Module 9
Legacy Code & Refactoring

CS 169A: Software Engineering

1 Overview

As your career in software engineering blossoms, you will find yourself reading and working with *legacy code* more often than writing code from scratch. Understanding, maintaining, and developing legacy code are important skills. This module discusses some of the guiding principles and techniques regarding how to navigate and improve a legacy codebase!

Agile techniques we've already learned have a place here! As covered in ESaaS 9.1, understanding legacy code can be briefly broken down into four steps:

1. Identify change points (Places to make changes in legacy code).
2. Add characterization tests to capture how code works if necessary.
3. Determine if change points need refactoring.
4. Create new regression tests for new code as described in modules 7, 8.

2 Characterization Tests

Let's assume that you are part of a software development team that's just been awarded by Hasbro to create an online version of the classic Monopoly Board Game. Congratulations! The first thing your team does is review the legacy codebase written by a previously contracted development team (unfortunately, this was because they were a group of Stanford students who employed P&D - this joke is not important to the problem).

You notice a disappointing lack of tests left by the previous team. Your team proposes several new characterization tests to make up for this deficit. For each of these tests, determine whether 1. The test qualifies as a characterization test (Yes/No) and 2. Justify your choice.

1. An RSpec unit test that tests the action of a user rolling two six sided die and having their avatar move forward a number of steps equivalent to the die' value. This app behavior currently has no coverage.

   Yes. One of the goals of characterization testing is to improve test coverage

2. An RSpec unit test that tests a corner case of the "purchase" function. Specifically, if players want to purchase a property, their balances are deducted and property ownership is transferred to their name. This tests verifies whether the purchase action is terminated if the player's balance is not enough.

   No. Characterization tests are meant to capture existing app behavior, not test boundary conditions or corner cases.

3. Documentation indicates that the previous team fixed a bug in the "pay_rent" function, which players call to send payments to one another. However, how the bug was fixed was not documented. Your team proposes adding a unit test to check that this behavior doesn't reappear.

   Either 1. Yes, One of the goals of characterization testing is to improve test coverage. OR 2. No, Characterization Testing is not meant to prevent regressions (a.k.a. reintroduction of earlier bugs)

3 One team's tests = Another team's tests(?)

Upon running the app's testing suite, your team notices that some of the existing tests are failing. Do you (a) remove these tests, (b) fix these tests, (c) replace these tests with new ones, or (d) mark them as pending? In two sentences, clearly indicate your answer choice letter + explain why.

d. Mark them as pending. As mentioned in ESaaS 9.3.1: "If the test suite is out-of-date, some tests may be failing red. Rather than trying to fix the tests before you understand the code, mark them as "pending" with a comment that reminds you to come back to them later to find out why they fail."

4 Quantified Style

After meeting with your customer, you start thinking about enforcing code style. Monopoly is a game with complicated logic arising from specific conditions. Your team plans to implement the game logic as a series of nested if conditionals. However, to avoid methods becoming cluttered or unreadable, **you want to avoid any one method having too many nested conditionals**.

Your team is considering using the following code quality metrics: Code to test ratio, C0 coverage, ABC score, and Cyclomatic Complexity. Out of these metrics, which **one** value is most helpful in quantifying whether the aforementioned code style is respected? In at most two sentences, indicate which single metric and why.

Cyclomatic Complexity. This code metric calculates the number of independent execution paths in a method. Conditionals create two or more different execution paths, and nested conditionals multiply the number of possible paths. For instance, a set of two nested if-else conditionals could have up to 4 paths of execution.

5 Potpourri

5.1

Which is not a goal of method level refactoring?

- Reducing code complexity
- Eliminating code smells
- Eliminating bugs
- Improving testability

While debugging is important, the goal of refactoring is to preserve the code's current behavior while changing its structure. Therefore, Eliminating bugs is not a priority at the stage of method level refactoring.

5.2

State whether each of the following is a goal of unit and functional testing, a goal of characterization testing, or both:

- Improve Coverage
- Test boundary conditions and corner cases
- Document intent and behavior of app code
- Prevent regressions (reintroduction of earlier bugs)

(i) and (iii) are goals of unit, functional, and characterization testing. (ii) and (iv) are goals of unit and functional testing, but non-goals of characterization testing.

5.3

What are some reasons it is important to get the app running in development even if you don't plan to make any code changes right away?

A few reasons among many include:

1. For SaaS, the existing tests may need access to a test database, which may not be accessible in production.

2. Part of your exploration might involve the use of an interactive debugger or other tools that could slow down execution, which would be disruptive on the live site.

3. For part of your exploration you might want to modify data in the database, which you can't do with live customer data.