Reviewing your Program

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 2.5

Introduction

- A program is like an essay or term paper.
- You wouldn't turn in a term paper without proofreading it for typos, checking for spelling, and generally making improvements.
- So you shouldn't turn in your program without checking it over and seeing if it can be improved.
- Our goal is to write *beautiful* programs– programs which are easily read and understood by others.

The Program Review Recipe

- On the next slide is a list of things to check in your program.
- Then we'll go through each of the items in more detail.

The Program Review Recipe

1. Do all the tests pass?
2. Are the contracts accurate?
3. Are the purpose statements and interpretations clear and accurate?
4. Are there ugly pieces of code that should be broken out into their own functions?
5. Are there pieces of code that are duplicated (or almost duplicated) and should be made into independent functions?

1. Do all the tests pass?

- Of course you wouldn't turn in a program if some your tests failed, but...
  - Did you achieve 100% code coverage?
  - Are your tests readable?
  - Are there comments or error messages so that the TA will be able to see the purpose of each test?
  - Are the tests written so that the TA can easily see that each test actually tests what it claims to test?

2. Are the contracts accurate?

- Are all the type names spelled correctly and consistently?
- Do the contract and function definition agree on the number and types of the arguments, and on the type of the result?
  - Maybe you discovered along the way that you had to change some of the arguments. Make sure that you've changed the contract to match.

3. Are the purpose statements clear and accurate?

- Each purpose statement is a set of short noun phrases describing *what value* the function is supposed to return.
  - They generally take the form GIVEN/RETURNS, where each of these keywords is followed by a short noun phrase.
  - The RETURNS clause must at least mention every one of the function arguments.

Review: what makes a good purpose statement?

- It gives more information than just the contract. For example

```
GIVEN: a Number and a Boolean
RETURNS: a Number
```

  is not a good purpose statement

- It is *specific.* Ideally, a reader should be able to figure out what a function returns just by reading the purpose statement
  - perhaps along with examples, other documentation, etc.
  - but WITHOUT reading the code!

Writing good purpose statements can be hard.

- Sometimes the arguments are the components of some thing, rather than the thing itself.
  - Here's a useful example:

    GIVEN: the x-coordinate, y-coordinate, and direction of some robot

    RETURNS: the robot moved forward by 10 pixels.
  - You may find this to be a good pattern in many examples.

Spelling Counts

- Spelling Counts. Always. Everywhere.
- Spelling errors show a lack of professionalism.
- They tell the reader that you are SLOPPY and you DON'T CARE.
- If you have a spelling error in your resume or cover letter, you will NOT get the job.

Reviewing your purpose statements (cont'd)

- Check your purpose statements for spelling.
- Make sure you are consistent and correct about English singular and plural.
- Try to use English articles, like "a" and "the", correctly.
  - this may be difficult if your first language does not have these.
- If English is not your first language, go find the best English-speaker you know and get help.
  - We are officially allowing this.

The same things hold for data definitions

- Go back and review your data definitions, too.
- Check over the English in your interpretations.
- Check the other items that you were supposed to review—
  - Here's the recipe, from Lesson 1.6

Reviewing a Data Design

1. Is the interpretation clear and unambiguous?
2. Can you represent all the information you need for your program?
3. Do you *need* all of the data in your representation?
4. Does every combination of values make sense? If not, document the meaningful combinations with a WHERE clause.

4. Are there ugly pieces of code that should be broken out into their own functions?

- Remember: one function = one task
- If there is complicated stuff in your function definition, replace it by a call to a help function.
- That way you can document what that stuff is supposed to do, and test it.

5. Are there pieces of code that are duplicated (or almost duplicated) and should be made into independent functions?

Principle: "Don't Repeat Yourself"

- We strive for no duplicated code.
- If there's a subtask that gets done several times, turn it into a function that you can document and test.
- We'll have much more to say about this in coming weeks.

Summary

- Our goal is to write not just working programs, but *beautiful* programs.
- We've given you a list of things to check *before* you turn in your program.
- Fixing these things will make your program more pleasant for the reader, whether that is your TA, your boss, or your successor at your job.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson