Test Driven Development (TDD), and Refactoring Legacy Code Using C#

**Duration:** 4 Days

**Price:** $2295

**Discounts:** We offer multiple discount options. [Click here](#) for more info.

**Delivery Options:** Attend face-to-face in the classroom, [remote-live](#) or [on-demand training](#).

### Students Will Learn

- Agile development and the test-driven development paradigm
- Creating tests from use cases and/or Agile methodology
- Unit testing using NUnit and Visual Studio
- Testing code that interacts with databases
- Using mocks, fakes, and stubs
- Automating tests, builds and check-ins using a continuous integration server
- Refactoring existing code to improve clarity, readability and maintainability
- Identifying patterns useful in TDD including the SOLID principles
- Identifying and eliminating dependencies that make code difficult to maintain and extend
- Tracking code coverage and analyzing other code metrics to improve code maintainability
- Using the seam model to identify appropriate places in the code to make changes safely
- Identifying and correcting various types of code smells
- Using effect sketches and pinch points to identify optimal places for tests
- Using feature sketches to identify opportunistic refactoring

### Course Description

This course provides students with hands on experience learning Test Driven Development (TDD) using NUnit and Microsoft’s Visual Studio. Students will build unit tests using mocks, fakes, stubs and drivers, and address issues working with databases and other systems. Students will create tests and code that will be more likely to meet and exceed requirements. Code that receives “test coverage” will not break existing systems, because tests are passed before code is checked in.

Students will spend time working with the issues involved in refactoring legacy code, safely cutting into an already deployed system. Students will work on looking for, or creating
“seams” to more safely improve code or add features, and work on identifying “code smells” that need attention in a productive system.

Finally, students will explore dependency issues as well as techniques to better understand and improve complex systems.

Students will also examine TDD and refactoring legacy code in other languages like Java to gain a broader view of options and issues working in a multi-language shop. Comprehensive labs using C# provide facilitated hands on practice crucial to developing competence and confidence with the new skills being learned.

**Course Prerequisites**

C# programming experience and an understanding of object-oriented design principles. HOTT’s [Learning to Program with C#](https://www.hott.com/courses/learning-to-program-with-c/) course or equivalent knowledge provides a solid foundation.

**Course Overview**

<table>
<thead>
<tr>
<th>Why TDD? Think Twice, Write Production Code Once</th>
<th>Basic Unit Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Utilizing a Safety Net of Automated Testing</td>
<td>■ NUnit and Visual Studio</td>
</tr>
<tr>
<td>■ Agile Development Concepts</td>
<td>■ Adding Complexity to Initial Simple Tests</td>
</tr>
<tr>
<td>■ Eliminating Bugs Early</td>
<td>■ Making Tests Easy to Run</td>
</tr>
<tr>
<td>■ Smoothing Out Production Rollouts</td>
<td>■ The TDD Pattern: Red, Green Refactor</td>
</tr>
<tr>
<td>■ Writing Code Faster via Testing</td>
<td>■ Using Methods of the Assert Class</td>
</tr>
<tr>
<td>■ Reducing Technical Debt</td>
<td>■ Boundary Testing</td>
</tr>
<tr>
<td>■ Practicing Emergent Design</td>
<td>■ Unit Test Limitations</td>
</tr>
<tr>
<td>■ Making Changes More Safe</td>
<td></td>
</tr>
<tr>
<td>■ The Importance of Regression Testing</td>
<td></td>
</tr>
</tbody>
</table>

**Comprehensive Unit Testing Concepts**

| ■ Using Declarative-Style Attributes | ■ TDD Development Patterns |
| ■ Using Constraints for More Complex Scenarios | ■ Naming Conventions for Better Code |
| ■ Using Test Categories | ■ Using Mock Objects |
| ■ Exception Handling in Tests | ■ Using Fakes |
| ■ NUnit Test Initialization and Clean Up Methods | ■ Using Stubs |
| ■ Writing Clean and Dirty Tests | ■ Test Doubles |
| ■ Testing with Collections, Generics and Arrays | ■ Manual Mocking |
| ■ Negative Testing | ■ Mocking with a Mock Framework |

**Mocks, Fakes, Stubs and Drivers**

| ■ TDD Development Patterns | ■ Self-Shunt Pattern |
| ■ Naming Conventions for Better Code | |
| ■ Using Mock Objects | |
| ■ Using Fakes | |
| ■ Using Stubs | |
| ■ Test Doubles | |
| ■ Manual Mocking | |
| ■ Mocking with a Mock Framework | |

**Database Unit Testing**

| ■ Mocking the Data Layer | ■ Refactoring Existing Code |
| ■ Identifying what Should Be Tested in Databases | ■ Restructuring |
| ■ Stored Procedure Tests | ■ Extracting Methods |
| ■ Schema Testing | ■ Removing Duplication |

**Refactoring Basics**

| ■ Refactoring Existing Code | ■ Reducing Coupling |
| ■ Restructuring | |
| ■ Extracting Methods | |
| ■ Removing Duplication | |
### Patterns and Anti-Patterns in TDD

- The SOLID Principles
- Factory Methods
- Coding to Interface References
- Checking Parameters for Validity Test
- Open/Closed Principle: Open to Extension, Closed to Change
- Breaking Out Method/Object
- Extract and Override Call
- Extract and Override Factory Method
- Singleton Pattern
- Decorator Pattern
- Facade Pattern
- State Pattern
- MVP, MVC and MVVM Patterns
- Finding and Removing Code Smells/Antipatterns

### Refactoring Legacy Code

- Reducing Risk of Change
  - Eliminating Dependencies
  - Characterization Tests as a Safety Net
  - Introducing Abstractions to Break Dependencies
- Analyzing Legacy Code
  - Identifying Pinch Points with Effect Analysis
  - Identifying Seams for Expansion and Testing
  - Listing Markup
- Minimizing Risk of Adding New Behavior
  - Sprout Method
  - Sprout Class
  - Wrap Method
  - Wrap Class
- Dealing with Code that's Difficult to Test
  - Globals and Singletons in Tests
  - Inaccessible Methods and Fields
- Using Smells to Identify What to Refactor
  - Dealing with Monster Methods
  - Dealing with Excessively Complex, Large Classes
  - Identifying and Eliminating Duplication
  - Other Smells
- Dealing with Large Legacy Systems
  - Preserving Signatures

### Code Coverage

- White Box vs Black Box Testing
- Planning to Increase Code Coverage Over Time
  - Goal 80% or More Test Coverage
  - Statement Coverage
  - Condition Coverage
  - Path Coverage
- Test Harnesses
- Unit-Testing Harnesses

### System, Regression and Acceptance Testing

- Statistical Sampling
- Usability Testing
- Test Protocols
- Regression Testing
- Acceptance Testing
Continuous Integration
Servers/Automated Testing

- Early Warning of Conflicts
- Metrics and Tools
- Checking into Repository
  - Team Foundation Server (TFS)
  - Subversion
- Continuous Integration Servers
  - CruiseControl.NET
  - Hudson
- Automating the Build/Deployment

Hands On Technical Training
The Best Way to Transfer Technology Skills

1 Village Square, Suite 8
14 Fletcher Street
Chelmsford, MA 01824

Copyright© 2019