book-99-bottles-of-oop

Estifanos Beyene

May 29, 2022

Contents

1 99 Bottles of OOP

2 pseudo-code

- pseudo-code is speculative, and it has already been used to reveal the code smells that would have arisen had you added a new conditional to generate other counting-down songs. That pseudocode quickly and painlessly revealed a shortcut to a better path.

3 wishful-thinking

- Coding by wishful thinking allows you to sketch software design ideas economically, with a low level of commitment. In other words, you can guess, unburdened by penalties for being wrong. This approach to writing code can feel unruly and indulgent, but it's a bona fide, efficient, and often elegant technique for making progress

4 oop-vs-procedural

- OOP: break code up which makes each cohesive piece each to change, although it can make the obscure what is really happening.
- procedural: simple ones are easy to understand, but the complicated one's are costly.
  - If you plan to adapt and grow, OOP is the way to go. tdd-growth

5 data-clump

- If you start noticing data clumps, it is likely that there is a concept those have in common, and it is imperative that you represent that deeper concept and name it.
- If you find yourself using empty lines to create separation within a scoped entity, it is likely that those things are separate responsibilities that needed to be moved to a new one.

6 removing-argument-using-small-changes

- Alter the method body to replace occurrences of the argument with references to the - property of the same name.
- Change every sender of the message to remove the parameter
- Confirm that you've updated every sender by adding a temporary guard clause to the - method body.
- Finally, delete the argument and guard clause from the method definition

7 abstract-as-objects

- The true power of OOP lies in the fact that besides creating objects that represent the real, you can represent the abstract as well.

8 characteristics-of-code

- Questions to ask to grasp the characteristics of code:
  - questions that look at the class as a whole and expose common qualities within it:
    1. Do any methods have the same shape?
       * You can easily identify same-shaped methods by doing the Squint Test
         · The Flocking Rules would return code that is shaped similar.
         · Differences increases the difficulty of future refactoring.
    2. Do any methods take an argument of the same name?
    3. Do arguments of the same name always mean the same thing?
    4. If you were going to break this class into two pieces, where is the dividing line?
  - question that look specific methods and expose common qualities among them:
    * Do the tests in the conditionals have anything in common?
    * How many branches do the conditionals have?
    * Do the methods contain any code other than the conditional?

    - Does each method depend more on the argument that got passed, or on the class as a whole?

9 creating-abstractions PROGRAMMINGINSTRUCTIONS

1. Describe its responsibility as you understand it at this moment,
2. Choose a good-name which reflects that responsibility.
3. Define a method for the concept.
4. Alter it to return one of the differences.
5. Replace that difference with a message send.
6. Add the number argument to the new method, with appropriate default.
7. Implement the conditional.
8. Pass the number argument from the current sender.
9. Send the message from the other branch, this time including the number argument.
10. Clean up.

10 consistent-code

- The rules lead to consistent code, and consistency matters deeply. It makes code easy to understand and enables future refactoring.
- Inconsistencies do not guarantee that something is wrong, but they should certainly motivate you to think more deeply about the underlying abstraction.

11 variants

- Underlying each concrete variant is a generalized abstraction. If you could find this abstraction, you could use it to reduce the duplication.

- DRYing out sameness has some value, but DRYing out difference has more.
- It might help to first decide what is not a difference among similar things.

12 changing-code

- Changing a code comes after make the code open for change. openClosedFlowchartPicture
- Although hard problems exist, they are usually the sum of small problems that are easier to fix. So to improve code, fix the easy problems within it. And when there is any doubt of the path, do not bother too much, search for code smells and fix them using the proper rules like the flocking-rules for identifying abstraction.
  - Process works, and that encountering errors while following it suggests that a closer look at the code is in order.
- Learning the art of transforming code one line at a time, while keeping the tests passing at every point, lets you undertake enormous refactoring efficiently. A small problem is a good place to practice this technique, in preparation for later tackling bigger ones.
- Trying testing code, it will let you know if it is coupled.

13 good-tests

- Good tests are only testing the interface and not the implementation. The first point to writing good tests is to avoid DRY and abstractions, both of whom lead to coupled-tests

14 coupled-tests

- If the test suite has coupled tests, changes and additions in the code would result in unrelated tests failing.
- You might think of using logic from code directly, but that would result in a test makes sense only for the current implementation.

15 fake-it-test TDD

- In this method, you write a test and you write the smallest amount of code of make that test pass, and repeat for the entire process.
- It will help reveal the correct abstraction/implementation. purpose-of-abstractions

16 three-ways-to-pass-tests TDD

- The three ways to pass tests are:
  - fake-it-test
  - triangulate-test
  - obvious-implementation-test
- In this method, instead of writing a test and the code to make it pass, you write multiple tests at once, and write code that make all the tests pass and in the process finding an abstraction that makes encloses all of them.

17 purpose-of-abstractions

- The purpose of abstractions is to make code easier to understand. If the change makes the code more complicated, it is most likely that it isn't the right abstraction to use.
- You might have been seeking-abstraction and maybe you should ask yourself making-abstractions-questions

18 making-abstractions-questions

- Is there am immediate benefit of adding the abstractions now?
  - If the abstraction added now provides little immediate benefit, it is unnecessary and the addition should be prolonged.
- When will I arrive at a necessary abstraction?
  1. Gather as much information from the problem
  2. Write tests for the gathered information

(a) duplicate tests that isolate independent examples are acceptable because they help reveal abstractions

(b) duplication that repeat already present examples masks the responsibility that the component has and makes it harder to understand

19 first-test **TDD**

- The first test doesn't have special significance. Just aim to test something small that you thoroughly understand.

20 over-planning

- While planning is important, don't overthink and instead, know that as you write more code and tests, the clearer the solutions to the problems become. more they would reveal about the problem.
  - link: first-test
- Over-planing can lead to wrong-level-abstractions, or as a result of have being guarded by our intentions, come to the wrong implementations when using obvious-implementation-test

21 obvious-implementation-test

- In this method, you write a test, and pick the code that will make that test pass no matter how many lines of code it is. This should be used only for small steps, as it could lead to the wrong abstractions if you start seeking-abstraction

22 tdd-cycle

- TDD Cycle has 3 steps:
  1. write a test
  2. write the smallest amount of code to make it pass
  3. write more tests that break code, and repeat until you fulfill requirements

23 tdd-growth

- It is interesting to note that as you write more tests, the code would have to be able to return the correct value for each, meaning it would need to cover larger and larger abstractions, which in turn makes the tests more generic.

24 tdd-process

- The process behind TDD is writing tests, passing tests by writing code and breaking tests by writing more tests. Just as its name says, the tests are what drive the program, and until all the tests have been written, the program isn't complete.
- tdd-growth
- tdd-cycle

25 judging-code-questions

- How difficult was it to write?
- How hard is it to understand?
- How expensive will it be to change?
- It also helps to write code-for-reader

26 good-name

- The general rule is that the name of a method should be one level of abstraction higher than the thing itself.
- The name of a class doesn't go with the same rule as method, but you should not limit to thinking it can only be a real object.
- The effort you put into selecting good names right now pays off by making it easier to recognize perfect names and liskov-principle

27 liskov-principle

- The liskov rule says that the objects should be what they promise they are. The sender should always fulfill its responsibility to the receiver.
  - You will be forced to write verbose code for precaution. ignoring-liskov
- If you choose a good-name

28 polymorphism-responsibility

- Polymorphism creates multiple classes that play a common role without requiring more from the message sender. They play the same role, liskov-principle

29 ignoring-liskov

- The likely reason you code isn't returning what it promised is because there is an insufferable use of polymorphism-responsibility

30 code-for-reader

- Although short code is impressive, it is useless if it is hard to read. With the reader in mind, make it a point to write with consistent code and clearly defined logic.
  - You should try to be aware of separating-within-scope

31 separating-within-scope

- If you find yourself using empty lines to create separation within a scoped entity, it is likely that those things are separate responsibilities that needed to be moved to a new one.

32 flocking-rules

- To identify abstractions:
  - Select the things that are most alike.

  - Find the smallest difference between them.
    * Make the simplest change to remove that difference:
      · parse the new code
      · parse and execute it
      · parse, execute and use its result
      · delete unused code

33 seeking-abstraction ABSTRACTION:PROGRAMMINGRULES

- Actively seeking abstractions, rather than using flocking-rules will lead to a wrong choice and after other aspects like wrong-level-abstractions

34 wrong-level-abstractions

- DRY at the wrong level will result in code that is hard to alter or extend because there are abstractions within the code that are yet to become revealed. Another sign that abstractions are not been revealed is the existence of cause-of-duplication

35 cause-of-duplication

- Duplication is most likely the result of abstractions that haven't been identified yet. At the right level of abstraction, DRY is used to eliminate duplication and reduce the overall size of the code.