Working Effectively with Legacy Code
What’s the book about?

• Software rots, get used to it – software entropy
• Techniques to understand code, get it under test, refactor it, and add features
• Making legacy code better, even if not perfect
• What is legacy code?
  – Code written by someone else
  – Code we don’t understand or difficult to change
  – Code not covered by tests
Four Reasons to Change Software

• Adding a feature
• Fixing a bug
• Improving the design
• Optimizing resource usage

• Can you think of any others?
Refactoring

• Software is more like gardening than construction
• Refactoring: changing the internal structure of code without changing its external behavior
  – Don’t try to refactor and add functionality at the same time
  – Have good tests and run them often when refactoring
  – Take short, deliberate steps
• Become familiar with automated refactoring tools
• See www.refactoring.com
What Changes?

<table>
<thead>
<tr>
<th></th>
<th>Adding a Feature</th>
<th>Fixing a Bug</th>
<th>Refactoring</th>
<th>Optimizing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure</strong></td>
<td>Changes</td>
<td>Changes</td>
<td>Changes</td>
<td></td>
</tr>
<tr>
<td><strong>New Functionality</strong></td>
<td>Changes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td></td>
<td>Changes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Resource Usage</strong></td>
<td></td>
<td></td>
<td></td>
<td>Changes</td>
</tr>
</tbody>
</table>
How Much Changes?

Existing Behavior

New Behavior
What are the Implications?

• Make sure that the small number of things that we change are changed correctly

• Preserve existing behavior
  – i.e. ensure that the vast majority of the behavior doesn’t change
How do we do this?

• Minimize number of changes?
  – May result in poor choices (broken windows)
    • E.g. add a little to a method, even though it makes the method more complex than it needs to be
  – “the move from figuring things out to making changes feels like jumping off a cliff to avoid a tiger. You hesitate and hesitate. ‘Am I ready to do it? Well, I guess I have to.’”
    • Ex. Sprint consultants convincing each other to deploy
Chapter 2

• Edit and Pray?
  – Study the code
  – Make the change
  – Do some testing to see if the new functionality works and if we broke anything
    • How do we know? There is a lot to test

• Cover and Modify – the best option
Software Vise

• Tests that detect change serve as a software vise.
Large vs. Small Tests

- Problems with large tests
  - Error localization
  - Execution time
  - Coverage (hard to cover just new code)

- Qualities of good unit tests
  - They run fast
    - If it takes 1/10th of a second, it is too slow
    - Don’t talk to db, over network, files, configuration
  - They help us localize problems
Cover and Modify

• Legacy Code Change Algorithm
  – Identify change points
  – Find test points
  – Break dependencies
  – Write tests
  – Make changes and refactor
Ch.3

- Two reasons to break dependencies
  - Sensing: accessing values our code computes
  - Separation: getting our code in a test harness
Fakes vs. Mocks

• Fakes are simpler objects that stand in for the real thing
• Mocks are more advanced fakes that can include assertions (e.g. what the object should be given and what it should return)
Ch. 4: Seams

• A seam is a place where you can alter behavior in your program without editing in that place.
  – Preprocessing seams (e.g. #ifdef)
  – Link seams (e.g. change classpath)
  – Object seams (e.g. override method)