Testing Driven Development

What Is It?
• A strategy for interleaving code development with automated test development
  • “Try out” the code while writing it
    – Most likely you already do this
    – Except that now we want to save these as tests
  • Write tests while writing the features
    – Sometimes write the test before the code

TDD Recipes
• Say that we need to write code for a task
  1. (Write some code to) setup the environment for the task (e.g., a DB table with some sample data)
     • This will be the “setUp” part of a test.
     • Write “tearDown” too
  2. (Write code to) invoke the code for task A
     • This is the first part (exercise) of “testA”
     • You get to design the inputs and output of your code

So far you have written some testing code that clarifies in what context will your new feature run

TDD Recipes (cont.)
3. Now start writing the code for task A
4. Run the test, put breakpoints in your IDE, clarify the detail design of your code
   • Keep refining the code until it works in the given scenario
5. Finish the test by writing the assertions part of the test
6. Repeat from step 1 or 2 (another scenario)

TDD fits well with agile development:
• Write some code, try it out, learn how to do better
• Also comes in handy for refactoring (see later)
### TDD by Example - Setup

- **Best described (and motivated) by example**
  - E.g., See Kent Beck book “TDD by Example”

- **Our example: we want to develop a program that scans a GIT repository and tabulates the bugs that were fixed and how many lines of code were changed for each fix**
  - We’ll call this system **bugaton**
  - We’ll use the public python git repository as example

### Bugaton – Example

- **The desired output:**
  ```
  shell> cd python_git_repo
  shell> ./bugaton.py
  7673 182 lines changed
  8202 7 lines changed
  9125 18 lines changed
  ...
  ```

- **The raw data is taken from “git log --shortstat”**

### Bugaton – Initial Design

- **Display bugs**
- **Parse commits**
- **Get commit logs**

### Demo

- Files in cs169_testdriven.tar.gz (see class web site)
- Instructions in README.txt
**Test Driven Development (TDD)**

- A strategy for interleaving code development with automated test development

Repeat until done:

1. Write one or more tests that fail
   - Maybe they do not even compile
2. Write code to make the tests compile and run
3. Debug and fix the current test
   - Quickest fix first
4. Check that all other tests STILL work
5. Refactor to clean code and tests
   - Make sure that ALL tests still work

**Lessons Learned – Write Tests First**

- Helps design a "use-driven-API"  
  - Clarifies the usage of a module:
    - Name and granularity of methods
    - Name and type of parameters
    - Subject of a future lecture
- As opposed to implementation-driven-API
  - If you design without thinking of usage first
- Tests will help exercise the code you will be writing
  - Faster debugging

**Lessons Learned – What Tests To Write?**

- Driven by design/implementation:
  - One or more tests for each method
  - Exercise corner cases

- Driven by bug found:
  - Write a system test that reproduces the bug
  - Repeat until all tests pass
    - Investigate the bug and narrow down the cause
    - Write "smaller" test that reproduces the bug
    - Fix code to pass the smallest test
    - Try to run the larger tests

**Regression Testing**

- Recall: Always run all tests when code changes
  - Regression testing ensures forward progress
  - We never go back to old bugs
  -Easy to diagnose the test failure when we know which change breaks the tests
- Continuous integration systems
  - Attempt to run the regression tests on every commit (or at least with high frequency)
  - E.g., Hudson/Jenkins, Cruise Control
Lessons Learned - Refactoring

- Refactor code
  - For clarity/maintainability/testability
- Refactor tests also

Refactor:
- Only when all tests pass
- Do not also change the functionality at same time
- Fix bugs until tests pass again
- Can refactor with courage once all tests pass
- Without tests, refactoring is often too scary

Lessons Learned – Cost of TDD

- Final LOC count:
  - 80 LOC for actual code
  - 130 LOC test
    - 70 LOC test code (sometimes more challenging to write than actual code)
    - 60 LOC test data
- Fairly typical (50-70% effort goes in testing)
- Cost pays off during maintenance
- Early investment in testing infrastructure
  - Small incremental cost to add a new test
  - Helps ensure that TDD is self-sustainable

TDD - Conclusions

- TDD results in fewer bugs and more maintainable code
- TDD fits well with agile process: mix coding with trying out the code
- Try it out in your project