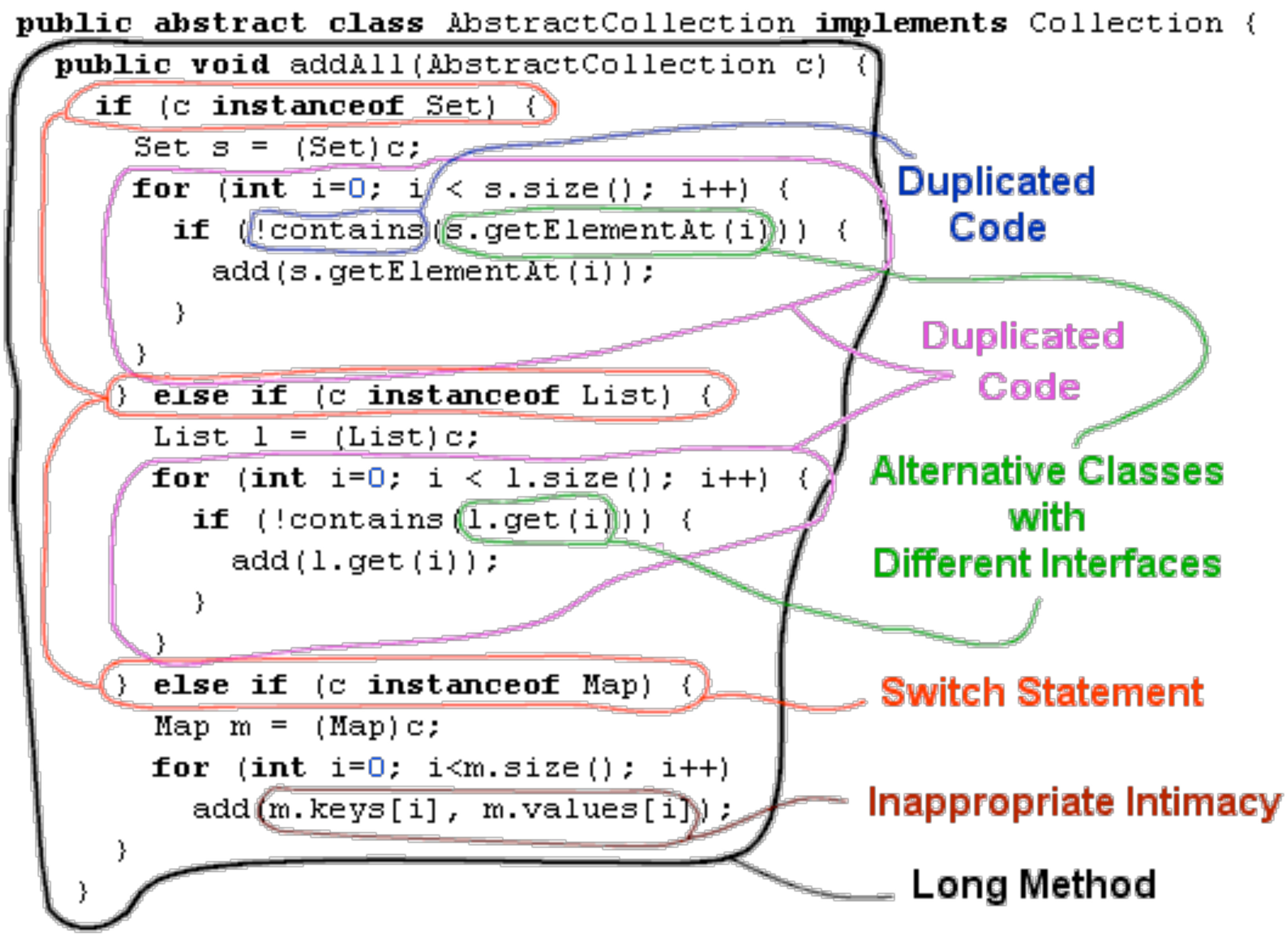


```
public abstract class AbstractCollection implements Collection {
  public void addAll(AbstractCollection c) {
    if (c instanceof Set) {
      Set s = (Set)c;
      for (int i=0; i < s.size(); i++) {
        if (!contains(s.getElementAt(i))) {
          add(s.getElementAt(i));
        }
      }
     } else if (c instanceof List) {
       List l = (List)c;
       for (int i=0; i < l.size(); i++) {
         if (!contains(l.get(i))) {
           add(l.get(i));
         }
       }
     } else if (c instanceof Map) {
       Map m = (Map)c;
       for (int i=0; i<m.size(); i++)
         add(m.keys[i], m.values[i]);
    }
  }
```

CPEN 321

Code Reviews, Code Anti-Patterns

Outline

- Types of code review
- What to look for in code reviews
- Automated tool
- Expectations for the “Code Review” milestone

Code Reviews

- A constructive review of a fellow developer's code.

- Analogy: when writing articles for a newspaper, what is the effectiveness of the author editing their own article?

Code Reviews: Why Do Them?

- Improved code quality
  - Prospect of someone reviewing your code raises the quality threshold.
  - Forces code authors to articulate their decisions
- Hands-on learning experience from peers
  - Direct feedback leads to better algorithms, tests, design patterns
- Better understanding of complex code bases
  - Reviewing others' code enhances the overall understanding of the system, reduces redundancy.

Code Reviews: Mechanics

- **Who**: original developer and reviewer(s), either in person or online
- **What**: reviewer gives suggestions for improvement on a logical and/or structural level, to conform to a common set of quality standards.
  - Feedback leads to discussion and modifications
  - Reviewer eventually approves code
- **When**: code author has finished a coherent system change that is ready for production
  - Before committing the code to the central repository or incorporating it into the new build.

Code Reviews: Who Does Them?

- **Everyone**: a common industry practice.
- A required sign-off from another team member before a developer is permitted to push changes or new code

Google

yelp

f

Microsoft

amazon

Common Types of Code Reviews

- Formal inspections
- Walkthroughs
- Pair programming

Formal Inspections

- Introduced in the mid-1970s
- A formalized code review with
  - roles (moderator, author, reviewer, scribe, etc.)
  - a specific **checklist** and workflow and processes

SEI CMM Recommendations

- Adequate **resources** and funding are provided for performing reviews on each software work product
- Review leaders receive required **training** in how to lead peer reviews.
- Reviewers who participate in reviews receive required **training** in the objectives, principles, and methods of peer reviews.
- Reviews are **planned**, and the plans are documented.
- Reviews are performed according to a **documented procedure**.
- Data on the conduct and results of the reviews are **recorded**.
- **Measurements** are made and used determine the status of the review activities.
- The software quality assurance group reviews and/or **audits** the activities and work products for peer reviews and reports the results.

Walkthroughs

- An informal discussion of code between author and a single reviewer.
  - The author walks the reviewer through a set of code changes.
- Advantages
  - Simplicity in execution: anyone can do it, anytime.
- Disadvantages
  - Not an enforceable process, no record of the review.
  - Easy for the author to unintentionally miss a change.
  - Reviewers rarely verify that defects were fixed.

Pair Programming

- Two developers writing code at a single workstation with
  - only one typing
  - continuous free-form discussion and review
- Advantages
  - Deep reviews, instant and continuous feedback.
  - Learning, sharing, team-building.
- Disadvantages
  - Some developers don’t like it.
  - No record of the review process.
  - Time-consuming.

In Practice …

- A mixture of techniques
  - Less formal than formal inspections but more formal than random walkthroughs
- Typically integrated into the project workflow
  - E.g., all changes are to be submitted via pull requests which require at least two approvals
  - Can rely on tools to facilitate the review process, e.g., Gerrit
- Typically testing and various types of static analysis are automatically performed before the manual review
  - E.g., Travis CI, Jenkins, static analysis to find null pointer exceptions, code smells, etc.

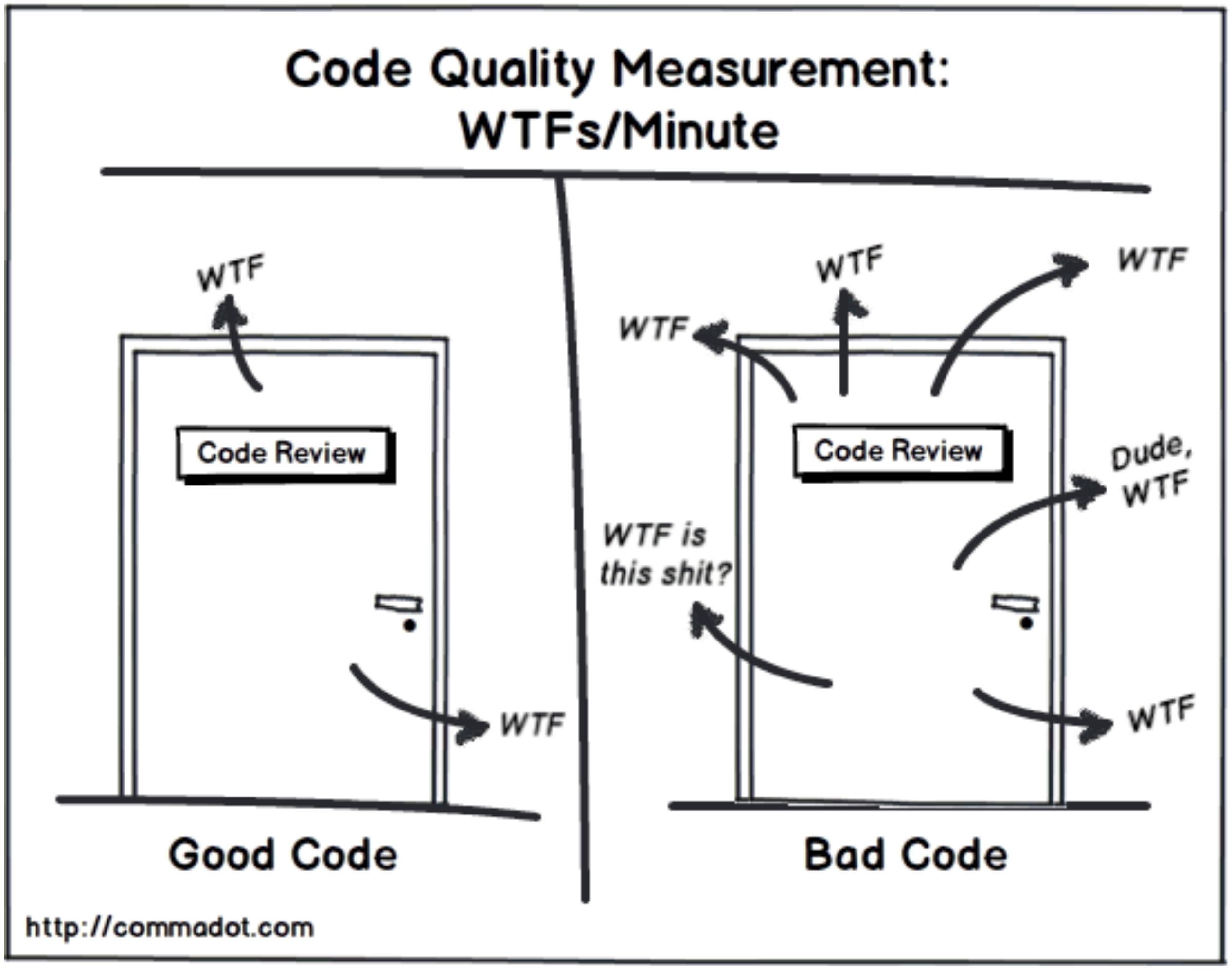
Example: what changes, if any, would you suggest?

```
public class Account {
   double principal, rate; int daysActive, accountType;
   public static final int STANDARD = 0, BUDGET = 1,
      PREMIUM = 2, PREMIUM_PLUS = 3;
   }

   public static double calculateFee(Account[] accounts) {
      double totalFee = 0.0;
      Account account;
      for (int i = 0; i < accounts.length; i++) {
         account = accounts[i];
         if (account.accountType == Account.PREMIUM ||
         account.accountType == Account.PREMIUM_PLUS)
            totalFee += .0125 * ( // 1.25% broker's fee
            account.principal * Math.pow(account.rate,
            (account.daysActive / 365.25)) - account.principal); // interest-
                                                                // principal
      }
      return totalFee;
}
```

Possible Changes

- Comment.
- Make fields `private`.
- Replace magic values (e.g., 365.25) with constants.
- Use an `enum` for account types
- Introduce intermediate variables, methods
- Use consistent whitespace, line breaks, etc.

What to look for?



A Quote From a Senior Software Developer (in one of the Top-5 Companies)

- First thing: check for **maintainability** by reading the CR without reading the title and description of the review. These won't be available when this code would need to be changed/updated/maintained in 6 months. Once you understand the code, compare your understanding to the title/description. A discrepancy between the two means that either the code doesn't do what it suppose to do or it is a confusing code.
- Bugs due to using **technologies that are complicated** - especially regexp (never saw a CR with a regexp that didn't have a scenario that the developer didn't consider).
- Look for **security / privacy issues**, especially with new endpoints or new parameters to existing ones (there are always security issues with these).
- Look for needed **refactoring**.
- See **tests**.
- Look for **code smells**.
- Ask for **meaningful comments**.
- See if there's some **specific person** that should look at the code and request that person to take a look (we get billion CR a day, need to select carefully what to review).

What to look for?

- Bugs
- Security vulnerabilities
- Performance
- Maintainability
- Readability
- Comments
- Adherence to coding standards and best practices

What to look for?

- Bugs
- Security vulnerabilities
- Performance
- Maintainability
- **Readability**
- **Comments**
- **Adherence to coding standards and best practices**

$\left.\begin{array}{l}\text{Bugs}\\\text{Security vulnerabilities}\\\text{Performance}\\\text{Maintainability}\end{array}\right\}$ Requires deep familiarity with the project

Code Smells

- A characteristic of a design that is a strong indicator it has poor structure, and should be modified (refactored)

- Code smells are rules of thumb
  - It's not always straightforward that a bad smell must lead to a problem. Have to use judgment.
  - Still, as new designers, bad code smells likely mean you should change your code.

Code Smells: In Short

- The most common design problems result from code that:
  - Is duplicated
  - Is unclear
  - Is complicated

- SOFA
  - Is it Short?
  - Does it do One thing?
  - Does it have Few arguments?
  - Is it a consistent level of Abstraction?

More Details

- Some well-known method-level code smells
- Some well-known class-level code smells

More Details

- **Some well-known method-level code smells**
- Some well-known class-level code smells

Magic Numbers

- Unique values with unexplained meaning or multiple occurrences which should be replaced with named constants
  - As in the earlier example

Comments

There's a fine line between comments that illuminate and comments that obscure.

Bad Comments

```
// Add one to i.
i++;
```

```
// Lock to protect against concurrent access.
SpinLock mutex;
```

```
// This function swaps the panels.
void swap_panels(Panel* p1, Panel* p2) {...}
```

Bad Comments

```
// Loop through every array index, get the
// third value of the list in the content to
// determine if it has the symbol we are looking
// for. Set the result to the symbol if we
// find it.
```

Comments

There's a fine line between comments that illuminate and comments that obscure.

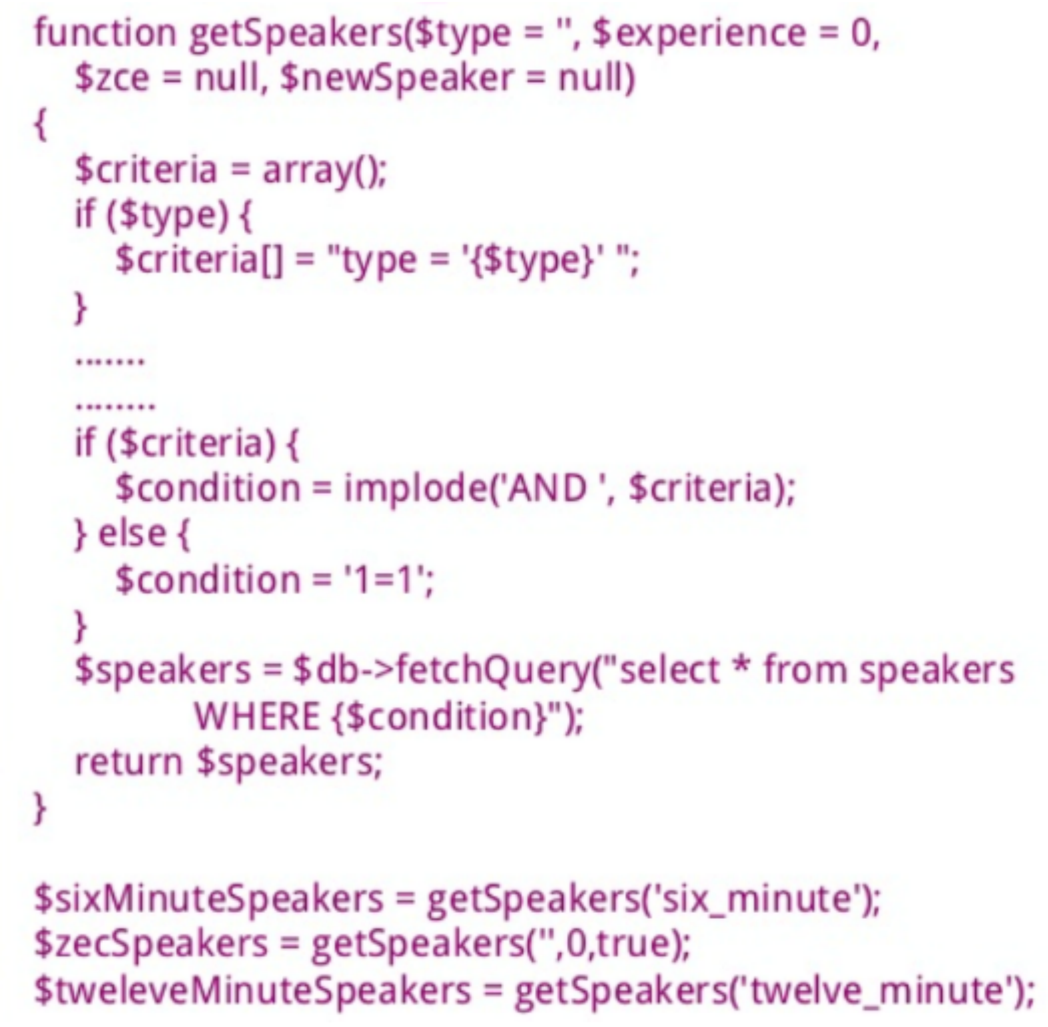
- Are the comments necessary?
- Do they explain "why" and not "what"?
- Can you refactor the code, so the comments aren't required?
  - And remember, you're writing comments for people, not machines.

Long Method

A method that has too many lines of code

- How long is too long? Depends. No hard rules.
- Over 20 is usually a bad sign.
- Under 10 lines is typically good.

Long (Overly Complex) Method: Example



```
function getSpeakers($type = '', $experience = 0,
    $zce = null, $newSpeaker = null)
{
    $criteria = array();
    if ($type) {
        $criteria[] = "type = '{$type}' ";
    }
    .......
    ........
    if ($criteria) {
        $condition = implode('AND ', $criteria);
    } else {
        $condition = '1=1';
    }
    $speakers = $db->fetchQuery("select * from speakers
                WHERE {$condition}");
    return $speakers;
}

$sixMinuteSpeakers = getSpeakers('six_minute');
$zecSpeakers = getSpeakers('',0,true);
$tweleveMinuteSpeakers = getSpeakers('twelve_minute');
```

Long Method

- Problem
  - The longer a method, the harder it is to understand, change, and reuse

- Fix: extract method
  - Take chunks of code from inside long method, and make a new method
  - Call new method inside the now-not-so-long method.

Long Parameter List

- The more parameters a method has, the more complex it is.
- Boolean arguments should be a yellow flag
  – If a method behaves differently based on a Boolean argument value, maybe it should be 2 methods
- If arguments “travel in a pack”, maybe you need to extract a new class (“*Data clump*”)

Duplicated Code

- The same or very similar code appears in many places (code clones)

- Problem
  - A bug fix in one code clone may not be propagated to all
  - Makes code larger than it needs to be

- Fix: *extract method* refactoring
  - Create new method that encapsulates duplicated code
  - Replace code clones with method calls

***DRY**: Don't Repeat Yourself!*

Duplicated Code: Example

```
extern int array_a[];
extern int array_b[];

int sum_a = 0;

for (int i = 0; i < 4; i++)
   sum_a += array_a[i];

int average_a = sum_a / 4;

int sum_b = 0;

for (int i = 0; i < 4; i++)
   sum_b += array_b[i];

int average_b = sum_b / 4;
```

```
int calc_average_of_four(int* array) {
   int sum = 0;
   for (int i = 0; i < 4; i++)
       sum += array[i];

   return sum / 4;
}
```

```
extern int array1[];
extern int array2[];

int average1 = calc_average_of_four(array1);
int average2 = calc_average_of_four(array2);
```

Inconsistent Names

- Pick a set of standard terminology and stick to it

- Strive for symmetry
  - For example, if you have open(), you should probably have close()
  - Having both abortParsing() and stopReading() is confusing

Goes without saying: use meaningful names!

Conditional Complexity

- Problem: large conditional logic blocks, switch statements

- Long code is hard to understand. Even more so when the code is filled with conditions:
  - While you are busy figuring out what the code in the `then` block does, you forget what the relevant condition was.
  - While you are busy parsing `else`, you forget what the code in `then` does.

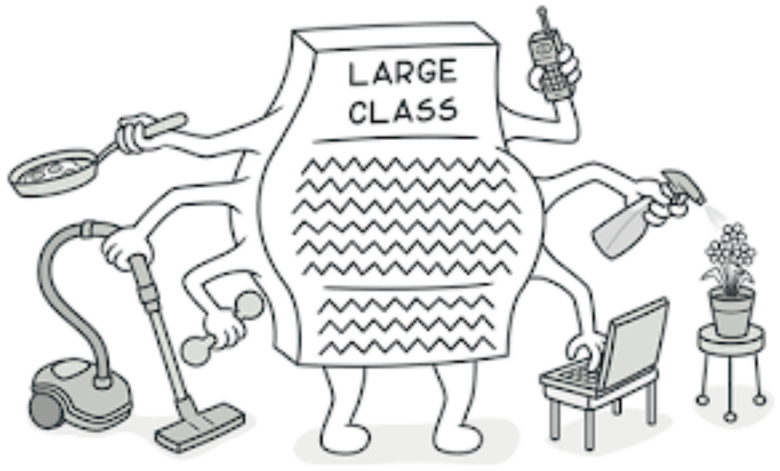
Switch Statements

- The cases in a `switch` statement contain logic for different types of instances of the same class
- This indicates that new submodules should be created

- Some well-known method-level code smells
- **Some well-known class-level code smells**

Large Class

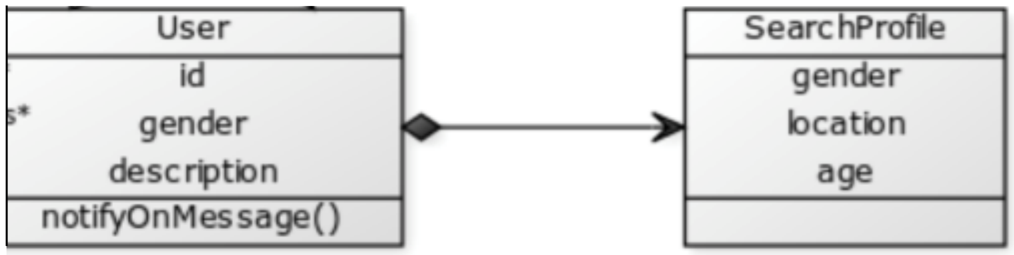
- Large class
  - A class is trying to do too much
  - Many fields
  - Many methods



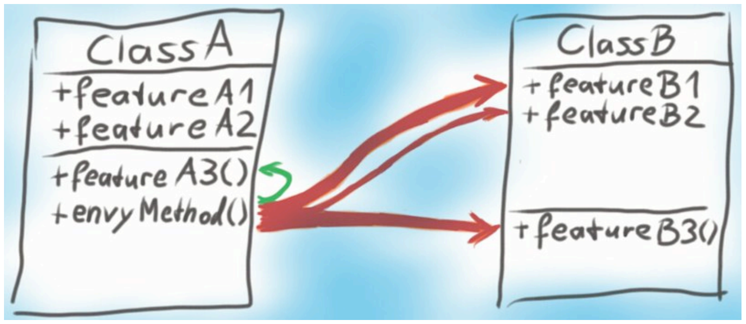
- Problem:
  - Indicates abstraction fault: there is likely more than one concern embedded in the code
- Fix:
  - Take a subset of the instance variables and methods and create a new class with them
  - Move one or more methods to other classes

Lazy Class

- Every additional class increases the complexity of a project.
- A class that isn't doing enough to "pay for itself" should be eliminated:
  - inlined?
  - combined into another class?
- It might be a class that was added because of changes that were planned but not made



Feature Envy



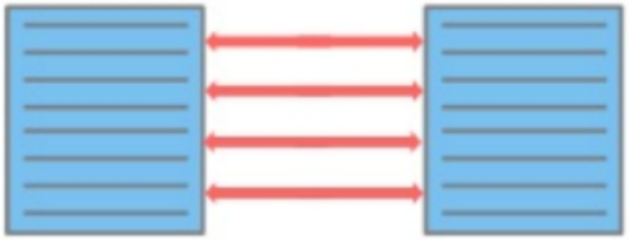
- A method in one class uses primarily data and methods from another class to perform its work
  - Seems "envious" of the capabilities of the other class
- Problem:
  - Indicates abstraction fault
  - Could indicate the method was incorrectly placed in the wrong class
- Fix:
  - Move the method with feature envy to the class containing the most frequently used methods and data items

Data Class

- Data class
  - A class that has only class variables, getter/setter methods/properties, and nothing else
  - Is just acting as a data holder
- Problem
  - Typically, other classes have methods with feature envy
  - That is, there are usually other methods that primarily manipulate data in the data class
  - Indicates these methods should really be in the data class

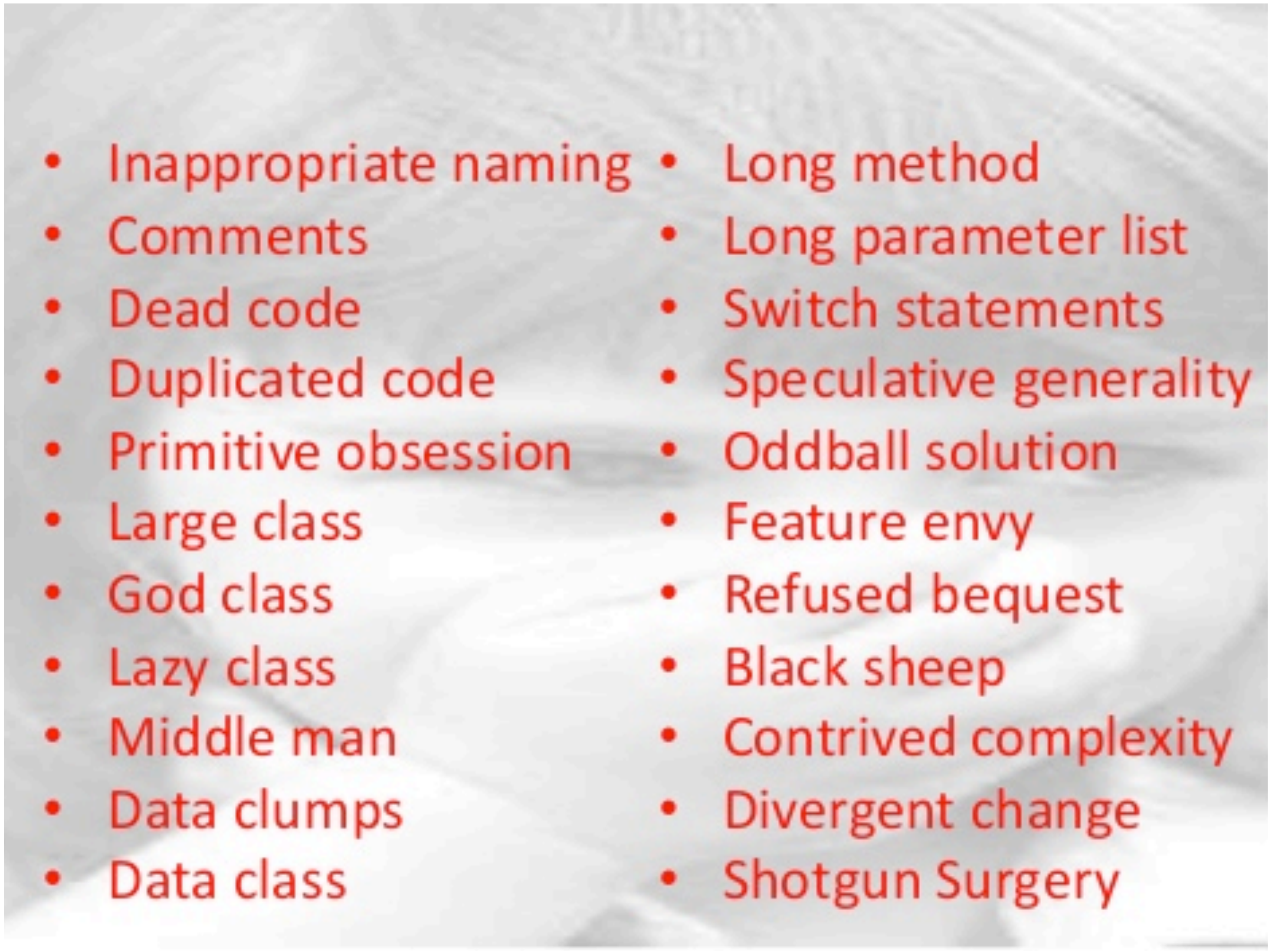
Inappropriate Intimacy

- Classes too coupled:
  Use each other methods and fields



- Solutions
  - separate out the common part
  - or merge, if you discovered “true love”

More Smells



- Inappropriate naming
- Comments
- Dead code
- Duplicated code
- Primitive obsession
- Large class
- God class
- Lazy class
- Middle man
- Data clumps
- Data class
- Long method
- Long parameter list
- Switch statements
- Speculative generality
- Oddball solution
- Feature envy
- Refused bequest
- Black sheep
- Contrived complexity
- Divergent change
- Shotgun Surgery

Reading Assignment for Next Week

- Coding Horror "Code Smells: https://blog.codinghorror.com/code-smells/
- A Taxonomy for Bad Code Smells: http://mikamantyla.eu/BadCodeSmellsTaxonomy.html

Quantitative Metrics & Static Analysis Tools

- Some quantitative metrics can be computed automatically

- There exist several static analysis tools for:
  - Code reviews
  - Defect detection (buffer overflow, null dereference, etc.)
  - Detection of security vulnerabilities
  - etc.

See for reference: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Automated Tools for Code Review

- **Codacy** – Static analysis and automated code reviews for Python, Ruby, PHP, Java, JavaScript, Scala, etc. Provides code metrics on code security, duplication, complexity, style, and coverage
- Codebeat − Swift, Obj-C, Ruby, Go, Python, Java, Kotlin, JavaScript, etc.
- Code Climate – JavaScript, Ruby, PHP, Obj-C, Java, etc..

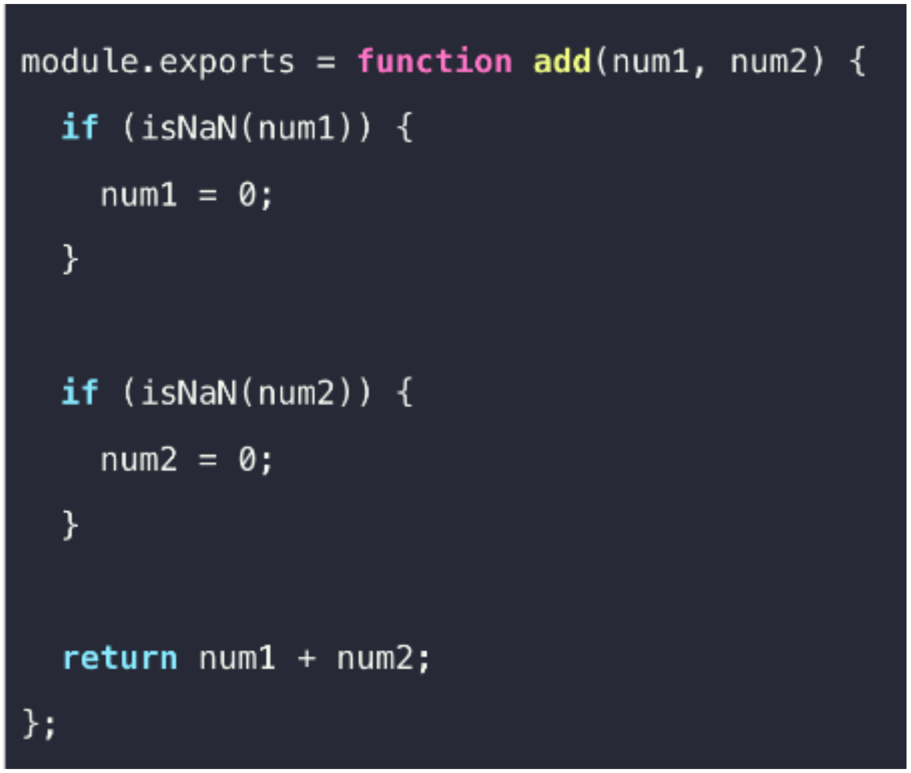
All integrate with public git repositories

Basic Metrics

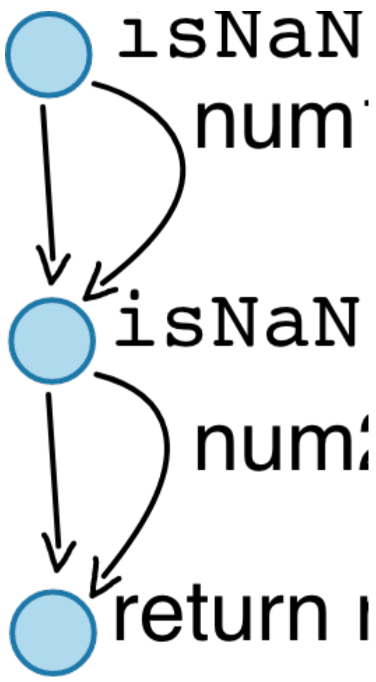
- Lines of code
- Number of methods
- Number of classes
- Size of each method
- etc.

NPath Complexity

- The number of acyclic execution paths in a method, module, class, or program
- Calculated using the control flow graph

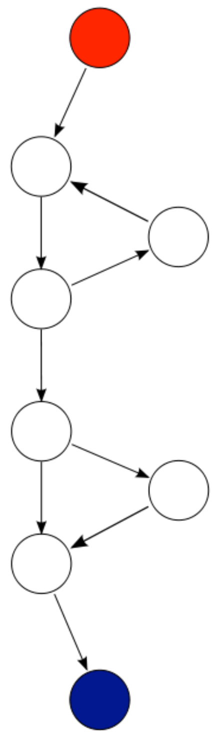


```
module.exports = function add(num1, num2) {
  if (isNaN(num1)) {
    num1 = 0;
  }

  if (isNaN(num2)) {
    num2 = 0;
  }

  return num1 + num2;
};
```



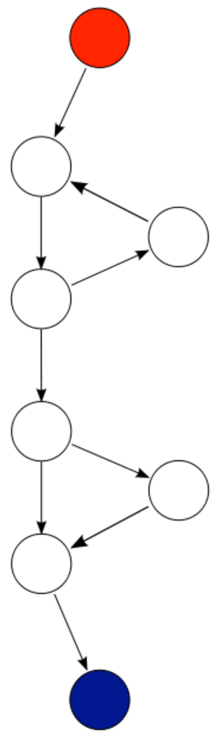
Cyclomatic complexity

- Measure of the number of linearly independent paths through a program's source code
  - a linearly independent path: a path through the application that introduces at least one new edge that is not included in any other linearly independent path.
- Computed using the control flow graph of a method
  - Can be extended to classes or other modules within a program.
- Rationale: many entangled conditional statements increase the complexity of a program



Cyclomatic complexity

- Calculations: $M = E - N + 2$
  - E = the number of edges of the graph.
  - N = the number of nodes of the graph.
- In this example:
  - E=9, N=8
  - M=9-8+2=3
- A standard threshold is 10
- (Implications for testing → to be discussed later)



ABC score

- The number of assignments (A), number of branches (B), and number of conditionals (C) in a program.

$$| < ABCScore > | = \sqrt{}(A^2 + B^2 + C^2)$$

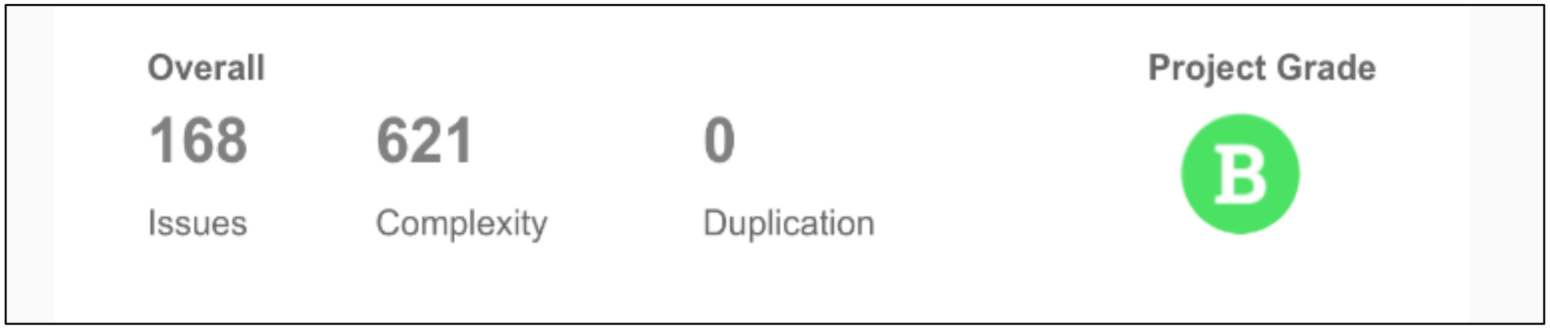
- Assignment: an explicit transfer of data into a variable
- Branch: an explicit forward program branch out of scope (goto, method call, etc.)
- Condition: a logical test (conditions in `if’, `while’, etc.)

ABC score

- Can be applied to individual methods, functions, classes, modules, or files within a program
- Rationale: help classify the program as ‘data intense’, ‘function intense’ and/or ‘logic intense’
- NIST (Natl. Inst. Stds. & Tech.): ≤20 /method

Codacy Example

- My project:

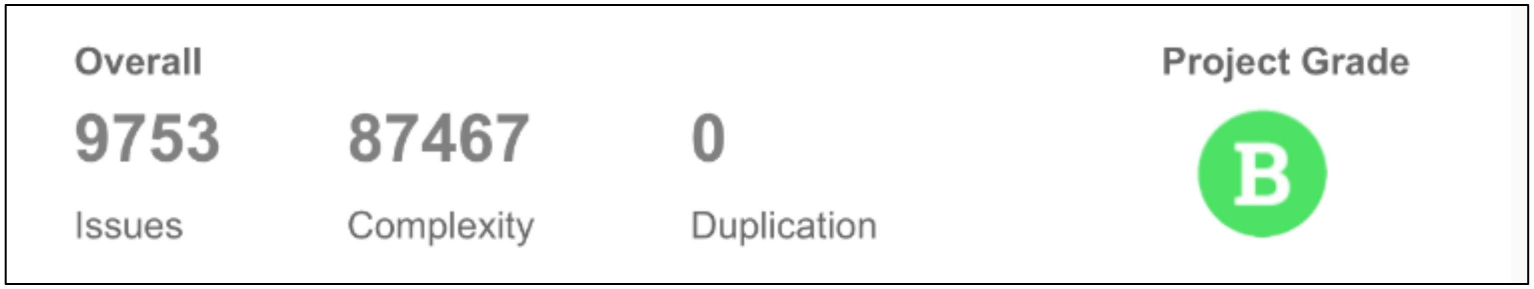


Overall

168	621	0
Issues	Complexity	Duplication

Project Grade

- Elastic search



Overall

9753	87467	0
Issues	Complexity	Duplication

Project Grade

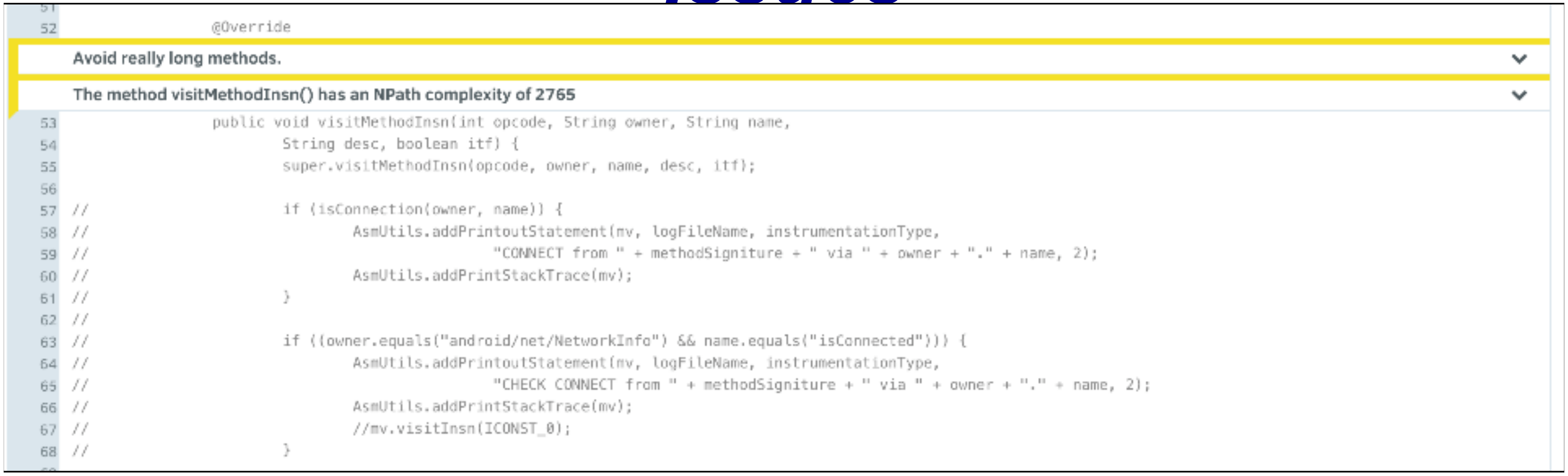
Codacy Grade

- Grade (A-F): a number of issues for each thousand lines of code (KLOC)

- Considered issues:
  - Code Complexity - Complex code to be simplified
  - Code Style - Rules to specify a coding standard
  - Unused Code - Dead code that could be removed
  - Security - Security issues that should be addressed
  - Documentation - Basic documentation checks
  - Performance - Issues that could have a performance impact in your code
  - Compatibility - Ensures backwards compatibility of your code
  - Error-prone - Code that can behave abnormally in production

- Steve McConnell, Code Complete: "Industry average is about 15 - 50 errors per 1000 lines of delivered code."

Issues

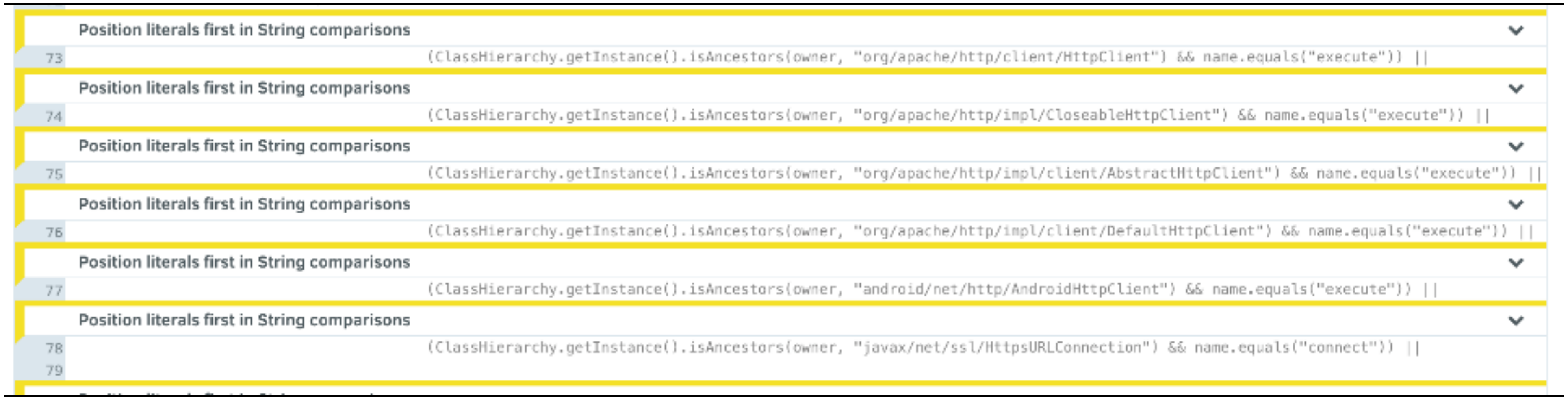


```
                @Override
```

Avoid really long methods.

The method visitMethodInsn() has an NPath complexity of 2765

```
                public void visitMethodInsn(int opcode, String owner, String name,
                        String desc, boolean itf) {
                        super.visitMethodInsn(opcode, owner, name, desc, itf);

//                      if (isConnection(owner, name)) {
//                              AsmUtils.addPrintoutStatement(mv, logFileName, instrumentationType,
//                                              "CONNECT from " + methodSigniture + " via " + owner + "." + name, 2);
//                              AsmUtils.addPrintStackTrace(mv);
//                      }
//
//                      if ((owner.equals("android/net/NetworkInfo") && name.equals("isConnected"))) {
//                              AsmUtils.addPrintoutStatement(mv, logFileName, instrumentationType,
//                                              "CHECK CONNECT from " + methodSigniture + " via " + owner + "." + name, 2);
//                              AsmUtils.addPrintStackTrace(mv);
//                              //mv.visitInsn(ICONST_0);
//                      }
```



Position literals first in String comparisons

```
(ClassHierarchy.getInstance().isAncestors(owner, "org/apache/http/client/HttpClient") && name.equals("execute")) ||
```

Position literals first in String comparisons

```
(ClassHierarchy.getInstance().isAncestors(owner, "org/apache/http/impl/CloseableHttpClient") && name.equals("execute")) ||
```

Position literals first in String comparisons

```
(ClassHierarchy.getInstance().isAncestors(owner, "org/apache/http/impl/client/AbstractHttpClient") && name.equals("execute")) ||
```

Position literals first in String comparisons

```
(ClassHierarchy.getInstance().isAncestors(owner, "org/apache/http/impl/client/DefaultHttpClient") && name.equals("execute")) ||
```

Position literals first in String comparisons

```
(ClassHierarchy.getInstance().isAncestors(owner, "android/net/http/AndroidHttpClient") && name.equals("execute")) ||
```

Position literals first in String comparisons

```
(ClassHierarchy.getInstance().isAncestors(owner, "javax/net/ssl/HttpsURLConnection") && name.equals("connect")) ||
```

Summary

- Code reviews improve code quality, teamwork, knowledge and skills
- Important and performed in industry
- Typically integrated into the product workflow
- Look for
  - Bugs, security vulnerabilities, performance issues, maintainability issues, readability, comments, code smells, adherence to coding standards and best practices
- Some problems can be detected by automated tools

Expectations for the Code Review Deliverable (W8 – October 29, M4)

1. Automated step:
   - Run an automated code review tool on your project
   - Fix the main issues
   - Run the tool again
   - Attach the reports before and after the fixes, and an explanation about the main issues you fixed
   - Attach an explanation about the main remaining issues

(cont.)

Expectations for the Code Review Deliverable (W8 – October 29, M4)

2. Manual step:
   - Review the code of your customer team
   - Attach a report describing one **major** issue you identified and how this issue was fixed (not about comments!)

3. Be ready to dig into your code with your customer and TA during the meeting