CSC290 Communication Skills for Computer Scientists

Lisa Zhang

Lecture 5; Feb 4, 2019

Announcements

- **Design Review Presentation Slides** Due Wednesday midnight
- **Blog Post 3** Due Sunday 9pm

Group Project

- Everyone must contribute to every portion of the project
- If there are students in your group that are not contributing, please let me know
- If you still did not get in touch with your group, let me know

How to break up the work?

Think carefully about what code is necessary!

For example, in a tic-tac-toe game, we need code to:

- Decide on how to represent the board
- Render "X" and "O" on the board
- Capture mouse clicks
- Determine which of the 9 squares the mouse click belongs to
- Restart game
- Determine whether there is a winning move
- Determine moves of computer player (?)

If you can't break up the work, it means you haven't designed and communicated enough!

Design Review Presentation

Your presentation should be clean (not flashy)

Focus on content and good delivery

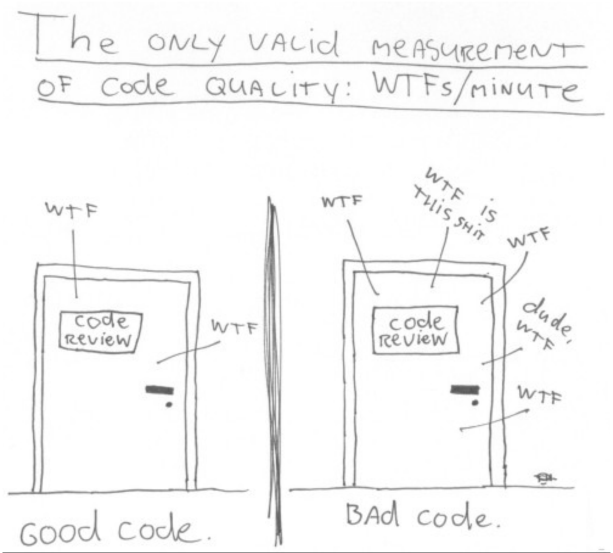
Practice, practice, practice!

Today

- Writing clean code
- Writing commit messages
- Reviewing Code
- Structuring your GitHub repo

Clean Code

It's hard to define "clean code"



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Worksheet

- There are four versions of a function.
- Which version has the “cleanest” code?
- Each version has some clear issues – what are they?

Work as a group.

Clean Code...

... does not stray from a reader's expectations. It ...

- Follows the appropriate **coding convention**
- Uses **meaningful names**.
- Contains little/**no duplication** of code
- Is **testable**
- Explains itself, and is **well documented**

We'll talk about each of those items, in turn.

Following appropriate conventions

Different organizations will have different conventions. Different projects may have different conventions.

- https://google.github.io/styleguide/javaguide.html
- https://github.com/google/styleguide/blob/gh-pages/pyguide.md

Larger organizations will have more formal conventions.

- Use tools to automatically check whether your code follow conventions

Python Conventions

```
# Version D:
def hasVowel(word):
    """Return whether word contains a lowercase vowel."""
    for vowel in VOWELS:
        if vowel in word: return True
    return False
```

Python Conventions

```
# Version D:
def hasVowel(word):
    """Return whether word contains a lowercase vowel."""
    for vowel in VOWELS:
        if vowel in word: return True
    return False

# Version C:
def index_of_first_vowel(word, vowels = "aeiou"):
    ...
    i = 0
    while word[i] not in vowels:
            i = i + 1
    return i
```

Naming

Are the names used in verisons A-D good?

Function Names

- `f`: bad name, does not say anything
- `translate_to_piglatin`: good name, starts with a verb
- `piglatin`: okay name
- `english_to_piglatin`: descriptive name

Helper Function Names

- `index_of_first_vowel`: descriptive, but a bit long
- `first_vowel_index`: just as descriptive, and shorter
- `hasVowel`: good name (other than breaking convention)

Functions that Return a Boolean

- Functions that return a boolean often starts with “has” or “is”
- These are verbs used to ask a yes/no question

Variable Names: Version A

- `w`: bad name, hard to search
- `i`: borderline, still hard to search

Naming Consideration

- Does the name fully and accurately describe what the variable represents?
- Name should have the right level of specificity
  - The larger the scope, the more specific the name
  - Reserve single characters names for short loops only
  - Use `i`, `j`, `k` for integer loop indicies (why not `l`?)
- Name should be easy to **search** (e.g. global search and replace)

Variable Names: Version B

- `word`: good name
- `i`: borderline, hard to search

Variable Names: Version C

- `vowels`: good name
- `i` in `index_of_first_vowel`: okay
- `i` in `piglatin`: not okay!

Variable Names: Version D

- `VOWELS`: reasonable name for a constant
- `vowel`: good name
- `str`: very bad name, because `str` means something in Python!

Avoid common, meaningless names

- flag
- status
- data
- variable
- tmp
- foo, bar, etc.

Duplication

```
# Version A has a lot of duplication:

if (w[0] == "a" or w[0] == "e" or w[0] == "i" or
                   w[0] == "o" or w[0] == "u"):

# Copy & paste introduces error

while i < len(w) and (w[i] != "a" and w[i] != "e" and
                      w[i] != "i" and
                      w[i] != "o" and w[i] == "u"):
```

Did you notice it?

Abstraction is Better

Instead of:

```
while i < len(w) and (w[i] != "a" and w[i] != "e" and
                      w[i] != "i" and w[i] != "o" and
                      w[i] == "u"):
```

Write:

```
while i < len(w) and all(w[i] != v for v in "aeiou"):
```

Or write a helper function as in Version C & D.

Reduce code repetition

```python
def make_egg():
    egg = take_out("egg")
    cooked_egg = cook(egg)
    plated_egg = plate(cooked_egg)
    return plated_egg

def make_ham():
    ham = take_out("ham")
    cooked_ham = cook(ham)
    plated_ham = plate(cooked_ham)
    return plated_ham
```

Don’t rewrite the builtins

```python
def round(num):
    frac = num % 1
    if frac >= 0.5:
        return (num - frac + 1)
    return (num - frac)
```

Don’t re-write code that other people in your project have already written.

Reduce nesting (exit early)

```
def piglatin(word):
    i = index_of_first_vowel(word)
    if i != len(word): # has vowel
        if i == 0:
            return word + "way"
        else:
            return word[i:] + word[:i] + "ay"
    else:
        return word
```

versus:

```
def piglatin(word):
    i = index_of_first_vowel(word)
    if i == 0: # begins with vowel
        return word + "way"
    if i == len(word): # no vowel
        return word
    return word[i:] + word[:i] + "ay"
```

Writing Testable Code

- **Unit test** verifies the behaviour of a small part of your code
  - Easy to write and run
- **Integration test** verifies that components interacts well with each other
  - Difficult to write and run

So what makes code easier to test?

Writing Testable Code

- **Unit test** verifies the behaviour of a small part of your code
  - Easy to write and run
- **Integration test** verifies that components interacts well with each other
  - Difficult to write and run

So what makes code easier to test?

- Each function should do one thing only.
- Isolate functions that interact with external systems (file system, database)
- Prefer **pure** functions
  - Function whose output is deterministic given its arguments

Code that is difficult to unit-test

```
def read_file_and_compute_total(file):
    total = 0
    for line in open(file):
        item, price = line.split(",")
        price = float(price)
        if item not in FOOD_LIST:
            total += price * 1.13
        else:
            total += price
    return total
```

Code that is easier to unit-test

```python
def read_product_price(file):
    products = []
    for line in open(file):
        item, price = line.split(",")
        products.append(item, float(price))
    return products

def compute_total(item, price):
    if item in FOOD_LIST:
        return price
    return price * 1.13

def read_file_and_compute_total(file):
    return sum([compute_total(item, price)
                for (item, price)
                in read_product_price(file)])
```

Comments

- Comments should **explain why** the code is what it is.
- Comments should never repeat the code.
- Ideally, the code will make sense without any comments.

Comments that repeat the code are bad!

```
def translate_to_piglatin(word):
    if word[0] in "aeiou":  # first character is a vowel
        return word + "way" # return the word + "way"
```

These are useless comments!

Comments that explain the code

```
def piglatin(word):
    i = index_of_first_vowel(word)
    if i == 0: # begins with vowel
        return word + "way"
    if i == len(word): # no vowel
        return word
    return word[i:] + word[:i] + "ay"
```

These are better comments.

Comments that mark the code

```
VOWELS = "aeiou" #TODO: include y?
```

. . . but clean these up, ideally before committing.

Other comments:

- Block comments to lay out code
- Comments that describe the code's intent
- Comments that summarizes a chunk of code
- Information like copyright notices, references, etc.