ProTech

protechtraining.com

"Charting the Course ...

... to Your Success!"

Taming Legacy Code (Java Edition)

Course Summary

Description

In this course, students will learn practical techniques to address technical debt in legacy Java code bases.

Topics

- Introduction and logistics
- Understanding the problem
- What does good code look like?
- Recognizing problems
- Fixing problems
- Visualizing the desired end state
- A quick greenfield TDD exercise
- The Six Great Fears about legacy code and their pragmatic answers; or:
- Stages of improvement
- Breaking up legacy code
- Wrap-up

Prerequisites

Students should have working knowledge of Java, curiosity and a desire to improve the technical debt situation in your shop.

Duration

Two days

ProTech

protechtraining.com

"Charting the Course ...

... to Your Success!"

Taming Legacy Code (Java Edition)

Course Outline

I. Introduction and logistics
- A. Introduction – Who are you, who am I, why are we here?
- B. Logistics – Class start/stop times, lunch time, break times, location of rest rooms, etc.

II. Understanding the problem
- A. Legacy code defined
- B. What are your problems with legacy code?

III. What does good code look like?
- A. Cohesion
- B. Coupling
- C. The SOLID principles of OO design
- D. Cyclomatic complexity

IV. Recognizing problems
- A. Code smells
- B. Separation of concerns
- C. The most common problems in Java legacy code

V. Fixing problems
- A. Refactoring
- B. The most common refactorings when working with Java legacy code
- C. Unit Tests
- D. IDE features that support legacy code remediation

VI. Visualizing the desired end state
- A. All code is under version control
- B. All changes are testdriven
- C. Team routinely practices incremental refactoring

VII. A quick greenfield TDD exercise
- A. Randoristyle hands on exercise
- B. Debrief – capture lessons learned
- C. List concerns you have about the feasibility of all this in your environment

VIII. The Six Great Fears about legacy code and their pragmatic answers; or:
- A. How to solve world hunger
- B. The code is complicated and it takes a long time to learn it.
- C. We don't have time to refactor the whole mess at once; there's just too much technical debt.
- D. Due to delivery pressure, we only have time to hack changes as fast as we can. These techniques add to development time, and we don't have extra time.
- E. Even if these techniques could help in the long run, we can't slow down even temporarily to learn them. All we can do is keep running as fast as possible on the treadmill.
- F. How can we drive development from tests when we don't have any tests? How can we even begin?
- G. A test case that is “meaningful” would cover so many conditions and would have so many external dependencies that it isn't practical to write unit tests. Maybe in an ideal world, but not here.

IX. Stages of improvement
- A. Refactor selectively to break dependencies
- B. Characterize current application behavior
- C. Start test driving changes

X. Breaking up legacy code
- A. Randori style Hands on exercise based on prepared code
- B. Debrief – capture lessons learned

XI. Breaking up legacy code
- A. Debrief – capture lessons learned

XII. Breaking up legacy code
- A. Randori style hands-on exercise based on prepared code
- B. Debrief – capture lessons learned

XIII. Breaking up legacy code
- A. Randori style hands-on exercise based on your actual legacy code
- B. Debrief – capture lessons learned

XIV. Breaking up legacy code
- A. Randori style hands-on exercise based on your actual legacy code
- B. Debrief – capture lessons learned

XV. Wrapup
- A. Discussion of lessons learned on the second day.
- B. Retrospective – Did we achieve our learning goals?
- C. Course evaluation
- D. Recommended books and websites for further information