Test Driven Development (TDD)

Long Development Cycles

- Historically, big software development proceeded “one step at a time”
  - Product envisioned from a user point of view
  - Requirements specified
  - Definition specified
  - Design specified
  - Code written
  - Testing performed
  - Documentation written
  - Sold to user (who no longer wants it)

Why did we miss the boat?

- Such software development cycles take months to years
- Requirements were frozen early
- Requirements change
  - User needs change
  - Market windows close
  - Competition emerges with different features
  - Regulations change
  - Users couldn’t describe what they wanted accurately
  - Etc.

Agile Methodologies

- Primary purpose – react to changing (or previously unknown) requirements
- Primary result – software that is closer to meeting the current requirements

- Examples of Agile Methodologies:
  - Continuous Software Evolution
  - Extreme Programming
  - Lean Programming

Continuous Software Evolution

- I invented this in ~1988. Nobody else knows what it is. It is not published.
- Included:
  - Iteration planning
  - Automated regression testing
  - Developers developed code, tests, internal doc, user doc.
  - Strong rule: a change to the code HAD to be accompanied by a change to the tests, the internal doc, and the user doc.
- Failed due to politics
  - I tied the scheduling and project management of docs and testing to the code. The existing management hierarchies were separate; the wanna-be empire-builders couldn't handle the change, and shut me down.

- In ~1993, I used the CSE technique anyway.
- Agreed on next features to implement with customer ~4 weeks. Requirements *always* changed.
- With a single command, I could build the code, build the internal doc, build the external doc, build the tests, run the tests, and make the Product Release Tape. The product was *always* ready to ship.
- New requirements were normal; we only implemented the features that were actually asked for.
- For me personally, the result was:
  - Success!
  - Confidence!
  - Faith !?!

How old is “agile”?

- It’s *old*
- Most of the mechanism that comprises “agile” is old
- Formalization of the techniques and standardization of the terminology is new
  - The Extreme Programming movement has done the best job of this so far

Extreme Programming

- Extreme Programming (XP) is an Agile Methodology developed by people with a good ability to formalize the techniques.
- One tool they use to meet the need to handle changing requirements and shorten the development cycle: TDD
  - In XP, TDD eliminates the need for requirements tracking and design specification
  - In XP, TDD provides automated unit testing, regression testing, acceptance testing

How to learn TDD

- Many books are articles now printed and published on the web
- It’s OK to read *one*. It almost doesn’t matter which one.
- *Test Driven Development*, Kent Beck

- Generally, people are still “learning by doing”, mostly by “doing it with somebody that already knows how”

Definition: Test Driven Development

- A *software development process*
  - Not a testing technique, per se, but depends heavily on testing as a tool
- Write tests first – the tests determine what code is to be written
- Testing is done in a fine-grained fashion

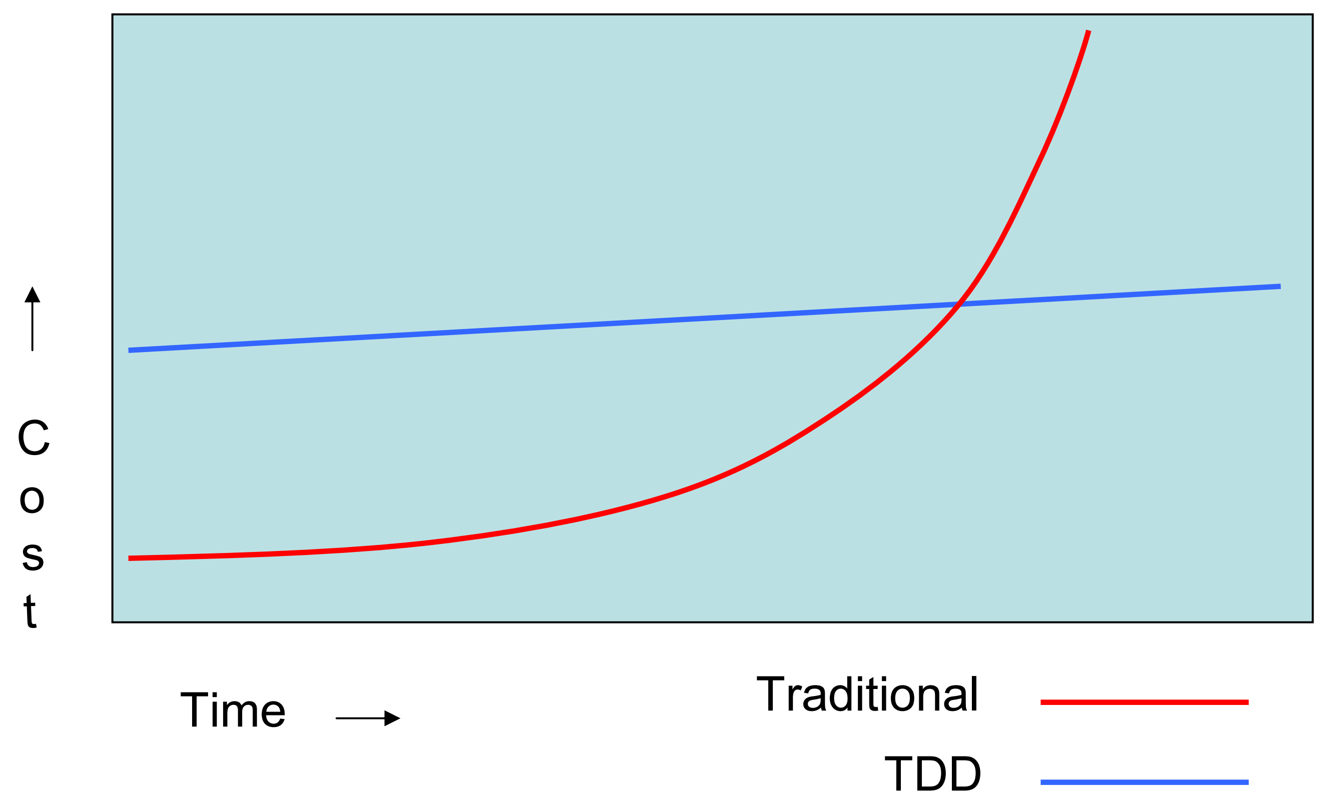
Characteristics of TDD

- Gets away from big-bang, artistic, “magic happens here” here software development
- Makes software development predictable on
  - Reliability
  - Scheduling, development cost
- Results in “clean code that works”
  - [Ron Jeffries]
- TDD is generally a white-box unit-testing mechanism
  - I will show later how to build it up to broader types of testing
- Taking small steps prevents bugs and the need for debugging
- Design optional; will emerge from the tests if necessary

Design not necessary

- First step in the procedure will always be to identify a small change to be made
- That change can be identified from
  - a formal design specification,
  - a requirement spec,
  - a user story (use case),
  - or an ad-hoc informal request from a user.
- All tests are saved forever, and are a record of requirements.
  - The tests replace the requirements and design specs

Cost of development



Personal Opinions

- Only “10%” of the programmers out there can handle TDD to the level of consistency that we need, without micromanagement.
  - (With micromanagement, nearly every programmer out there can handle it)
- Only “10%” of the programmers out there can handle Extreme Programming (XP), due to the need to wear too many hats.

Rules for developers

- Unit testing is not separable from coding
- Start as simply as possible
- Write new code ONLY if a test is failing
  - The tests provide the reason for writing a line of code
  - Write a failing test before writing a line of code
- Eliminate duplication of code and simplify code ruthlessly
  - Fewer lines of code mean fewer tests to write and maintain, prevents mushrooming of the test base
- ALL tests are saved in the automated regression test suite

Technique

- Write a single failing test
- Run the failing test
  - “Proves” that the test is correct
- Write *minimal* code that fixes the test
- Run the test again
  - “Proves” that the code is correct
- Refactor towards a better design
- Run the test again
  - “Proves” that the better code is still correct

Technique 2

- Identify a “smallest possible” change to be made
- Implement test and (the one line of) code for that change (see previous slide)
- Run *all* tests
- Save test and code together in source control system
- Repeat

Elements of TDD unit tests

- Testing and reporting tool (xUnit)
- Test suites (groups of tests)
- Tests
- Mock resources
- Test library (assert implementations, etc.)

- Product-specific setup library

The Form of a test

- Test-group (Container) Setup
- Test-specific Setup
- Invoke functionality
- Test results
- Test-specific Teardown (if any)
- Test-group Teardown

Organization

- Use “libraries” (in OO, these may be classes)
  - Global Utilities (e.g., for assertions)
  - Test-group Utilities (e.g., for setups)
  - Test-specific Utilities for test convenience
  - Utilities for test setups (initialization)

Define: refactoring

- Rewriting already-working code for the purposes of:
  - Elimination of duplicate code
  - Simplify testing and coding
  - Following accepted software engineering principles

Performance

- The entire test suite needs to run in a few minutes, to encourage programmers to run them all regularly
- There are thousands of tests

- Oh, yeah. Like *that's* going to work!
- Use "Mock" resources if necessary to speed up the tests
  - E.g., in-memory database, microcontroller emulator

Why does TDD work?

- The (sometimes tedious) routine leads the programmers to think about details they otherwise don’t (because they’ve bitten off more than they can chew)
- Specifically, test cases are thought through before the programmer is allowed to think about the “interesting part” of how to implement the functionality

Why does TDD work?

- Encourages “divide-and-conquer”
- Programmers are *never* scared to make a change that might “break” the system
- The testing time that is often squeezed out of the end of a traditional development cycle *cannot* be squeezed out.

Building from the ground up

- Tests and code are written “bottom up”
- Tests build upon each other until they represent user actions and acceptance tests (sequences of user actions)

Technique 3

- Test and implement a low-level function (using previous Techniques)
    - (Notice I didn’t say “implement and *then* test a function”? Subtle, eh? You will be assimilated.)
- Test and implement a higher-level function that invokes the lower-level function
- Test all the logic in the higher-level function as expected; use as many tests as necessary
- Include *one* test that convinces you that the higher-level function called the lower-level one

Technique 4

- Build higher- and higher-level tests
- Build tests that represent user actions such as entering a piece of data and hitting “OK”
- Build tests that string together a series of user actions that represent Acceptance Test cases
- Demonstrate the Acceptance Tests to the user(s) regularly

More on higher-level testing

- To support using TDD as the mechanism for writing tests beyond unit tests, such as functional tests and acceptance tests, the most important rule to follow is to use descriptive test names

- Examples of real tests in Sphygmochron
- Demo of COMUnit on Sphygmochron
- Drawings of elements of unit tests

- Next steps
  – Implement features in spreadsheet using TDD
  – Write a Phoenix Sphygmochron Development Process, TDD is at the core, in form specified by Chris' Phoenix Process Framework, references FDA GMP CFR QSR to show where the QSR is being met