



John E. Boal
TestDrivenDeveloper.com

A PRACTICAL GUIDE TO UNIT TESTING

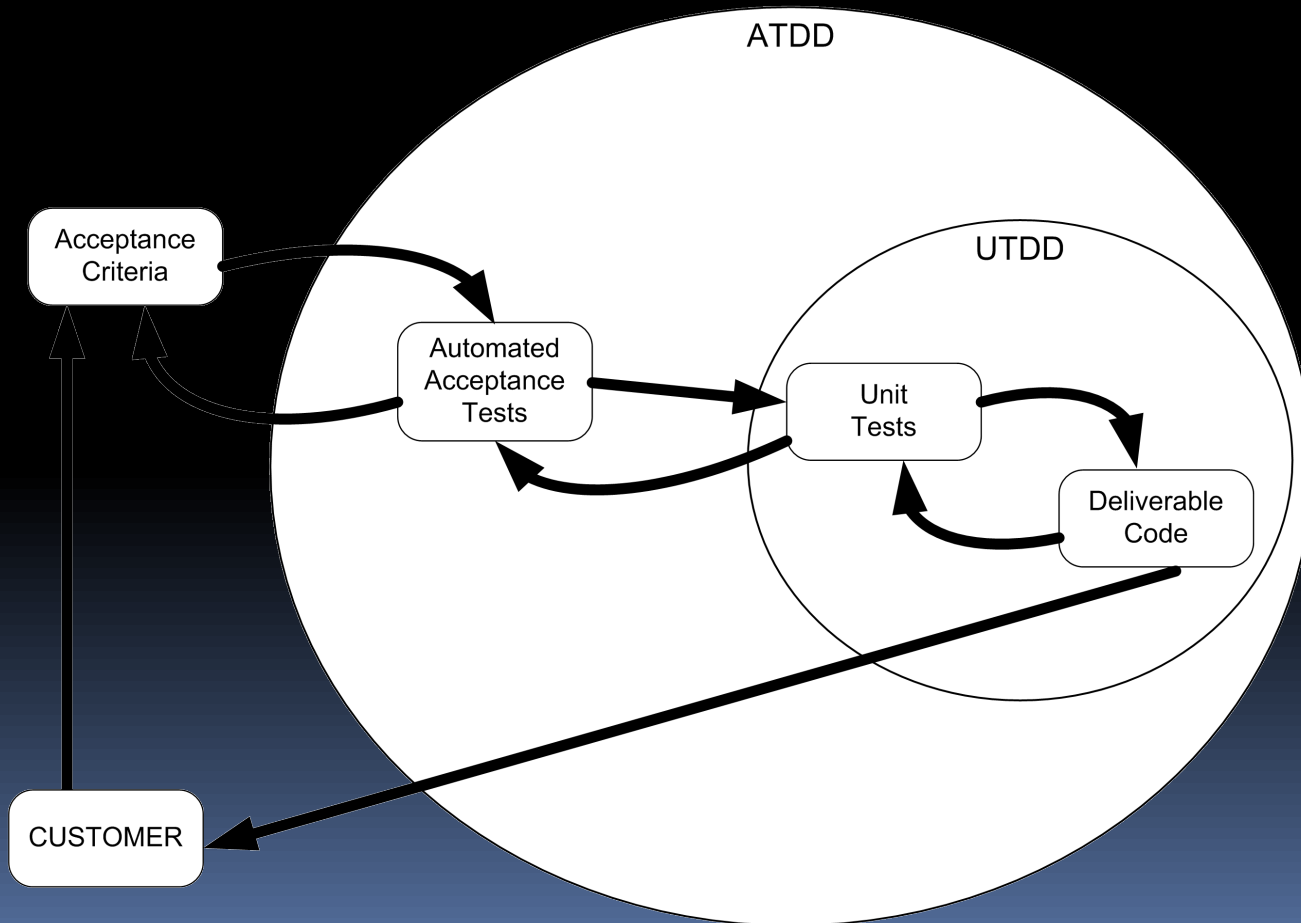


Who is John E. Boal?

- I am an experienced agile practitioner with many years experience in development and consulting.
- I am a certified Scrum Master, and experienced XP practitioner.
- I worked for Microsoft, Rockwell, Premera Blue Cross (among others), and now Velocity Partners
- I have a BS degree in Computer Science and an electronics engineering and networking background.
- I am a Test-Driven Developer
- You got a test for that?

The Agile Test-Driven Cycle

- Here is a diagram that illustrates how we can deliver software iteratively using tests to guide our delivery.



Why do we write unit tests?

- To document the programmer's intent.
 - (right, wrong, or otherwise – this is what the programmer wanted the code to do.)
- Ensure the code units function as intended.
- A suite of UTs gives us a “safety net” when we change code so that we don't break it by altering it or adding something new.
- UTs are a good indicator of how well we understand the code we are writing.

Levels of Unit Testing

- Level 0 – huh?
 - What's a unit test? [Note: U R DOING IT WRONG.]
- Level 1 – we write some unit tests.
 - We wrote some tests that tell us if our code mostly works.
- Level 2 – we write (most of) our unit tests first.
 - We write a unit test then some code, then some more code.
- Level 3 – we do Test-Driven Development
 - We write a unit test, then we write some code to make it pass. Then we refactor, and do it all over again.
- Level 4 – we do Acceptance Test-Driven development
 - We automate our acceptance criteria into acceptance tests first, and then use failing AT's as a guide to work using UTDD building code and tests until our AT's pass.

What is a unit test exactly?

- A small, FAST test that examines a little piece of code (like a single method)
- It provides a stimulus to the Code Under Test [CUT] and verifies an expected response.
- It is NOT a “scenario,” an end-to-end, or use case verification. These are different kinds of tests and should not be in with true unit tests.
- The Unit test and the code it tests should all fit on one screen.

Qualities of a good UT

- Reveals the intent of the developer who wrote it
- Independent
 - No side effects, and is isolated from other tests
 - Has no dependencies on external systems or code
 - UTs Can be run in any order
- Automated (this goes without saying...)
- Executes very fast (milliseconds)
- Unique – tests something not tested elsewhere
- Short and Clear – 5 to 10 lines of code
- Coded to same standards as main line code

What do we unit test?

- SMALL code units.
- A unit test ideally should cause execution of no more than 20 lines of main-line code.
 - (this is only a guideline)
- Unit tests are focused on a specific narrow piece of functionality rather than on a broader set.
- Method-level, or usually even smaller focus.
- We should have many tests for each method.

Testing the “right” things

- We want to ensure that our code works, so we write unit tests to illustrate how the code behaves in specific circumstances.
- Test for things that may be null
- Test for empty strings...
- Test all but the simplest properties. Don't usually need unit tests for this:

```
• public String Name  
• {  
•     get  
•     {  
•         return this.name;  
•     }  
•     set  
•     {  
•         this.name = value;  
•     }  
• }
```

When is it enough?

- How do we know if we have enough UT's?
 - 100% code coverage?
 - Zero bugs?
 - All Acceptance Tests Passing?
- ***Nope...***
- We have enough tests when we feel CONFIDENT in our code quality.
- If we don't feel confident that our code works and it's safe to refactor, keep writing tests...

Dangers of unit tests

- Unit tests are code. They need to be maintained.
- They cost time to write and they cost more time to maintain.
- Beware of writing tests for their own sake.
- Trying to hit 100% code coverage is usually not necessary (but it is a usual byproduct of TDD).
- Don't duplicate testing. Test only what hasn't yet been covered.
- Don't trust UTs that are not well written.
- Don't write complex tests. Keep them simple.

Refactoring

- “Changing existing code to refine its structure without changing its behavior.”
- **MAKE ONLY SMALL CHANGES AT ONE TIME.**
 - Then test them. With ALL the tests.
 - Then review them with someone. Then check them in...
 - This should happen a LOT.
- Consider the whole design – not just the classes and methods you happen to be touching.
 - Think globally, act locally...
- Look at the overall design and make refactoring changes accordingly.
- Refactor **MERCILESSLY.**



Code Quality

- Code quality should not be negotiable.
- It's implied that our code should just *work*.
- UT's and other tests in our test suite help us to know quickly if our code really does what we intended.
- UT's help us know in a few seconds whether a change we make has impact on the code and might affect quality.

Bugs !

- There are always bugs.
- If there is a bug, it means someplace we forgot a test! [do we have good acceptance criteria?]
- ***Find a bug, write a test!***
- DO NOT FIX A BUG WITHOUT A TEST!
- Write the test FIRST. This is important.
- First the test fails – illustrating the bug. When we fix the bug, the test passes.
- We now have a test that ensures the bug will never return. No more regressions!

Test-Driven development

- Figure out what the code needs to do – *exactly* [note: this is the HARD part.]
- Write a test that illustrates the code does the right thing.
- The test fails because there is no code yet.
- Write just enough code to make the test pass (this can mean hard-coding the right answer)
- Refactor the code keeping all the tests passing, until the overall code and design are good enough to stand up to peer review.
- Lather, rinse, repeat.

TDD: 1. Writing a Test

- Suggested practice is either Acceptance or Unit rather than other test types.
- The test describes precisely what the code is to do (behavior).
- The test provides specific, controlled input to the code.
- The test also controls the environment so the code can operate in a completely pre-determined way.
- The test cleans up after itself (no side-effects)

TDD: 2. Writing the code

- We now have a failing test. If we don't – go back.
- Write ONLY enough code to pass the test.
- This usually means literally hard-code the expected return value.
- We do this important step because we need to test the TEST itself.
- We saw the test fail, now we want to see it pass. Now we have confidence in the test.

TDD: 3. Refactor

- We had a failing test. Now it's passing. We now know that the test itself works, so we can now use it as an indicator of real success.
- Now is the time when it is SAFE for us to go write real production code!
- Write just enough main-line code to do the *real* job at hand while keeping the entire set of tests passing at all times.
- This should be like 3 lines of code AT MOST.

TDD: 4. Refactor

- Didn't we just do this? YES.
- We have some working code, some working tests, and everything is functional.
- Here we refactor design.
- Look at the overall design, examine how things are constructed and work together.
- Look for code smells...
- Fix them NOW. Don't wait. Really. This is important! Don't do anything else until fixed.

TDD: 5. Refactor ... continued


- OK, this is getting old... Maybe... but we need to look at the entire system with respect to the changes we've introduced with the following critical views:
- Performance - Is it fast enough? Did we slow something down with the changes?
- Integration - Do we play well with others? Did any interfaces change? If so we probably have lots more work to do...
- Security - Do we have a threat model that needs updating? What kinds of security issues might we have introduced?
- Acceptance - Last but MOST IMPORTANT... Does it ALL REALLY WORK? Automated Acceptance Tests: run 'em if you 've got 'em.

TDD: 6. Lather Rinse Repeat

- We are happy with all of our code in the system. Now is the time for review.
- Get a code review (it should be really quick because it's only a small amount of code and a test or two).
- Check in the change.
- When we know our build is good, we move on to the next small task and go back to Step 1.
- Do this a lot. Really. Every 2 - 4 hours is good.



TDD Techniques

- Common Unit Test Patterns
 - Simple-test – provide input and check result for acceptance or failure.
 - Code-path – provide values that force the code through a specific path.
 - Parameter-range – pass a set of parameters that cause success, and a set that cause failure.
 - Data-driven – pass in sets of data and their expected outcomes
- 

More TDD Techniques - Mocks

- Mock Objects
 - When programming to interfaces, we get free mock objects, courtesy of frameworks like Rhino Mocks.
 - Mocks are used to remove dependencies on external systems or components.
 - Example: Mock Database
 - If we use an interface to specify how we communicate with storage, we can have a “database” and a “mock” implementation.
 - The mock will just be programmed to return certain packages of data according to which test is being run.

Mocks: Dependency Injection

- “Inversion of Control” pattern (for you pattern buffs)
- We can supply our own “version” of a dependency, to suit our own test’s needs.
- We’re not testing the dependent component, just how OUR code operates using it... so mock it out.
- As long as everyone implements the interface, the main code should not care about what the actual implementation is.
- Production code factories construct the “real” ones while test factories construct mocks.
- We can eliminate dependencies on external systems, and make our tests FAST too!

TDD on UI code

- We can use tools like Selenium RC to operate our UI code natively through a browser, under our test's control.
- Tests run under NUnit, MS Test, or anything else we can invent.
- Open a page, type in text, click a button.
- JavaScript executes, and the test can experience the code just like the user does.
- We can write a UI test, and write the code to make it pass just like any other unit test.

UI Testing Caveats

- Tests run in an actual browser, when it takes time to process things in that browser, the test must wait for it...
- Pre-Set wait times cause problems when the system is running slower than we expected
- Selenium has `WaitForEvent()` that we can use (ideally) to wait just long enough for something to happen.
- Pop-up windows are a pain to work with...

UI Testing – JavaScript

- We all write a lot of JavaScript code now for our pages. Web 2.0 and Ajax require lots of stuff to happen on the client now.
- We should be unit-testing the JavaScript code, just like all the other code we write.
- JUnit is one tool that can make unit testing JavaScript much easier.
- Don't mix your JavaScript logic with UI manipulation, separate the functionality. Test logic with JUnit, and UI code with Selenium.

TDD and the Database

- Database code needs tests also!
- We can write SQL unit tests in SQL
- I prefer to write nUnit tests for SQL code, and drive it from the unit test framework.
- Either way works...
- Independence – design tests so that one test does not affect others
- We can still use TDD to write the test first ... the paradigm still holds.

TDD and the Database ctd.

- The big trick is putting the database into a “known state” for each test, then undoing the changes at the end of the test...
- The test can wrap everything in a transaction using a try block, then roll it back in the finally() block
- Test Tables.
- Test Views.
- Test Stored Procedures and Functions.

Testing the Table/View

- Create a way to get the metadata
- “describe()” is one way to do it.
- Have the UT framework setUp() create the describe procedure, and the tearDown() remove it (zero residual side effects)
- Write tests that verify the column count, order, type, length, and nullability
- These guarantee we have the right structure.

Describe()

```
CREATE PROCEDURE describe (@table_name VARCHAR(90))
AS
SELECT DISTINCT
    sc.column_id AS ColumnNumber,
    cols.column_name AS Name,
    cols.data_type AS Type,
    ISNULL(cols.character_maximum_length, 0) AS Length,
    cols.is_nullable AS Nullable
FROM
    information_schema.columns cols
INNER JOIN sys.columns sc ON
    cols.column_name = sc.name
    AND OBJECT_NAME(sc.object_id) = @table_name
ORDER BY sc.column_id
```

Tools

- NUnit – the standard.
- TestDriven.Net – VS- integrated test runner
- Selenium RC – the best in UI testing
 - Selenium Recorder:
 - <http://seleniumrecorder.mozdev.org>
- Rhino Mocks – mock object framework
- JUnit – JavaScript Unit Testing
- ReSharper - if you use C# and Visual Studio