Lecture Notes
September 7, 2017

First team assignment (team formation)

Continuation of version control from Tuesday

First homework (practice with github) - for students remaining on waitlist and who have signed up with me for canvas access, I will sign add forms after you submit this assignment

Basic functionality of most version control systems:

Backup and restore apply to coordinated collections of files, not just individual files

Synchronization among multiple users

- LOCO model - lock on checkout, unlock on checkin (need way to break lock)
- MOM model - merge on modify

Short term undo - return to last known good version

Long term undo - return to old version as of specific date, specific release, etc.

Track changes - commit messages with who, when, (hopefully) why

Terminology:

Repository (repo) = the file database

What is kept in a repo?

- Source, tests, scripts, resources, configuration
- Usually **not** executables or other generated files built from the stored files
  (these are what is “built” during continuous integration)

Server = where the repo lives

Client = developer machine
command line shell, special GUI client, file system snapin, IDE or code editor plugin

Working set/working copy = local file directory on client where developer makes changes

Main = master = trunk

- Primary set of versions in the repository
- Think of a tree (with branches)

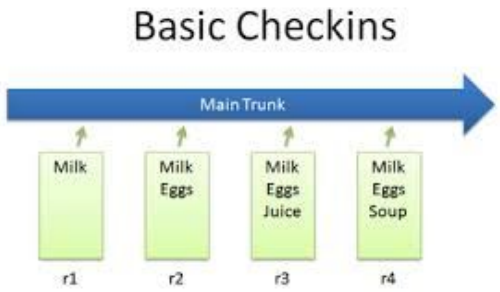
Head = latest revision

Basic actions applied to a repository:

Add file to repo - doesn't work to just add to local copy, need to tell the VCS to track

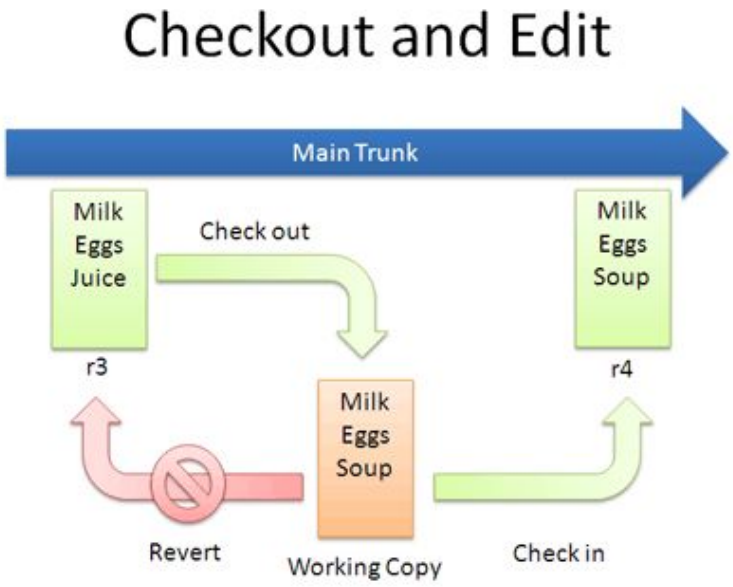
Checkin/Commit/push - upload or copy from local working set to repo

- Updates revision number
- Records commit message and who made the change in changelog/history
- Might be integrated with issue tracker, e.g., associating a bugfix commit with the original bug report
- Here is where most of the problems arise!
- - Accidental or intentional overwrites of other changes



Checkout/pull - download or copy from repo to local working set

- Clone = initial copy
- LOCO (pessimistic) vs. MOM (optimistic)
- Limitations/problems with both, no perfect solution
- Some VCSs distinguish read-only vs. read-write checkout



Revert - throw away local changes and reload latest revisions from repo

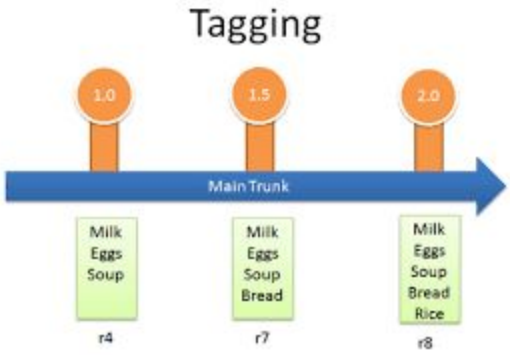
Update/sync

- Get latest revisions of files from repo, may have changed since checkout
- Be careful not to overwrite any local changes (stash)

Advanced actions:

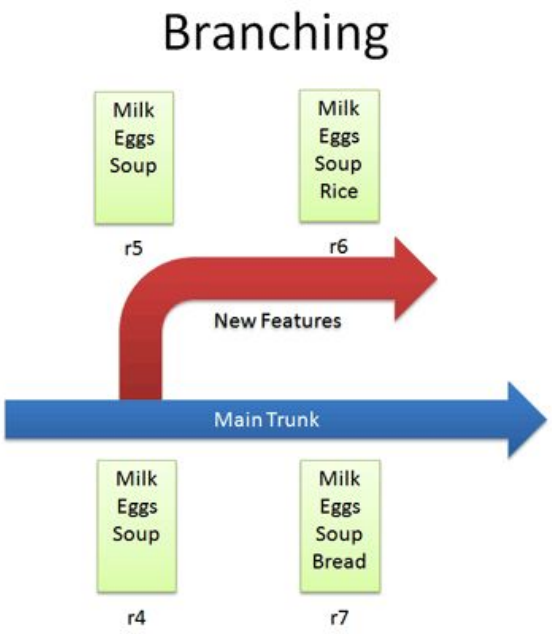
Tagging

- Not the same as revision numbering
- Different files may be revised at different paces, one file rarely changes while another is changed several times per day
- Tagging marks the set of all file revisions contributing to some distinguished milestone, e.g., demo or release
- May keep checkpoint snapshot



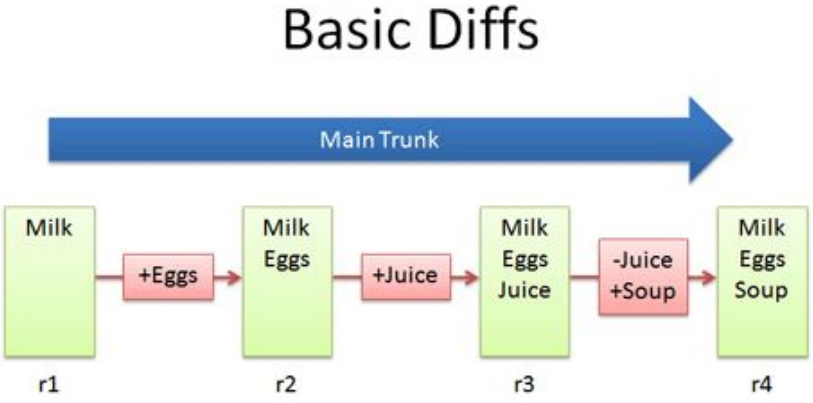
Branching and merging - branch is both a noun and a verb (the branch, to branch)

Fork copy of code base and track changes separately, may or may not later merge back to main line or with another branch



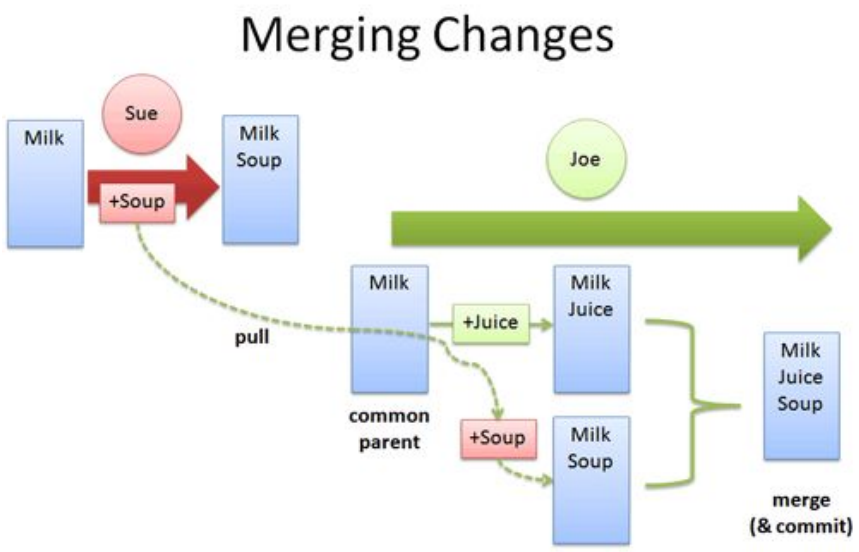
Diff/delta - find differences between two revisions of a file, usually sequential revisions

- Many “diff” algorithms used by different tools, helpful for merging
- Can also reduce storage required for saving many revisions
- Maintain original file, and then store only diffs from that revision to the next
  - to get latest version need to reapply all those diffs
- Maintain latest file, and then store only reverse-diffs to the previous revision
  - to get latest version just grab it, only need to reapply diffs to get older versions



Merge

- Integrate changes from working set into previous revision
- Also applies to full branches
- May be automatic when there are no conflicts



Conflicts may be detected automatically

- Usually in context of simultaneous changes by different developers
- When pending changes contradict each other
- Usually purely lexical (dumb), same line or same method is modified
- Ideally semantic (smart), requires sophisticated static analysis such as slicing

Resolution usually manual

- One developer integrates two or more sets of changes (good time to use pair programming even if not normally used!)
- Then commits corrected version

Team software development cannot function without a shared VCS

SCCS 1972 … RCS 1982 first generation (probably others used internally even earlier)
There is nothing novel or researchy about VCS, just use it!

Static Analysis

Second homework (practice with static analysis)

Static analysis is anything a tool can do just by looking at the code, without execution

Static analysis applies to a range of inputs, possibly “all” inputs - or all valid inputs

Dynamic analysis requires execution, besides testing includes “checking”, monitoring, debugging, etc.

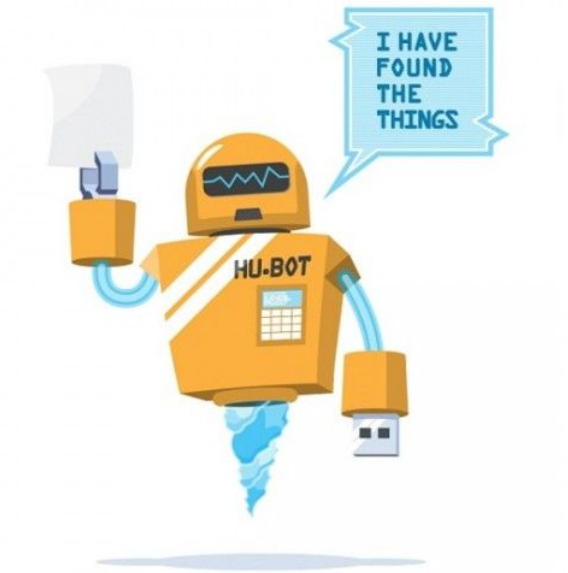
Many interesting properties we would like to check are undecidable

e.g., halting problem

```
SELF-HALT(program)
{
    if(DOES-HALT(program, program))
        infinite loop
    else
        halt
}
```

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

Sometimes lexical, but usually works with an internal representation of program - like a compiler uses - rather than the source code text



Style Checkers

Most programming and scripting languages have a publicly defined coding convention, often called a coding style or code standard

Some languages have multiple coding conventions that are sometimes mutually incompatible

Example style guides for Python:

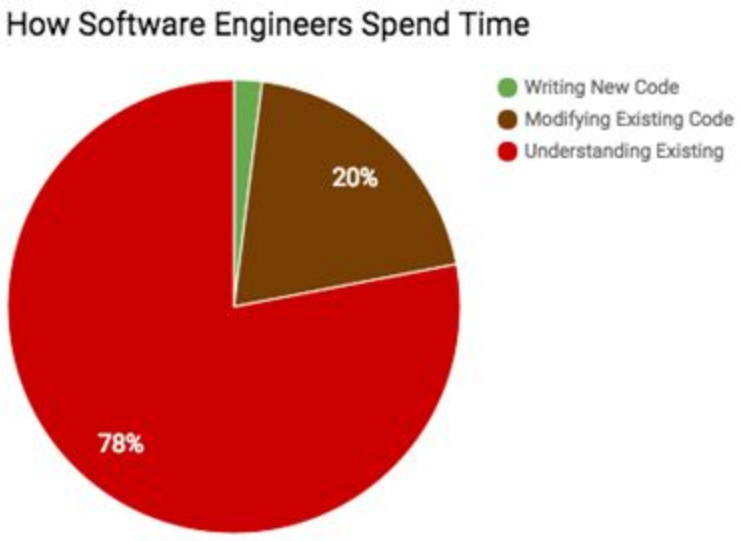
- PEP 8 - Style Guide for Python Code (PEP = Python Enhancement Proposal)
- Google Python Style Guide
- The Hitchhiker's Guide to Python

Google has style guides for everything: Google Style Guides

The main purpose of coding conventions is to make it easy (or easier) for **other** developers to understand **your** code, even for the now you to communicate with the future you

If previous developers didn't use good coding conventions, how will 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 following code and tell me which tests pass
- neither, first, second, both

```java
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);
    }

}
```

The coding style violation in this example concerns indentation

Coding conventions govern how and when to use comments, appropriate use of white space, proper naming of variables and functions, code grouping and organization, sometimes file/directory structure

- Patterns to be used and Patterns to be avoided
- Make errors more obvious - unfamiliar patterns jump out of the code when you look at it

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 style guidelines, which may run standalone or as code editor, IDE, build, CI plugins - or as part of a general static analyzer

Numerous linters for Atom editor

Checkstyle for Java can be configured to almost any coding standard, numerous plugins for development environments and build processes

Python Prospector combines pylint with other checkers

SonarQube provides code analyzers for many languages

Code Smells

Some static analyzers can detect “code smells” = badly written code

Code smell refers to symptom in the source code that possibly indicates deeper problem

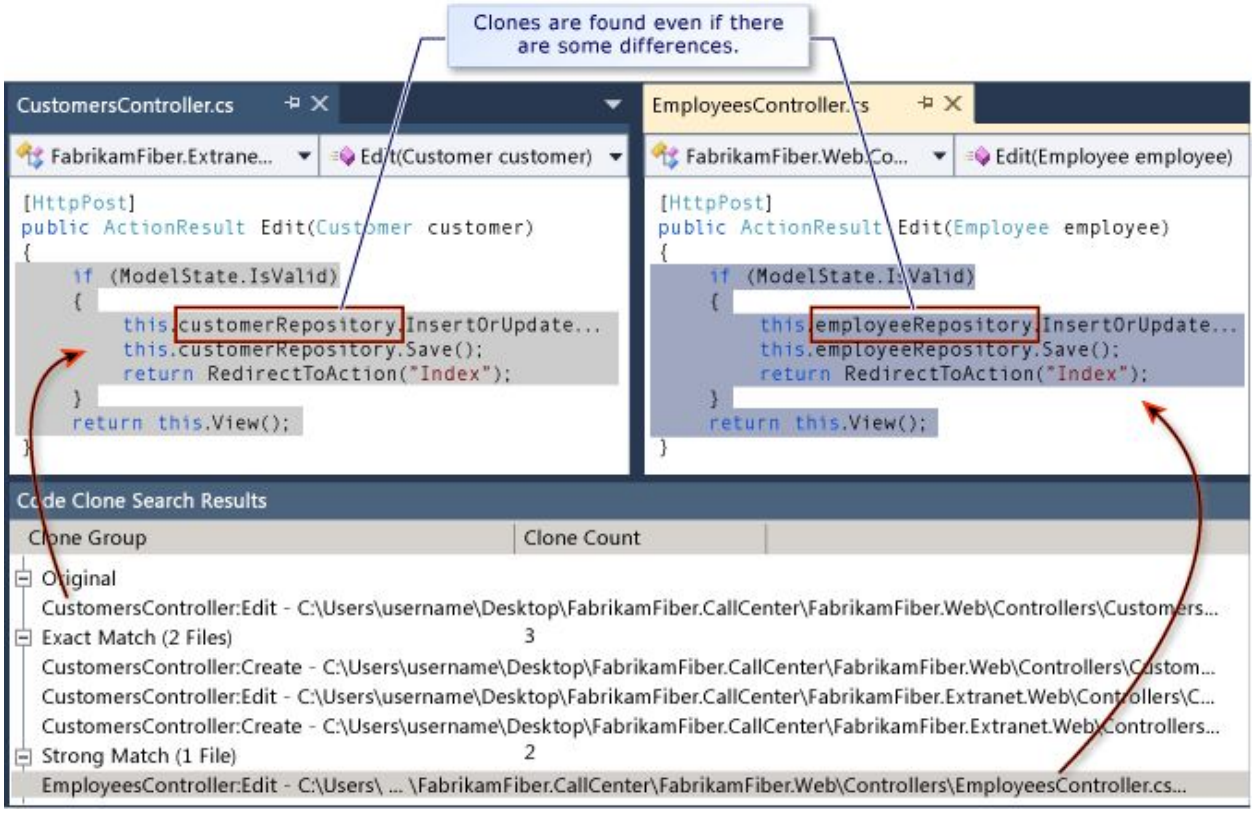
> Does not necessarily indicate bugs, but tends to make changes more difficult and more likely to lead to bugs

“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

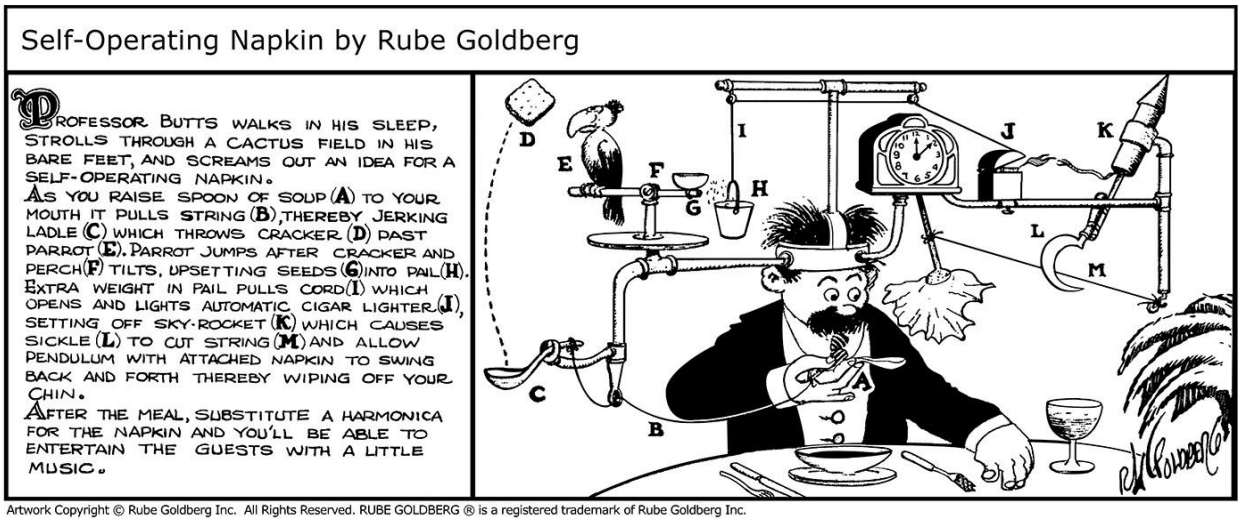
> Analogous to monetary debt - if technical debt not repaid quickly, can accumulate “interest” that adds up substantially over time

Some application-level smells:

Duplicated code (“code clones”) - “dispensable” identical or very similar code exists in more than one location, typically arises via copy/paste

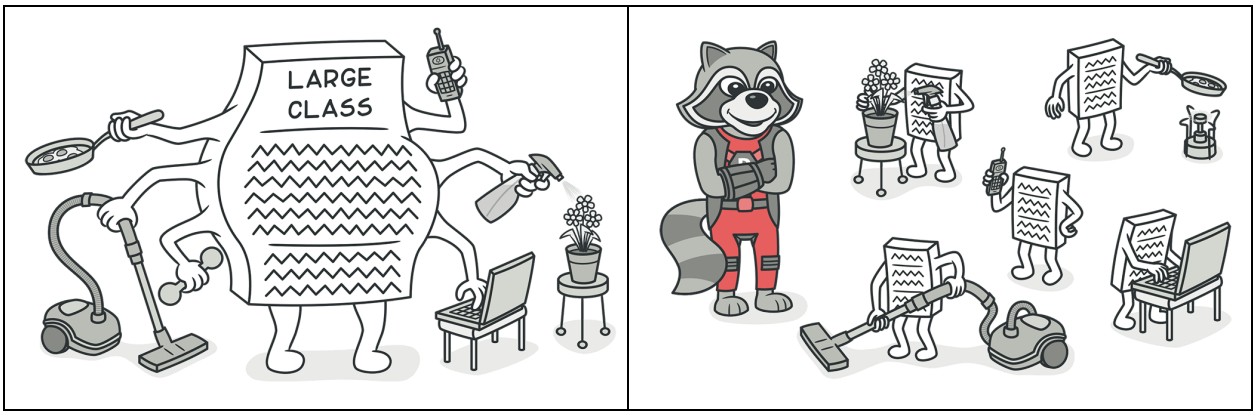


Contrived complexity - forced usage of overcomplicated design where simpler design would suffice (some design patterns covered later in course)



Some class-level smells:

Large class (god object) - a “bloater” class that has grown too large, typically accumulated over time as program evolves

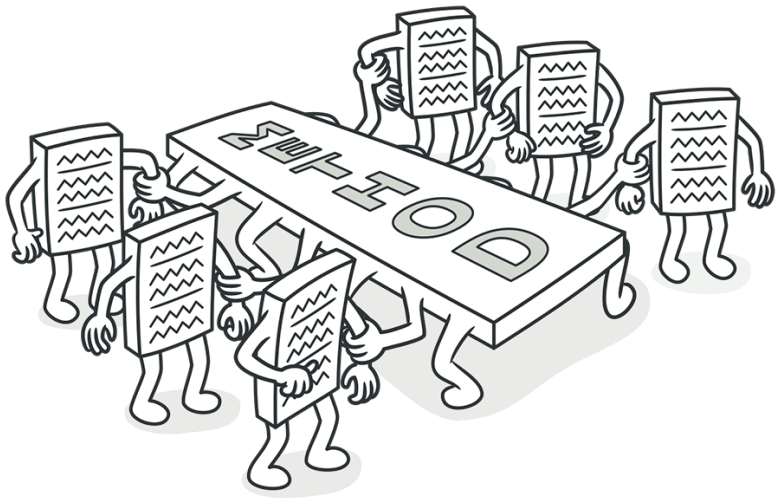


Feature envy - a “coupler” class that uses public methods of another class excessively

```
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();
   }
```

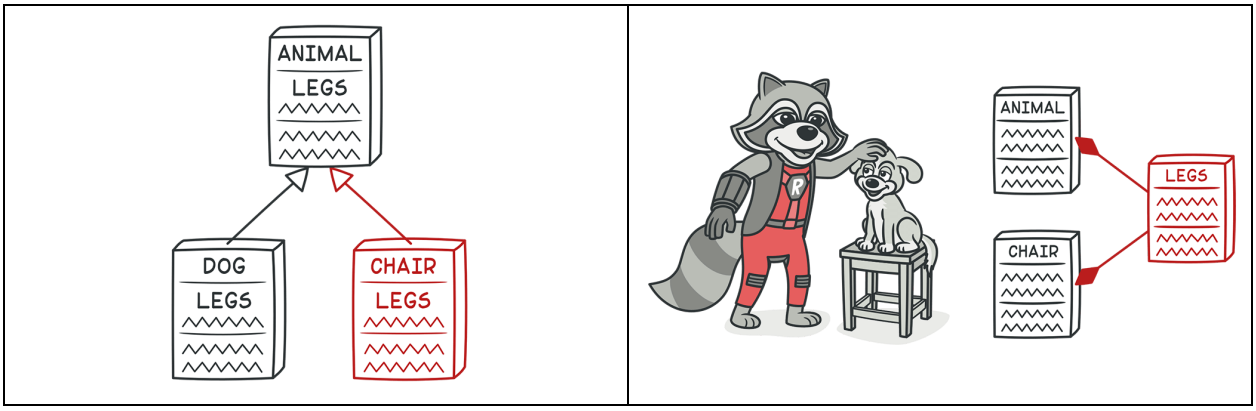
```
public class Phone {
   private final String unformattedNumber;
  public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
  private String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
  private String getPrefix() {
      return unformattedNumber.substring(3,6);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

Inappropriate intimacy - another kind of “coupler” class that has dependencies on implementation details (internal fields and methods) of another class



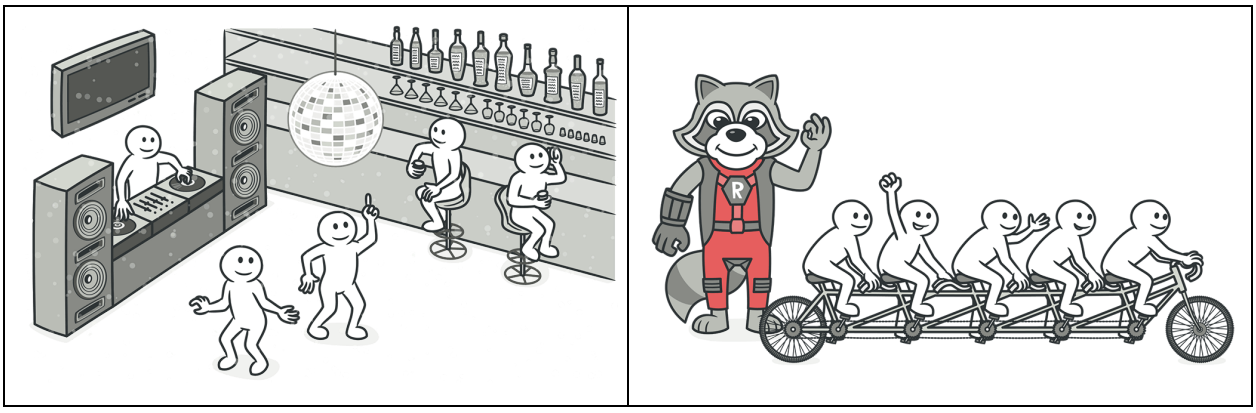
Refused bequest - an “object-orientation abuser” class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class

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



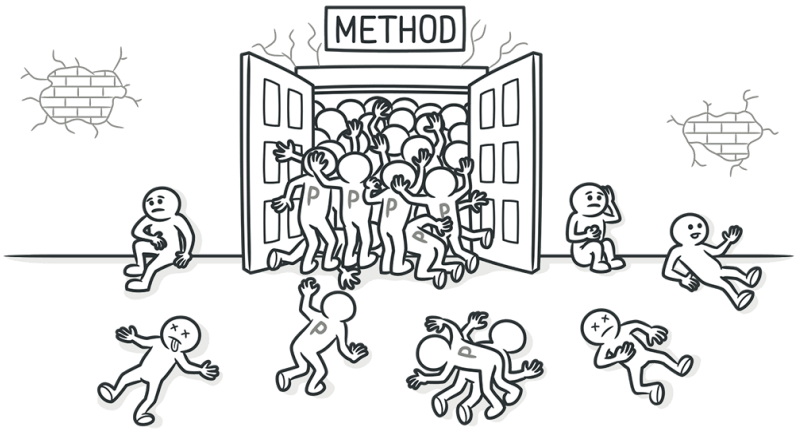
Data clump - Another “bloater” that occurs when group of variables passed around together in various parts of code

> More appropriate to formally group the different variables together into a single object, and pass around only this object instead

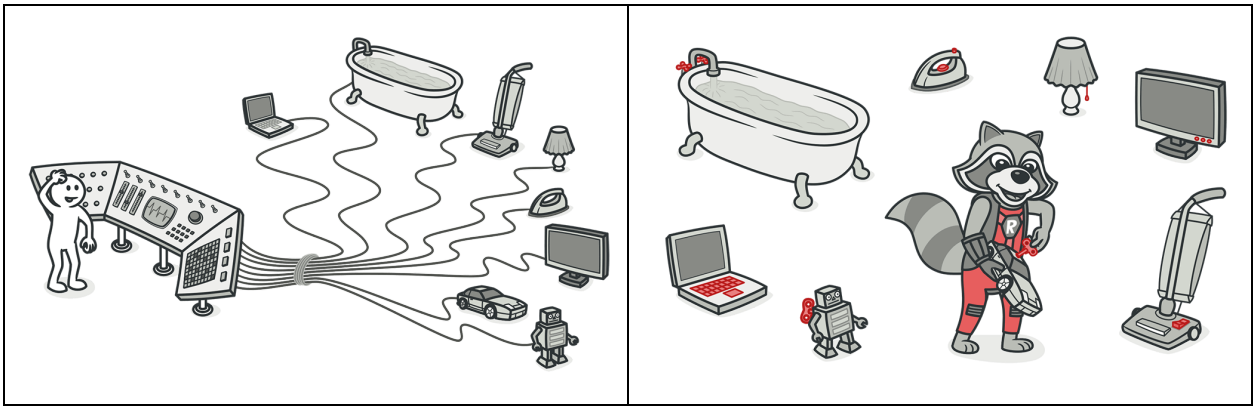


Some method-level smells:

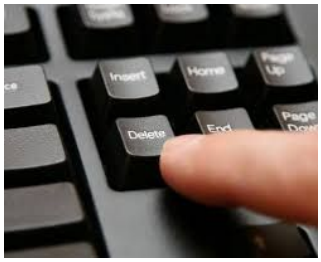
Too many parameters - a long “bloater” list of parameters is hard to read, and makes calling and testing the function complicated



Switch statements - complex switch operator or sequence of if statements “object-orientation abuser”



Dead code, speculative generality - “dispensible” unused code, sometimes previously used, sometimes created “just in case” for future (YAGNI = you ain’t gonna need it)



Message chains - 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 …

```
# 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
```

Many other generic code smells, some language-specific, e.g., Java code smells

Refactoring tools, typically provided as IDE plugin, can find some code smells and suggest/automate rewriting code to remove the smell and leave cleaner code

Refactoring will be covered later on

Bug Finders

Bug finders (or bug checkers) are sophisticated tools that look for generic bugs as well as code smells

- They do not need to know anything about the intended functionality (features) and in most cases do not require any “specification” for the software
- Sometimes encode domain-specific rules, e.g., web client/server messaging formats and protocols, or knowledge of error specifications, e.g., library functions return 0 on success and a negative integer on failure

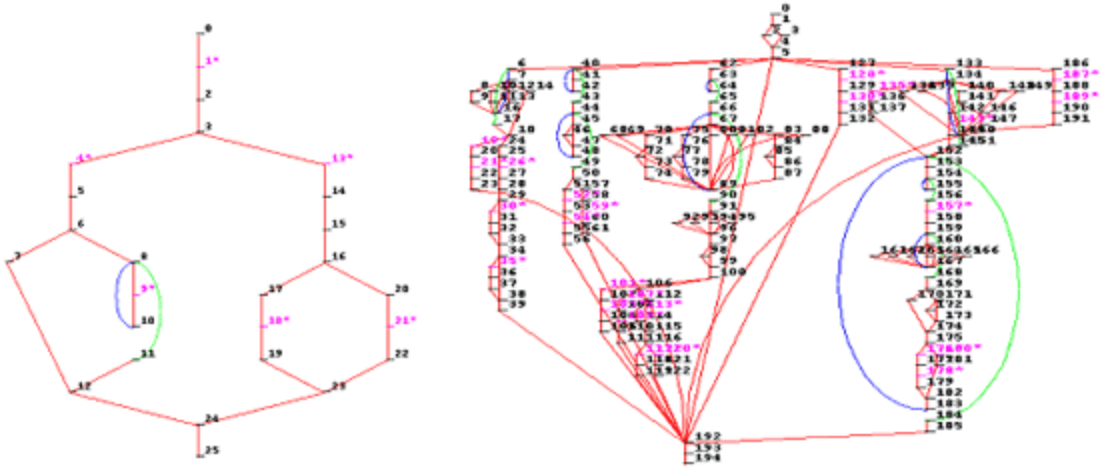
[FindBugs for Java]

Examples of what bug checkers can detect:

Secret keys, tokens or passwords embedded in code

Using third-party libraries with known security vulnerabilities

Too many paths in a given function or method (cyclomatic complexity)



Simple Code vs. Complex Code

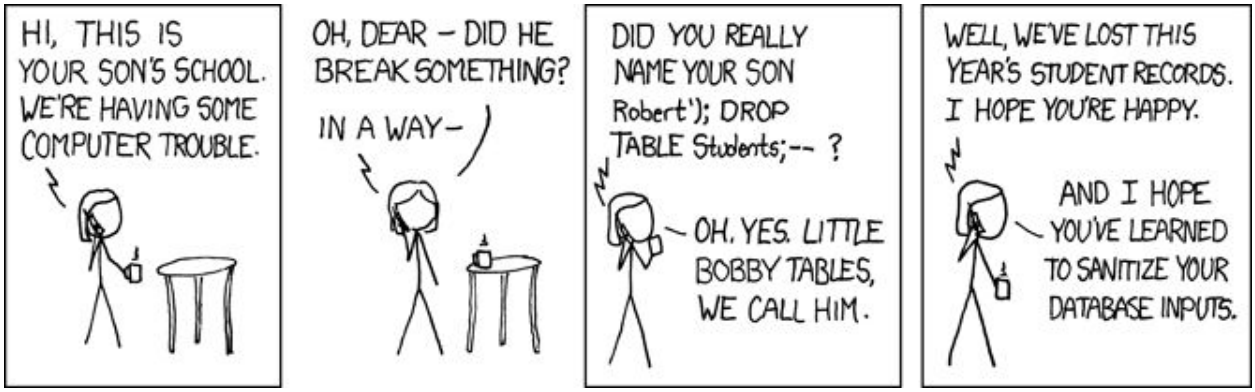
Code does not handle every error code that can be returned from an API function or system call

[System Error Codes: 1 to 15841]

Dereferencing a null pointer

[You're dereferencing a null pointer!]

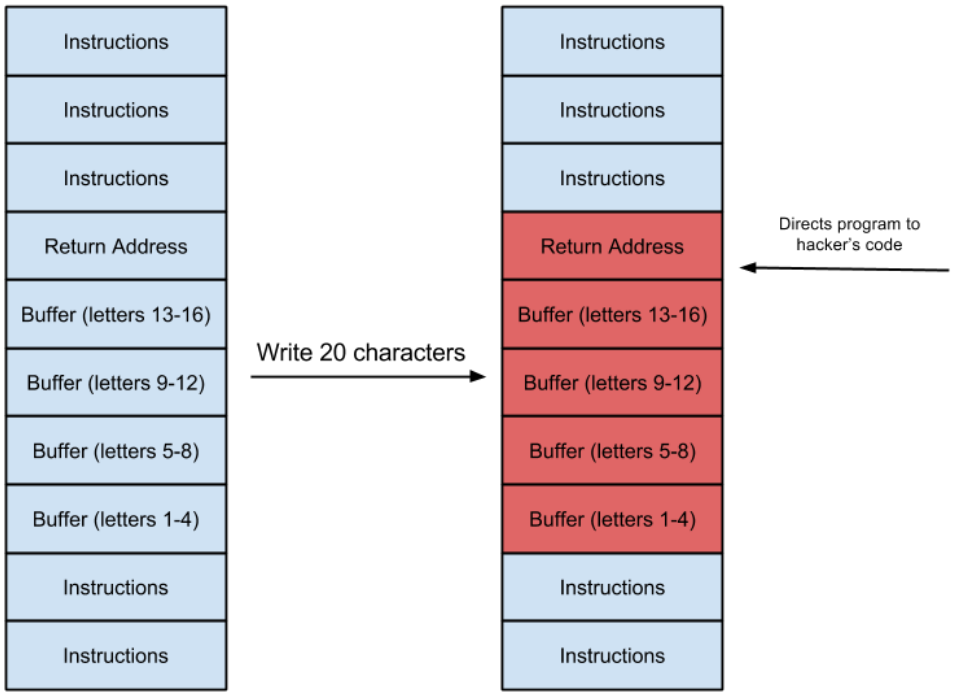
Input from user passed unsanitized to framework/database code - sanitize

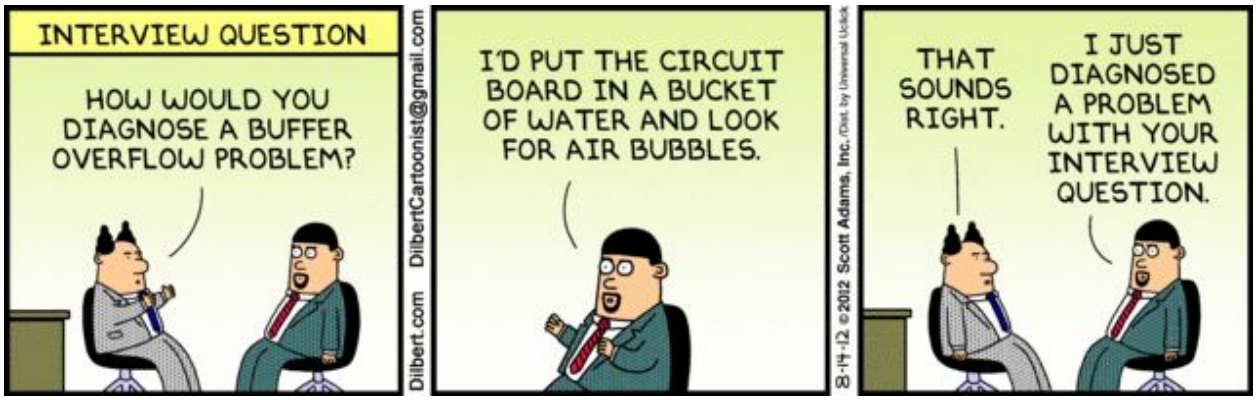


In a non-memory managed language, checks every path from memory allocation to make sure there's corresponding memory deallocation - and that there are no further uses of the memory after it has been deallocated ("use after free")

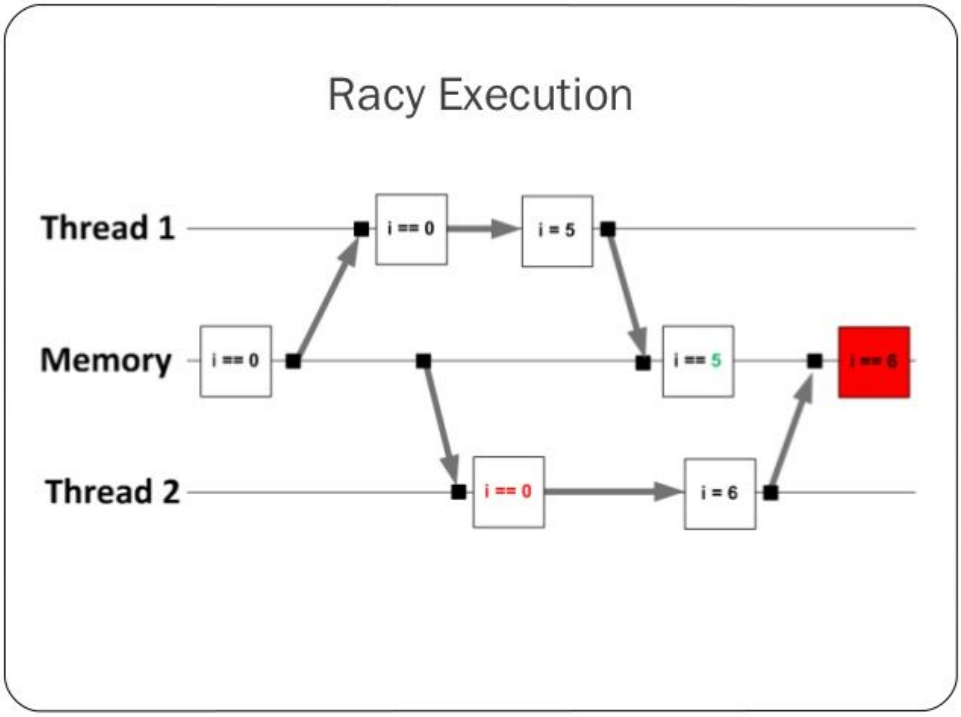
CWE-416: Use After Free

Also looks for checks/limits on array bounds, to prevent buffer overflow





In multi-threaded code, check that data dependencies cannot race and locks are matched with unlocks along every execution path from the lock



Static analysis bug checkers typically produce many *false positives*, e.g., a missing free or unlock on an infeasible execution path, so various kinds of error/warning messages can be suppressed

Static analysis bug checkers cannot avoid *false negatives*, they cannot find “all” bugs and cannot “prove” that no bug exists

Static analyzers do not replace testing - static analysis finds bugs that *might* happen with some input, dynamic analysis finds bugs that *did* happen with specific input but cannot consider all possible inputs

You do not need to understand how they work internally in order to use static analysis tools