TDD: Refactoring

"Code with Passion!"

Topics

- What is and Why refactoring?
- Things to refactor
- Refactoring legacy code
- How to improve your refactoring skills?

What is and Why Refactoring?

What is refactoring?

- A series of small steps, each of which changes the program's internal structure without changing its external behavior (Martin Fowler)
  - Verify “no change in external behavior” by Unit testing

- Example scenarios that require refactoring
  - Code smells
  - We have too much duplicate code
  - The class is too big
  - ...

Why refactoring?

- Helps us deliver more business value faster
- Minimize technical debt
  - To “fix broken windows” (Pragmatic Programmers)
- Improves the design of the software
  - Easier to maintain and understand
  - Easier to facilitate change
  - Increased re-usability
- Understand unfamiliar code
- To help find bugs

Things to refactor

Remove Complexity: Make future change easier

- Bad names (variables, methods, classes)
- Big class
- Long methods
- Deep conditionals
- Magic numbers
- Local variables
- Improper variable scoping
- Missing encapsulation
- Irrelevant comments
- Feature envy - A method accesses the data of another object more than its own data

Remove Cleverness: Make code more readable

- Cryptic code
- Abbreviated code
- Hijacking a method (changing its intent for your own purpose)

Refactoring Legacy Code

What is Legacy Code?

- Code without tests
  - “I can't test this code”

- Code with “code smells”
  - Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality

Technical Challenges of Legacy Code

- Challenges
  - Usage of static method
  - Usage of singleton
  - Creation of dependency inside the code

- Approach to take for refactoring
  - Write testing code without changing the target code first – use seam approach
  - Once testing code is done, start refactoring the target code – dependency should be injected (dependency injection)

Best Practices with working with Legacy Code

- Start testing from shortest to deepest branch
  - The shortest branch provides the highest level of abstraction to be tested
- Start refactoring target code from deepest to the shortest branch
  - The deepest branch provides smallest amount of code that needs to be refactored without disrupting the rest of the target code

How to Improve Your Refactoring Skills?

Tips for improving your refactoring skills

- Practice with Kata examples
  - Kata is Karate form that can be practiced repeatedly each time making a small progress
  - http://codekata.com
- Take some legacy code and start refactoring
  - The more practice, the better you will be in refactoring

Lab:

Exercise 1: Trip service application refactoring
1657_tdd_refactoring.zip

Trip service legacy code

- Trip service has a method that returns list of trips of a friend of a logged in user
- Problems
  - It does not have any test code
  - It has code smells – usage of local variables, long method, conditional
  - It uses static method
  - It uses singleton
  - It creates a dependency object inside
- Things to do – read “readme.txt” of the starter project
  - Write testing code
  - Refactor the code removing code smells
  - Inject dependencies

Lab:

Exercise 2: Movie rental application refactoring
1657_tdd_refactoring.zip

Movie Rental legacy code

- Customer class has a single big method that prints out customer receipt of renting movies
- Problems – code smells
  - Long method
  - Deep conditionals
  - Feature envy
  - Local variables
- Things to do – read “readme.txt” of the starter project
  - Initial testing code is provided
  - Refactor the code removing code smells

Code with Passion!