Legacy Code Retreat

Software Craftsmanship New York
2015 December

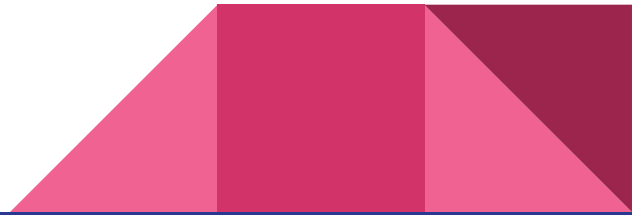
Rafael Huaman, Andrew Leung, Li-Hsuan Lung, Balint Pato

Legacy code

The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above.

It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.

/Uncle Bob’s foreword to Working Effectively With Legacy Code (Michael Feathers)/



Dealing with legacy code

- Legacy code definitions
  - Old code
  - Code that is no longer supported
  - Code that was written by X who left the company
  - Code that doesn’t have any tests - Michael Feathers
  - Any code that is in production
- Changing legacy code options
  - Edit and Pray
  - Cover And Modify

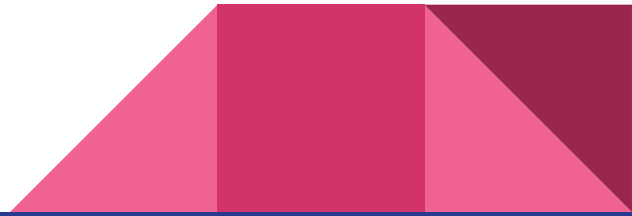
What is refactoring?

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

-Martin Fowler in Refactoring Improving The Design of Existing Code

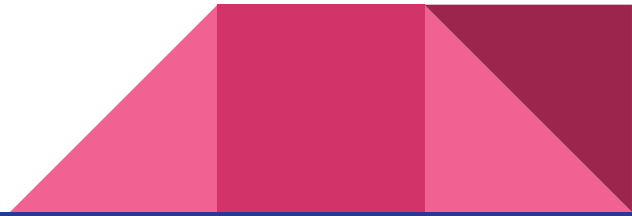
Kent Beck’s 4 Rules of Simple Design

1. Passes the tests
2. Reveals Intention (expressive code)
3. No Duplication (DRY)
4. Fewest Elements or Simplest Design



Rules for Legacy Code Retreat

- Rule 1: Don’t change code unless it is tested!
- Rule 2: See Rule 1
- Rule 3: Switch pairs after every session.
- Rule 4: Throw away your changes after every session.

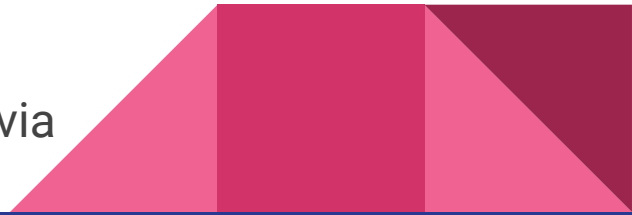


Ready?...

Session 1- Understanding the code

- Don’t touch the code
- Add characterization tests
- Setup a test coverage tool
- 45 minutes

https://github.com/software-craftsmanship-new-york/trivia



Session Retro

Session 2 - Golden Master

- Golden master = record behaviour -> characterization tests
  - 1. function weird(int a, int b) { return a + b/(a-b)}
  - 2. covering the state space on the inputs:
    - ? a: 1..5, b: 6..7
    - -1..1, -1..1
  - 3. capture input/output pairs: {(-1,-1) -> DivBy0, (0,-1) -> -1, …}
  - 4. assert on it: assertThat(output, is(expectedOutput))
- Some tricks to record our ugly trivia's behaviour
  - seeding the random generator
    - try it first with one seed
  - capture output - ideas?

https://github.com/software-craftsmanship-new-york/trivia

Session Retro

Session 3 - Subclass for testing

- Reduce test complexity by controlling behavior of dependent objects
- Make use of the testing frameworks mocking and stubbing facilities
- Look at how objects interact with each other

https://github.com/software-craftsmanship-new-york/trivia

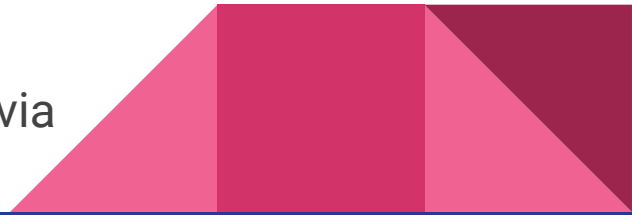
Session Retro

Lunch

Session 4 - Extract Classes

- Single Responsibility Principle
- A class should only have one reason to change.
- So a responsibility is a family of functions that serves one particular actor.
- An actor for a responsibility is the single source of change for that responsibility.

https://github.com/software-craftsmanship-new-york/trivia



Session Retro

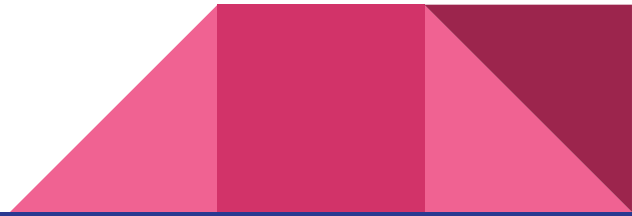
Session 5 - Dependency Injection

- Instantiation: hard to test
- Instead: Pass it in!
  - Constructor vs. setter
  - Frameworks vs. “hand crafted”
  - Who creates the object? Factory/Container/your class
- Other benefits besides testability
  - use interfaces - you can swap the implementation
  - prod/tests are just the first “scopes” for “wiring” - think: “profiles” or even “request”
- Don’t throw away your changes!

Session Retro

Session 6 - Free For All

- You are free to code however you want.
- Some suggestions:
  - Pure Functions - Make sure your methods have no side-effects.
  - Cowboy style - No testing (Run them at the end see how you did).
  - Golden Master Reprise



Day Retro

- What, if anything, did you learn today?
- What, if anything, surprised you today?
- What, if anything, will you do differently in the future?
- Would you use anything from today at your day job on Monday?