Test Driven Development

Kirrily Robert
The problem

Good

Cheap  Fast
No silver bullet
Time taken to fix bugs

- Design
- Implementation
- QA
- Post-release
Cheap programmers

- Best programmers 10x as effective
- Testing can close the gap (somewhat)
Software quality

- “Instinctive”
- Hard to measure
The solution

- Testing
- Test Driven Development
Testing

Design

Implement

Test
How to do it

• Design: figure out what you want to do
• Test: write a test to express the design
  • It should **FAIL**
• Implement: write the code
• Test again
  • It should **PASS**
The subroutine add() takes two arguments and adds them together. The result is returned.
use Test::More tests => 1;

is(add(2,2), 4, "Two and two is four");
$ prove -v add.t
add....Undefined subroutine &main::add called at add.t line 3.
# Looks like your test died before it could output anything.
1..1
dubious
    Test returned status 255 (wstat 65280, 0xff00)
DIED. FAILED test 1
    Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail  List of Failed
-----------------------------------------------------------------------------------
add.t        255 65280     1    2  1
 Failed 1/1 test scripts. 1/1 subtests failed.
Files=1, Tests=1, 0 wallclock secs ( 0.02 cusr + 0.01 csys = 0.03 CPU)
 Failed 1/1 test programs. 1/1 subtests failed.
Implement

sub add {
    my ($first, $second) = @_;  
    return $first + $second;
}
$ prove -v add.t
add....1..1
ok 1 - Two and two is four
ok
All tests successful.
Files=1, Tests=1, 0 wallclock secs ( 0.02 cusr + 0.01 csys = 0.03 CPU)
Wait...

- What if there are fewer than two arguments?
- What if there are more than two arguments?
- What if the arguments aren’t numeric?
Iterate

- Design
- Implement
- Test
- Test
Design

- The subroutine `add()` takes two arguments and adds them together. The result is returned.

- If fewer than two arguments are provided, `add()` will return `undef`.

- If more than two arguments are provided, `add()` will return the sum of the first two.

- If any argument is non-numeric, `add()` will return `undef`. 
use Test::More tests => 4;

is(add(2,2), 4,  
   "Simple case: two and two is four");

is(add(3), undef, 
   "Return undef for < 2 args");

is(add(2,2,2), 4, 
   "Only add first 2 args");

is(add("foo", "bar"), undef, 
   "Return undef for non-numeric args");
prove -v add.t
add....1..4
ok 1 - Two and two is four
ok 2 - Return undef for < 2 args
ok 3 - Only add first 2 args
ok 4 - Return undef for non-numeric args
ok
All tests successful.
Effective tests must be automated
Write once, run often

- Write tests once
- Keep them somewhere sensible
- Run frequently (one click)
- No human input
- Machine-parsable output
Test coverage

- How much of the code is tested?
- What areas still need testing?
- Where are the greatest risks?
TDD in summary

A. First we write a test.

B. Then we write code to make the test pass.

C. Then we find the best possible design for what we have - refactoring (Relying on the existing tests to keep us from breaking things while we are at it)

TDD goals

- **TDD** is a technique for improving the software’s *internal* quality

**Well-written code**
- Good design
- A balanced division of responsibilities
- Without duplication of responsibility
- Maintainability and smooth evolution
**Build it right: TDD**

- TDD: building up the system incrementally, knowing that we’re never far from a working baseline.
  - A test is our way of taking that next small step.
- The term *refactoring* is used to better communicate that the last step is about transforming the current design toward a better design.

**First we write a test**

- We are writing a test. Also, we are making design decisions:
  - We are designing the API—the interface for accessing the functionality we’re testing.
  - The test case that we design will be the first “client” of the functionality that we are going to implement.
  - One of the fundamental lessons in designing an interface is that we only evaluate a design effectively and objectively when we try to use it.
Then we write just enough code

- The second step of the TDD cycle is to write just enough code to make the test pass.
- You’re satisfying an explicit, unambiguous requirement expressed by a test.

And then we refactor

- Take a step back, look at our design, and figure out ways of making it better.
- It is all about keeping your software in good health—at all times.
- Refactoring is about applying refactorings on code in a controlled manner.
Keeping code healthy with refactoring

• “a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior” : Martin Fowler

Refactoring Example

• Replace Inheritance with Delegation
  – Motivation: A subclass uses only part of a superclass interface or does not want to inherit data
  – Summary: Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.
**Refactoring Example**

- Mechanics
  1. Create a field in the subclass that refers to an instance of the superclass. Initialize it to `this`.
  2. Change each method defined in the subclass to use the delegate field.
  3. Compile and test after changing each method.

**Refactoring Example**

- Mechanics
  4. Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.
  5. For each superclass method used by a client, add a simple delegating method.
Refactorings alter internal structure

• Many of the refactorings are very low-level
  – rename method
  – Rename variable

• Low-level refactorings are the fundamental building blocks to achieving larger refactorings
  – Moving the responsibilities around in your code
  – Introducing or removing an inheritance hierarchy

Refactorings preserve behavior

• whatever transformations you apply to the existing code, those transformations should only affect the code’s design and structure—not its externally visible behavior or functionality.
  –Renaming a method that is part of a class’s public interface - ???
  –how can we be sure that our refactorings haven’t changed the code’s external behavior? - ???