W4156

Good Code, Bad Code, Ugly Code and Refactoring

Recap

We are up and away: we can define requirements, write and test code

However,

1. We want to write *good* code. What is ‘good code’?

2. We briefly mentioned ***‘refactoring’*** during last lecture where we said we could *change* code to be ‘better’ if we didn’t like it. How do we ‘refactor?

Agenda

- ❏ Why does good code matter?
- ❏ What is Good Code?
- ❏ Smells
- ❏ Refactoring

Note: we split ‘good’ into ‘good code’ today and ‘good designs’ in a later lecture

Good Code as a NFR

We have actually already met ‘good code’ within the non-functional requirements. There were a set of internal quality attributes such as maintainability and testability ……

Ok - it matters but what is 'good code'?

Possible Definitions	Discussion
Meets customer needs / External Quality	- Mandatory - but does not include other users of the code (developers) - Can I have bad code that still functions?
Internal Quality (maintainability, modifiability, reusability, testability, etc)	- Yes - some of these are *consequences* - Others equally hard to measure quantitatively
Well formatted / 'Beautified'	- Yes/ one aspect but neglects many other issues?
Well designed	- Yes - again - how do we define 'well designed' - Can I have well designed but poor code?
Aesthetics	- Yes - but a bit hand wavy

Readability

*Communication with other **people** is the motivation behind the quest for the Holy Grail of self-documenting code.*

The computer doesn't care whether your code is readable. It's better at reading binary machine instructions than it is at reading high-level-language statements. You write readable code because it helps other people to read your code. Readability has a positive effect on all these aspects of a program:

- *Comprehensibility, Reviewability, Error rate, Debugging, Modifiability*
- *Development time—a consequence of all of the above*
- *External quality—a consequence of all of the above*

[McConnell Code Complete]

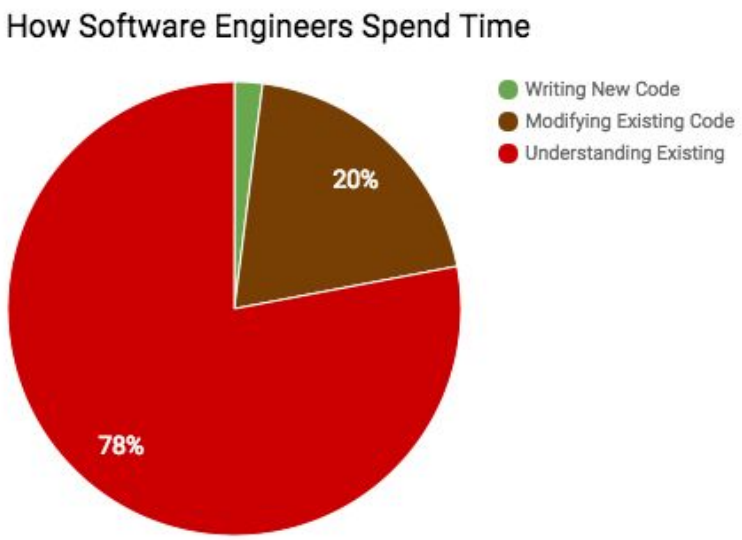
I love this definition.
The many qualities we want are *enabled/manifestations of* readable code

Readability

"Indeed, the ratio of time spent reading versus writing is well over **10 to 1**. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write." **Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship**

The *majority* of code of software is *maintenance*.

Systems exist for 5, 10, 15, 20 years



Peter Haal

Why Good Code Matters

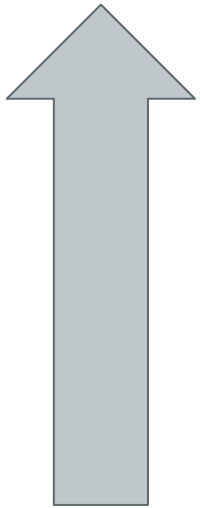
Learning to program we write new code, hand it in for grading then walk on.

In Industry systems exist for 1, 2, 5, 10, 15* years. Each month we add new features or change how existing functionality works as the business evolves.

Question: What is the ratio of reading code to writing code?

* one of the engineers on my floor runs a system written in small-talk started in the 90s. Still heavily used and critical. Although it is still changing the code has been read 100/1000 times more than written

Readability



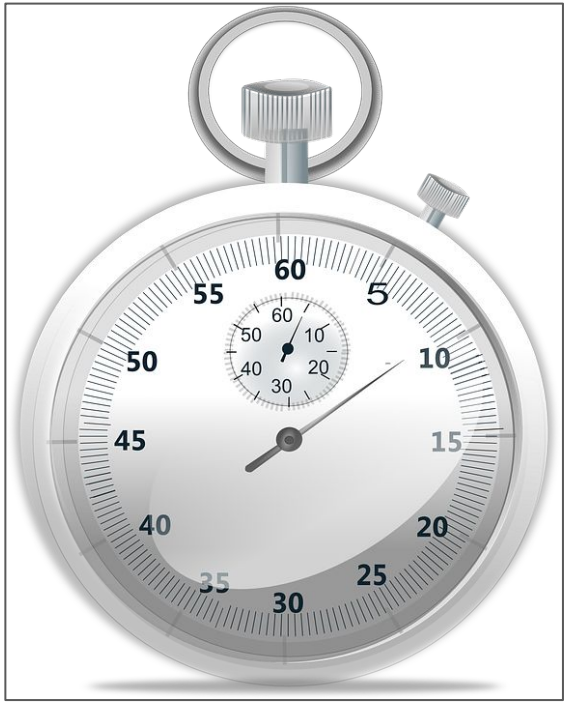
- Between Classes
- Within Classes
- Coding Conventions

Coding Conventions

A basic foundation of readability is coding conventions

1. Naming conventions (variable, method, class, module)
2. Indentation & Line Length
3. Comments
4. Language Features
5. Project Layout
6. Other conventions

Readability: Can I convince you?



As quick as possible

1. Analyze the code
2. Which unit tests pass?
3. Be ready to answer
   A) Neither
   B) First
   C) Second
   D) Both

Which Tests Pass?

```
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);
    }

}
```

Which pass:

A) Neither
B) First
C) Second
D) Both

Answer

```
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);
    }

}
```

Answer B)
First one passes
Second one fails

- x *= 2 is not guarded by the 'if'
- despite indentation (not py)
- 3 *= 2 = 6
- 6 + 5 = 11

(if(increment) { x++; x*=2 } does pass)

Sorry

Hopefully x% of the class got this wrong. (Sorry)

But think about yourself as a software engineer on 10-50k LOC, in an IDE, 20 files open, making a change to how the business/system works, scanning code, 80mg Caffeine.

Code should be ***hard to misunderstand*** versus ***understandable with effort***

This Happened

Example was based off a production incident:

1. Developer wrote part of the code code without braces

2. Change to add the second line
   a. The second developer added it indented and didn't format code
   b. The lack of braces introduced a bug
   c. Not caught in testing

3. When debugging the code line by line we *still* didn't see the bug
   (we thought there was a problem between the code in our IDE and the binary we were debugging)

Readability and Coding Conventions

Most teams/companies have coding conventions which dictate naming convention, layout, language features, etc.

It is not pedantry. It establishes common basics to enable focus on business logic and intent.

Tool Support:

- Most IDEs support coding standard/convention within IDE
- Linters / static analysis tools also cover coding standards (and others)
- IDE can also auto-format before commit
- Some teams put lint check build or CI (next lecture)

Coding Convention

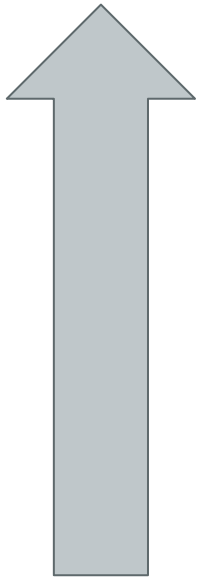
Google Java Coding Standards

4.1.1 Braces are used where optional

Braces are used with `if` , `else` , `for` , `do` and `while` statements even when the body is empty or contains only a single statement.

https://www.python.org/dev/peps/pep-0008/

Readability



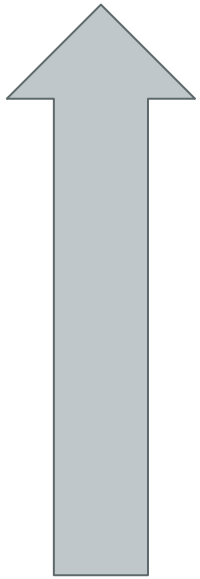
- ❑ Between Classes
- ❑ Within Classes
- ✓ Coding Conventions

Code Smells

“A code smell is a surface *indication* that usually corresponds to a deeper problem in the system”
[Fowler]

- Name comes from the fact they are *indicators* worthy of investigation.
- You may investigate but choose not to deal with (or tradeoff not worth it)

Readability



- ❑ Between Classes
- ✓ Within Classes
- ✓ Coding Conventions

Smells: Within Classes

Smell	Description
Excessively Long Methods	Self explanatory (google C++ convention is <= 40 lines)
Excessive Parameter List	**def** my_function(numberPeople, minAcceptable, maxAcceptable, nettingAllowed, removeDups, ….)
Duplicated Code	Duplication of code in other functions or branches within functions (often 4 line duplication)
Conditional Complexity	**if** cond1 && cond2 && foo < bar -1 && etc **then**
Large Class (a.k.a. God Class)	Single class has grown too large and controls other classes in system
Type Embedded in Name	**def** addIntegers
Inconsistent Naming	Function names are not consistent / no linguistic conventions open() and stop() are paired
Dead Code	**if**(false) then //dead endif
Speculative Generality	Attempting to design the system to be too generic (where requirements don't exist yet)

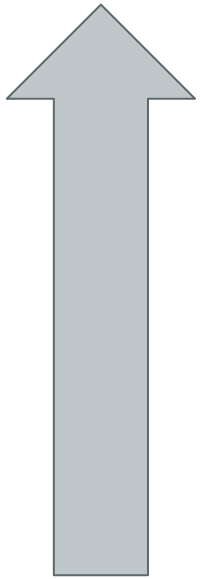
Smell: Duplication of Code

- Remember code is *not an asset*. More code == more code to manage
- Duplicate code is dangerous because a rule is implemented in more than one place
  - When you want to change the rule you need to find all copies and change
  - If you miss a copy you may introduce a bug
- Duplication is often a result of copy paste of function
  (Previous boss banned any copy paste while coding - I laughed the first time he said it but he then explained, “Why are you copy pasting? If you are copy pasting large chunks you have done something wrong. If chunks are small then type it out while you think it out”)
- Think about the *functionality*. Don’t copy-paste. Extract common functionality.
- Automated tools detect duplication
  (I have seen legacy system with 60% of code is duplicated to some extent. They copy pasted whole classes TicketEntryWindow.java and TicketEntryWindowAsia.java)

Smells: Between Classes

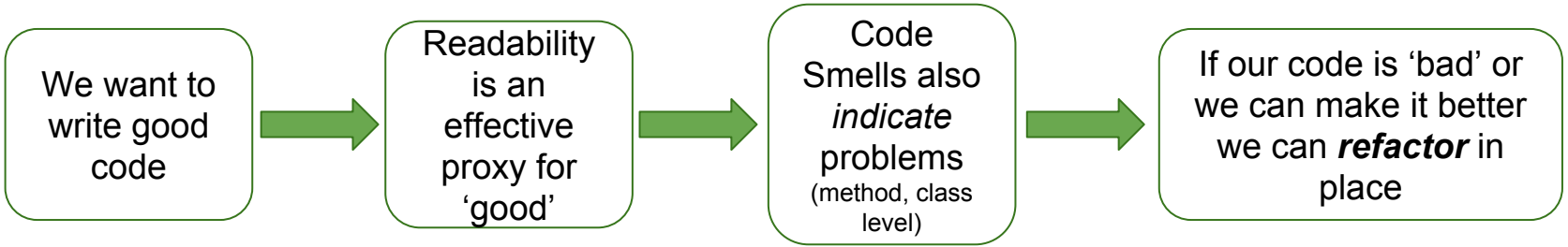
Smell	Description
Primitive Obsession	Use of primitives rather than thinking OO and recognizing objects. Would have day, month, year as primitives rather than recognizing concept of ‘date’ and encapsulating behavior and data
Data Class	Classes that contain only data and no behavior (ok sometimes) but often seen when C developers learn Java (Objects replace structs)
Inappropriate intimacy	One class accessing the internals of another class which is ‘tentacle’ into internal design
Indecent exposure	Classes that unnecessarily expose inner workings (fragile to change and inflexible)
Feature envy	Methods that make such extensive use of another class maybe they belong there?
Shotgun Surgery	Changes in a single class that require cascading changes to other classes
Incomplete Library Class	Method missing from library but can’t be added. Gets ‘tagged’ on somewhere

Readability



- ✓ Between Classes
- ✓ Within Classes
- ✓ Coding Conventions

Recap / Thought Progression



Refactoring

We have identified bad code or smells that turn out to be bad code. Now we want to fix them …..

Refactoring: *“The process of changing a software system in such a way that it does not alter the behavior of the code yet improves its internal structure”* [Fowler]

- You have already been refactoring every time you ‘cleaned up code’
- “Refactoring” book is very mechanical inventory of tactics
- “Boy Scout Principle”: Leave code better than you found it

Refactoring

Smell	Refactoring Solutions
Duplicate Code	Extract Method Extract Class Pull Up Method Form Template Method
Inappropriate Intimacy	Move Method Move Field Extract Method
Long Parameter List	Replace Param w/ Method Introduce Parameter Object Preserve Whole Object
…….	…….
…….	……..

Long set of
- symptoms
- procedures

Read to build your ‘toolkit’

You will ‘see’ it in code

Refactoring: Association with <?>

We have code that functionally works but we want to significantly refactor.

Q1: How do we know we didn't break the code during refactor?

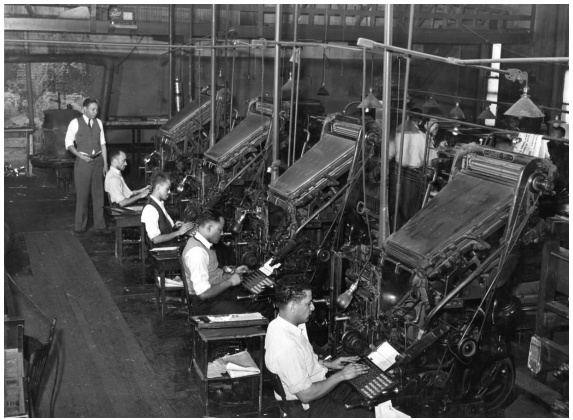
Q2: What happens to codebase over time if we are reluctant to refactor?

Q3: What does this tell us?

Final Thought

McConnell in code complete has an analogy for software.
Is software ‘printed’ or ‘accreted’?

Printing Press



https://en.wikipedia.org/wiki/The_Chicago_Defender

Pearl Formation in Oyster

http://www.livescience.com/32856-what-makes-a-black-pearl.html

Project

1. One of the homeworks is to integrate a linter into the build process (enforces)

2. Whichever IDE you use select one of the coding standards

3. My personal recommendation is to make coding standard violations *errors* rather than *warnings* while learning

Pop Quiz

Question	Answer
One measure of ‘good code’ is <>?	
Code smells are <potential/definite> indicators of problems?	
When we choose to change the internal structure of code we are <>?	
When we refactor we <keep/change> the functional behavior?	

Reading

Reading	Optionality
Google Code Conventions [Java or Python]	Required
Code Smells	Required
Code Smells (I do not require encyclopedic knowledge of all refactorings but you should know 5-10 smells and associated refactoring)	Required
Refactoring	Required
Refactoring Misuse	Required
Beginning S.E. pg. 155-169	Required