Test Driven Development … in a nut shell

By Phlip

A rapid immersion into the software implementation cycle that merges designing, coding, and testing by operating them backwards—testing, coding, and designing.

The Simplicity Principles

Simplicity is the opposite of complexity. It does not mean "turn your brain off" or survive with simple-minded code. That grows tangled and complex. Simple reductions enforce elegance.

After adding each tiny ability, run this checklist, in order, and refactor until your code:

0. Obeys your team's Sane Subset and style guidelines
1. Passes All Tests (*TDD starts here*)
2. Expresses Intent Clearly
3. Duplicates No Behavior
4. Minimizes Classes, Methods, and Statements.

Repeat those checks, throughout a project's lifecycle, to keep it simple and understandable at all times. Apply them in order—don't remove that last bit of duplication if the result wouldn't express intent clearly. Some languages permit more obfuscation than others do. At least put duplicated things next to each other.

Simple code is easy to test. Tested code is easy to simplify. These extremes reinforce each other. Writing minimal Test Cases forces narrow dependencies between modules. Then tests permit refactoring, and squeezing code down to a minimum. If you try to remove some detail of complexity and make a mistake, the tests that required the features that required the complexity should stop you.

The best way to go fast is to know exactly when to stop. Test-Driven Development is a framework in which programmers automate the system that triggers a red light.

Mindshare

Before 150 years ago, doctors believed evil spirits caused disease. Then anesthetics permitted doctors more time to perform surgeries. These doctors typically washed their hands only *after* operating—to clean the blood off. Patients frequently caught post-operative infections. "The operation was a success but the patient died."

Around the 1850s, one Dr. Ignaz Philippe Semmelweiss experimented by washing his hands *before* helping women give birth, instead of only afterwards. He reported a very high success rate, but doctors had yet to learn better systems to report and value experiments and statistics, and Semmelweiss used no physical model to advertise the results.

Then Louis Pasteur and Robert Koch spread the meme that diseases were tiny animals eating our bodies from the inside. This physical model explained the hand-washing experiments, backed by early research with microscopes. Doctors such as Scot Joseph Lister

added antiseptic solutions to their regimens. However, many doctors still refused to wash or sterilize their instruments—even though these procedures obviously could not harm patients. Doctors in leading and teaching roles, such as Pierre Pachet, the Professor of Physiology at Toulouse, as late as 1872, would declaim that, "Louis Pasteur's theory of germs is ridiculous fiction."

Today we recognize healthy bodies contain millions of tiny creatures living in a balance of cooperation and clumsiness. Doctors decrease the risk of upsetting this balance by always scrubbing a long list of things before even minor operations, using soaps and procedures that are now required by law.

The programming industry lives in interesting times. While many programmers still believe lack of design planning causes bugs, they have no conceivable reason not to write test code before writing the tested code. That certainly could not hurt the patient. Many believe it helps the patient, and some feel it's the best way to design the patient. Evidence from the field will quell these debates in due time—especially as Agile companies out-flank and surround the slower ones, and collapse their leaders' abilities to make decisions.

Here's the operation.

The TDD Cycle

When you develop, use a test runner, written in the same language as the Production Code, that provides some kind of visual feedback at the end of the test run. Either use a GUI-based test runner that displays a Green Bar on success or a Red Bar on failure, or a console-based test runner that displays "All tests passed" or a cascade of diagnostics, respectively.

Write failing tests, and briefly make them pass. Then refactor to improve maintainability; testing every few edits, to ensure all tested behaviors stay the same.

Engage each action in this algorithm:

- Locate the next missing **code ability** you want to add
- **Write a test** that will only pass if the ability is there
- Run the test and ensure it **fails for the correct reason**
- Perform the **minimal edit** needed to make the test pass
- When the tests pass and you get a Green Bar, **inspect the design**
- While the design (anywhere) is low quality, **refactor** it
- Only after the design is squeaky clean, **proceed to the next ability**.

That algorithm needs more interpretation. Looking closely into each **bold** item reveals a field of nuances. All are beholden to this algorithm and to the intent of each action. Each action leverages different intents; they often conflict directly with other actions' intents. Our behaviors during each action differ in opposing ways. Repeated edits with opposing intents anneal code's structure and lubricate its articulations.

A **code ability**, in this context, is the current coal face in the mine that our picks swing at. It's the location in the program where we must add new behavior, or change current behavior. Typically, this location is near the bottom of our most recent function. If we can envision one more line to add there, or one more edit to make there, then we must perforce be able to envision the complementing test that will fail without that line or edit.

"**Write a test**" can mean to write a new Test Case, and get as far as one assertion. If the new test lines assume facts not in evidence—if, for example, they reference a class or method

name that does not exist yet—run the test anyway and predict a compiler diagnostic. This test collects valid information just like any other (and it minimizes all possible reasons *not* to hit that test button). If the test inexplicably passes, you may now understand you were about to write a new class name that conflicted with an existing one.

Alternately, "**Write a test**" can mean to take an existing test function, and add new assertions to it. Tests decouple new code from its own mechanics. Many small tests, with as few as one assertion in each, often leads to clean code that assumes very little about its environment. However, when our tests address libraries, such as GUI Toolkit libraries, designed to provide dozens of coherent side-effects, we often need to end each test case with many assertions. This re-uses the test's scenario. GUI Toolkits optimize screen display time, often at the expense of preparation time in memory. Our tests, by contrast, will create many GUI objects in memory and then throw them away without displaying them. This can waste time, so many of our tests will re-use existing objects a few more times before destroying them.

Work on the assertion and the code's structure (but not behavior) until the test **fails for the correct reason**. If it passes, step thru and inspect the code to ensure you understand it, and ensure the true reason was indeed correct; then proceed to the next feature.

To fail for a correct reason, all other tests must still pass. Tests share code in fixtures, but developing a new test might tweak those. The fixtures must still perform correctly for all other tests before this one may proceed. Often one can write enough of the test that it executes, then add an assertion to the test, and switch the assertion so it accepts the code's current state. After a successful run, adjust the assertion to now expect the improved behavior.

Just before passing the test is the most efficient point in the cycle to test the test, and make sure it accurately enforces the required behavior.

Check that the diagnostic is what you expect. If the test fails for the wrong reason, your mental model of the code's situation may be wrong. If you fix a test failing for the wrong reason, your fix could be wrong. Alternately, the test may have discovered a fault. Then change the Test Case's name, and use it to fix the problem.

All this work prepares you to make that **minimal edit**. Go ahead and write that line which you have been anxious to get out of your system for the last seven paragraphs.

The **edit** is **minimal** because we live on borrowed time until the Bar turns Green. Correct behavior and happy tests come (just slightly) before design quality. We might pass the test by cloning a method and changing one line in it. If that's the minimum number of edits, do it. Or, re-write a method from scratch, even if it turns out very similar to an existing method. And often the simplest edit naturally extends a clean design that won't need refactoring, yet.

Ironically, one should work harder to ensure a test **fails for the correct reason** than one should work to make it pass! The **edit** is **minimal** because it may do anything, including lying, to pass the test. More tests will force out the lie. Practice this technique, even when you don't know the production code lies.

"**Minimal**" here means comprehensible, not sparse. Of course complete variable names, proper formatting, etc, are worth typing. But could you manually reverse the **edit**, without the Undo button? You should be able to manually find your changes and remove them. Don't keep too many edits in memory until getting back to a Green Bar, and don't tangle the new code up with the old, yet. Only change a few small areas.

If the **minimal edit** fails, and if the fault is not obvious and simple, just hit the Undo button and try again. Anything else is preferable to bug hunting, and an ounce of prevention is worth a pound of cure.

Now that we have a Green Bar, we **inspect the design**. Per the **minimal edit** principle, the most likely design flaw is duplication. So, to teach us to improve things, we spread the definition of "duplication" as wide as possible, beyond mere code cloning.

The book *Design Patterns* advises, "Abstract the thing that varies." This is the reverse way to say, "Merge the duplication that does not vary." So merging duplicated behavior together may tend to approach an object model with the quality of a Pattern.

To **refactor**, we inspect our code, and try to envision a design with fewer moving parts, less duplication, shorter methods, better identifiers, separated concerns, and deeper abstractions. Start with the code we just changed, and feel free to involve any other code in the project. But, during this step, never change functionality—only design. But we may change it anywhere in the program. This recovers the design from the **minimal edit** in only one small area. Frequent refactors irresistibly pull much of your code, and tests, towards methods with only 1 to 3 statements or operations in them.

If we cannot envision a better design, we can proceed to the next step anyway. Seek minimal edits that will either improve the design or lead to a series of related edits that might lead to an improvement. Between each edit, run all the tests. If any test fails, hit Undo and start again.

If a design contains related problems, don't refactor in order from hard to easy. Refactor from easy to easy. Start by picking the low-hanging fruit.

The level of cleanness is important here. You may have code quality that formerly would have passed as "good enough". Or you may become enamored of some new abstraction that new code *might* use, possibly months from now, or minutes. Snap out of it. The path from cruft to new features is always harder than the path from minimal elegance to new features. Fix the problems, including removing *any* speculative code, while the problems are still small.

If you see duplication, but can't imagine how to improve its design without obfuscating what it does (or can't imagine any way at all), move all the duplicating lines next to each other. This practice forms little tables, with columns that are easy to document and scan.

We may add assertions at nearly any time; while refactoring the design, and before **proceed**ing **to the next ability**. Whenever we learn something new, or realize there's something we don't know, we take the opportunity to write new assertions that express this learning, or query the code's abilities. As the TDD cycle operates, and individual abilities add up to small features, we take time to collect information from the code about its current operating parameters and boundary conditions.

Boundary conditions are the limits between defined behavior and regions where bugs might live. Set boundaries for a routine well outside the range you know production code will call it. Research "Design by Contract" to learn good strategies; these roll defined ranges of behaviors up from the lower routines to their caller routines. Within a routine, simplifying its procedure will most often remove discontinuities in its response.

Parameters between these limits now typically cause the code to respond smoothly with linear variations. The odds of bugs occurring between the boundaries are typically lower than elsewhere. For example, today's method that takes 2, 3 and 5 and returns 10, 15 and 25, respectively, tomorrow is unlikely to take 4 and return 301. Like algebraic substitutions reducing an expression, duplication removal forces out special cases.

After creating a function, other functions soon call it. Their tests engage our function too. Our tests cover every statement in a program, and they approach covering every path in a program. We add features in order of business value, so the code of highest value—written

earliest—experiences the highest testing pressure and the most test events for the remaining duration of the project. The cumulative pressure against bugs make them extraordinarily unlikely.

If you are curious, or code does something unexpected, or you receive a bug report, always write a new test. Then use what you learned to improve design, and write more tests of this category. If you treat the situation "this code does not yet have that ability" as a kind of bug, then the TDD cycle is nothing but a specialization of the principle "capture bugs with tests".

Computer Science vs. Software Engineering

Researching new, complex algorithms is Computer Science; an open-ended quest into the unknown. Software Engineering is a system to turn the results of research into money, so avoid inefficient endeavors, such as researching new, complex algorithms.

TDD is a very good tool for researching computer science, but its results are excessively sensitive to initial conditions. Programs are simple algorithms—iteration, recursion, etc. Refactoring searches the space of designs, for simple algorithms, to find a good fit. Programs reuse published algorithms behind simple interfaces, such as `std::sort<>()`. When TDD seeks a new complex algorithm, early refactors that obeyed the Simplicity Principles can lead to abstractions that prevent, not enable, a clear and efficient algorithm later on.

Strong the Dark Side Is

TDD's force can also cause problems. Used alone, it cannot gather requirements or prevent gold plating. Used incompletely, it can devolve code. If you frequently test but skip refactoring, and leave huge matching gaps in testage and code, the code will fill up with cruft. If your tests make wild assumptions about your exact environment, and if you infrequently integrate, your colleagues will blame your tests. Frequent testing aggressively reinforces these activities, and provides a very high apparent velocity.

Teach your colleagues to write tests on your code, and learn to write tests on theirs too. If they write a test that fails due to missing abilities, not faults in deployed abilities, treat the failing test as a feature request, and get it prioritized.

Test Cases

Here's a detail of a simple test in a familiar language, without the production code that makes it pass:

```
int main()
{

    Source aSource("a b\nc,  d");

    string
    token = aSource.pullNextToken();  assert("a" == token);
    token = aSource.pullNextToken();  assert("b" == token);
    token = aSource.pullNextToken();  assert("c" == token);
    token = aSource.pullNextToken();  assert("d" == token);
    token = aSource.pullNextToken();  assert(""  == token);
                                          //  EOT!
}
```

Passing the test requires objects of type `Source` to parse strings, ignoring spaces and commas. But to write that test, following the above rules, one only adds one line at a time, and makes the code pass it, before adding the next line.

This technique often generates tests with a stack of statements and assertions, like the above example. If that were production code, one should "roll it up" into a table, or something, to make it shorter and easier to understand & extend. But tests' Coding Standard differs from Production Code's. They trade parsimony for self-documentation.

That test explicitly documents a boundary condition—`End Of Text` will return a blank string `""`. Production Code, parsing a known string, wouldn't bother to call the parser one more time.

But projects larger than this should not pack all their assertions into `main()`. Split assertions up into functions called Test Cases. Each is essentially one "paragraph" of testage, following the "Triple A" format:

- **Assemble** the sample object & data: `Source aSource("a b\nc,  d");`
- **Activate** the target feature: `token = aSource.pullNextToken();`
- **Assert** the results: `assert("a" == token);`

While tests code must appear "obvious", and should not indulge in advanced architectures, tests that duplicate behavior should merge it into common methods. More than one Test Case sharing the same private Test Fixtures form a Test Suite—typically as methods of a Test Suite class. However, Test Cases must operate independent of each other, so they could execute in any order. Some Test Suites execute their cases in alphabetical order, some in their source order. If one Test Case writes a little file, for example, and programmers know which Test Case comes next, programmers must not exploit this information to re-use that file. The most common two fixtures are `setUp()` and `tearDown()`. Test Suites call these before and after calling each case, respectively. `setUp()` will initialize Test Suite member variables that each of its Cases can use, and `tearDown()` will clean up any after-effects of each case.

Some Test Fixtures are important (especially the kinds that this book will request), so they should merge into test utility modules that support many Test Suites.

So the pseudo-code to run each Test Case in a Suite is:

- Construct a Test Suite object
- Call `setUp()`
- Call the Test Case
    - Assemble
    - Act
    - Assert
- Call `tearDown()`
- Destroy the Test Suite object

Some test runners count the number of failing assertions. Test rigs in general need that ability, but Test-Driven Development does not strictly require it. A single failing assertion is cause to suspect the most recent edit.

Here's a suite that calls a fixture (`test_a_b_d()`) to Act and Assert, but not Assemble. The fixture verifies that whatever input we Assemble, the parser shall pull the tokens `a`, `b` & `d`, but `c` is always commented out:

```
struct TestTokens:  TestCase
{

void test_a_b_d(string input)
    {
    Source aSource(input);
    string
    token = aSource.pullNextToken();
    CPPUNIT_ASSERT_EQUAL("a", token);
    token = aSource.pullNextToken();
    CPPUNIT_ASSERT_EQUAL("b", token);

 // token = aSource.pullNextToken();
 // CPPUNIT_ASSERT_EQUAL("c", token);

    token = aSource.pullNextToken();
    CPPUNIT_ASSERT_EQUAL("d", token);
    token = aSource.pullNextToken();
    CPPUNIT_ASSERT_EQUAL("", token);  //  EOT!
    }

};

TEST_(TestTokens, elideComments)
{
    test_a_b_d("a b\n //c\n  d");
    test_a_b_d("a b\n // c \"neither\" \n  d");
    test_a_b_d("//\na b\n // c \"neither\" \n  d//");
…
    test_a_b_d(" // \na b\n // c \"neither\" \n  d//");
}


TEST_(TestTokens, elideStreamComments)
{
    test_a_b_d("a b\n /*c*/\n  d");
    test_a_b_d("a b\n /* c \"neither\" */\n  d");
…
    test_a_b_d("//c\na b\n // c \"neither\" \n  d/* */");
}
```

The `TEST_()` macro builds a list of Test Cases, each inheriting `TestTokens`, so a Test Runner can run them all. Without the TEST_() macro, we would need to copy the name of each test case's method into a list of tests. If we forget one, we'd get a false positive Green Bar.

The … mark indicates we fear boring the compiler less than the reader. Those tests contain many more lines that cover various situations.

When a Test Fixture, such as `test_a_b_d()`, abstracts from its sample data, such as `"a b\n //c\n  d"`, this book call the sample data a Test Resource. As Test Fixtures grow useful, their Test Resources can move into a database, with a simple user interface. That

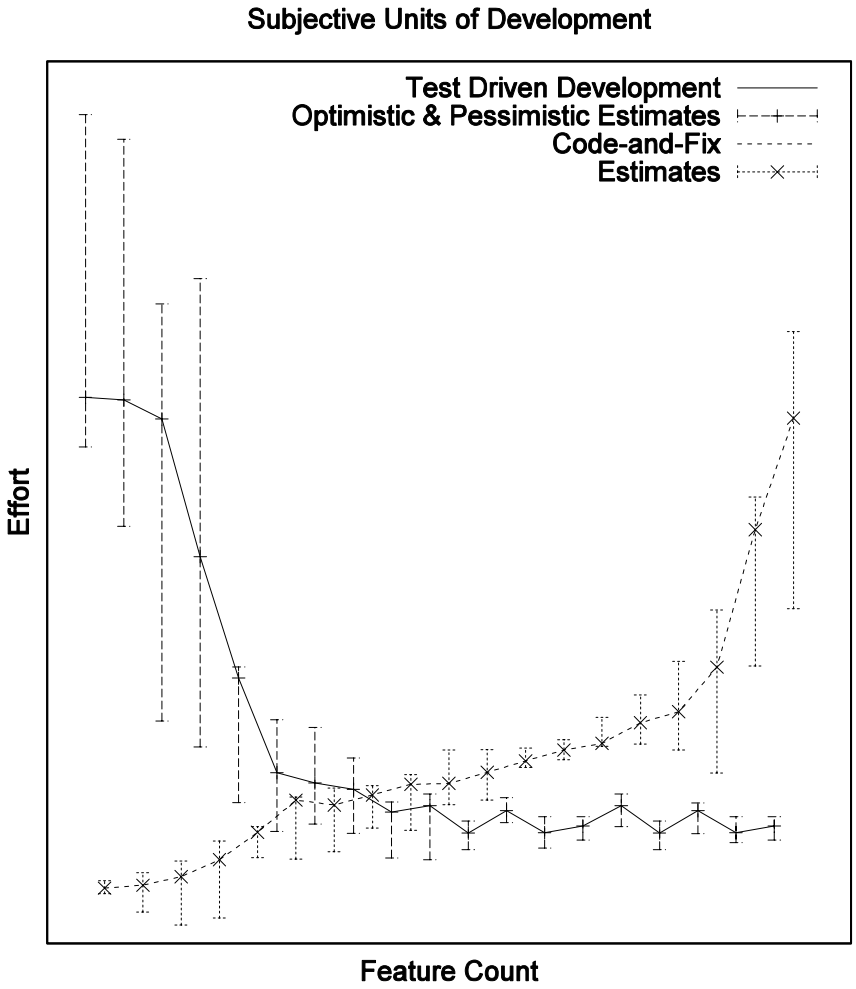
permits anyone who thinks of sample data to enter it, without programming, and see their results. Some "test resource databases" form part of a project's Customer Acceptance Tests.

Continuous Testing

The most ambitious research in TDD focuses on "Continuous Testing", where your editor chronically runs your tests, at every relevant juncture, providing Zero Button Testing. Your editor continuously reflects the testing status, and might color failing test cases pink. Less frantically, editors might test each time programmer saves the code. These techniques require more practice, and more tuning and tweaking of your editor's abilities.

Time vs. Effort

If our libraries had themselves emerged test-first, we would not need to begin our projects with so much research to build Test Fixtures that apply the OBT principles. So I must make a case, again, for how important these principles are to the rest of the project's schedule. The up-front investment will pay for itself.



The Code-and-Fix line starts easy, especially under systems like GUI Toolkits that bundle with Wizards and form painters, to write the first dozen features for you. But as you run out of

wiggle room to rapidly change code without causing obscure bugs (and as you go where such Wizards can't follow), that line trends up into unsustainable regions. Once there, each new effort adds bugs, and these take time. When they are fixed, you have the choice between letting design quality slide, or adding more bugs improving it.

Everyone has experienced code bases maintained for a few years which became too hard to change, and were thrown away. For example, Microsoft replaced MSDev.exe (Visual Studio 6) with DevEnv.exe (Visual Studio 7), instead of upgrading. The excuse "but MSDev.exe is very big, and DevEnv.exe is very different" should not matter.

Many libraries immediately make Test-Driven Development very difficult. You must invest up-front time learning how operate those libraries nearly opposite to the way their tutorials advertise. Lack of experience writing tests also afflicts this phase.

The high-effort areas of each kind of project do not just take a lot of time. They interfere with tracking and estimating. The higher a point on the chart, the wider the spread between estimates and actual times, and the higher risks of delays to research into mysteries and to concoct patches & compromises. High-risk activities cause cruft, because refactoring their excess code away similarly adds risk. Risks cause stress.

 Heroism is not sustainable

After you cross the peak, and enter the low-effort phase, your environment permits easy Flow with minimal distractions. Expect to encounter such a peak, and schedule time to cross it, for each new library. Software lifecycle books call this the "Exploration Phase", or a "Spike".

If a new library troubles you, determine if you'll need an exploration phase. Convert its sample code into a Learner Test, then see how easily that test starts answering your questions. We'll call the exploration phase for a specific library "Bootstrapping", before we learn to write the first relevant test for a given situation.

Down in your own coal mine, tests are canaries whose nervousness shows the presence of evil design vapors. However, when entering someone else's coal mine—a library you must construct tests around—expect to expend many nervous canaries!

Your occupational behavior may raise naïve concerns here. Suppose the program needs to Get a complex variable, and you first spend a lot of time learning to Set that variable—so your test can transmit sample data into the tested function, of course. The potential benefits might not be apparent.

 If it does not have a test (or a visual check), it does not exist

Adding features without their tests creates "artificial velocity" that defers the hard part of programming until later, when it grows much harder. Adding tests and features without refactoring creates artificial velocity, too.

The Agile literatures speaks of many fairly aggressive coding activities—Test Fixtures, Merciless Refactoring, Incremental Testing, Continuous Testing, Continuous Integration—etc. They all represent a significant effort to learn, and to configure your build scripts and environment.

These only burden a project as it starts. When enough tests grow and share useful fixtures, when code grows flexible, and when build scripts robustly enable a team's practices, then we will write new complex fixtures less frequently. We will refactor in tiny nudges, and rely on our environment to take care of administrative details. A mature TDD development cycle consists mostly of re-using existing fixtures, adding 2- or 3-line tests, and trivially passing them.

The feeling of propulsion and momentum is unprecedented in software engineering.