Test Driven Development

How we typically create classes?

- We think about what a class must do
- We focus on its implementation
- We write fields
- We write methods
- We may write a few test cases to see if it works
- We hand it off to users of our code
- We then wait for them to come back with feedback (problems)
TDD vs. TFD

- Test Driven Development
  - Written to gain insight into design and implementation issues
  - Quickly alerts when things fall apart

- Test First Development
  - Test for code written before any code is written
  - You start with a failing test and write minimal code to make it work

- TFD is a subclass of TDD

- TFD very helpful at times; TDD adequate at other times

Test First Coding

- How about starting with a test case even before we have any code for our class?

- How about first write test that fail because the code to support it does not exist?

- How about adding functionality to our system by adding tests incrementally and then adding code to make those tests succeed?
Test First Coding Benefits

• It would
  • completely revert the way we develop
  • We think about how our class will be used first
    • Helps us develop better interfaces that are easier to call and use
  • Would change the way we perceive things
  • Will have code that verifies operations
  • Will increase robustness of code
  • Will verify changes we make
  • Will give us more confidence in our code

Test First Coding Benefits...

• Forces us to make our code testable

• Tests decouple the program from its surroundings

• Serves as invaluable form of documentation
  • Shows others how to use our code
Why TDD?

- “Clean code that works,” – Ron Jeffries
- Predictable, no worry of mounting bugs
- Better quality of code
- Makes you dependable
- You are confident... feel good

Programming by Intention

- Making your intent clear
  - Avoid Opacity – that is code that is hard to understand
- It goes a long way in writing code
  - Choose appropriate names for methods, fields, classes, etc.
  - Strive for simplicity
- Test First, Code next
Simple Code that works

- Striving for simplicity is important
- When you write your test, you think of how the object should be used
- And you do that before you implemented it
- This leads to finding the simplest way to communicate
- Avoid unnecessary complication or over abstraction

Simple Code that works...

- Lie your way as much as you can
  - I don’t recommend this in real life–especially to the spouse
- Write minimal code that works
- Write only code if a test fails
- You strive to reduce complexity
- You strive for simplicity
**Types of Tests**

- **Black-box testing**
  - Does not know and depend on internal structure of modules being tested

- **Acceptance testing**
  - Written by customers, QA
  - Focuses on functionality of the system

- **White-box testing**
  - Knows and depends on internal structure of modules being tested

- **Unit Testing**
  - Drives the design
  - Validates changes made
  - Insufficient as verification tool however

---

**Acceptance Testing**

- Manual testing is not the preferred way

- Need to find ways to automate this as well

- Promotes separation of business logic from UI

- May be written using scripts, XML, etc.
**Unit Testing**

- Unit Testing is an act of design than an act of verification
- It helps provide instant feedback when code is changed
  - Substantially improves robustness of app
- Works as a great form of documentation
- Safety net when refactoring code

**System functionality and TDD**

- Evolutionary design
- Requirements change
- Functionality is added constantly
- Test validate to make sure your code change abides by the contract and expectations of the rest of the code
Tenets of TDD

- Write code only if a test case fails
- Minimum code that works
- Seek pragmatic and simple design
- Eliminate duplication

TDD Mantra

- Red—write a test that doesn’t work
- Green—make the test work (minimum code that makes it work)
- Refactor—eliminate any duplication
- Red / Green / Refactor
Let’s develop an application using Test Driven Development techniques

Part I – TickTackToe playing the game

Creating task list
Using JUnit
Quality of tests
  Positive, negative and exception
Test isolation
Setup
Practices for Unit Testing

- Use Unit Testing as Design Tool
- Consider untested code as unfinished code
- Use testing as safety net
- Test on each platform supported
- Test Business Logic

Use Unit Testing As Design Tool

- You move towards a more pragmatic and simpler design
- You can avoid bloating classes with unnecessary methods and members
- You seek a minimal yet complete design
Untested code is unfinished code

- “Done” means different things to different people
- What does “done” mean to you?
- Is it finishing the coding or making sure it actually works?
- If you throw code across fence and then find out it does work…
  - More expensive to fix
  - You have to switch context
  - Make others wait
  - It’s embarrassing
- How do you justify the time it takes for UT?
  - Measure the time to actually complete work, not just type in code

Use testing as safety net

- Would you tightrope walk without a safety net?
- Why would you modify your code without test cases?
- Unit test are great safety net that
  - provides instant feedback
  - makes your code robust
  - can be a good design tool
  - is a confidence booster
  - act as probes when solving problems
  - are a form of reliable documentation
  - are good learning tests
**Test on each platform supported**

- It works on my machine!
- Learnt this the hard way
- Tests pass on my machine, code doesn’t work on colleagues?
- Difference between XP and 2003!

- If your product will run on different OS, different virtual machines, CLR, etc…
- Run your tests on each supported platform

---

**Test Business Logic**

- Critical Business Logic must be tested by customers
- Get it into their hands early
- One of the risks to be avoided
- Why not automate these tests?
- Look at FIT
When and Who should write it?

- Best written
  - before writing code or
  - as it is being written
- Written by the developer
- Can a QA person write it?
- Not effective if others write it
- Not effective if written at other times

When not to write Unit test?

- Not effective to take existing code and start writing tests
- Why?
  - You can’t think of the specific issues to be tested all at once
  - No point going on a marathon of writing tests for legacy code
- What’s the solution then?
  - Write unit tests as you refactor or evolve code instead
When not to write Unit test?...

- Not very effective to write unit test for GUI
- People are trying...
- Different tools and techniques being developed and suggested
- Answer right now is to keep the UI layer thin

Pragmatic Unit Testing

- From Pragmatic Unit Testing by Andy Hunt and Dave Thomas
- Often we get in to trouble with boundary conditions, consistency related issues,...
- Acronyms that help us focus on testing
  - CORRECT
  - RIGHT-BICEP
Consider CORRECT

- Conformance
- Ordering
- Range
- Reference
- Existence
- Cardinality
- Time

Right-BICEP

- Are the results Right (Positive Test)
- Look for these when writing your tests
- Boundary conditions
- Check Inverse Relationships
- Cross-check using other means
- Force Error Conditions (Exception Tests)
- Test Performance Characteristic
Organizing test cases

- You want to not only test your public methods, but also your friend and protected methods
- Test cases should go in the same package as your classes
  - Makes it harder if you want to remove them at deployment time, though
- You may also want to test some select private methods
  - Why not write your test as static inner class?

Red/Green/Refactor

- First write a test code that fails
- Implement enough code to make the test succeed
  - Go ahead lie your way though it
  - Only so much you can lie
  - Keep it simple; do not complicate things
- Refactor the code to improve it
Stay one step from Green

- At any time, we should be one step away from a green bar

- Why?
  - Gives confidence with change
  - Let’s us focus on one thing well
  - Avoids tendency to write up a bunch of test cases at one time

Where should your test go?

- Not only public methods are tested
- How do you test protected or package friendly methods?

- Test code should be in the same package as your class being tested
Isolate your Tests

- One test should not affect another test
- One test should not fail because another test failed
- Provides order independence
- You can pick arbitrary set of tests to run

Test First & Assert First

- When should you write a test?
- Before writing the code to be tested!
- Remember red/green/refactor
- Write your tests with Asserting for result/conditions in mind
**Writing Test Stubs?**

- What if you plan to implement an interface method later?
  - You plan to leave a dummy implementation in place
    - You have no time for it now
  - Why not write a Test that will fail as soon as the method is implemented?
    - Have the method throw an exception when called
    - In the Test, Assert for the receipt of that Exception

**How good are your Test?**

- Your test are not good if they
  - Have long setup code
  - Have setup duplication
  - Take long duration to run
  - Are fragile
Where to run?
- It is not enough to run your tests on developer’s machine
- Tests should run on each platform supported by the product!
- Why?
  - You do not want to miss variations or differences in behavior on different platforms
  - Learning tests come in to picture here as well
- Consider continuous integration (discussed later)

OK, we found a bug?!
- We have solid tests (or so we thought)
- A bug is found
- What should we do?
- Understand the bug and fix it, right?
- Nope
- First write a (missing) test case that will bring the bug to surface
- Get the red bar on it first
- Then fix the bug to get to the green bar
- Refactor as necessary
Accommodate change?

- First write a test case for the change
- Make the change
- Run all the tests to make sure you have not broken any existing tests as well as any new tests
- Get a good green bar before you make further change
- Only one change at a time though