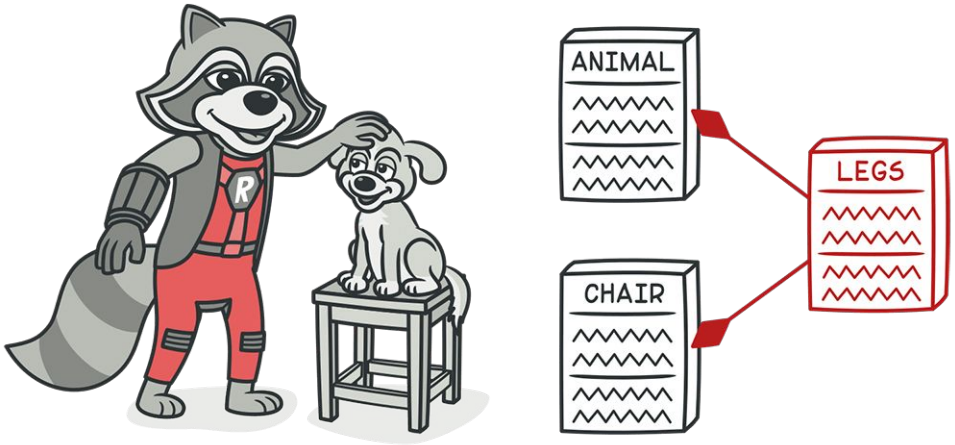
What is the relationship between the dog and the stool? They both have four legs...



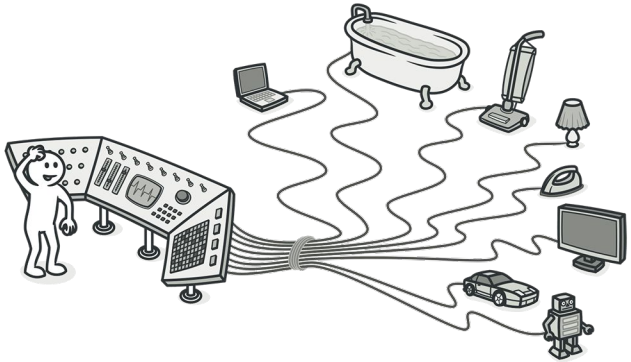


Refused Bequest

Refactor to replace inheritance with
object composition =
‘has-a’ rather than ‘is-a’



Another Object-Oriented Abuser



“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

Lots more code smells and how to recognize and remove them here

Upcoming Assignments

First Iteration due October 24, next Monday!

First Iteration Demo due October 31

➔ You can keep coding and testing between first iteration submission and first iteration demo, but tag both separately! Also see github releases

First Individual Assessment
available 12:01am November 1, due 11:59pm November 4

Next Class

gtest/gmock demo

Finding Bad Code

Ask Me Anything