Example: xUnit

TDD requires tests. I suppose that goes almost without say, but I'll say it anyway, because it requires not just any tests, but a special kind of test. The tests must be:
Draw influence diagrams for each of these

- Easy to write for programmers—The basic value system of TDDs is that code is king. The question you want to answer at the end of the day is how much functionality did you get working? Test tools that require unfamiliar environments or languages will be less likely to be used, even if they are technically superior.
- Easy to read for programmers—Unsynchronized documentation is scarce. The tests will be more valuable if they are readable, giving an interesting second perspective on the messages hidden in the source code.
- Quick to execute—If the tests don't run fast, they won't get run. If they don't get run, they won't be valuable. If they aren't valuable, they won't continue to be written. Transitive closure—if the tests don't run fast, they won't get written. Any test tool that requires you to bring up the application is probably doomed before we start.
- Order independent—If one test breaks, we'd like the other to succeed or fail independently. I once stopped automatically testing a system about the tenth time I received a panicked call about hundreds of failing tests that turned out to have a single source of error.
- Deterministic—Tests that run one time and don't run the next give negative information. The times they run you have unwarranted confidence in the system. This implies that TDD as described here is not suitable for the synchronization parts of multi-thread programming.
- Piecemeal—We'd like to be able to write the tests a few at a time.
- Composable—We'd like to be able to run tests in any combination.
- Versionable—The source of the tests should play nicely with the rest of the source in the system.
- A priori—We should be able to write the tests before they can possibly run.
- Automatic—The tests should run with no human intervention. The cycle that kills quality is that when stress increases, errors also increase, which increases stress, which… Fully automated tools break this cycle. Every time you run a suite of tests successfully, your stress level goes down, so you are encouraged to run more tests when stress increases
- Helpful when thinking about design—Writing the tests a priori should be a learning experience. Tools that operate within the programming concepts of the system can help with design, while tools that operate the system as a black box cannot help with structuring the internals.

There are several possible testing tools we could use to write our tests. You can go through the list above and eliminate GUI-based tools, script-language-based tools, and simple-minded source code hacks on one or more counts. Such tools certainly have a place in the well-stocked testing-bag-o-tricks. However, they aren't suitable for TDD.

JUnit and its cousins are one way to negotiate this tricky, sometimes contradictory, set of constraints. The basic decisions are:

- Tests are expressed in ordinary source code
- The execution of each test is centered on an instance of a TestCase object
- Each TestCase, before it executes the test, has the opportunity to create an environment for the test, and to destroy that environment when the test finishes
- Groups of tests can be collected together, and their results of running them all will be reported collectively
- We use the language's exception handling mechanism to catch and report errors

xUnit Test-First

How, oh how, to talk about the implementation of a tool for test-driven development? Test-driven, naturally.

The xUnit architecture comes out very smoothly in Python, so I'll switch to Python for this section. Don't worry, I'll skip all the backtracking and boo-boos (which I will be making behind the scenes, never fear), so you'll be left with just the good parts. I'll also give a little commentary on Python, for those of you who haven't seen it before.

Now, writing a testing tool test-first, using itself as the tool, may seem a bit like performing brain surgery on yourself ("Don't touch those motor centers—oh, too bad, game over"). It will get weird from time to time. However, the logic of the testing framework is more complicated than the wimpy money example above. You can read this chapter as a step towards test-driven development of "real" software. You can read this chapter as a computer-sciency exercise in self-referential programming. Or you can skip it, and move on to the next chapter, which gives a design-oriented overview of xUnit.

First, we need to be able to create a TestCase and run a test method. For example: TestCase("testMethod").run(). We have a bootstrap problem. We are writing test cases to test a framework that we will be using to write the test cases. Since we don't have a framework yet, we will have to verify the operation of the first tiny step by hand. Fortunately, we are well rested and relaxed and unlikely to make mistakes, which is why we will go in teensy tiny steps, verifying everything six ways from Sunday.

We are still working test-first, of course. For our first proto-test, we need a little program that will print out true if a test method gets called, and false otherwise. If we have a test case that sets a flag inside the test method, we can print the flag after we're done and make sure it's correct. Once we have verified it manually, we can automate the process.

Python executes statements as it reads a file, so we can start with invoking the test method manually:

```
test= WasRun("testMethod")
print test.wasRun
test.testMethod()
print test.wasRun
```

We expect this to print "None" (None in Python is like null or nil, and stands for false, along with 0 and a few other objects) before the method was run, and "1" afterwards. It doesn't, because we haven't defined the class WasRun yet (test-*first*, test-*first*).

```
WasRun
class WasRun:
        pass
```

(The keyword "pass" is used when there is no implementation of a class or method.) Now we are told we need an attribute "wasRun". We need to create the attribute when

we create the instance is created (the constructor is called "__init__" for convenience). In it, we set the wasRun flag false.

```
WasRun
class WasRun:
        def __init__(self, name):
                self.wasRun= None
```

Running the file faithfully prints out "None", then tells us we need to define the method "testMethod" (wouldn't it be great if your IDE noticed this, provided you with a stub, and opened up an editor on it? Nah, too useful…)

```
WasRun
        def testMethod(self):
                pass
```

Now when we execute the file, we see "None" and "None". We want to see "None" and "1". We can get it by setting the flag in testMethod():

```
WasRun
        def testMethod(self):
                self.wasRun= 1
```

Now we get the right answer (the green bar, hooray!). Now we have a bunch of refactoring to do, but as long as we maintain the green bar, we know we have made progress.

Next we need to use our real interface, run(), instead of calling the test method directly. The test changes to:

```
test= WasRun("testMethod")
print test.wasRun
test.run()
print test.wasRun
```

The implementation we can hardwire at the moment to:

```
WasRun
        def run(self):
                self.testMethod()
```

And our test is back to printing the right values again. Lots of refactoring has this feel—separating two parts so you can work on the separately. If they go back together when you are finished, fine, if not, you can leave them separate. In this case, we expect to create a superclass TestCase, eventually, but first we have to differentiate the parts of our one example. There is probably some clever analogy with mitosis in here, but I don't know enough cellular biology to explain it.

The next step is to dynamically invoke the testMethod. If the name attribute of the instance of WasRun is the string "testMethod", then we can replace the direct call to "self.testMethod()" with "exec "self." + self.name + "()"" (the dynamic invocation of methods is called Pluggable Selector, and should be used sparingly, and only if there are no reasonable alternatives).

```
WasRun
class WasRun:
        def __init__(self, name):
                self.wasRun= None
                self.name= name
        def run(self):
                exec "self." + self.name + "()"
```

Here is another general pattern of refactoring—take code that works in one instance and generalize it to work in many by replacing constants with variables. Here the constant was hardwired code, not a data value, but the principle is the same. Test-first makes this work well by giving you running concrete examples from which to generalize, instead of having to generalize purely with reasoning.

Now our little WasRun class is doing two distinct jobs—one is keeping track of whether a method was invoked or not, the other is dynamically invoking the method. Time for a little of that mitosis action. First we create an empty TestCase superclass, and make WasRun a subclass:

```
TestCase
class TestCase:
    pass
WasRun
class WasRun(TestCase): ...
```

Now we can move the “name” attribute up to the superclass:

```
TestCase
    def __init__(self, name):
        self.name= name
WasRun
    def __init__(self, name):
        self.wasRun= None
        TestCase.__init__(self, name)
```

Finally, the run() method only uses attributes from the superclass, so it probably belongs in the superclass (I’m always looking to put the operations near the data.)

```
TestCase
    def __init__(self, name):
        self.name= name
    def run(self):
        exec "self." + self.name + "()"
```

(Between every one of these steps I run the tests to make sure I’m getting the same answer.)

We’re getting tired of looking to see that “None” and “1” are printed every time. Using the mechanism we just built, we can now write:

```
TestCaseTest
class TestCaseTest(TestCase):
    def testRunning(self):
        test= WasRun("testMethod")
        assert(not test.wasRun)
        test.run()
        assert(test.wasRun)
TestCaseTest("testRunning").run()
```

The body of the test is just the print statements turned into assertions, so you could just see what we have done as a complicated form of Extract Method.

I'll let you in on a little secret. I look at the size of the steps in the development above and it looks ridiculous. On the other hand, I tried it with bigger steps, probably six hours in all (I had to spend a lot of time looking up Python stuff), starting from scratch twice, and both times I thought I had the code working when I didn't. This is about the worst possible case for TDD, because we are trying to get over the bootstrap step. However, the promise stands—you can work absolutely confidently in little tiny steps and go fast as a result.

It is not necessary to work in such tiny steps as these. Once you've mastered TDD, you will be able to work in much bigger leaps of functionality between test cases. However, to master TDD you need to be able to work in such tiny steps when they are called for.

Reviewing, we:

- After a couple of hubris-fueled false starts, figured out how to begin with a tiny little step
- Implemented functionality hardwired, then made it more general by replacing constants with variables
- Used Pluggable Adaptor, which we promise not to use again for four months, minimum, because it makes code hard to statically analyze
- Bootstrapped our testing framework, all in tiny steps

Set the Table

When you begin writing tests, you will discover a common pattern:

1. Create some objects
2. Stimulate them
3. Check the results

While the stimulation and checking steps are unique test-to-test, the creation step is often familiar. I have a 2 and 3. If I add them, I expect 5. If I subtract them, I expect –1, if I multiply them, I expect 6. The stimulation and expected results are unique, the 2 and the 3 don't change.

If this pattern repeats at different scales (and it does), then we're faced with the question of how often do we want to create new objects. Looking back at our initial set of constraints, two constraints come into conflict:

- Performance—we would like our tests to run as quickly as possible
- Isolation—we would the success or failure of one test to be irrelevant to other tests

For performance sake, assuming creating the objects (we'll call them collectively the "fixture") is expensive, we would like to create them once and then run lots of tests. But sharing objects between tests creates the possibility of test coupling. Test coupling can have an obvious nasty effect, where breaking one test causes the next ten to fail even though the code is correct. Test coupling can have a subtle really nasty effect, where the order of tests matters. If I run A before B, they both work, but if I run B before A, then A fails. Worse, the code exercised by B is wrong, but because A ran first, the test passes.

Test coupling—don't go there. Let's assume for the moment we can make object creation fast enough. In this case, we would like to create the objects for a test every time the test runs. We've already seen a disguised form of this in WasRun, where we wanted to have a flag set to false before we ran the test. Taking steps towards this, first we need a test:

```
TestCaseTest
    def testSetUp(self):
        test= WasRun("testMethod")
        test.run()
        assert(test.wasSetUp)
```

Running this, (by adding the last line "TestCaseTest("testSetUp").run()" to our file) Python politely informs us that there is no "wasSetUp" attribute. Of course not. We haven't set it. This method should do it.

```
WasRun
    def setUp(self):
        self.wasSetUp= 1
```

It would if we were calling it. Calling setUp is the job of the TestCase, so we turn there:

```
TestCase
    def setUp(self):
        pass
    def run(self):
        self.setUp()
        exec "self." + self.name + "()"
```

That's two steps to get a test case running, which is too many in such ticklish circumstances. Perhaps it will work. We'll see. Yes, it does pass. However, if you want to learn something, try to figure out how we could have gotten the test to pass by changing no more than one method at a time.

We can immediately use our new facility to shorten our tests. First, we can simplify WasRun by setting the wasRun flag in setUp:

```
WasRun
    def setUp(self):
        self.wasRun= None
        self.wasSetUp= 1
```

We have to simplify testRunning not to check the flag before running the test. Are we willing to give up this much confidence in our code? Only if testSetUp is in place. This is a common pattern—one test can be simple iff another test is in place and running correctly.

```
TestCaseTest
    def testRunning(self):
        test= WasRun("testMethod")
        test.run()
        assert(test.wasRun)
```

We can also simplify the tests themselves. In both cases we create an instance of WasRun, exactly that fixture we were talking about earlier. We can create the WasRun in setUp, and use it in the test methods. Each test method is run in a clean instance of TestCaseTest, so there is no way the two tests can be coupled (assuming the objects don't interact in some incredibly ugly way, like setting global variables.)

```
TestCaseTest
        def setUp(self):
                self.test= WasRun("testMethod")
        def testRunning(self):
                self.test.run()
                assert(self.test.wasRun)
        def testSetUp(self):
                self.test.run()
                assert(self.test.wasSetUp)
```

Garbage collectors take care of deallocating objects for us, but tests will from time to time also need to allocate external resources in setUp(). If we want the tests to remain independent, a test that allocates external resources should release them before it is done. Am I violating the “don’t code it until you need it” rule? Yes, a little. However, symmetry also cries for a tearDown() to go with setUp(), and I have a cool testing idea in mind I want to show you, so away we go.

The simple minded way to write the test is to introduce yet another flag. All those flags are starting to bug me, and they are missing an important aspect of the methods—setUp() is called before the test method is run, and tearDown() is called afterwards. I’m going to change the testing strategy to keep a little log of what methods are called. By always appending to the log, we will preserve the order in which the methods are called.

```
WasRun
    def setUp(self):
        self.wasRun= None
        self.wasSetUp= 1
        self.log= "setUp "
```

Now we can change testSetUp() to look at the log instead of the flag:

```
TestCaseTest
    def testSetUp(self):
        self.test.run()
        assert("setUp " == self.test.log)
```

Now we can delete the wasSetUp flag. We can record the running of the test method, too:

```
WasRun
    def testMethod(self):
        self.wasRun= 1
        self.log= self.log + "testMethod "
```

This breaks testSetUp, because the actual log contains “setUp testMethod ”. We change the expected value:

```
TestCaseTest
    def testSetUp(self):
        self.test.run()
        assert("setUp testMethod " == self.test.log)
```

Now this test is doing the work of both tests, so we can delete testRunning and rename testSetUp:

```
TestCaseTest
    def setUp(self):
        self.test= WasRun("testMethod")
    def testTemplateMethod(self):
        self.test.run()
        assert("setUp testMethod " == self.test.log)
```

Unfortunately, we are only using the instance if WasRun in one place, so we have to undo our clever setUp hack:

```
TestCaseTest
    def testTemplateMethod(self):
        test= WasRun("testMethod")
        test.run()
        assert("setUp testMethod " == test.log)
```

Doing a refactoring based on a couple of early uses, then having to undo it soon after is fairly common. Some folks wait until they have three or four uses before refactoring because they don't like undoing work. I prefer to spend my thinking cycles on design, so I just reflexively do the refactorings without worrying about whether I will have to undo them immediately.

Now we are ready to implement tearDown(). Got you! We are ready to test for tearDown:

```
TestCaseTest
    def testTemplateMethod(self):
        test= WasRun("testMethod")
        test.run()
        assert("setUp testMethod tearDown " == test.log)
```

This fails. Making it work is simple:

```
TestCase
    def run(self, result):
        result.testStarted()
        self.setUp()
        exec "self." + self.name + "()"
        self.tearDown()
WasRun
    def setUp(self):
        self.log= "setUp "
    def testMethod(self):
        self.log= self.log + "testMethod "
    def tearDown(self):
        self.log= self.log + "tearDown "
```

Surprisingly, we get an error, not in WasRun, but in the TestCaseTest. We don't have a no-op implementation of tearDown() in TestCase:

```
TestCase
    def tearDown(self):
        pass
```

This time we got value out of using the same testing framework we are developing. Yippee…

Counting

I was going to implement making sure tearDown() is called regardless of exceptions during the test method. However, if we make a mistake implementing this, we won't be able to see it because we have to catch Exceptions to make the test work (I know, I just tried it, and backed it out.) In general, the order of implementing the tests is important. The best general advice I can give on picking the next test is to find a test that will teach you something but which you have confidence you can make work. If you get that test working but get stuck on the next one, consider backing up two steps. It would be great if your IDE helped you with this, where you could instantly take snapshots of the world every time all the tests ran and quickly go backwards and forwards in time.

What we would like to see is the results of running any number of tests—"5 run, 2 failed, TestCaseTest.testFooBar—ZeroDivideException, MoneyTest.testNegation—AssertionError". Then if the tests stop getting called, or results stop getting reported, at least we have a chance of catching the error. Having the framework automatically report all the test cases it knows nothing about seems a bit far-fetched, at least for the first test case.

TestCase.run() will return a TestResult object that records the results of running the test (singular for the moment, but we'll get to that.)

```
TestCaseTest
    def testResult(self):
                test= WasRun("testMethod")
        result= test.run()
        assert("1 run, 0 failed" == result.summary())
```

We'll start with a stub implementation:

```
TestResult
class TestResult:
    def summary(self):
        return "1 run, 0 failed"
```

and return a TestResult as the result of TestCase.run()

```
TestCase
    def run(self):
        self.setUp()
        exec "self." + self.name + "()"
        self.tearDown()
        return TestResult()
```

Now that the test runs, we can realize (as in "make real") the implementation of summary() a little at a time. First, we can make the number of tests run a symbol constant:

```
TestResult
    def __init__(self):
        self.runCount= 1
    def summary(self):
        return "%d run, 0 failed" %self.runCount
```

But runCount shouldn't be a constant, it should be computed by counting the number of tests run. We can initialize it to 0, then increment it every time a test is run.

```
TestResult
    def __init__(self):
        self.runCount= 0
    def testStarted(self):
        self.runCount= self.runCount + 1
    def summary(self):
        return "%d run, 0 failed" %self.runCount
```

We have to actually call this groovy new method:

```
TestCase
    def run(self):
        result= TestResult()
        result.testStarted()
        self.setUp()
        exec "self." + self.name + "()"
        self.tearDown()
        return result
```

We could turn the constant string "0" for the number of failed tests into a variable in the same way as we realized runCount. However, the tests don't demand it. So, we write another test.

```
TestCaseTest
    def testFailedResult(self):
        test= WasRun("testBrokenMethod")
        result= test.run()
        assert("1 run, 1 failed", result.summary)
```

Where:

```
WasRun
    def testBrokenMethod(self):
        raise Exception
```

The first thing we notice is that we aren't catching the exception thrown by WasRun.testBrokenMethod. We would like to catch the exception and make a note in the result that the test failed. We'll put this test on the shelf for the moment.

We'll write a smaller grained test to be sure that if we note a failed test, we print out the right results.

```
TestCaseTest
    def testFailedResultFormatting(self):
        result= TestResult()
        result.testStarted()
        result.testFailed()
        assert("1 run, 1 failed" == result.summary())
```

These are the messages we expect to send to the result. If we can get the summary correct, then our problem is reduced to how to get these messages sent. Once they are sent, we expect the whole thing to work. The implementation is to keep a count of failures:

```
TestResult
    def __init__(self):
        self.runCount= 0
        self.errorCount= 0
    def testFailed(self):
        self.errorCount= self.errorCount + 1
```

With the count correct (which I suppose we could have tested for, if we were taking teensy, weensy, tiny steps, but I won't bother, the coffee has kicked in now), we can print correctly:

```
TestResult
    def summary(self):
        return "%d run, %d failed" %(self.runCount, self.failureCount)
```

Now we expect if we call testFailed() correctly, we will get the expected answer. When do we call it? When we catch an exception in the test method:

```
TestCase
    def run(self):
        result= TestResult()
        result.testStarted()
        self.setUp()
        try:
            exec "self." + self.name + "()"
        except:
            result.testFailed()
        self.tearDown()
        return result
```

There is a subtlety hidden inside this method. The way it is written, if a disaster happens during setUp(), the exception won't be caught. That can't be what we mean—we want our tests to run independently of each other. However, we need another test before we can change the code (I taught my oldest daughter TDD as her first programming style and she thinks the browser won't work for new code unless there is a test broken. The rest of us have to muddle through reminding ourselves to write the tests.) That next test and its implementation are left as an exercise for the reader (sore fingers, again.)

How Suite It Is

We can't leave xUnit without visiting TestSuite. The end of our file is looking pretty ratty:

```
print TestCaseTest("testTemplateMethod").run().summary()
print TestCaseTest("testResult").run().summary()
print TestCaseTest("testFailedResultFormatting").run().summary()
print TestCaseTest("testFailedResult").run().summary()
```

Duplication is always a bad thing, unless you look at it as motivation to find the missing design element. What we would like here is the ability to compose tests and run them together (working hard to make them run in isolation doesn't do us much good if we only ever run one at a time). Another good reason to implement TestSuite is that it gives us a pure example of Composite—we want to be able to treat single tests and groups of tests exactly the same from a programmatic perspective.

We would like to be able to create a TestSuite, add a few tests to it, then get collective results from running it.

```
TestCaseTest
    def testSuite(self):
        suite= TestSuite()
        suite.add(WasRun("testMethod"))
        suite.add(WasRun("testBrokenMethod"))
        result= suite.run()
        assert("2 run, 1 failed" == result.summary())
```

Implementing the add() method just adds tests to a list:

```
TestSuite
class TestSuite:
    def __init__(self):
        self.tests= []
    def add(self, test):
        self.tests.append(test)
```

The run method is a bit of a problem. We want a single TestResult to be used by all the tests that run. Therefore, we should write:

```
TestSuite
    def run(self):
        result= TestResult()
        for test in tests:
            test.run(result)
        return result
```

However, one of the main constraints on Composite is that the collection has to respond to the same messages as the individual items. If we add a parameter to TestCase.run(), we have to add the same parameter to TestSuite.run(). I can think of three alternatives:

- Use Python's default parameter mechanism. Unfortunately, the default value is evaluated at compile time, not run time, and we don't want to be reusing the same TestResult
- Split the method into two parts, one which allocates the TestResult and the other which runs the test given a TestResult
- Allocate the TestResults in the caller

I can't think of good names for the two parts of the method, so we will allocate the TestResults in the callers. This pattern is called Collecting Parameter.

```
TestCaseTest
    def testSuite(self):
        suite= TestSuite()
        suite.add(WasRun("testMethod"))
        suite.add(WasRun("testBrokenMethod"))
        result= TestResult()
        suite.run(result)
        assert("2 run, 1 failed" == result.summary())
```

This solution has the advantage that run() now has no explicit return:

```
TestSuite
    def run(self, result):
        for test in tests:
            test.run(result)
TestCase
    def run(self, result):
        result.testStarted()
        self.setUp()
        try:
            exec "self." + self.name + "()"
        except:
            result.testFailed()
        self.tearDown()
```

Now we can clean up the invocation of the tests at the end of the file:

```
suite= TestSuite()
suite.add(TestCaseTest("testTemplateMethod"))
suite.add(TestCaseTest("testResult"))
suite.add(TestCaseTest("testFailedResultFormatting"))
suite.add(TestCaseTest("testFailedResult"))
suite.add(TestCaseTest("testSuite"))
result= TestResult()
suite.run(result)
print result.summary()
```

There is substantial duplication here, which we could eliminate if we had a way of constructing a suite automatically given a test class. However, first we have to fix the 4 failing tests (they use the old no-argument run interface):

```
TestCaseTest
    def testTemplateMethod(self):
        test= WasRun("testMethod")
        result= TestResult()
        test.run(result)
        assert("setUp testMethod tearDown " == test.log)
    def testResult(self):
        test= WasRun("testMethod")
        result= TestResult()
        test.run(result)
        assert("1 run, 0 failed" == result.summary())
    def testFailedResult(self):
        test= WasRun("testBrokenMethod")
        result= TestResult()
        test.run(result)
        assert("1 run, 1 failed" == result.summary())
    def testFailedResultFormatting(self):
        result= TestResult()
        result.testStarted()
        result.testFailed()
        assert("1 run, 1 failed" == result.summary())
```

Notice that now each test allocates a result, exactly the problem solved by setUp(). We can simplify the tests (at the cost of making them a little more difficult to read), by creating the TestResult in setUp():

```
TestCaseTest
    def setUp(self):
        self.result= TestResult()
    def testTemplateMethod(self):
        test= WasRun("testMethod")
        test.run(self.result)
        assert("setUp testMethod tearDown " == test.log)
    def testResult(self):
        test= WasRun("testMethod")
        test.run(self.result)
        assert("1 run, 0 failed" == self.result.summary())
    def testFailedResult(self):
        test= WasRun("testBrokenMethod")
        test.run(self.result)
        assert("1 run, 1 failed" == self.result.summary())
    def testFailedResultFormatting(self):
        self.result.testStarted()
        self.result.testFailed()
        assert("1 run, 1 failed" == self.result.summary())
    def testSuite(self):
        suite= TestSuite()
        suite.add(WasRun("testMethod"))
        suite.add(WasRun("testBrokenMethod"))
        suite.run(self.result)
        assert("2 run, 1 failed" == self.result.summary())
```

All those extra “self.”s are a bit ugly, but that’s Python. If it was an object language, the self would be assumed and references to global variables would require qualification. Instead, it is a scripting language with object support (excellent object support, to be sure) added, so global reference is implied and referring to self is explicit.

xUnit Retrospective

If the time comes for you to implement your own testing framework, the above sequence can serve as your guide. The details of the implementation are not nearly as important as the test cases. If you can support a set of test cases like the ones above, you can write tests that are isolated and composeable, and you will be on your way to being able to develop test-first.

xUnit has been ported to more than 30 languages at this writing. Your language is likely to already have an implementation. There are a couple of reasons for implementing it even if there is a version already available:

- Mastery—The spirit of xUnit is simplicity. Martin Fowler said, “Never in the annals of software engineering was so much owed by so many to so little code.” Some of the implementations have gotten a little complicated for my taste. Rolling your own will give you a tool over which you have a feeling of mastery.
- Exploration—Say you are faced with a new programming language. By the time you have implemented the first 8-10 test cases, you will have explored many of the facilities you will be using in daily programming

When you begin using xUnit, you will discover a big difference between assertions that fail and other kinds of errors while running tests—assertion failures consistently take much longer to debug. Because of this, most implementations of xUnit distinguish between failures—meaning assertion failures—and errors. The GUIs present them differently, often with the errors on top.

JUnit declares a simple Test interface that is implemented by both TestCase and TestSuite. If you want your tests to be runnable by JUnit tools, you can implement the Test interface, too.

```
public interface Test {
    public abstract int countTestCases();
    public abstract void run(TestResult result);
}
```

Languages with optimistic typing don’t even have to declare their allegiance to an interface, they can just implement the operations. If you write a test scripting language, Script can implement countTestCases() to return 1 and run to notify the TestResult on failure and you can run your scripts along with the ordinary TestCases.

Section III: Patterns

What follows are the “greatest hits” patterns for TDD. Some of the patterns are TDD tricks, some are design patterns, and some are refactorings.

The goal in these patterns is not to be comprehensive. If you want to understand testing, design patterns, or refactoring you will have to go elsewhere for mastery. If you are not familiar with these topics, there is enough here to get you going. If you are familiar with one of these topics, the patterns here will show you how the topics play with TDD.

Patterns for Test-Driven Development

Test *n.*

How do you test your software? Write an automated test.

No programmers release even the tiniest change without testing, except the very confident and the very sloppy. I'll assume that if you've gotten this far, your're neither. While you may test your changes, testing changes is not the same as *having* tests. Why does a test that runs automatically feel different than poking a few buttons and looking at a few answers on the screen?

(What follows in an influence diagram, *a la* Gerry Weinberg's Quality Software Management. An arrow between nodes means an increase in the first node implies an increase in the second. An arrow with a circle means an increase in the first node implies a decrease in the second.)

What happens when the stress level rises?

Figure 1 has Stress negatively connected to Testing negatively connected to Errors positively connected to Stress.

This is a positive feedback loop. The more stress you feel, the less testing you will do. The less testing you do, the more errors you will make. The more errors you make, the more stress you feel. Rinse and repeat.

How do you get out of such a loop? Either introduce a new element, replace one of the elements, or change the arrows. In this case we'll replace "testing" with "automated testing".

Figure 2 has Stress positively connected to Automated Testing negatively connected to Errors and Stress, and Errors positively connected to Stress.

"Did I just break something else with that change?" When I have automated tests, when I start to feel stress I run the tests. "No, the tests are all still green." The more stress I feel, the more I run the tests. Running the tests immediately gives me a good feeling, and reduces the number of errors I make, which further reduces the stress I feel.

"We don't have time to run the tests. Just release it!" The second picture isn't guaranteed. If the stress level rises high enough, it breaks down. However, with the automated tests you have a chance to choose your level of fear.

Isolated Test

How should the running of tests affect each other? Not at all.

When I was a young programmer, long long ago when we had to dig our own bits out of the snow and carry heavy buckets of them bare-footed back to our cubicles leaving bloody little footprints for the wolves to follow… Sorry, just reminiscing. My first experience of automated tests was having a set of GUI-based tests (you know, record