Go Test Driven Development

Course Summary

Description

Test Driven Development (TDD) has become a standard best practice for developers, especially those working in an Agile development environment. TDD is more than just automated unit testing, it is a team and individual development discipline which, when followed correctly, increases productivity of both individual developers and entire teams. From the programmer's perspective, TDD has another benefit – it allows programmers to eliminate the tedious tasks of debugging and reworking code so that programmers can focus on the creative work of designing and writing code. It makes programming fun again.

The course integrates two primary learning streams.

The first is the how to effectively implement TDD in a production or development environment and intel-grating TDD practices with other practices like software craftsmanship, agile design practices, continuous integration and best practices in program design and Go development.

The second learning stream is an in depth and hands on deep dive into Go TDD tools, starting with the standard Go test framework but also including a number of useful TDD third party tools that are popular in the Golang community.

The class is designed to be about 50% hands on labs and exercises, about 25% theory and 25% instruct-tor led hands on learning where students code along with the instructor.

Topics

- The TDD process - "red, green, refactor"
- Eliminating technical debt with TDD
- Why TDD works
- Integrating the TDD discipline into a development process
- Using TDD to support programming and design best practices
- Developing a TDD project using the Go test framework
- Go test concepts, architecture and features
- Go assertions, using the testify package, hamcrest, gocheck, etc
- The concepts of using mocks.
- Mocking in Go: libraries (testify, gomock) and generators (counterfeit, minimock)
- How to developing good tests and test suites
- Best practices when using TDD to improve development
- TDD and concurrency in Go
- Migrating to TDD as a programming discipline

Audience

This course is intended for Go programmers who have successfully completed ProTech's "Introduction to Go Programming for Developers" (PT20182) or have an equivalent level of Go knowledge and experience.

Prerequisites

This course is intended for Go programmers who have successfully completed ProTech's "Introduction to Go Programming for Developers" (PT20182) or have an equivalent level of Go knowledge and experience. Students who do not have this prerequisite will find the course difficult to follow and the labs too fast pasted. Because the course does not have time that can be allocated to doing additional Go remedial instruction, this prerequisite is essential.

Duration

Three days

Go Test Driven Development

Course Outline

I. Go Test Driven Development Introduction: The TDD process as a discipline
- A. How TDD improves efficiency and effectiveness of programming
- B. Eliminating technical debt, debugging and rework
- C. Integrating TDD with best practices in program design and coding
- D. TDD as a core Agile practice
- E. The Agile testing quadrants
- F. The importance of test automation
- G. Refactoring: what it is and why we do it

II. An Introduction to the Go test framework
- A. The Go testing package design and architecture
- B. The TDD process implemented with Go test
- C. Runners, fixtures test execution and reporting
- D. Tests versus benchmarks
- E. The testing.T and testing.B structures
- F. Go assertions
- G. Test fixtures with TestMain()
- H. Using subtests
- I. Test coverage reporting
- J. The “go test” options

III. TDD Best Practices I
- A. Testing through interfaces and function signatures
- B. Command / Query segregation
- C. Functional testing concepts
- D. Relationship between good component design and ease of testing
- E. Understanding Unit testing – component isolation
- F. Planning the development to minimize work
- G. How to decide what test to add next
- H. TDD best practices
- I. Common errors when implementing TDD

IV. More Go Testing
- A. Design by contract
- B. Writing tests for preconditions, post-conditions and invariants
- C. Testing for error correctness (e.g. if the correct error is returned)
- D. Selective execution of tests and grouping of tests
- E. Using gosuite for test setup and teardown
- F. Test coverage reporting

V. Testing Concepts
- A. Criteria for good testing: validity, accuracy and reliability
- B. Why our tests have to be correct
- C. Common sources of test case errors
- D. Systematic and algorithmic test case development
- E. Deriving test cases from acceptance tests
- F. Functional coverage measures
- G. Determining optimal numbers of tests
- H. Using test cases to identify requirements and specification problems
- I. Dealing with valid, invalid and outlier test cases
- J. Combinatorial versus stateful testing

VI. Assertions and Predicates
- A. Checking versus testing conditions
- B. Writing assertion with the Assertions package
- C. Writing assertions with testify and hamcrest
- D. Using assertions to examine structures, lists, etc
- E. Writing complex assertion predicates
- F. Comparison of different assertion packages

Go Test Driven Development

Course Outline (cont'd)

VII. Mocking and Mock Libraries
- A. Mocks, stubs, drivers and component unit testing
- B. Implementing interfaces with a mock object
- C. Overview of mocking libraries: stretchr/mock, gomock, etc.
- D. How mocking libraries work
- E. Generating mocks for interfaces using minimock
- F. Implementing and using mocks from a mocking library
- G. Designing and implementing a mocking strategy

VIII. TDD Best Practices II
- A. Organizing and maintaining the test environment
- B. Testing the test code
- C. Developing tests and code design together
- D. Deciding intensive and comprehensive should be
- E. TDD implementation patterns
- F. Metrics for TDD

IX. Refactoring
- A. Refactoring as controlled code changes
- B. Using TDD to implement a refactoring
- C. Code smells – driver for refactoring
- D. Code refactoring versus design refactoring
- E. Refactoring to a design pattern
- F. Using refactoring to reduce technical debt
- G. Refactoring best practices

X. Using TDD to Develop Concurrent Code
- A. The challenges of testing concurrent code
- B. Adapting the TDD approach to concurrent applications
- C. The "well-defined" and "well-designed" requirements for TDD
- D. A model template for developing concurrent code with TDD

XI. Implementing TDD
- A. Review of TDD best practices
- B. Review of Go testing best practices
- C. Planning a TDD project
- D. Integrating TDD into a development process
- E. Integrating TDD with coding excellence
- F. The importance of tracking metrics
- G. Developing an implementation plan
- H. Pitfalls, snares and traps to avoid