Test-Driven Development By Example

Kent Beck, Three Rivers Institute

Notes to reviewers:

- Are there diagrams that would help orient the examples?
- Section I: Money Example is now completely re-written. Does the new style work better than the old one? I have noticed several changes—shorter chapters, more careful adherence to "the rules", less American-isms. Better, worse, same?
- Please suggest your favorite glossary items
- How does the new how/why refactoring format work? Do I need an example, or is it sufficient to point people to Martin's book?

Publically availabe at
http://groups.yahoo.com/group/testdrivendevelopment/files/

To Do

Glossary

To-do lists, chapter hooks, and reviews for xUnit

Convert to Frame (sigh…)

Finish missing patterns

Bold source code changes

Run Money through Jester and a coverage tool

Deadend in Money. Where, oh where?

Random Thoughts

Another mental picture—programming is like exploring a dark house. You go from room to room to room. Writing the test is like turning on the light. Then you can avoid the furniture and save your shins (the clean design resulting from refactoring). Then you're ready to explore the next room.

I need an adjective which means "can be made to work in one step". Atomic? Achievable? Progressive?

At the different stages of the cycle you are solving different problems, so the aesthetics change: Write a test-what should the API be? Make it compile—do as little as possible to satisfy the compiler. Make it run—get back to green so you have confidence. Refactor—remove duplication to prepare for the next test.

Interesting error. I had two tests, one USD->USD and one USD->GBP. If I had kept the two assertions in the same test I wouldn't have gone off the rails. What's the rule there? When do you add assertions to existing tests and when do you write a new test?

Splitting into orthogonal dimensions didn't happen in either example. What up with that? I thought that was such an important technique. Maybe that's what "isolate change" is really about, and taking smaller steps than I usually do results in making progress along one dimension before having to make progress in the other.

More orientation material at the beginning of the example chapters—UML, lists of tests running, to do list

Brian Marick on test-first tests as tests?

TDD as a gesture—technical, political, aesthetic, emotional. Relationship to other practices.

Since people are likely to read the chapters in the example one or two at a time, it is important to provide context at the beginning of each one-UML, maybe a list of the test cases that are running at the beginning.

Test coverage. Use data structures that make special cases go away- iterators, number-like numbers.

Balancing reasoning and testing. Every one of those reasoning steps is subject to error, which adds risks. Replacing each and every reasoning step with a concrete test is extremely expensive (impossibly expensive, really). There is some tradeoff. Maybe that's part of being a TDD—being aware of tradeoffs and intelligently choosing the crossover point for this particular situation.

Assuming a certain geekoid value system—you want to do well by doing good (or vice versa). That is, you like clean code, you enjoy the feeling of designing and building well, and you want to be seen to be successful by managers and customers.

Code aesthetics. For any given set of test cases, we are trying to minimize a complicated cost function—number of classes, number of methods, number of unique selectors, number of arguments, complexity of flow of control, visibility of methods

and members, coupling and cohesion. Either that or we are trying to give ourselves a glimpse through a tiny keyhole at an eternal realm of dynamic order.

Once and only once—part of philosophical underpinnings. Also emergence. How about the attractor stuff Phlip talks about? Make it run, make it right, make it fast. Concrete to abstract, existential to universal.

“Clever” play on words in the title. Test-driven development *is* development by example. The book is also structured by example.

One paragraph of my history with TDD (preface?)

What exactly is the relationship between test cases and design patterns? Test cases and refactorings?

Tease apart “test-driven development”.

This book is another example of my overall quest to find fundamental rules underlying effective software development. I’m looking for a theory in the physics sense, but I always take something I enjoy doing, subject it to a microscopic examination, and see if following simple rules enhances my enjoyment. Software patterns in general, SBPP, XP, and now this all have the same form.

Contents

Preface

Test-Driven Development:

- Don’t write a line of new code unless you first have a failing automated test.
- Eliminate duplication.

Two simple rules, but they generate complex individual and group behavior. Some of the technical implications are:

- You must design organically, with running code providing feedback between decisions
- You must write your own tests, since you can’t wait twenty times a day for someone else to write a test
- Your development environment must provide rapid response to small changes
- Your designs must consist of many highly cohesive, loosely coupled components, just to make testing easy

The two rules imply an order to the tasks of programming:

- Red—write a little test that doesn’t work, perhaps doesn’t even compile at first
- Green—make the test work quickly, committing whatever sins necessary in the process
- Refactor—eliminate all the duplication created in just getting the test to work

Red/green/refactor. The TDDs mantra.

Assuming for the moment that such a style is possible, it might be possible to dramatically reduce the defect density of code and make the subject of work crystal clear to all involved. If so, writing only code demanded by failing tests also has social implications:

- If the defect density can be reduced enough, QA can shift from reactive to proactive work
- If the number of nasty surprises can be reduced enough, project managers can estimate accurately enough to involve real customers in daily development
- If the topics of technical conversations can be made clear enough, programmers can work in minute-by-minute collaboration instead of daily or weekly collaboration
- Again, if the defect density can be reduced enough, we can have shippable software with new functionality every day, leading to new business relationships with customers

So, the concept is simple, but what’s my motivation? Why would a programmer take on the additional work of writing automated tests? Why would a programmer work in tiny little steps when their mind is capable of great soaring swoops of design? Fear.

Fear

Test-driven development (TDD) is a way of managing fear during programming. I don't mean fear in a bad way, pow widdle prwogwammew needs a pacifiew, but fear in the legitimate, this-is-a-hard-problem-and-I-can't-see-the-end-from-the-beginning sense. If pain is nature's way of saying "Stop!", fear is nature's way of saying "Be careful." The problem is that fear has a host of other effects:

- Makes you tentative
- Makes you grumpy
- Makes you want to communicate less
- Makes you shy from feedback

None of these effects are helpful when programming, especially when programming something hard. So, how can you face a difficult situation and

- Instead of being tentative, begin learning concretely as quickly as possible.
- Instead of clamming up, communicate more clearly.
- Instead of avoiding feedback, search out helpful, concrete feedback.
- (You'll have to work on grumpiness on your own.)

Imagine programming as turning a crank to pull a bucket of water from a well. When the bucket is small, a free-spinning crank is fine. When the bucket is big and full of water, you're going to get tired before the bucket is all the way up. You need a ratchet mechanism to enable you to rest between bouts of cranking. The heavier the bucket, the closer the teeth need to be on the ratchet.

The tests in test-driven development are the teeth of the ratchet. Once you get one test working, you know it is working, now and forever. You are one step closer to having everything working than you were when the test was broken. Now get the next one working, and the next, and the next. By analogy, the tougher the programming problem, the less ground should be covered by each test.

Readers of Extreme Programming Explained will notice a difference in tone between XP and TDD. TDD isn't an absolute like Extreme Programming. XP says, "Here are things you must be able to do to be prepared to evolve further." TDD is a little fuzzier. TDD is an awareness of the gap between decision and feedback during programming, and control over that gap. You could have only application-level tests and be doing TDD. The gap between decision and feedback would be large—days, even—but for extremely skilled programmers that might be enough feedback.

TDD gives you control over feedback. When you are cruising along in overdrive and the snow begins to fall, you can shift into 4WD Low and keep making progress. When the road clears, you can up shift and away you go.

That said, most people who learn TDD find their programming practice changed for good. “Test Infected” is the phrase Erich Gamma coined to describe this shift. You might find yourself writing more tests earlier, and working in smaller steps than you ever dreamed would be sensible. On the other hand, some programmers learn TDD and go back to their earlier practices, reserving TDD for special occasions when ordinary programming isn’t making progress.

There are certainly programming tasks that can’t be driven primarily by tests (or at least, not yet). Security software and concurrency, for example, are two topics where TDD has no obvious application. The ability to write concrete, deterministic, automated tests is a prerequisite for applying TDD.

Once you are finished reading this book, you should be ready to:

- Start simply
- Write automated tests
- Refactor to add design decisions one at a time

This book is organized into three sections.

1. An example of writing typical model code using TDD. The example is one I got from Ward Cunningham years ago, and have used many times since, multi-currency arithmetic. In it you will learn to write tests before code, grow a design organically, and fail with grace (there is a dead end in the example which I swear I put in for pedagogical purposes.)
2. An example of testing more complicated logic, including reflection and exceptions, by developing a framework for automated testing. This example also serves to introduce you to the xUnit architecture that is at the heart of many programmer-oriented testing tools. In the second example you will learn to work in even smaller steps than in the first example, including the kind of self-referential hooha beloved of computer scientists.
3. Patterns for TDD. Included are patterns for the deciding what tests to write, how to write tests using xUnit, and a greatest hits selection of the design patterns and refactorings used in the examples.

I wrote the examples imagining a pair programming session. For me, joking and banter are signs of respect between peers, and an important outlet for tension. If you like looking at the map before wandering around, you may want to go straight to the patterns in section 3 and use the examples as illustrations. If you prefer just wandering around and then looking at the map to see where you’ve been, try reading the examples through, refering to the patterns when you want more detail about a technique, then using the patterns as a reference.

Acknowledgements

Thanks to all my many brutal and opinionated reviewers. I take full responsibility for the contents, but this book would have been much less readable and useful without their help. In the order in which I typed them in, they were: Steve Freeman, Frank

Westphal, Ron Jeffries, Dierk König, Edward Hieatt, Tammo Freese, Jim Newkirk, Johannes Link, Manfred Lange, Steve Hayes, Alan Francis, Jonathan Rasmusson, Shane Clauson, Simon Crase, Kay Pentecost, Murray Bishop, Ryan King, Bill Wake,

To all of the programmers I've test-driven code with, I certainly appreciate your patience going along with what was a pretty crazy sounding idea, especially in the early years. I've learned far more from you all than I could ever think of myself. Not wishing to offend everyone else, but Massimo Arnoldi, Ralph Beattie, Ron Jeffries, and last but certainly not least Erich Gamma stand out in my memory as partners from whom I've learned much.

My life as a real programmer started for me with patient mentoring from and continuing collaboration with Ward Cunningham. Sometimes I see TDD as an attempt to give any programmer, working in any environment, the sense of comfort and intimacy we had with our Smalltalk environment and our Smalltalk programs. There is no way to sort out the source of ideas once two people have shared a brain. If you assume all the good ideas here are Ward's, you won't be far wrong.

It is a bit of a cliché to recognize the sacrifices a family makes once one of its members catches the peculiar mental affliction that results in a book. It is a cliché because family sacrifices are as necessary to book writing as paper. To my children who waited breakfast until I could finish a chapter, and most of all to my wife who spent two months saying everything three times, my profoundest and least adequate thanks.

Finally, to the unknown author of the book which I read as a weird 12-year-old that suggested you type in the expected output tape from a real input tape, then code until the actual results matched the expected result, thank you, thank you, thank you.

Story Time

Tell the WyCash multi-currency story, perhaps with a time line “0900 – management asks for the impossible, 0910 – etc.”

- WyCash was a system for managing portfolios of fixed income securities. Initially, it had been written for the US market, and …
- One day they needed multi-currency arithmetic.
- Ward invents Money and MoneyBag.
- At the end of the day, the system was working.

This was a moment business crave. Investing one day of a few programmers’ time multiplied the value of WyCash, already worth tens of millions of dollars, by several times. In fact, the business of software is making a bunch of bets like this and holding on long enough for one to pay off. The only reason sensible business people put up with the eccentricity, unreliability, and general orneriness of programmers is because occasionally the pony-tailed freaks spin straw into gold.

Programmers, too, live for this kind of moment. Creativity, courage, and spark of genius combined to accomplish the impossible. Moments like this write a story that will keep the programmer in late-night conference beer for years, if told properly.

The users experienced magic, too. Handling multiple currencies was, to them, a perfectly simple, understandable request. The users probably had the experience of making such perfectly simple, understandable requests of programmers before, and of receiving bizarre replies. “At least six months. But if you’d told me about this a year ago it would have been easy.” Instead, they made a simple request and a few days later, they got what they wanted.

Moments that multiply the value of a project are a combination of method, motive and opportunity:

- Method—Ward and the WyCash team needed to have constant experience growing the design of the system little-by-little, so the mechanics of the transformation were well practiced.
- Motive—Ward and team had to understand clearly from the business the importance of making WyCash multi-currency, and to have the courage to start such a seemingly impossible task.
- Opportunity— The combination of comprehensive, confidence-generating tests; a well-factored program; and a programming language that made it possible to isolate design decisions meant that there were few sources of error, and those errors were easy to identify.

You can’t control whether you ever get the motive to multiply the value of your project by spinning technical magic. Method and opportunity, however, are entirely under your control. Ward and his team created method and opportunity by a combination of superior talent, experience, and discipline. Does this mean that if you

are not one of the ten best software engineers on the planet and you don't have a wad of cash in the bank so you can tell your boss to take a hike, you're going to take the time to do this right, that such moments are forever beyond your reach?

No. You absolutely can place your projects in a position for you to work magic, even if you are a programmer with ordinary skills and you sometimes buckle under and take shortcuts when the pressure builds. Test-driven development is a set of techniques any programmer can follow, that encourage simple designs and test suites that inspire confidence. If you are a genius, you don't need these rules. If you are a dolt, the rules won't help. For the vast majority of us in between, though, following these two simple rules can lead us to work much closer to our potential:

- Always write a failing automated test before you write any code
- Always remove duplication

How exactly to do this, the subtle gradations in applying these rules, and the lengths to which can push these two simple rules are the topic of this book. We'll start with the object Ward created in his moment of inspiration—multi-currency money.

Section I: Money Example

In this section we will develop typical model code completely driven by tests (except when we slip, purely for educational purposes). My goal is for you to see the rhythm of test-driven development:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

The surprises are likely to be:

- How each test can cover a small increment of functionality
- How small and ugly the changes can be to make the new tests run
- How often the tests are run
- How many teensy tiny steps make up the refactorings

Money Example

We'll start with the object Ward created at WyCash, multi-currency money. Suppose we have a report like this:

Instrument	*Shares*	*Price*	*Total*
IBM	1000	25	25000
GE	400	100	40000
		Total:	75000

To make a multi-currency report, we need to add currencies:

Instrument	*Shares*	*Price*	*Total*
IBM	1000	25 USD	25000 USD
Novartis	400	150 CHF	40000 CHF
		Total:	75000 USD

We also need to specify exchange rates:

From	*To*	*Rate*
CHF	USD	1.5

What behavior will we need to produce the revised report? Put another way, what is the set of tests which, when passed, will demonstrate the presence of code we are confident will compute the report correctly?

- We need to be able to add amounts in two different currencies and convert the result given a set of exchange rates.
- We need to be able to multiply an amount (price per share) by a number (number of shares) and receive an amount.

We'll make a to-do list to remind us what all we need to do, keep us focused, and tell us when we are finished:

```
To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 * 2 = $10
```

What object do we need first? Trick question. We don't start with objects, we start with tests (I keep having to remind myself of this, so I will pretend you are as dense as I am).

Try again. What test do we need first? Looking at the list, that first test looks complicated. Start small or not at all. Multiplication, how hard could that be? We'll work on that first.

When we write a test, we imagine the perfect interface for our operation. We are telling ourselves a story about how the operation will look from the outside. Our story

won't always come true, but better to start from the best possible API and work backwards than to make things complicated, ugly, and "realistic" from the get go.

Here's a simple example of multiplication:

```
public void testMultiplication() {
        Dollar five= new Dollar(5);
        five.times(2);
        assertEquals(10, five.amount);
}
```

(I know, I know, public fields, side-effects, integers for monetary amounts and all that. Small steps. We'll make a note of the stinkiness and move on. We have a failing test and we want it to go green as quickly as possible.)

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 * 2 = $10
Make "amount" private
Dollar side-effects?
Money rounding?

The test we just typed in (I'll explain where and how we type it in later, when we talk more about JUnit) doesn't even compile. That's easy enough to fix. What's the least we can do to get it to compile, even if it doesn't run? We have four compile errors:

- No class "Dollar"
- No constructor
- No method "times(int)"
- No field "amount"

Let's take them one at a time (I always search for some numerical measure of progress). We can get rid of one error by defining the class Dollar:

Dollar

```
class Dollar
```

Now we need the constructor, but it doesn't have to do anything just to get the test to compile:

Dollar

```
Dollar(int amount) {
}
```

We need a stub implementation of times(). Again we'll do the least work possible just to get the test to compile:

Dollar

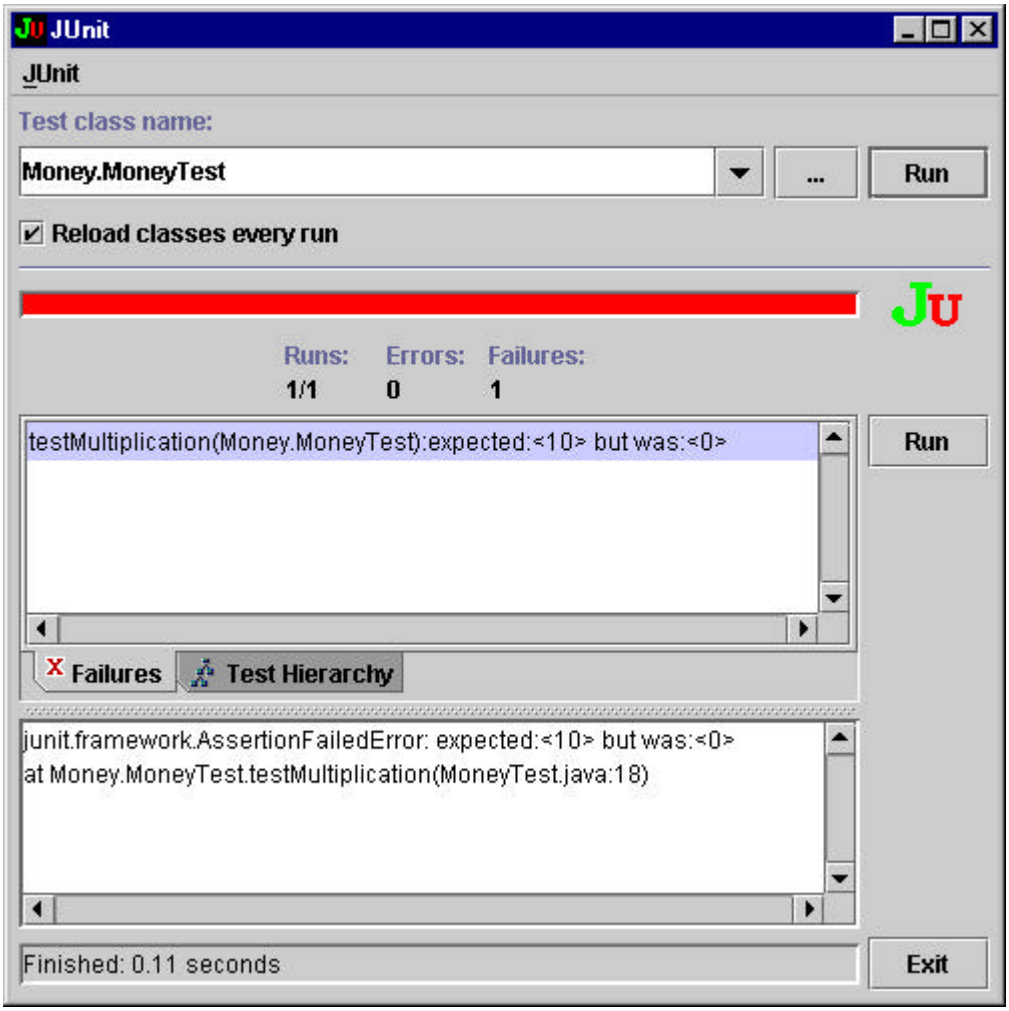
```
void times(int multiplier) {
}
```

Finally, we need an amount field:

Dollar

```
int amount;
```

Now we can run the test and watch it fail.



You are seeing the dreaded red bar. Our testing framework (JUnit, in this case) has run the little snippet of code we started with, and noticed that although we expected "10" as a result, we saw "0". Sadness.

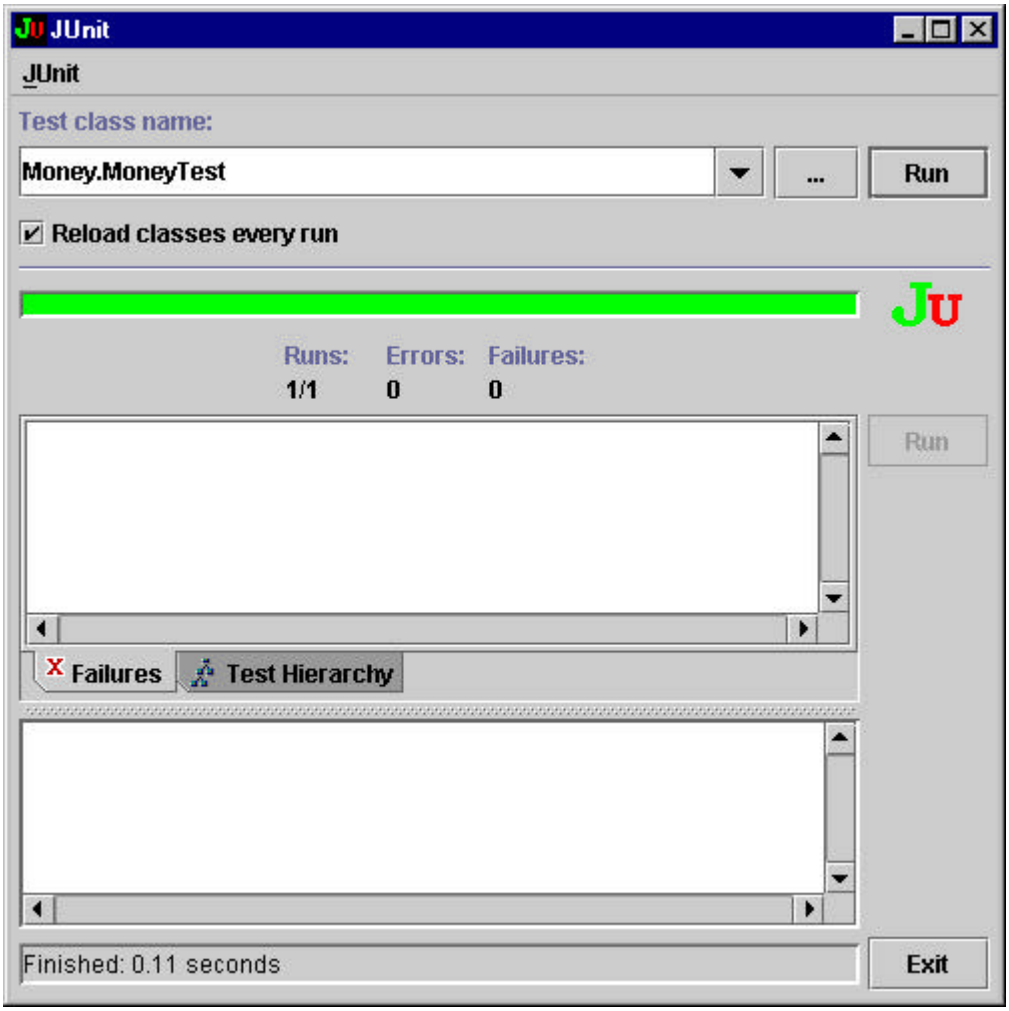
No, no. Failure is progress. Now we have a concrete measure of failure. That's better than just vaguely knowing we are failing. Our programming problem has been transformed from "give me multi-currency" to "make this test work, and then make the rest of the tests work." Much simpler. Much smaller scope for fear. We can make this test work.

You probably aren't going to like the solution, but the goal right now is not to get the perfect answer, the goal is to pass the test. We'll make our sacrifice at the altar of truth and beauty later.

Here's the smallest change I could imagine that would cause our test to pass:

```
Dollar
    int amount= 10;
```

Now we get the green bar, fabled in song and story.



Oh joy, oh rapture! Not so fast, hacker boy (or girl). The cycle isn't complete. There are very few inputs in the world that will cause such a limited, such a smelly, such a naïve implementation to pass. We need to generalize before we move on. Remember, the cycle is:

1. Add a little test
2. Run all tests and fail
3. Make a little change
4. Run the tests and succeed
5. Refactor to remove duplication

Sidebar: Dependency and Duplication

Steve Freeman pointed out that the problem with the test and code as it sits is not duplication (which I have not yet pointed out to you, but I promise to as soon as this digression is over.) The problem is the dependency between the code and the test—you can't change one without changing the other. Our goal is to be able to write another test that "makes sense" to us, without having to change the code, something that is not possible with the current implementation.

Dependency is the key problem in software development at all scales. If you have details of one vendor's implementation of SQL scattered throughout the code and you decide to change to another vendor, you will discover that your code is dependent on the database vendor. You can't change the database without changing the code.

If dependency is the problem, duplication is the symptom. Duplication most often takes the form of duplicate logic—the same conditional expression appearing in multiple places in the code. Objects are excellent for abstracting away the duplication of logic.

Duplication also appears in data. Symbolic constants were introduced to eliminate dependencies between code and magic numbers. Once you use a symbolic constant, your code is no longer dependent on the actual number. You can change the number all you want without having to touch the code.

Unlike most problems in life, where eliminating the symptoms only makes the problem pop up elsewhere in worse form, eliminating duplication in programs eliminates dependency. That's why the second rule appears in TDD. By eliminating duplication before we go on to the next test, we maximize our chance of being able to get the next test running with one and only one change.

Now back to your regularly scheduled puzzling example.

But where is the duplication? Usually you see duplication between two pieces of code. Here the duplication is between the data in the test and the data in the code. Don't see it? How about if we write?

Dollar

```
int amount= 5 * 2;
```

That "10" had to come from somewhere. We did the multiplication in our heads so fast we didn't even notice. The "5" and "2" are now in two places, and we must ruthlessly eliminate duplication before moving on.

There isn't a single step that will eliminate the 5 and 2. However, what if we move the setting of the amount from object initialization to the times() method?

Dollar

```
int amount;

void times(int multiplier) {
    amount= 5 * 2;
}
```

The test still passes, the bar stays green. Happiness is still ours.

Do these steps seem too small to you? Remember, TDD is not about taking teensy tiny steps, it's about being able to take teensy tiny steps. Would I code day-to-day with steps this small? No. But when things get the least bit weird, I'm glad I can.

Defensiveness aside, where were we? Ah, yes, we were getting rid of duplication between the test code and the working code. Where can we get a 5? That was the value passed to the constructor, so if we save it in the amount variable:

```
Dollar
    Dollar(int amount) {
        this.amount= amount;
    }
```

we can use it in times():

```
Dollar
    void times(int multiplier) {
        amount= amount * 2;
    }
```

The value of the parameter “multiplier” is 2, so we can substitute the parameter for the constant:

```
Dollar
    void times(int multiplier) {
        amount= amount * multiplier;
    }
```

To demonstrate our thorough-going knowledge of Java syntax, we will want to use the “*=” operator (which does, it must be said, reduce duplication):

```
Dollar
    void times(int multiplier) {
        amount *= multiplier;
    }
```

We can now mark off the first test as done:

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
Make “amount” private
Dollar side-effects?
Money rounding?

Next we’ll take care of those strange side effects. First, though, let’s review. We:

- Made a list of the tests we knew we needed to have working
- Told a story with a snippet of code about how we wanted to view one operation
- Ignored the details of JUnit for the moment
- Made the test compile with stubs
- Made the test run by committing horrible sins
- Gradually generalized the working code, replacing constants with variables
- Added items to our to-do list rather than addressing them all at once

Degenerate Objects

We got one test working, but in the process we noticed something strange—when we perform an operation on a Dollar, the Dollar changes. I would like to be able to write:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

I can't imagine a clean way to get this test working. After the first call to times(), five isn't five any more, it's really ten. If, however, we return a new object from times(), we can multiply our original five bucks all day and never have it change. We are changing the interface of Dollar when we make this change, so we have to change the test. That's okay. Our guesses about the right interface are no more likely to be perfect than our guesses about the right implementation.

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(10, product.amount);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

The new test won't compile until we change the declaration of Dollar.times():

```
Dollar
    Dollar times(int multiplier) {
        amount *= multiplier;
        return null;
    }
```

Now the test compiles, but it doesn't run. Progress! Making it run requires that we return a new Dollar with the correct amount:

```
Dollar
    Dollar times(int multiplier) {
        return new Dollar(amount * multiplier);
    }
```

In the last chapter when we made a test work we started with a bogus implementation and gradually made it real. Here, we typed in what we thought was the right implementation and prayed while the tests ran (short prayers, to be sure, because running the test takes a few milliseconds.) Because we got lucky and the test ran, we can cross off another item:

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side-effects?~~
Money rounding?

These are two of the three strategies for quickly getting to green:

- Fake It—return a constant and gradually replace constants with variables until you have the real code
- Obvious Implementation—type in the real implementation

When I use TDD in practice, I commonly shift between these two modes of implementation. When everything is going smoothly and I know what to type, I put in obvious implementation after obvious implementation (running the tests all the time to ensure that what's obvious to me is still obvious to the computer). As soon as I get an unexpected red bar, I back up, shift to faking implementations, and refactor to the right code. When my confidence is back, I go back to obvious implementations.

There is a third style of driving development, triangulation, which we will demonstrate in the next chapter. However, to review, we:

- Translated a design objection (side effects) into a test case that failed because of the objection
- Got the code to compile quickly with a stub implementation
- Made the test work by typing in what seemed like the right code

The translation of a feeling (disgust at side effects) into a test (multiply the same Dollar twice) is a common theme of TDD. The longer I do this, the better able I am to translate my aesthetic judgements into tests. When I can do this, my design discussions become much more interesting. First we can talk about whether the system should work like *this* or like *that*. Once we decide on the correct behavior, we can talk about the best way of achieving that behavior. We can speculate about truth and beauty all we want over beers, but while we are programming we can leave airy-fairy discussions behind and talk cases.

Equality for All

If I have an integer and I add 1 to it, I don't expect the original integer to change, I expect to use the new value. Objects usually don't behave that way. If I have a Contract and I add one to its coverage, the Contract's coverage should change (yes, yes, subject to all sorts of interesting business rules which do *not* concern us here.)

We can use objects as values, as we are using our Dollar now. The pattern for this is Value Object. One of the constraints on Value Objects is that the values of the instance variables of the object never change once they have been set in the constructor.

There is one huge advantage to using value objects—you don't have to worry about aliasing problems. Say I have one Check and I set its amount to $5, and then I set another Check's amount to the same $5. Some of the nastiest bugs in my career have come when changing the first Check's value inadvertently changed the second Check's value. This is aliasing.

When you have value objects, you don't have to worry about aliasing. If I have $5, I am guaranteed that it will always and forever be $5. If someone wants $7, they have to make an entirely new object.

One implication of Value Object is all operations must return a new object, as we saw in the previous chapter. Another implication is that value objects should implement equals(), since one $5 is pretty much as good as another:

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side-effects?~~
Money rounding?
Equals()

If you use Dollars as the key to a hash table, you have to implement hashCode() if you implement equals(). We'll put that in the list, too, and get to it when it's a problem.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side-effects?~~
Money rounding?
Equals()
HashCode()

You aren't thinking about the implementation of equals(), are you? Good. Me neither. After snapping the back of my hand with a ruler, I'm thinking about how to test equality. First, $5 should equal $5:

```
public void testEquality() {
        assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

The bar turns obligingly red. The fake implementation is to just return true:

```
Dollar
    public boolean equals(Object object) {
        return true;
    }
```

You and I both know that “true” is really “5 == 5” which is really “amount == 5” which is really “amount == dollar.amount”. If I went through these steps, though, I wouldn’t be able to demonstrate the third and most conservative implementation strategy, triangulation.

If two receiving stations at a known distance from each other can both measure the direction of a radio signal, there is enough information to calculate the range and bearing of the signal (if you remember more trigonometry than I do, anyway.) This calculation is called triangulation.

By analogy, when we triangulate, we only generalize code when we have two more more examples. We briefly ignore the duplication between test and model code. When the second example demands a more general solution, then and only then do we generalize.

So, to triangulate we need a second example. How about $5 != $6?

```
public void testEquality() {
        assertTrue(new Dollar(5).equals(new Dollar(5)));
        assertFalse(new Dollar(5).equals(new Dollar(6)));
}
```

Now we need to generalize equality:

```
Dollar
    public boolean equals(Object object) {
        Dollar dollar= (Dollar) object;
        return amount == dollar.amount;
    }
```

We could have used triangulation to drive the generalization of times(), also. If we had $5 x 2 = $10 and $5 x 3 = $15 we would no longer have been able to return a constant.

Triangulation feels funny to me. I only use it when I am completely unsure of how to refactor. If I can see how to eliminate duplication between code and tests and create the general solution, I just do it. Why would I need to write another test to give me permission to write what I probably could have written the first time?

However, when the design thoughts just aren’t coming, triangulation gives you a chance to think about the problem from a slightly different direction. What axes of variability are you trying to support in your design? Make some of the them vary and the answer may become clearer.

So, equality is done for the moment. (What about comparing with null and comparing with other objects? Add those to the list.)

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object

Now that we have equality, we can directly compare Dollars to Dollars. That will let us make amount private, as all good instance variables should be. Reviewing the above, though, we:

- Noticed that our design pattern (Value Object) implied an operation
- Tested for that operation
- Implemented it simply
- Didn't refactor immediately, but instead tested further
- Refactored to capture the two cases at once

Privacy

Now that we have defined equality, we can use it to make out tests more “speaking”. Conceptually, the operation Dollar.times() should return a Dollar whose value is the value of the receiver times the multiplier. Our test doesn’t exactly say that:

```
public void testMultiplication() {
        Dollar five= new Dollar(5);
        Dollar product= five.times(2);
        assertEquals(10, product.amount);
        product= five.times(3);
        assertEquals(15, product.amount);
}
```

We can rewrite the first assertion to compare Dollars to Dollars.

```
public void testMultiplication() {
        Dollar five= new Dollar(5);
        Dollar product= five.times(2);
        assertEquals(new Dollar(10), product);
        product= five.times(3);
        assertEquals(15, product.amount);
}
```

That looks better, so we rewrite the second assertion, too:

```
public void testMultiplication() {
        Dollar five= new Dollar(5);
        Dollar product= five.times(2);
        assertEquals(new Dollar(10), product);
        product= five.times(3);
        assertEquals(new Dollar(15), product);
}
```

Now the temporary variable “product” isn’t helping much, so we can inline it:

```
public void testMultiplication() {
        Dollar five= new Dollar(5);
        assertEquals(new Dollar(10), five.times(2));
        assertEquals(new Dollar(15), five.times(3));
}
```

This test speaks to us more clearly, as if it were an assertion of truth, not a sequence of operations.

With these changes to the test, Dollar is now the only class using its “amount” instance variable, so we can make it private:

```
Dollar

    private int amount;
```

And we can cross another item off the list:

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object

Notice that we have opened ourselves up to a risk. If the test for equality fails to accurately check that equality is working, the test for multiplication could also fail to accurately check that multiplication is working. That is a risk you actively manage in TDD. We aren't striving for perfection. By saying everything two ways, as both code and tests, we hope to reduce our defects enough to move forward with confidence. From time to time our reasoning will fail us and a defect will slip through. When that happens, we learn our lesson about the test we should have written and move on. The rest of the time we go forward boldly under our bravely flapping green bar (my bar doesn't actually flap, but one can dream.)

Reviewing, we:

- Used functionality just developed to improve a test
- Noticed that if two tests fail at once we're sunk
- Proceeded in spite of the risk
- Used new functionality in the object under test to reduce coupling between the tests and the code

Franc-ly Speaking

How are we going to approach the first test on that list?

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object

That's the test that's most interesting. It still seems to be a big leap. I'm not sure I can write a test that I can implement in one little step. A pre-requisite seems to be having an object like Dollar, but to represent Francs. If we can get Francs working like Dollars work now, we'll be closer to being able to write and run the mixed addition test.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
5 CHF * 2 = 10 CHF

We can copy and edit the Dollar test:

```
public void testFrancMultiplication() {
        Franc five= new Franc(5);
        assertEquals(new Franc(10), five.times(2));
        assertEquals(new Franc(15), five.times(3));
}
```

(Aren't you glad we simplified the test in the last chapter? That made our job here easier. Isn't it amazing how often things work out like this in books? I didn't actually plan it that way this time, but I won't make promises for the future.)

What short step will get us to a green bar? Copying the Dollar code and replacing "Dollar" with "Franc".

Stop. Hold on. I can hear the aesthetically inclined among you sneering and spitting. Copy and paste reuse? The death of abstraction? The killer of clean design?

If you're upset, take a cleansing breath. In…hold…out. There. Now, our cycle has different phases (they go by quickly, often in seconds, but they are phases.):

1. Write a test
2. Make it compile

3. Make it run
4. Remove duplication

The different phases have different purposes. They call for different styles of solution, different aesthetic viewpoints. The first three phases need to go by quickly, so we get to a known state with the new functionality. You can commit any number of sins to get there, because speed trumps design, just for that brief moment.

Now I'm worried. I've given you a license to abandon all the principles of good design. Off you go to your teams—"Kent says all that design stuff doesn't matter." Halt. The cycle is not complete. A three legged horse can't gallop. The first three steps of the cycle won't work without the fourth. Good design at good times. Make it run, make it right.

There, I feel better. Now I'm sure you won't show anyone except your partner your code until you've removed the duplication. Where were we? Ah, yes. Violating all the tenets of good design in the interest of speed (penance for our sin will occupy the next several chapters.)

Franc

```
class Franc {
    private int amount;

    Franc(int amount) {
        this.amount= amount;
    }

    Franc times(int multiplier) {
        return new Franc(amount * multiplier);
    }

    public boolean equals(Object object) {
        Franc franc= (Franc) object;
        return amount == franc.amount;
    }
}
```

Because the step to running code was so short, we were even able to skip the "make it compile" step.

Now we have duplication galore, and we have to eliminate it before writing our next test. We'll start by generalizing equals(). However, we can cross off an item, even though we have to add two more:

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
Common equals

Reviewing, we:

- Couldn't tackle a big test, so we invented a small test that represented progress
- Wrote the test by shamelessly duplicating and editing
- Even worse, made the test work by copying and editing model code wholesale
- Promised ourselves we wouldn't go home until the duplication was gone

Equality for All, Redux

There is a fabulous sequence in Wallace Stegner's *Crossing to Safety* where he describes a character's workshop. Every item is perfectly in place, the floor is spotless, all is order and cleanliness. The character, however, has never made anything. "Preparing has been his life's work. He prepares, then he cleans up." (This is also the book that sent me audibly blubbering in business class on a trans-Atlantic 747, so please read it with caution.)

We have avoided this trap in the last chapter. We actually got a new test case working. However, we sinned mightily so we could do it quickly. Now it is time to clean up.

One possibility is to make one of our classes extend the other. I tried it, and it hardly saves any code at all. Instead, we are going to find a common superclass for the two classes (I tried this already, too, and it works out great, although it will take a while.)

Add a picture here

What if we had a Money class to capture the common equals code? We can start small:

Money

```
class Money
```

All the tests still run (not that we could possibly have broken anything, but that's a good time to run the tests anyway.)

If Dollar extends Money, that can't possibly break anything.

Dollar

```
class Dollar extends Money {
    private int amount;
}
```

Can it? No, the tests still all run. Now we can move the "amount" instance variable up to Money:

Money

```
class Money {
    protected int amount;
}
```

Dollar

```
class Dollar extends Money {
}
```

The visibility has to change from private to protected so the subclass can still see it. (If we'd wanted to go even slower we could have declared the field in Money in one step, and then removed it from Dollar in a second step. I'm feeling bold.)

Now we can work on getting the equals() code ready to move up. First we change the declaration of the temporary variable:

```
Dollar
    public boolean equals(Object object) {
        Money dollar= (Dollar) object;
        return amount == dollar.amount;
    }
```

All the tests still run. Now we change the cast:

```
Dollar
    public boolean equals(Object object) {
        Money dollar= (Money) object;
        return amount == dollar.amount;
    }
```

To be communicative, we should also change the name of the temporary variable:

```
Dollar
    public boolean equals(Object object) {
        Money money= (Money) object;
        return amount == money.amount;
    }
```

Now we can move it from Dollar to Money:

```
Money
    public boolean equals(Object object) {
        Money money= (Money) object;
        return amount == money.amount;
    }
```

Now we need to eliminate Franc.equals(). First we notice that the tests for equality don't cover comparing Francs to Francs. Our sins in copying code are catching up with us. Before we change the code, we'll write the tests that should have been there in the first place.

You will often be TDDing in code that doesn't have adequate tests (at least for the next decade or so). When you don't have enough tests, you are bound to come across refactorings that aren't supported by tests. You could make a refactoring mistake and the tests would all still run. What do you do?

As here, write the tests you wish you had. If you don't, you will eventually break something while refactoring. Then you'll get bad feelings about refactoring and stop doing it so much. Then your design will deteriorate. You'll be fired. Your dog will leave you. Your teeth will go bad. So, to keep your teeth healthy, retroactively test before refactoring.

Fortunately, here the tests are easy to write. We just copy the tests for Dollar:

```
public void testEquality() {
        assertTrue(new Dollar(5).equals(new Dollar(5)));
        assertFalse(new Dollar(5).equals(new Dollar(6)));
        assertTrue(new Franc(5).equals(new Franc(5)));
        assertFalse(new Franc(5).equals(new Franc(6)));
}
```

More duplication, two lines more! We'll atone for these sins, too.

Tests in place, we can have Franc extend Money:

Franc

```
class Franc extends Money {
    private int amount;
}
```

We can delete Franc's field "amount" in favor of the one in Money:

Franc

```
class Franc extends Money {
}
```

Franc.equals() is almost the same as Money.equal(). If we make them precisely the same, we can delete the implementation in Franc without changing the meaning of the program. First we change the declaration of the temporary variable:

Franc

```
    public boolean equals(Object object) {
        Money franc= (Franc) object;
        return amount == franc.amount;
    }
```

Then we change the cast:

Franc

```
    public boolean equals(Object object) {
        Money franc= (Money) object;
        return amount == franc.amount;
    }
```

Do we really have to change the name of the temporary variable to match the superclass? I'll leave it up to your conscience… Okay, we'll do it:

Franc

```
    public boolean equals(Object object) {
        Money money= (Money) object;
        return amount == money.amount;
    }
```

Now there is no difference between Franc.equals() and Money.equals(), so we delete the redundant implementation in Franc. And run the tests. They run.

What happens when we compare Francs and Dollars? We'll get to that in the next chapter. Reviewing what we did here, we:

- Stepwise moved common code from one class (Dollar) to a superclass (Money)
- Made a second class (Franc) also a subclass
- Reconciled two implementations (equals()) before eliminating the redundant one

Apples and Oranges

The thought struck us at the end of the last chapter—what happens when we compare Francs and Dollars? We dutifully turned our dreadful thought into an item on our to-do list. But we just can't get it out of our heads. What does happen?

```
public void testEquality() {
        assertTrue(new Dollar(5).equals(new Dollar(5)));
        assertFalse(new Dollar(5).equals(new Dollar(6)));
        assertTrue(new Franc(5).equals(new Franc(5)));
        assertFalse(new Franc(5).equals(new Franc(6)));
        assertFalse(new Franc(5).equals(new Dollar(5)));
}
```

It fails. Dollars are Francs. Before you Swiss shoppers get all excited, let's try to fix the code. The equality code needs to check that it isn't comparing Dollars and Francs. We can do this right now by comparing the class of the two objects—two Moneys are equal only if their amounts and classes are equal.

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount && getClass().equals(money.getClass());
}
```

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Francs != Dollars~~

Using classes like this in model code is a bit smelly. We would like to use a criteria that made sense in the domain of finance, not the domain of Java objects. However, we don't currently have anything like a currency, and this doesn't seem like sufficient reason to introduce one, so this will have to do for the moment.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Francs != Dollars~~
Currency?

Now we really need to get rid of the common times() code, so we can get to mixed currency arithmetic. Before we do, though, we can review our grand accomplishments of this chapter:

- Took an objection that was bothering us and turned it into a test
- Made the test run a reasonable, but not perfect way (getClass())
- Decided not to introduce more design until we had a better motivation

Makin' Objects

The two implementations of times() are remarkably similar:

Franc

```
Franc times(int multiplier) {
    return new Franc(amount * multiplier);
}
```

Dollar

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

We can take a step towards reconciling them by making them both return a Money:

Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier);
}
```

Dollar

```
Money times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

The next step forward is not obvious. The two subclasses of Money aren't doing enough work to justify their existence, so we would like to eliminate them. However, we can't do it with one big step, because that wouldn't make a very effective demonstration of TDD.

Okay, we would be one step closer to eliminating the subclasses if there were fewer references to the subclasses directly. We can introduce a Factory Method in Money that returns a Dollar. We would use it like this:

```
public void testMultiplication() {
    Dollar five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

The implementation creates and returns a Dollar:

Money

```
static Dollar dollar(int amount) {
    return new Dollar(amount);
}
```

But we want references to Dollars to disappear, so we need to change the declaration in the test:

```
public void testMultiplication() {
    Money five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Our compiler politely informs us that times() is not defined for Money. We aren't ready to implement it just yet, so we make Money abstract (I suppose we should have done that to begin with, shouldn't we?) and declare Money.times():

Money

```
abstract class Money
    abstract Money times(int multiplier);
```

Now we can change the declaration of the factory method:

Money

```
static Money dollar(int amount) {
    return new Dollar(amount);
}
```

The tests all run, so at least we haven't broken anything. We can now use our factory method everywhere in the tests:

```
public void testMultiplication() {
    Money five = Money.dollar(5);
    assertEquals(Money.dollar(10), five.times(2));
    assertEquals(Money.dollar(15), five.times(3));
}
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
    assertFalse(new Franc(5).equals(Money.dollar(5)));
}
```

We are now in a slightly better position than before. No client code knows that there is a subclass called Dollar. By de-coupling the tests from the existence of the subclasses, we have given ourselves freedom to change inheritance without affecting any model code.

Before we go blindly changing the testFrancMultiplication, we notice that it isn't testing any logic that isn't tested by the test for Dollar multiplication. If we delete the test, will we lose any confidence in the code? Still a little, so we leave it there. But it's suspicious.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Francs != Dollars~~
Currency?
Delete testFrancMultiplication?

```
public void testEquality() {
        assertTrue(Money.dollar(5).equals(Money.dollar(5)));
        assertFalse(Money.dollar(5).equals(Money.dollar(6)));
        assertTrue(Money.franc(5).equals(Money.franc(5)));
        assertFalse(Money.franc(5).equals(Money.franc(6)));
        assertFalse(Money.franc(5).equals(Money.dollar(5)));
}

public void testFrancMultiplication() {
        Money five = Money.franc(5);
        assertEquals(Money.franc(10), five.times(2));
        assertEquals(Money.franc(15), five.times(3));
}
```

The implementation is just like Money.dollar():

```
Money
    static Money franc(int amount) {
        return new Franc(amount);
    }
```

We'll get rid of the duplication of times() next. For now, reviewing, we:

- Took a step towards eliminating duplication by reconciling the signatures of two variants of the same method (times())
- Moved at least a declaration of the method to the common superclass
- Decoupled test code from the existence of concrete subclasses by introducing factory methods
- Noticed that when the subclasses disappear some tests will be redundant, but took no action

Times We're Livin' In

What is there on our list that might help us eliminate those pesky useless subclasses?

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Francs != Dollars~~
Currency?
Delete testFrancMultiplication?

What about currency? What would happen if we introduced the notion of currency?

How do we want to implement currencies at the moment? I blew it, again. Before the ruler comes out, I'll rephrase. How do we want to test for currencies at the moment? There. Knuckles saved. For the moment.

We may want to have complicated objects representing currencies, with flyweight factories to ensure we create no more objects than we really need. However, for the moment Strings will do:

```
public void testCurrency() {
    assertEquals("USD", Money.dollar(1).currency());
    assertEquals("CHF", Money.franc(1).currency());
}
```

First we declare currency() in Money:

```
Money
    abstract String currency();
```

Then we implement it in both subclasses:

```
Franc
    String currency() {
        return return "CHF";
    }
Dollar
    String currency() {
        return return "USD";
    }
```

We want the same implementation to suffice for both classes. We could store the currency in an instance variable and just return the variable. (I'll start going a little

faster with the refactorings in the instance of time. If I go too fast, please tell me to slow down. Oh, wait, this is a book. Perhaps I just won't speed up much.)

Franc

```
private String currency;
Franc(int amount) {
    this.amount = amount;
    currency = "CHF";
}
String currency() {
    return currency;
}
```

We can do the same with Dollar:

Dollar

```
private String currency;
Dollar(int amount) {
    this.amount = amount;
    currency = "USD";
}
String currency() {
    return currency;
}
```

Now we can push up the declaration of the variable and the implementation of currency(), since they are identical:

Money

```
protected String currency;
String currency() {
    return currency;
}
```

If we move the constant strings "USD" and "CHF" to the static factory methods, the two constructors will be identical and we can create a common implementation.

First we'll add a parameter to the constructor:

Franc

```
Franc(int amount, String currency) {
    this.amount = amount;
    this.currency = "CHF";
}
```

This breaks the two callers of the constructor:

```
Money
    static Money franc(int amount) {
        return new Franc(amount, null);
    }
Franc
    Money times(int multiplier) {
        return new Franc(amount * multiplier, null);
    }
```

Wait a minute! Why is Franc.times() calling the constructor instead of the factory method? Do we want to make this change now, or will we wait? The dogmatic answer is that we'll wait, not interrupting what we're doing. The answer in my practice is that I will entertain a brief interruption, but only a brief one, and I will never interrupt an interruption (rule thanks to Jim Coplien). To be realistic, we'll fix times() before proceeding:

```
Franc
    Money times(int multiplier) {
        return Money.franc(amount * multiplier);
    }
```

Now the factory method can pass "CHF":

```
Money
    static Money franc(int amount) {
        return new Franc(amount, "CHF");
    }
```

And finally we can assign the parameter to the instance variable:

```
Franc
    Franc(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }
```

I'm feeling defensive again about taking such teeny-tiny steps. Am I recommending that you actually work this way? No. I'm recommending that you be able to work this way. What I actually did just now was I worked in larger steps and made a stupid mistake half way through. I unwound a minute's worth of changes, shifted to a lower gear, and did it over with little steps. I'm feeling better now, so we'll see if we can make the analogous change to Dollar in one swell foop:

```
Money
    static Money dollar(int amount) {
        return new Dollar(amount, "USD");
    }
Dollar
    Dollar(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }
    Money times(int multiplier) {
        return Money.dollar(amount * multiplier);
    }
```

And it worked first time. Whew!

This is the kind of tuning you will be doing constantly with TDD. Are the teeny-tiny steps feeling restrictive? Take bigger steps. Are you feeling a little unsure? Take smaller steps. TDD is a steering process—a little this way, a little that way. There is not right step size, now and forever.

The two constructors are now identical, so we can push up the implementation:

```
Money
    Money(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }
Franc
    Franc(int amount, String currency) {
        super(amount, currency);
    }
Dollar
    Dollar(int amount, String currency) {
        super(amount, currency);
    }
```

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Francs != Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

We're almost ready to push up the implementation of times() and eliminate the subclasses, but first, to review, we:

- Were a little stuck on big design ideas, so we worked on something small we noticed earlier
- Reconciled the two constructors by moving the variation to the caller (the factory method)
- Interrupted a refactoring for a little twist (using the factory method in times())
- Repeated an analogous refactoring (doing to Dollar what we just did to Franc) in one big step
- Pushed up the identical constructors

The Root of all Evil

When we are done with this chapter we will have a single class to represent Money. The two implementations of times() are close, but not identical.

```
Franc
    Money times(int multiplier) {
        return Money.franc(amount * multiplier);
    }
Dollar
    Money times(int multiplier) {
        return Money.dollar(amount * multiplier);
    }
```

There's not an obvious way to make them identical. Sometimes you have to go backwards to go forwards, a little like a Rubik's Cube. What happens if we inline the factory methods? (I know, I know, we just called the factory method for the first time just one chapter ago. Frustrating, isn't it?)

```
Franc
    Money times(int multiplier) {
        return new Franc(amount * multiplier, "CHF");
    }
Dollar
    Money times(int multiplier) {
        return new Dollar(amount * multiplier, "USD");
    }
```

In Franc, though, we know that the currency instance variable is always "CHF", so we can write:

```
Franc
    Money times(int multiplier) {
        return new Franc(amount * multiplier, currency);
    }
```

That works. The same trick words in Dollar:

```
Dollar
    Money times(int multiplier) {
        return new Dollar(amount * multiplier, currency);
    }
```

We're almost there. Does it really matter whether we have a Franc or a Money? We could carefully reason about this given our knowledge of the system. However, we have clean code and we have tests that give us confidence. Rather than apply minutes of suspect reasoning, we can just ask the computer by making the change and running the tests. In teaching TDD I see this situation all the time—excellent programmers spending 5-10 minutes reasoning about a question that can be answered by the computer in 15 seconds. Without the tests you have no choice, you have to reason.

With the tests you can decide whether an experiment would answer the question faster. Sometimes you should just ask the computer.

To run our experiment we change Franc.times() to return a Money:

Franc

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

The compiler tells us that Money must be a concrete class:

Money

```
class Money
    Money times(int amount) {
        return null;
    }
```

And we get a red bar. The error message says, “expected:<Money.Franc@31aebf> but was: <Money.Money@478a43>”. Not as helpful as we would perhaps like. We can define toString() to give us a better error message:

Money

```
public String toString() {
    return amount + " " + currency;
}
```

Whoa! Code without a test? Can you do that? We could certainly have written a test for toString() before we coded it. However:

- We are about to see the results on the screen
- Since toString() is only used for debug output, the risk of it failing is low
- We already have a red bar, and we’d prefer not to write a test when we have a red bar

Exception noted.

Now the error message says: “expected:<10 CHF> but was:<10 CHF>”. That’s a little better, but still confusing. We got the right data in the answer, but the class was wrong—Money instead of Franc. The problem is in our implementation of equals():

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount && getClass().equals(money.getClass());
}
```

We really should be checking to see that the currencies are the same, not that the classes are the same.

We’d prefer not to write a test when we have a red bar. However, we are about to change real model code, and we can’t change model code without a test. The conservative course is to back out the change that caused the red bar so we’re back to

green. Then we can change the test for equals(), fix the implementation, and re-try the original change.

This time, we'll be conservative (sometimes I plough ahead and write a test on a red, but not while the children are awake.)

Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier, currency);
}
```

That gets us back to green. The situation that we had was a Franc(10, "CHF") and a Money(10, "CHF") that were reported to be not equal, even though we would like them to be equal. We can use exactly this for our test:

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

It fails, as expected. The equals() code should compare currencies, not classes:

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount && currency().equals(money.currency());
}
```

Now we can return a Money from Franc.times() and still pass the tests:

Franc

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

Will the same will work for Dollar.times()?

Dollar

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

Yes! Now the two implementations are identical, so we can push them up.

Money

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
~~Common times~~
~~Francs != Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

The two subclasses have only their constructors, so we can replace references to the subclasses by references to the superclass without changing the meaning of the code. First Franc:

Franc

```
static Money franc(int amount) {
    return new Money(amount, "CHF");
}
```

Then Dollar:

Dollar

```
static Money dollar(int amount) {
    return new Money(amount, "USD");
}
```

Since there are no references to Dollar, we can delete it. Franc still has one reference, in the test we just wrote. Looking at the equality test:

```
public void testEquality() {
        assertTrue(Money.dollar(5).equals(Money.dollar(5)));
        assertFalse(Money.dollar(5).equals(Money.dollar(6)));
        assertTrue(Money.franc(5).equals(Money.franc(5)));
        assertFalse(Money.franc(5).equals(Money.franc(6)));
        assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

it looks like we have the cases for equality well covered, too well covered, actually. We can delete the third and fourth assertions since they duplicate the exercise of the first and second assertions:

```
public void testEquality() {
        assertTrue(Money.dollar(5).equals(Money.dollar(5)));
        assertFalse(Money.dollar(5).equals(Money.dollar(6)));
        assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

The test we wrote forcing us to compare currencies instead of classes only makes sense if there are multiple classes. Since we are trying to eliminate the Franc class, a test to ensure that the system works if there is a Franc class is a burden, not a help. Away testDifferentClassEquality() goes, and Franc goes with it.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
~~Dollar/Franc duplication~~
~~Common equals~~
~~Common times~~
~~Francs != Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

Similarly, there are separate tests for dollar and franc multiplication. Looking at the code, we can see there is no difference in the logic at the moment based on the currency (there was a difference when there were two classes). We can delete testFrancMultiplication() without losing any confidence in the behavior of the system.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side-effects?~~
Money rounding?
~~Equals()~~
HashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
~~Dollar/Franc duplication~~
~~Common equals~~
~~Common times~~
~~Francs != Dollars~~
~~Currency?~~
~~Delete testFrancMultiplication?~~

Multiplication in place, we are ready to tackle addition. First, to review, we:

- Reconciled two methods (times()) by first inlining the methods they called and then replacing constants with variables
- Wrote a toString() without a test just to help us debug
- Tried a change (returning Money instead of Franc) and let the tests tell us whether it worked

- Backed out an experiment and wrote another test. Making the test work made the experiment work.
- Finished gutting subclasses and deleted them
- Eliminated tests that made sense with the old code structure but were redundant with the new code structure

Addition, Finally

It's a new day, and our to-do list is getting a little cluttered, so we'll copy the pending items to a fresh list:

```
To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
```

(I like physically copying to-do items to a new list. If there are lots of little items, I tend to just take care of them rather than copy them. Little stuff that otherwise might build up gets taken care of just because I'm lazy. Play to your strengths.)

I'm not sure how to write the story of the whole addition, so we'll start with a simpler example—$5 + $5 = $10.

```
To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 + $5 = $10
```

```
public void testSimpleAddition() {
    Money sum= Money.dollar(5).plus(Money.dollar(5));
    assertEquals(Money.dollar(10), sum);
}
```

We could fake the implementation by just return "Money.dollar(10)", but the implementation seems obvious. We'll try:

```
Money
    Money plus(Money addend) {
        return new Money(amount + addend.amount, currency);
    }
```

(In general, I will begin speeding up the implementations to save trees and keep your interest. Where the design isn't obvious I will still fake the implementation and refactor. I hope you will see through this how TDD gives you control over the size of steps.)

Having said that I was going to go much faster, I will immediately go much slower, not in getting the tests working, but in writing the test itself. There are times and tests that call for careful thought. How are we going to represent multi-currency arithmetic? This is one of those times for careful thought.

The most difficult design constraint is that we would like most of the code in the system to be unaware that it is (potentially) dealing with multiple currencies. One possible strategy is to immediately convert all money values into a reference currency (I'll let you guess which reference currency American imperialist pig programmers generally choose). However, this doesn't allow exchange rates to vary easily.

Instead we would like a solution that lets us conveniently represent multiple exchange rates, and still allows most arithmetic-like expressions to look like, well, arithmetic.

Objects to the rescue. When the object you have doesn't behave like you want, make another object with the same external protocol (an Imposter), but a different implementation.

This probably sounds a bit like magic. How do you know to think of creating an imposter here? I won't kid you—there is no formula for flashes of design insight. Ward came up with the "trick" a decade ago and I haven't seen it independently duplicated yet, so it must be a pretty tricky trick. TDD can't guarantee that you will have flashes of insight at the right moment. However, confidence-giving tests and carefully factored code give you preparation for insight, and preparation for applying that insight when it comes.

The solution is to create an object that acts like a Money, but represents the sum of two Moneys. I've tried several different metaphors to explain this idea. One is to treat the sum like a Wallet—you can have several different notes of different denominations and currencies in the same wallet.

Another metaphor is "expressions", as in "(2 + 3) * 5", or in our case "($2 + 3 CHF) * 5". A Money is the atomic form of an expression. Operations result in Expressions, one of which will be a Sum. Once the operation (like adding up the value of a portfolio) is complete, the resulting Expression can be reduced back a single currency given a set of exchange rates.

Applying this metaphor to our test, we know what we end up with:

```
public void testSimpleAddition() {
    ...
    assertEquals(Money.dollar(10), reduced);
}
```

The reduced Expression is created by applying exchange rates to an Expression. What in the real world applies exchange rates? A bank. We would like to be able to write:

```
public void testSimpleAddition() {
    ...
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

(It's a little weird to be mixing the "bank" metaphor and the "expression" metaphor. We'll get the whole story told, and then we'll see what we can do about literary value.)

The Bank in our simple example doesn't really need to do anything. As long as we have an object we're okay:

```
public void testSimpleAddition() {
    ...
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

The sum of two Moneys should be an Expression:

```
public void testSimpleAddition() {
    ...
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

At least we know for sure how to get five dollars:

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

How do we get this to compile? We need an interface Expression (we could have a class, but an interface is even lighter weight):

Expression

```
interface Expression
```

Money.plus() needs to return an Expression:

Money

```
Expression plus(Money addend) {
    return new Money(amount + addend.amount, currency);
}
```

Which means that Money has to implement Expression (which is easy, since there are no operations yet):

Money

```
class Money implements Expression
```

We need an empty Bank class:

Bank

```
class Bank
```

Which stubs out reduce():

Bank

```
Money reduce(Expression source, String to) {
    return null;
}
```

Now it compiles, and fails miserably. Hooray! Progress! We can easily fake the implementation, though:

Bank

```
Money reduce(Expression source, String to) {
    return Money.dollar(10);
}
```

We’re back to a green bar, and ready to refactor. First, reviewing, we:

- Reduced a big test to a smaller test that represented progress ($5 + 10 CHF to $5 + $5)
- Thought carefully about the possible metaphors for our computation
- Re-wrote our previous test based on our new metaphor
- Got the test to compile quickly
- Made it run
- Looked forward with a bit of trepidation to the refactoring necessary to make the implementation real

Make It

We can't mark our test for $5 + $5 done until we've removed all the duplication. We don't have code duplication, but we do have data duplication. The $10 in the fake implementation:

```
Bank
    Money reduce(Expression source, String to) {
        return Money.dollar(10);
    }
```

is really the same as the "$5 + $5" in the test:

```
    public void testSimpleAddition() {
        Money five= Money.dollar(5);
        Expression sum= five.plus(five);
        Bank bank= new Bank();
        Money reduced= bank.reduce(sum, "USD");
        assertEquals(Money.dollar(10), reduced);
    }
```

Before when we've had a fake implementation, it's been obvious how to work backwards to the real implementation. It's just been a matter of replacing constants with variables. This time, though, it's not obvious to me how to work backwards. So, even though it feels a little speculative, we'll work forwards.

First, Money.plus() needs to return a real Expression, a Sum, not just a Money (perhaps later we'll optimize the special case of adding two identical currencies, but that's later.)

```
To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 + $5 = $10
Return Money from $5 + $5
```

The sum of two Moneys should be a Sum:

```
    public void testPlusReturnsSum() {
        Money five= Money.dollar(5);
        Expression result= five.plus(five);
        Sum sum= (Sum) result;
        assertEquals(five, sum.augend);
        assertEquals(five, sum.addend);
    }
```

(Did you know that the first argument to addition is called the "augend"? I didn't until I was writing this. Geek joy.)

The test above is not one I would expect to live a long time. It is deeply concerned with the implementation of our operation, not its externally visible behavior. However, if we make it work, we expect we've moved one step closer to our goal.

To get it to compile, all we need is a Sum class with two fields, augend and addend:

Sum

```
class Sum {
    Money augend;
    Money addend;
}
```

This gives us a ClassCastException, because Money.plus() is returning a Money, not a Sum:

Money

```
Expression plus(Money addend) {
    return new Sum(this, addend);
}
```

Sum needs a constructor:

Sum

```
Sum(Money augend, Money addend) {
}
```

And Sum needs to be a kind of Expression:

Sum

```
class Sum implements Expression
```

Now the system compiles again, but the test is still failing, this time because the Sum constructor is not setting the fields (we could fake the implementation by initializing the fields, but I said I'd start going faster):

Sum

```
Sum(Money augend, Money addend) {
    this.augend= augend;
    this.addend= addend;
}
```

Now Bank.reduce() is being passed a Sum. If the currencies in the Sum are all the same, and the target currency is also the same, the result should be a Money whose amount is the sum of the amounts:

```
public void testReduceSum() {
    Expression sum= new Sum(Money.dollar(3), Money.dollar(4));
    Bank bank= new Bank();
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(7), result);
}
```

I carefully chose parameters that would break the existing test. When we reduce a Sum, the result (under these simplified circumstances) should be a Money whose amount is the sum of the amounts of the two Moneys and whose currency is the currency to which we are reducing.

```
Bank
    Money reduce(Expression source, String to) {
        Sum sum= (Sum) source;
        int amount= sum.augend.amount + sum.addend.amount;
        return new Money(amount, to);
    }
```

This is immediately ugly on two counts:

- The cast. This code should work with any Expression.
- The public fields, and two levels of references at that

Easy enough to fix. First, we can move the body of the method to Sum and get rid of some of the visible fields. We are "sure" we will need the Bank as a parameter in the future, but this pure, simple refactoring, so we leave it out (actually, just now I put it in because I "knew" I would need it—shame, shame on me.)

```
Bank
    Money reduce(Expression source, String to) {
        Sum sum= (Sum) source;
        return sum.reduce(to);
    }
Sum
    public Money reduce(String to) {
        int amount= augend.amount + addend.amount;
        return new Money(amount, to);
    }
```

(Which brings up the point of how we are going to implement, er… test, Bank.reduce() when the argument is a Money.)

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 + $5 = $10
Return Money from $5 + $5
Bank.reduce(Money)

Let's write that test, since the bar is green and there is nothing else obvious to do with the code above:

```
    public void testReduceMoney() {
        Bank bank= new Bank();
        Money result= bank.reduce(Money.dollar(1), "USD");
        assertEquals(Money.dollar(1), result);
    }
Bank
    Money reduce(Expression source, String to) {
        if (source instanceof Money) return (Money) source;
        Sum sum= (Sum) source;
        return sum.reduce(to);
    }
```

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 + $5 = $10
Return Money from $5 + $5
Bank.reduce(Money)
Reduce Money with conversion

Ugly, ugly, ugly. However, we now have a green bar, and refactoring is possible. Any time you are checking classes explicitly, you should be using polymorphism instead. Since Sum implements reduce(String), if Money implemented it, too, we could then add it to the Expression interface.

```
Bank
    Money reduce(Expression source, String to) {
        if (source instanceof Money) return (Money) source.reduce(to);
        Sum sum= (Sum) source;
        return sum.reduce(to);
    }
Money
    public Money reduce(String to) {
        return this;
    }
```

If we add reduce(String) to the Expression interface:

```
Expression
    Money reduce(String to);
```

We can eliminate all those ugly casts and class checks:

```
Bank
    Money reduce(Expression source, String to) {
        return source.reduce(to);
    }
```

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
$5 + $5 = $10
Return Money from $5 + $5
~~Bank.reduce(Money)~~
Reduce Money with conversion

I'm not entirely happy with the name of the method being the same in Expression and in Bank, but having different parameter types. I've never found a satisfactory general solution to this problem in Java. In languages with keyword parameters, communicating the difference between Bank.reduce(Expression, String) and Expression.reduce(String) is well supported by the language syntax. With positional parameters, it's not so easy to make the code speak for you about how the two are different.

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
Reduce Money with conversion
Reduce(Bank, String) vs
  reduce(String)

Next we'll actually exchange one currency for another. First, reviewing, we:

- Didn't mark a test as done because the duplication had not been eliminated
- Worked forwards instead of backwards to realize the implementation
- Wrote a test to force the creation of an object we expected to need later (Sum)
- Started implementing faster (the Sum constructor)
- Implemented code with casts in one place, then moved the code where it belonged once the test were running
- Introduced polymorphism to eliminate explicit class checking

Change

Change is worth embracing (especially if you have a book out with “embrace change” in the title). Here, though, we are thinking about a much simpler form of change—we have 2 francs and we want a dollar. That sounds like a test case already:

```
public void testReduceMoneyDifferentCurrency() {
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(Money.franc(2), "USD");
    assertEquals(Money.dollar(1), result);
}
```

When I go from francs to dollars, I divide by two (we’re still studiously ignoring all those nasty numerical problems.) We can make the bar green in one piece of ugliness:

Money

```
public Money reduce(String to) {
    int rate = (currency.equals("CHF") & to.equals("USD"))
        ? 2
        : 1;
    return new Money(amount / rate, to);
}
```

Now, suddenly, Money knows about exchange rates. Yuck. The Bank should be the only place we care about exchange rates. We’ll have to pass the Bank as a parameter to Expression.reduce() (see, we *knew* we would need it, and we were right. In the words of the grandfather in *The Princess Bride*, “You’re very clever…”) First the caller:

Bank

```
Money reduce(Expression source, String to) {
    return source.reduce(this, to);
}
```

Then the implementors:

```
Expression
    Money reduce(Bank bank, String to);
Sum
    public Money reduce(Bank bank, String to) {
        int amount= augend.amount + addend.amount;
        return new Money(amount, to);
    }
Money
    public Money reduce(Bank bank, String to) {
        int rate = (currency.equals("CHF") & to.equals("USD"))
            ? 2
            : 1;
        return new Money(amount / rate, to);
    }
```

The methods have to be public because methods in interfaces have to be public (for some excellent reason, I'm sure.)

Now we can calculate the rate in the Bank:

```
Bank
    int rate(String from, String to) {
        return (from.equals("CHF") & to.equals("USD"))
            ? 2
            : 1;
    }
```

And ask the bank for the right rate:

```
Money
    public Money reduce(Bank bank, String to) {
        int rate = bank.rate(currency, to);
        return new Money(amount / rate, to);
    }
```

That pesky "2" still appears in both the test and the code. To get rid of it, we need to keep a table of rates in the Bank and look up a rate when we need it. We could use a Hashtable mapping pairs of currencies to rates. Can we use a two element array containing the two currencies as the key? Does Array.equals() check to see if the elements are equal?

```
    public void testArrayEquals() {
        assertEquals(new Object[] {"abc"}, new Object[] {"abc"});
    }
```

Nope. The test fails, so we have to create a real object for the key:

Pair

```
private class Pair {
    private String from;
    private String to;

    Pair(String from, String to) {
        this.from= from;
        this.to= to;
    }
}
```

Because we are using Pairs as keys, we have to implement equals() and hashCode(). I'm not going to write tests for these, because we are writing this code in the context of a refactoring. If we get to the payoff of the refactoring and all the tests run, we expect the code to have been exercised. If I was programming with someone who didn't see exactly where we were going with this, or if the logic became the least bit complex, I would begin writing separate tests.

Pair

```
    public boolean equals(Object object) {
        Pair pair= (Pair) object;
        return from.equals(pair.from) & to.equals(pair.to);
    }

    public int hashCode() {
        return 0;
    }
```

"0" is a terrible hash value, but it has the advantage that it's easy to implement and it will get us running quickly. Currency lookup will look like linear search. When we get lots of currencies, we can do a more thorough job with real usage data.

We need somewhere to store the rates:

Bank

```
private Hashtable rates= new Hashtable();
```

We need to set the rate when told:

Bank

```
void addRate(String from, String to, int rate) {
    rates.put(new Pair(from, to), new Integer(rate));
}
```

And then we can look up the rate when asked:

```
Bank
    int rate(String from, String to) {
        Integer rate= (Integer) rates.get(new Pair(from, to));
        return rate.intValue();
    }
```

Wait a minute!? We got a red bar. What happened? A little snooping around tells us that if we ask for the rate from USD to USD, we expect the value to be 1. Since this was a surprise, let's write a test to communicate what we discovered:

```
    public void testIdentityRate() {
        assertEquals(1, new Bank().rate("USD", "USD"));
    }
```

Now we have three errors, but we expect them all to be fixed with one change:

```
Bank
    int rate(String from, String to) {
        if (from.equals(to)) return 1;
        Integer rate= (Integer) rates.get(new Pair(from, to));
        return rate.intValue();
    }
```

Green bar!

To do:
$5 + 10 CHF = $10 if CHF:USD is 2:1
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
Reduce(Bank, String) vs
reduce(String)

Next we'll implement our last big test, $5 + 10 CHF. Several significant techniques have slipped into this chapter:

- Added a parameter, in seconds, that we expected we would need
- Factored out the data duplication between code and tests
- Wrote a test (testArrayEquals) to check an assumption about the operation of Java
- Introduced a private helper class without distinct tests of its own
- Made a mistake in a refactoring and chose to forge ahead, writing another test to isolate the problem

Mixed Currencies

Now we are finally ready to add the test that started it all, $5 + 10 CHF:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

This is what we'd like to write. Unfortunately, there are a host of compile errors. When we were generalizing from Money to Expression, we left a lot of loose ends laying around. I was worried about them, but I didn't want to disturb you. It's disturbing time, now.

We won't be able to get the test above to compile quickly. We will make the first change that will ripple to the next and the next. We have two paths forward. We can make it work quickly by writing a more specific test and then generalizing, or we can trust our compiler not to let us make mistakes. I'm with you—let's go slow (in practice I would probably just fix the rippling changes one at a time).

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Money tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

The test doesn't work. We get 15 USD instead of 10 USD. It's as if Sum.reduce() isn't reducing the arguments. It isn't:

```
Sum
public Money reduce(Bank bank, String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

If we reduce both of the arguments, the test should pass:

```
Sum
    public Money reduce(Bank bank, String to) {
        int amount= augend.reduce(bank, to).amount + addend.reduce(bank,
to).amount;
        return new Money(amount, to);
    }
```

And it does. Now we can begin pecking away at Moneys that should be Expressions. To avoid the ripple effect, we'll start at the edges and work our way back to the test case. For example, the augend and addend can now be Expressions:

```
Sum
    Expression augend;
    Expression addend;
```

The arguments to the Sum constructor can also be Expressions:

```
Sum
    Sum(Expression augend, Expression addend) {
        this.augend= augend;
        this.addend= addend;
    }
```

(Sum is starting to remind me of Composite, but not so much that I want to generalize. The moment we want a Sum with other than two parameters, though, I'm ready to transform it.) So much for Sum. How about Money?

The argument to plus() can be an Expression:

```
Money
    Expression plus(Expression addend) {
        return new Sum(this, addend);
    }
```

Times() can return an Expression:

```
Money
    Expression times(int multiplier) {
        return new Money(amount * multiplier, currency);
    }
```

This suggests that Expression should include the operations plus() and times(). That's all for Money. We can now change the argument to plus() in our test case:

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

When we change fiveBucks to an Expression, we have to make several changes. Fortunately we have the compiler's to-do list to keep us focused. First we make the change:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

We are politely told that plus() is not defined for Expressions. We define it:

```
Expression
    Expression plus(Expression addend);
```

And then we have to add it to Money and Sum. Money? Yes, it has to be public in Money:

```
Money
    public Expression plus(Expression addend) {
        return new Sum(this, addend);
    }
```

We'll just stub out the implementation in Sum, and add it to our list:

```
Sum
    public Expression plus(Expression addend) {
        return null;
    }
```

Now that the program compiles, the tests all run.

To do:
~~$5 + 10 CHF = $10 if CHF:USD is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
Reduce(Bank, String) vs
  reduce(Expression, String)
Expression.plus
Sum.plus
Expression.times

We are ready to finish generalizing Money to Expression, but first we'll review. We:

- Wrote the test we wanted, then backed off to make it achievable in one step
- Generalized (used a more abstract declaration) from the leaves back to the root (the test case)
- Followed the compiler when we made a change (Expression fiveBucks) which caused changes to ripple (added plus() to Expression, etc.)

Abstraction, Finally

We need to implement Sum.plus() to finish Expression.plus, and then we need Expression.times(), and then we're finished with the whole example. Here's the test for Sum.plus():

```
public void testSumPlusMoney() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).plus(fiveBucks);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(15), result);
}
```

We could have created a Sum by adding fiveBucks and tenFrancs, but the form above, where we explicitly create the Sum, communicates more directly. You are writing these tests not just to make your experience of programming more fun and rewarding, but also as a Rosetta Stone for future generations to appreciate your genius. Think, oh think, of your readers.

The test, in this case, is longer than the code. The code is the same as the code in Money (do I hear an abstract class in the distance?):

Sum

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

You will likely end up with about the same number of lines of test code as model code when TDDing. For TDD to make economic sense, either you will have to be able to write twice as many lines per day as before, or write half as many lines for the same functionality. You'll have to measure and see what effect TDD has on your own practice. Be sure to factor debugging, integrating, and explaining time into your metrics, though.

If we can make Sum.times() work, then declaring Expression.times() will be one simple step. The test is:

```
public void testSumTimes() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).times(2);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(20), result);
}
```

Again, the test is longer than the code (you JUnit geeks will know how to fix that—the rest of you will have to read Fixture):

```
Sum
    Expression times(int multiplier) {
        return new Sum(augend.times(multiplier), addend.times(multiplier));
    }
```

Since we abstracted augend and addend to Expressions in the last chapter, we now have to declare times() in Expression before the code will compile:

```
Expression
    Expression times(int multiplier);
```

Which forces us to raise the visibility of Money.times() and Sum.times():

```
Sum
    public Expression times(int multiplier) {
        return new Sum(augend.times(multiplier), addend.times(multiplier));
    }
Money
    public Expression times(int multiplier) {
        return new Money(amount * multiplier, currency);
    }
```

And it works.

To do:
~~$5 + 10 CHF = $10 if CHF:USD is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
Reduce(Bank, String) vs
  reduce(Expression, String)
~~Expression.plus~~
~~Sum.plus~~
~~Expression.times~~

The only loose end to tie up is to experiment with returning a Money when we add $5 + $5. The test would be:

```
public void testPlusSameCurrencyReturnsMoney() {
    Expression sum= Money.dollar(1).plus(Money.dollar(1));
    assertTrue(sum instanceof Money);
}
```

This test is a little ugly, because it is testing the guts of the implementation, not the externally visible behavior of the objects. However, it will drive us to make the changes we need to make, and this is only an experiment, after all. Here is the code we would have to modify to make it work:

Money

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

There is no obvious, clean way (not to me, anyway, I'm sure you could think of something) to check the currency of the argument if and only if it is a Money. The experiment fails, we delete the test (which we didn't like much anyway), and away we go.

To do:
~~$5 + 10 CHF = $10 if CHF:USD is 2:1~~
~~$5 + $5 = $10~~
~~Return Money from $5 + $5~~
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
Reduce(Bank, String) vs
reduce(Expression, String)
~~Expression.plus~~
~~Sum.plus~~
~~Expression.times~~

The final item on the list, finding a better name for the helper method for Bank.reduce(), still isn't obvious. Our design can only reflect our understanding. Less than perfect understanding implies a less than perfect design, and there is no such thing as perfect understanding. We still have to ship. A moment of silence, then, for our less than perfect design… Thank you. Now ship it.

To do:
~~$5 + 10 CHF = $10 if CHF:USD is 2:1~~
~~$5 + $5 = $10~~
~~Return Money from $5 + $5~~
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String) vs~~
~~reduce(Expression, String)~~
~~Expression.plus~~
~~Sum.plus~~
~~Expression.times~~

Reviewing, we:

- Wrote a test with future readers in mind
- Suggested an experiment comparing TDD with your current programming style

- Once again had changes of declarations ripple through the system, and once again followed the compiler's advice to fix them
- Tried a brief experiment, then discarded it when it didn't work out

Money Retrospective

Let's take a look back at the Money example, both the process we used and the results. We will look at:

- Metaphor—the dramatic affect metaphor has on the structure of the design
- JUnit Usage—when we ran tests and how we used JUnit
- Code Metrics—a numerical abstract of the resulting code
- Process—we say red/green/refactor, but how much work goes into each step?
- Test Quality—how do TDD tests stack up by conventional test metrics?

Metaphor

The biggest surprise in coding this example is how different it came out this time. I have programmed Money in production at least three times that I can think of. I have used it as an example in print another half dozen times. I have programmed it live on stage (relax, it's not as exciting as it sounds…) another fifteen times. I coded another three or four times preparing for writing (I ripped out Section I and rewrote it based on early reviews.) Then, while I was writing this, I thought of using Expression as the metaphor and the design went in a completely different direction than it has gone before.

I really didn't expect the metaphor to be so powerful. A metaphor should just be a source of names, shouldn't it? Apparently not.

The metaphor Ward used for "several monies together with potentially different currencies" was a vector, like a mathematic vector where the coefficients were currencies instead of x^2. I used MoneySum for a while, then MoneyBag (which is nice and physical), and finally Wallet (which is commoner in most folks' experience). All of these metaphors imply that the collection of Money's is flat. For example, "2 USD + 5 CHF + 3 USD" would result in "5 USD + 5 CHF". Two values with the same currency would be merged.

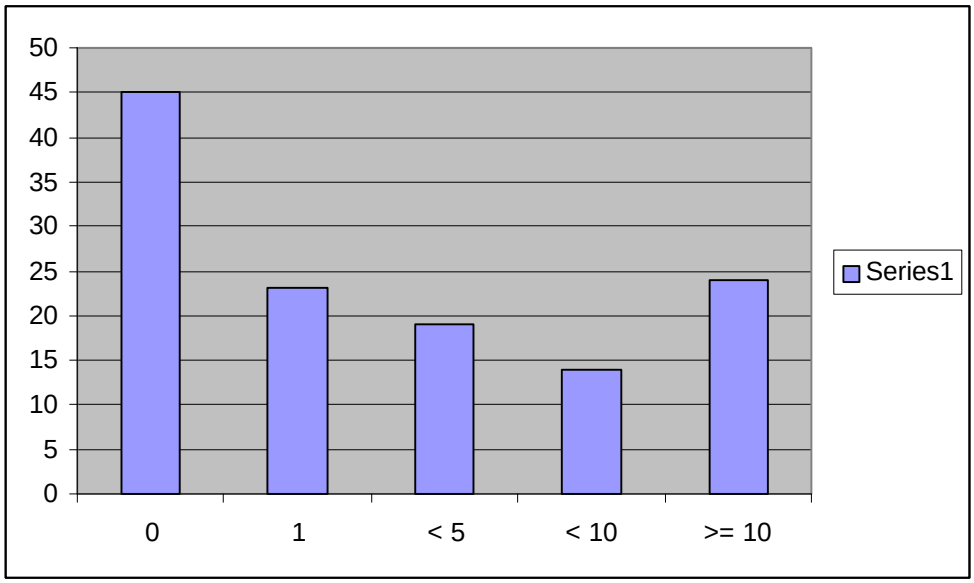
The Expression metaphor freed me from a bunch of nasty issues about merging duplicated currencies. The code came out cleaner and clearer than I've ever seen it before. I'm concerned about the performance of Expressions, but I'm happy to wait until I see some usage statistics before I start optimizing.

What if I got to rewrite everything I ever wrote 20 times? Would I keep finding insight and surprise every time? Is there some way to be more mindful as I program so I can squeeze all the insight out of the first three times? The first time?

JUnit Usage

I had JUnit keep a log while I was coding the Money example. I pressed the Run button precisely 125 times. Because I was writing at the same time as I was programming, the interval between runs isn't representative, but during the times I was just programming I ran the tests about once a minute. Only once in that whole time was I surprised by either success or failure, and that was a refactoring done in haste.

Here is a histogram of the time interval between test runs. The large number of large intervals is most likely because of the time I spent writing:



Code Statistics

Here are some statistics on the code: *Replace these with the real numbers*

	Functional	Test
Classes	5	1
Functions (1)	22	15
Lines (2)	91	89
Cyclomatic complexity (3)	1.04	1
Lines/function	4.1 (4)	5.9 (5)

1. Because we haven't implemented the whole API, we can't evaluate the absolute number of functions, or the number of functions per class, or lines per class. However, the ratios are instructive. There are roughly as many lines and functions in the test and functional code.

2. The number of lines of test code can be reduced by extracting common fixtures. The rough correspondance between lines of model code and lines of test code will remain, however.
3. Cyclomatic complexity is a measure of conventional flow complexity. Test complexity is 1 because there are no branches or loops in test code. Functional code complexity is low because of the heavy use of polymorphism as a substitute for explicit control flow.
4. This includes the function header and trailing brace.
5. Lines/function in the tests is inflated because we have not factored out common fixture-building code, as explained in the section on JUnit.

Process

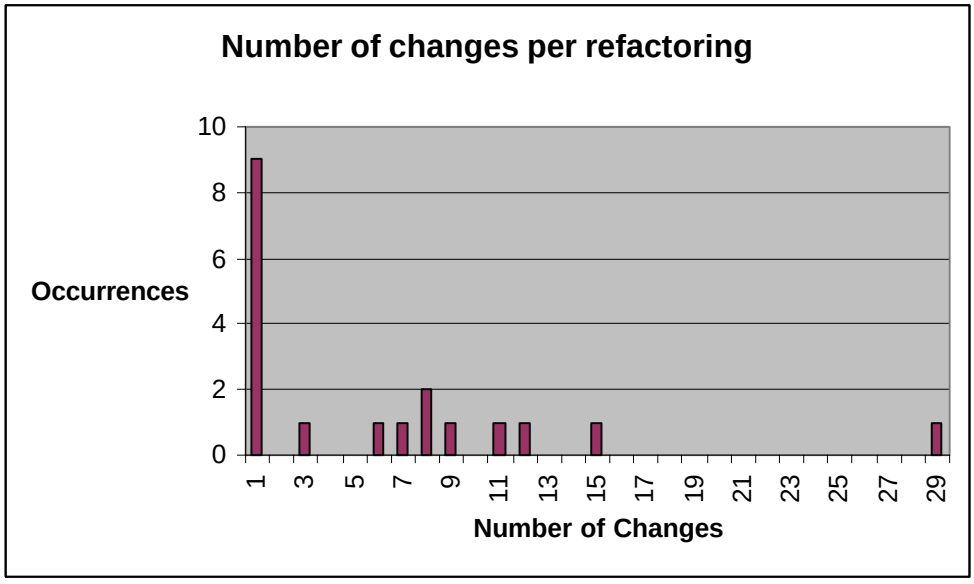
The TDD cycle is:

1. Write a test
2. Make it compile
3. Make it run
4. Remove duplication

Assuming that writing a test is a single step, how many changes does it take to compile, run, and refactor? (By change, I mean changing a method or class definition.) Here is the raw data: *(Tufte, where are you when I need you?)*

Compile	Run	Refactor
4	1	7
1	2	0
0	1	0
0	1	0
1	0	8
0	0	6
0	1	11
3	2	28
0	1	7
1	1	0
5	1	0
5	1	0
0	1	2
0	1	5

0	1	14
0	1	0
0	1	10
0	1	0
0	1	0



I expect that if we gathered data for a large project, the number of changes to compile and run would remain fairly small (they could be even smaller if the programming environment understood what the tests were trying to tell it—creating stubs automatically, for instance). However, (here's at least a master's thesis) the number of changes per refactoring should follow a "fat tail" or leptokurtotic profile, which is like a bell curve but with more large changes. Many measurements in nature follow this profile, like price changes in the stock market.[1]

Test Quality

You professional testers out there please give me some hints. What are tests that I missed that won't run the first time?

Measure Coverage

Try defect insertion—Jester?

[1] *Fractals and Scaling in Finance*, Benoit Mandelbrot, editor, Springer-Verlag, 1997, ISBN 0387983635

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

the keystrokes and mouse events and play them back) for a debugger I was working on (hi Jothy, hi John!). Every morning when I came in there would be a neat stack of paper on my chair describing last nights test runs (hi Al!). On good days there would be a single sheet summarizing that nothing broke. On bad days there would be many many sheets, one for each broken test. I began to dread days when I saw a pile of paper on my chair.

I took two lessons from this experience. First, make the tests so fast to run that I can run them myself, and run them often. That way I can catch errors before anyone else sees them, and I don't have to dread coming in in the morning. Second, I noticed after a while that a huge stack of paper didn't usually mean a huge list of problems. More often it meant that one test had broken early, leaving the system in an unpredictable state for the next test.

We tried to get around this problem by starting and stopping the system between each test, but it took to long, which taught me another lesson about seeking tests at a smaller scale than the whole application. But the main lesson I took was that tests should be able to ignore each other completely. If I had one test broken, I wanted one problem. If I had two tests broken, I wanted two problems.

One convenient implication of isolated tests is that the tests are order independent. If I want to grab a subset of tests and run them, I can do so without worrying that a test will break now because a prerequisite test is gone.

Performance is the usual reason cited for having tests share data. A second implication of isolated tests is that you have to work, sometimes work hard, to break your problem into little orthogonal dimensions, so setting up the environment for each test is easy and quick. Isolating tests encourages you to compose solutions out of many highly cohesive, loosely coupled objects. I always heard this was a good idea, and I was happy when I achieved it, but I never knew exactly how to regularly achieve high cohesion and loose coupling until I started writing isolated tests.

Test List

What should you test? Before you begin, write a list of all the tests you know you will have to write.

The first part of our strategy for dealing with programming stress is to never take a step forward unless we know where our foot is going to land. When we sit down to a programming session, what is it we intend to accomplish?

One strategy for keeping track of what we're trying to accomplish is to hold it all in our heads. I tried this for several years, and found I got into a positive feedback loop. The more experience I accumulated, the more things I knew that might need to be done. The more things I knew might need to be done, the less attention I had for what I was doing. The less attention I had for what I was doing, the less I accomplished. The less I accomplished, the more things I knew that needed to be done.

Just ignoring random items on the list and programming at whim did not appear to work to break this cycle.

I got in the habit of writing down everything I wanted to accomplish over the next few hours on a slip of paper next to my computer. I had a similar list, but with weekly or monthly scope pinned on the wall. As soon as I had all that written down, I knew I wasn't going to forget something. When a new item came up, I would quickly and consciously decide whether it belonged on the "now" list, the "later" list, or it didn't really need to be done at all.

Applied to test-driven development, what we put on the list are the tests we want to implement. If you want a comprehensive treatment of this subject, I recommend Bob Binder's "???". The material here is just enough to get you started. First, put on the list examples of every operation that you know you need to implement. Next, for those operations that don't already exist, put the null version of that operation. Finally, list all the refactorings that you think you will have to do to have clean code at the end of this session.

Instead of outlining the tests, we could just go ahead and implement them. There are a couple of reasons this hasn't worked for me. First, every test you implement is a bit of inertia when you have to refactor. With automated refactoring tools this is less of a problem, but when you've implemented ten tests and *then* you discover the arguments need to be in the opposite order, you are just that much less likely to go clean up. Second, if you have ten tests broken, you are a long way from the green bar. If you want to get to green quickly, you have to throw all ten tests away. If you want to get all the tests working, you are going to be staring at a red bar for a long time.

Conservative mountain climbers have a rule that you have to have three out of your four hands and feet attached at any one time. Dynamic moves where you let go of two at once are much more dangerous. The extreme form of TDD, where you are never more than one change away from a green bar, is like that three out of four rule.

As you make the tests run, the implementation will imply new tests. Write the new tests down on the list. Likewise with refactorings. "This is getting ugly." "<sigh> Put it on the list. We'll get to it before we check in."

Items that are left on the list when the session is done need to be taken care of. If you are really half way through a piece of functionality, use the same list later. If you have discovered larger refactorings that are out of scope for the moment, move them to the "later" list. I can't recall ever moving a test case to the "later" list. If I can think of a test that might not work, nothing is more important than getting it to work.

Test-First

When should you write your tests? Before you write the code that is to be tested.

You won't test after. Your goal as a programmer is running functionality.

You need a way to think about design

You need a method for scope control

Let's look at the usual influence diagram relating stress and testing (but not stress testing, that's different):

Stress above negatively connected to testing below negatively connected to stress.

The more stress you feel, the less likely you are to test enough. When you know you haven’t tested enough, you add to your stress. Positive feedback loop. Once again, there needs to be a way to break the loop.

What if we adopted the rule that we would always test first. Then we can invert the diagram and get a virtuous cycle:

Test-first above negatively connected to stress below negatively connected to test-First..

When we test-first, we reduce the stress, which makes us more likely to test. There are lots of other elements feeding into stress, however, so the tests must live in other virtuous cycles or they will be abandoned when stress increases enough.

Assert First

When should you write the asserts? Try writing them first.

Don't you just love self-similarity?

- Where should you start building a system? With the stories that you will be able to tell about the system when it is done.
- Where should you start writing a bit of functionality? With the tests that will run when it is done.
- Where should you start writing a test? With the asserts that will pass when it is done.

Jim Newkirk introduced me to this technique. When I test assert-first I find it has a powerful simplifying effect. When you are writing a test, you are solving several problems at once, even if you no longer have to think about the implementation.

- Where does the functionality belong? Is it a modification of an existing method, a new method on an existing class, an existing method name implemented in a new place, or a new class?
- What should the names be called?
- How are you going to check for the right answer?
- What is the right answer?
- What other tests does this test suggest?

Pea-sized brains like mine can't possibly do a good job of solving all these problems at once. The two problems from the list that can be easily separated from the rest are "what is the right answer?" and "how am I going to check?"

Here's an example. Suppose we want to communicate with another system over a socket. When we're done, the socket should be closed and we should have read the string "abc".

```
testCompleteTransaction() {
    …
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Where does the buffer come from? The socket, of course:

```
testCompleteTransaction() {
    …
    Buffer reply= reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

And the socket? We create it by connecting to a server:

```
testCompleteTransaction() {
    …
    Socket reader= Socket("localhost", defaultPort());
    Buffer reply= reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

But before this, we need to open a server:

```
testCompleteTransaction() {
    Server writer= Server(defaultPort(), "abc");
    Socket reader= Socket("localhost", defaultPort());
    Buffer reply= reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Now we may have to adjust the names based on actual usage, but we have created the outlines of the test in teensy tiny steps, informing each decision with feedback within seconds.

Test Data

What data do you use for test-first tests? Use data that makes the tests easy to read and follow.

You are writing tests to an audience. Don’t scatter data values around just to be scattering data values around. If there is a difference in the data, it should be meaningful. If there isn’t a conceptual difference between 1 and 2, use 1.

Test Data isn’t a license to stop short of full confidence. If your system has to handle multiple inputs, your tests should reflect multiple inputs. However, don’t have a list of 10 items as the input data is a list of 3 items will lead you to the same design and implementation decisions.

The alternative to Test Data is Realistic Data, where you use data from the real world. Realistic Data is useful when:

- You are testing real-time systems using traces of external events gathered from the actual execution
- You are matching the output of the current system with the output of a previous system (Parallel Testing)
- You are refactoring a simulation and expect precisely the same answers when you are finished, particularly if floating point accuracy may be a problem

Evident Data

How do you represent the intent of the data? Include expected and actual results in the test itself, and try to make their relationship apparent.

You are writing tests for a reader, not just the computer. Someone in decades to come will be asking themselves the question, "What in the heck was this joker thinking about?" You'd like to leave as many clues as possible, especially if that frustrated reader is going to be you.

Here's an example. If we convert from one currency to another, we take a 1.5% commission on the transaction. If the exchange rate from USD to GBP is 2:1, then if we exchange $100, we should get 50 GBP - 1.5% = 49.25 GBP. We could write this test like this:

```
Exchange bank= new Exchange().
bank.addRate("USD", "GBP", STANDARD_RATE);
bank.commission(STANDARD_COMMISSION);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(49.25, "GBP"), result);
```

or we could try to make the calculation obvious:

```
Exchange bank= new Exchange().
bank.addRate("USD", "GBP", 2);
bank.commission(0.0015);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(100 / 2 * (1 - 0.0015), "GBP"), result);
```

Draw lines from the source data to the assertion data. I can read this test and see the connection between the numbers used in the input and the numbers used to calculate the expected result.

One beneficial side effect of Evident Data is that it makes programming easier. Once we've written the expression in the assertion, we know what we need to program. Somehow we have to get the program to evaluation a division and a multiplication. We can even use Fake It Til You Make It to discover where the operations belong incrementally.

Evident Data seems to be an exception to the rule that you don't want magic numbers in your code. *Why is this?*

Implementation Strategies

TDD is not about blindly following a set of rules for how to program. It is about intelligently choosing the size of your programming steps and the amount of feedback depending on conditions. The patterns in this section are all ways of taking small steps forwards.

Even though I'm in an sharing, caring, "can't-we-all-just-get-along" mood, it is only fair to tell you that programmers practicing TDD consistently report that they take smaller and smaller steps over time. Your brain is likely to suffer the same rot if you continue. Don't say I didn't warn you.

Fake It ('Til You Make It)

What is your first implementation once you have a broken test? Return a constant. Once you have the test running, gradually transform the constant into an expression using variables.

A simple example occurred in our implementation of xUnit.

```
return "1 run, 0 failed"
```

became:

```
return "%d run, 0 failed" % self.runCount
```

became:

```
return "%d run, %d failed" % (self.runCount , self failureCount)
```

Fake It is a bit like driving a piton above your head when you are climbing a rock. You haven't really gotten there yet (the test is there but the code structure is wrong). However, when you do get there, you know you will be safe (the test will still run).

There are a couple of effects that make Fake It Til You Make It powerful:

- Psychological—Having a green bar is completely different than not having a green bar. When the bar is green, you know where you stand. Refactoring from there you can do with confidence.
- Scope control—Programmers are good at imagining all sorts of future problems. Starting with one concrete example and generalizing from there prevents you from prematurely confusing yourself with extraneous concerns. You can do a better job of solving the immediate problem because you are focused. When you go to implement the next test case, you can focus on that one, too, knowing that the previous test is guaranteed to work.

Does Fake It violate the rule that says you don't write any code that isn't needed? I don't think so, because in the refactoring step you are eliminating duplication of data between the test case and the code. When I write[2]:

[2] Thanks to Dierk König for the example.

```
        assertEquals(new MyDate("28.2.02"), new MyDate("1.3.02").yesterday());
    MyDate
        public MyDate yesterday()
        {
            return new MyDate("28.2.02");
        }
```

There is duplication between the test and the code. I can shift it around by writing

```
    MyDate
        public MyDate yesterday()
        {
            return new MyDate(new MyDate("31.1.02").days()-1);
        }
```

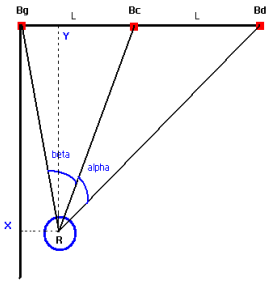
But there is still duplication. However, I can eliminate the data duplication (because this = MyDate("31.1.02") for the purposes of my test) by writing:

```
    MyDate
        public MyDate yesterday()
        {
            return new MyDate(this.days()-1);
        }
```

Not everyone is convinced by this bit of sophistry, which is why you can Triangulate, at least until you are sick of it.

When I use Fake It, I'm reminded of long car trips with kids in the back. I write the first test, I make it work some ugly way, and then, "Don't make me stop this car and write another test. If I have to pull over, you'll be sorry." "Okay, okay, Dad. I'll clean the code up. You don't have to get all huffy."

Triangulate



How do you most conservatively drive abstraction with tests? Only abstract when you have two or more examples.

Here's an example. Suppose we want to write a function that will return the sum of two integers. We write:

```
public void testSum() {
    assertEquals(4, plus(2, 2));
}

private int plus(int augend, int addend) {
    return 4;
}
```

Do we need to have the discussion about unused arguments again? Probably. Sigh…

If we are triangulating to the right design, we have to write:

```
public void testSum() {
    assertEquals(4, plus(2, 2));
    assertEquals(7, plus(3,4));
}
```

(We were careful to give the two parameters different values in the second example, so we couldn't self-righteously claim we only needed one parameter.) When we have the second example, we can abstract the implementation of plus():

```
private int plus(int augend, int addend) {
    return augend + addend;
}
```

Triangulation is attractive because the rules for it seem so clear. The rules for Fake It, where we are relying on our sense of duplication between the test case and the piggy implementation to drive abstraction, seem a bit vague and subject to interpretation. While they seem simple, the rules for triangulation create an infinite loop. Once we have the two assertions and we have abstracted the correct implementation for plus, we can delete one of the assertions on the grounds that it is completely redundant with the other. If we do that, however, we can simplify the implementation of plus() to just return a constant, which requires us to add an assertion.

I only use triangulation when I'm really, really unsure about the correct abstraction for the calculation. Otherwise I rely on either Obvious Implementation or Fake It.

Obvious Implementation

How do you implement simple operations? Just implement them.

Fake It and Triangulation are teensy-weensy tiny steps. Sometimes you are sure you know how to implement an operation. Go ahead.

For example, would I really use Fake It to implement something as simple as plus()? Not usually. I would just type in the obvious implementation. If I noticed I was getting surprised by red bars, I would go to smaller steps.

Keep track of how often you get surprised by red bars using Obvious Implementation. You want to maintain that red/green/refactor rhythm. Obvious Implementation is second gear. Be prepared to downshift if your brain starts writing checks your fingers can't cash.

One to Many

How do you implement an operation that works with collections of objects? Implement it without the collections first, then make it work with collections.

For example, suppose we are writing a function to sum an array of numbers. We can start with one:

```
public void testSum() {
    assertEquals(5, sum(5));
}

private int sum(int value) {
    return value;
}
```

(I am implementing sum() in the TestCase class to avoid writing a new class just for one method.)

We want to test sum(new int[] {5, 7}) next. First we add a parameter to sum() taking an array of values:

```
public void testSum() {
    assertEquals(5, sum(5, new int[] {5}));
}

private int sum(int value, int[] values) {
    return value;
}
```

You can look at this step as an example of Isolate Change. Once we add the parameter in the test case we are free to change the implementation without affecting the test case.

Now we can use the collection instead of the single value:

```
private int sum(int value, int[] values) {
    int sum= 0;
    for (int i= 0; i<values.length; i++)
        sum += values[i];
    return sum;
}
```

Now we can delete the unused single parameter:

```
public void testSum() {
    assertEquals(5, sum(new int[] {5}));
}

private int sum(int[] values) {
    int sum= 0;
    for (int i= 0; i<values.length; i++)
        sum += values[i];
    return sum;
}
```

The previous step is also an example of Isolate Change, where we change the code so we can change the test cases without affecting the code. Now we can enrich the test case as planned:

```
public void testSum() {
    assertEquals(12, sum(new int[] {5, 7}));
}
```

Process

These patterns are about when you write a test, where you write tests, and when you stop.

One Step Test

Which test should you pick next from the list? Pick a test that will teach you something and that you are confident you can implement.

Each test should represent one step towards your overall goal. If we are looking at the following Test List, which test should we pick next?

- Plus
- Minus
- ~~Times~~
- Divide
- ~~Plus like~~
- ~~Equals~~
- ~~Equals null~~
- ~~Null exchange~~
- Exchange one currency
- Exchange two currencies
- Cross rate

There is no right answer. What is one step for me, never having implemented these objects before, will be one tenth of a step to you, with your vast experience.

When I look at a Test List, I think, “That’s obvious, that’s obvious, I have no idea, obvious, what was I thinking about with that one, ah, this one I can do.” That’s the test I implement next.

If you don’t find any test on the list that represents one step, add some new tests that would represent progress towards the items there.

A program grown from tests like this can appear to be written top-down, because you can begin with a test that represents a simple case of the entire computation. A program grown from tests can also appear to be written bottom-up, because you start with small pieces and aggregate them larger and larger.

Neither top-down nor bottom-up really describes the process helpfully. First, a vertical metaphor is a simplistic visualization of how programs change over time. “Growth” implies a kind of self-similar feedback loop where the environment affects the program and the program affects the environment. Second, if we have to have a direction in our metaphor, “known-to-unknown” is a helpful description. Known-to-unknown implies that we have some knowledge and experience on which to draw, and that we expect to learn in the course of development. Put these two together and we have programs growing from known to unknown.

Starter Test

Which test should you start with? Start by testing a variant of an operation that doesn't do anything.

The first question you have to ask with a new operation is "Where does it belong?" Until you've answered this question, you don't know what to type for the test. In the spirit of solving one problem at a time, how can we answer just this question and no other?

If you write a “realistic” test first, you will find yourself solving a bunch of problems at once:

- Where does the operation belong?
- What are the correct inputs?
- What is the correct output given those inputs?

If you begin withAnswering these question a realistic will leave you too long without feedback. Red/green/refactor, red/green/refactor. You want that loop to be minutes.

You can shorten the loop by choosing inputs and outputs that are trivially easy to discover. For example, a poster on the Extreme Programming newsgroup asked about how to write a polygon reducer test-first. The input is a mesh of polygons and the output is a mesh of polygons that describes precisely the same surface, but with fewer polygons. “How can I test-drive this problem since getting a test to work requires reading Ph.D. theses?”

Starter Test provides an answer:

- The output should be the same as the input. Some configurations of polygons are already normalized, incapable of further reduction.
- The input should be as small as possible, like a single polygon, or even an empty list of polygons.

Bing! First test is running. Now for all the rest of the tests on the list.

One Step Test applies: Pick a Starter Test that will teach you something but that you are certain you can get working quickly. If you are implementing something for the Nth time, pick a test that will require an operation or two. You will be justifiably confident you can get it working. If you are implementing something hairy and complicated for the first time, you need a little courage pill immediately.

Explanation Test

How do you spread the use of automated testing? Ask for and give explanations in terms of tests.

It can be frustrating to be the only TDD on a team. Soon, you will notice fewer integration problems and defect reports in tested code, and the designs will be simpler and easier to explain. It has even happened before that folks get downright enthusiastic about testing, and testing first.

Beware the enthusiasm of the newly converted. Nothing will stop the spread of TDD faster than pushing it in people's faces. If you're a manager or leader, you can't force anyone to change the way they work.

What can you do? A simple start is to start asking for explanations in terms of test cases. "Let me see if I understand what you're saying. For example, if I have a Foo like this and a Bar like that then the answer should be 76?" A companion technique is to start giving explanations in terms of tests. "Here's how it works now. When I have a Foo like this and a Bar like that, the answer is 76. If I have a Foo like that and a Bar like this, though, I would like the answer to be 67."

You can do this at higher levels of abstraction. If someone is explaining a sequence diagram to you, you can ask for permission to convert it to a more familiar notation. Then you type in a test case that contains all the externally visible objects and messages in the diagram.

Another Test

How do you keep a technical discussion from straying off topic? When a tangential idea arises, add a test to the list and go back to the topic.

I love wandering discussions (you've read most of the book now, so you've probably reached that conclusion yourself). Keeping a conversation strictly on course is a great way to stifle brilliant ideas. You hop from here to there to there, and how did we get here? who cares, this is cool!

Some programming is like this. You're looking for a breakthrough to get you going. Most programming, though, is a bit more pedestrian. You have ten things to implement and you've only implemented three of them. I'm an accomplished procrastinator at such times. Retreating to hummingbird conversation is a way of avoiding work (and maybe the fear that goes along with it.)

Whole unproductive days have taught me that at times it's best to stay on track. When I'm feeling this way, new ideas are greeted with respect, but not allowed to divert my attention. I write them down on the list, and get back to what I was working on.

Regression Test

What's the first thing you do when a defect is reported? Write the smallest possible test that fails, and that once it runs, the defect will be repaired.

Regression tests are tests that, with perfect foreknowledge, you would have written when coding originally. Every time you have to write a regression test, think about how you could have known to write the test in the first place.

You will also gain value by testing at the level of the whole application. Regression tests for the application give your users a chance to speak concretely to you about what is wrong and what they expect. Regression tests at the smaller scale are a way for you to improve your testing. The defect report will be about a bizarre large negative number in a report. The lesson for you is that you need to test for integer rollover when you are writing your test list.

You may have to refactor the system before you can easily isolate the defect. The defect in this case was your system's way of telling you, "You aren't quite done designing me yet."

Break

What do you do when you feel tired or stuck? Take a break.

Take a drink, take a walk, take a nap. Wash your hands clean of your emotional commitment to the decisions you just made and the characters you typed.

Often, this amount of distance is all it will take to break loose the idea you've been lacking. You'll just be standing up when you realize, "I haven't tried it with the parameters reversed!" Take the break anyway. Give yourself a couple of minutes. The idea won't go away.

If you don't get "the idea", review your goals for the session. Are they still realistic or should you pick new goals? Is what you were trying to accomplish impossible? If so, what are the implications for the team?

Dave Ungar calls this his Shower Methodology. If you know what to type, type. If you don't know what to type, take a shower. Many teams would be happier, more productive, and smell a whole lot better if they took his advice.

Here is an influence diagram that shows the positive feedback loop at work:

Fatigue negatively affects judgement which negatively affects fatigue

You're getting tired, so you're less capable of realizing that you're tired, so you keep going and get more tired.

The way out of this loop is to introduce an additional outside element.

- At the scale of hours, keep a water bottle by your keyboard so biology provides the motivation for regular breaks.
- At the scale of a day, commitments after regular work hours can help you stop when you need sleep before progress.
- At the scale of a week, weekend commitments help get your conscious, energy-sucking thoughts off work. (My wife swears I get my best ideas Friday evening.)

- At the scale of a year, mandatory vacation policies help you refresh yourself completely. The French do this right—two contiguous weeks of vacation aren't enough. You spend the first week decompressing, and the second week getting ready to go back to work. Therefore three weeks, or better four, are necessary for you to be your most effective the rest of the year.

There is a flip side to taking breaks. Sometimes when faced with a tough problem what you need to do is press on, push through it. However, programming culture is so infected with macho, "I'll ruin my health, alienate my family, and kill myself if necessary," spirit that I don't feel compelled to give any advice along these lines. If you find yourself caffeine-addicted and making no progress whatsoever, perhaps you shouldn't take quite so many breaks. In the meantime, take a walk.

Do Over

What do you do when you are feeling lost? Throw away the code and start over.

You're lost. You've taken the break, rinsed your hands in the brook, sounded the Tibetan temple bell, and still you're lost. The code that was going so well an hour ago is now a mess, you can't think of how to get the next test case working, and you've thought of 20 more tests that you really should implement.

This has happened to me several times in writing this book. I would get the code a bit twisted. "But I have to finish the book. The children are starving and the bill collector's are pounding on the door." My gut reaction would be to untwist it just enough to move on. After a pause for reflection, starting over always made more sense. The one time I pressed on regardless, I had to throw away 25 pages of manuscript because it was based on an obviously stupid programming decision.

My favorite example of Do Over is a story Tim Mackinnon told me. He was interviewing someone by the simple expedient of asking her to pair program with him for an hour. At the end of the session, they'd implemented several new test cases and done some nice refactoring. It was the end of the day, though, and they felt tired when they were done, so they discarded their work.

Switching pair programming partners is a good way to motivate productive Do Overs. You'll try to explain the complicated mess you made for a few minutes when your new partner, completely uninvested in the mistakes you've made, will gently take the keyboard and say, "I'm terribly sorry for being so dense, but what if we started like this…"

Cheap Desk, Nice Chair

What physical setup should you use for test-driven development? Get a really nice chair, skimping on the rest of the furniture if necessary.

You can't program well if your back hurts. Yet, organizations that will spend a hundred thousand dollars a month on a team won't spend ten thousand dollars on decent chairs.

My solution is to use cheap, ugly folding tables for my computers, but buy the best chairs I can find. I have plenty of desk space, and I can easily get more, and I am fresh and ready for programming in the afternoon and the morning.

Get comfortable when you're pair programming. Clean off the desk surface enough that you can slide the keyboard back and forth. Each partner should be able to sit comfortably directly in front of the keyboard when they are driving. One of my favorite coaching tricks is to come up behind a pair that is hacking away and gently slide the keyboard so it is comfortably placed for the person typing.

Manfred Lange points out that careful resource allocation also applies to computer hardware. Get cheap/slow/old machines for individual email and surfing, and the hottest possible machines for shared development.

Testing Techniques

Child Test

How do you get a test case running that turns out to be too big? Write a smaller test case that represents the broken part of the bigger test case. Get the smaller test case running. Reintroduce the larger test case.

The red/green/refactor rhythm is so important for continuous success that when you are at risk of losing it, it is worth extra effort to maintain it. This commonly happens to me when I write a test that accidentally requires several changes to make work. Even ten minutes with a red bar gives me the willies.

When I write a test that is too big, I first try to learn the lesson. Why was it too big? What could I have done differently that would have made it smaller? How am I feeling right now.

Metaphysical navel gazing accomplished, I delete the offending test and start over. "Well, getting these three things working at once was too much. If I had A, B, and C working, though, getting the whole thing working would be a cinch."

Can I tell you a secret? Sometimes I don't even bother to delete the offending test. Shhhhh… Do as I say, not as I do. I live with two, count 'em two, broken tests for a matter of a couple of minutes while I get the child test working. I could be making a mistake when I do this. Two broken tests could easily be a holdover from my bad old test-last-if-ever days.

Try it both ways yourself. See if you feel different, program different, when you have two tests broken. Respond as appropriate.

Mock Object

How do you test an object that relies on an expensive or complicated resource? Create a fake version of the resource that answers constants.

There is at least a book's worth of material in Mock Object, but this will serve as an introduction.

The classic example is a database. Databases take a long time to start, they are difficult to keep clean, and if they are located on a remote server, they tie your tests to a physical location on a network. The database is also a fertile source of error in development.

The solution is not to use a real database most of the time. Most tests are written in terms of an object that acts like a database, but is really just sitting in memory.

```
public void testOrderLookup() {
        Database db= new MockDatabase();
        db.expectQuery("select order_no from Order where cust_no is 123");
        db.returnResult(new String[] {"Order 2" ,"Order 3"});
        ...
}
```

If the MockDatabase does not get the query it expects, it throws an exception. If the query is correct, it returns something that looks like a result set constructed from the constant strings.

Another value of mocks, aside from performance and reliability, is readability. You can read the test above from one end to another. If you have a test database full of realistic data, when you see that a query should have resulted in 14 replies, you have no idea why 14 is the right answer.

One of the costs of using mocks is you can't easily store expensive resources in global variables (even if they masquerade as singletons). In the case of the database example, you need to pass the database as a parameter wherever it will be used.

There have been times when I was furious at this restriction. Massimo Arnoldi and I were working on some code relying on a set of exchange rates stored in a global variable. Each test needed different subsets of the data, and sometimes they needed different exchange rates. After a while of trying to get the global variable to work, we decided one morning (courageous design decisions come more often in the morning for me) to just pass the Exchange around wherever we needed it. We thought we would have to modify hundreds of methods. In the end, we added a parameter to ten or fifteen methods, and cleaned up other aspects of the design along the way.

Mocks will encourage you down the path of carefully considering the visibility of every object, reducing the coupling in your designs.

Mock Objects add a risk to the project—what if the mock doesn't behave like the real object? You can reduce this strategy by having a set of tests for the mock that can also be applied to the real object when it becomes available.

Self Shunt

How do you test that one object communicates correctly with another? Have the object under test communicate with the test case instead of with the object it expects.

Suppose we wanted to dynamically update the green bar on the testing user interface. If we could connect an object to the TestResult, it could be notified when a test ran, when it failed, when a whole suite started and finished, and so on.

First we need an object to count the number of notifications:

ResultListener

```
class ResultListener:
    def __init__(self):
        self.count= 0
    def startTest(self):
        self.count= self.count + 1
```

Then we need to create a ResultListener, attach it to a TestResult, run a test, and see that the count has been incremented:

ResultListenerTest

```
    def testNotification(self):
        result= TestResult()
        listener= ResultListener()
        result.addListener(listener)
        WasRun("testMethod").run(result)
        assert 1 == listener.count
```

But wait. Why do we need a separate object for the listener? We can just use the test case itself:

ResultListenerTest

```
    def testNotification(self):
        self.count= 0
        result= TestResult()
        result.addListener(self)
        WasRun("testMethod").run(result)
        assert 1 == self.count
    def startTest(self):
        self.count= self.count + 1
```

Tests written with Self Shunt tend to read better than tests written without. The test above is a good example. The count was 0, and then it was 1. How did it get to be 1? Someone must have called startTest(). How did startTest() get called? It must happen when running the test. This is another example of symmetry—the second version of the test method has the two values for count in one place, where in the first version the count is set to 0 in one class and expected to be 1 in another.

Self Shunt may require that you use Extract Interface to get an interface to implement. You will have to decide whether extracting the interface is easier or if testing the existing class as a black box is easier. I have noticed, though, that interfaces extracted for shunts tend to get their third and subsequent implementations soon thereafter.

As a result of using Self Shunt, you will see tests in Java implementing all sorts of bizarre interfaces *what is the goofiest example?* In optimistically typed languages, the test case class need only implement those operations that are actually used in the running of the test. In Java, however, you have to implement all the operations of the interface, even if most of the implementations are empty, so you would like interfaces to be as narrow as possible.

Log String

How do you test that the sequence in which messages are called is correct? Keep a log in a string, and append to the string when a message is called.

This is just a handy little trick, hardly worth a pattern of its own, but I like it when the alternative is a bunch of flags getting set.

The example from xUnit serves. We have a Template Method which we expect to call setUp(), a testing method, and tearDown(), in that order. By implementing the methods to record in a string that they were called, the test reads nicely:

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

And the implementation is simple, too:

```
WasRun
    def setUp(self):
        self.log= "setUp "
    def testMethod(self):
        self.log= self.log + "testMethod "
    def tearDown(self):
        self.log= self.log + "tearDown "
```

Log Strings are particularly useful when you are implementing Observer and you expect notifications to come in a certain order. If you expected certain notifications but you didn't care about the order, you could keep a set of strings, and use set comparison in the assertion.

Log String works well with Self Shunt. The test case implements the methods in the shunted interface by adding to the log and then returning reasonable values.

Crash Test Dummy

How do you test error code that is unlikely to be invoked? Invoke it anyway with a special object that throws an exception instead of doing real work.

Code that isn't tested doesn't work. This seems to be the safe assumption. What to do with all those odd error conditions, then? Do you have to test them, too? Only if you want them to work.

Let's say we want to test what happens to our application when the file system is full. We could go to a lot of work to create many big files and fill the file system, or we could fake it. "Fake it" doesn't sound dignified, does it? We'll simulate it.

Here's our crash test dummy for a file:

```
private class FullFile extends File {
    public FullFile(String path) {
        super(path);
    }
    public boolean createNewFile() throws IOException {
        throw new IOException();
    }
}
```

Now we can write our Expected Exception test:

```
public void testFileSystemError() {
    File f= new FullFile("foo");
    try {
        saveAs(f);
        fail();
    } catch (IOException e) {
    }
}
```

A Crash Test Dummy is like a Mock Object, except you don't need to Mock up the whole object. Java's anonymous inner classes work well for sabotaging just the right method to simulate the error we want to exercise.

Broken Test

How do you leave a programming session when you're programming alone? Leave the last test broken.

Richard Gabriel taught me the trick of finishing a writing session in mid-sentence. When you sit back down, you look at the half sentence and you have to figure out what you were thinking when you wrote it. Once you have the thought thread back, you finish the sentence and continue. Without the urge to finish the sentence, you can spend many minutes first sniffing around for what to work on next, then trying to remember your mental state, then finally getting back to typing.

I tried the analogous technique for my solo projects and I really like the effect. Finish a solo session by writing a test case and running it to be sure it doesn't pass. When you come back to the code, you have an obvious place to start, you have an obvious, concrete bookmark to help you remember what you were thinking, and making that test work should be quick work, so you'll quickly get your feet back on that victory road.

I thought it would bother me to have a test broken overnight. It doesn't, I think because I know that the program isn't finished. A broken test doesn't make the program any less finished, it just makes the status of the program manifest. The ability to quickly pick up a thread of development after weeks of hiatus is worth that little twinge of walking away from a red bar.

Clean Check-in

How do you leave a programming session when you're programming in a team? Leave all the tests running.

> "Do I contradict myself? Tough."
>
> –Bubba Whitman, Walt's stevedore brother

When you are responsible to your teammates, the picture changes completely. When you start programming on a team project, you don't know in detail what has happened to the code since you saw it last. You need to start from a place of confidence and certainty. Therefore, always make sure all the tests are running before you check in your code (a bit like how each test case leaves the world in a known-good state, if you are prone to computer metaphors for human behavior, which I'm not (usually)).

Here is a sample Ant script for building, testing, and checking in the Money example:

Help!!!

The test suite you run when you check in may be more extensive than the one you are running every minute during development (don't give up on running the whole suite all the time until it is slow enough to be annoying). You will occasionally find a test broken in the integration suite when you try to check in. What to do?

The simplest rule is to just throw away your work and start over. The broken test is pretty strong evidence that you didn't know enough to program what you just programmed. If the team adopted this rule, there would be a tendency for folks to check in more often because the first person to check in doesn't risk losing any work. Checking in more often is probably a good thing.

A slightly more libertine approach is to give you a chance to fix the defect and try again. To keep from dominating the integration resources, you should probably give up after a few minutes and start over. It goes without saying, so I'll say it anyway, that commenting out tests to make the suite pass is strictly verboten, and grounds for some serious beer purchasing at that Friday late afternoon's offsite planning meeting.

Using xUnit

Assertion

How do you check that tests worked correctly? Write boolean expressions that automate your judgment about whether the code worked.

If we are going to make the tests fully automated, every bit of human judgment has to be taken out of the evaluation of the results.

Be specific--assertNotNull(result) can be satisfied by almost any code.

Many xUnit implementations have a special assertion for testing equality. Testing for equality is common, and if you know you are testing equality you can write an informative error message.

I am challenged sometimes to think about the object as a black box. If I have a Contract with a Status that can either be an instance of Offered or Running, I might want to write a test like:

```
contract= Contract() # Offered status by default
contract.begin() # Changes status to Running
assert(contract.status.class = Running)
```

This test is really talking about the current implementation of status. If the representation of status could change to a boolean, how could we write the test so that it still worked? Perhaps once the status changes to Running, it is possible to ask for the actual start date.

```
assert(contract.startDate() == ...) # Throws an exception if the status is Offered
```

I'm aware that I am swimming against the tide in insisting that all tests be written using only unprotected protocol. There is even a package that extends JUnit, JXUnit, that allows testing the value of variables, even those declared private.

Wishing for white box testing is not a testing problem, it is a design problem. Any time I want to use a variable as a way of checking to see whether code ran correctly or not, I have an opportunity to improve the design. If I give in to my fear and just check the variable, I lose that opportunity. That said, if the design idea doesn't come, it doesn't come. I'll check the variable, shed a tear, make a note to come back on one of my smarter days, and move on.

The original SUnit had simple assertions. If one broke, a debugger popped up, you fixed the code, and away you went. Because the IDEs for Java aren't so sophisticated, and because building Java-based software often happens in a batch environment, it makes sense to add information about the assertion which will be printed if it ever fails.

In JUnit, this takes the form of an optional first parameter (I know, optional parameters are supposed to come at the end, but we tried it that way and it didn't work). If you write "assertTrue("Should be true", false)", when the test is run you will see an error message something like "Assertion failed: Should be true". This is often enough information to send you straight to the source of the error in the code. Some teams adopt the convention that all assertions must be accompanied by an informative error message. Try it both ways and see if the investment in the error messages pays off for you.

Fixture

How do you create common objects needed by several tests? Convert the local variables in the tests into instance variables. Override setUp() and initialize those variables.

Talk about the tradeoff, where you have a bunch of implicit state, so you can't just read the tests in isolation any more. Writing tests is faster, but reading them requires establishing context first.

The relationship of classes and instances of TestCase is one of the most confusing parts of xUnit. Each new kind of fixture should be a new subclass of TestCase. Each

new instance of the fixture is created in an instance of that subclass, used once, then discarded. *UML diagram*

This implies that there is no simple relationship between test classes and model classes. Sometimes one fixture serves to test several classes (although this is rare). Sometimes two or three fixtures are needed for a single model class. In practice, you usually end up with roughly the same number of test classes as model classes, but not because for each and every model class you write one and only one test class.

External Fixture

How do you release external resources in the fixture? Override tearDown() and release the resources.

Remember that the goal of each test is to leave the world in exactly the same state as before it ran. For example, if you open a file in a test, you need to be sure to close it before the test completes. You could write:

```
testMethod(self):
    file= File("foobar").open()
    try:
        ...run the test...
    finally:
        file.close()
```

If the file was used in several tests, you could make it part of the common fixture:

```
setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    try:
        ...run the test...
    finally:
        self.file.close()
```

First, there is that pesky duplication of the finally clause telling us that we are missing something in the design. Second, this method is error prone because it is easy to forget the finally clause, or forget to close the file altogether. Lastly, there are three lines of noise in the test.

xUnit guarantees that a method called tearDown() will be run after the test method. TearDown() will be called regardless of what happens in the test method (although if setUp() fails tearDown() won't be called). We can transform the above to:

```
setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    ...run the test...
tearDown(self):
    self.file.close()
```

Test Method

How do you represent a single test case? As a method (whose name begins with “test” by convention.)

Exception Test

How do you test for expected exceptions? Catch expected exceptions and ignore them, failing only if the exception isn’t thrown.

Let’s say we’re writing some code to look up a value. If the value isn’t found, we want to throw an exception. Testing the lookup is easy enough.

```
public void testRate() {
    exchange.addRate("USD", "GBP", 2);
    int rate= exchange.findRate(“USD”, “GBP”);
    assertEquals(2, rate);
}
```

Testing the exception may not be so obvious. Here’s how we do it:

```
public void testMissingRate() {
    try {
        exchange.findRate("USD", “GBP");
        fail();
    } catch (IllegalArgumentException expected) {
    }
}
```

If findRate() doesn’t throw an exception, we will call fail(), an xUnit method which reports that the test failed. Notice that we are careful only to catch the particular exception we expect, so if the wrong kind of exception is thrown, we will also be notified.

AllTests

How do you run all tests together? Make a suite of all the suites, one for each package and one aggregating the package tests for the whole application.

This is really a JUnit idiom, and one that is only necessary because the version of JUnit (3.7 at the time of this writing) doesn’t support collecting bunches of tests automatically.

What you’d really like to have happen is that when you add a TestCase subclass to a package and you add a test method to that class, the next time all the tests run that test method would run, too (there’s that test-driven stuff—the preceding is the outline for a

test that I would probably just go and implement if I wasn't busy writing a book.) Because this isn't supported, yet, each package should declare a class AllTests that implements a static method suite() that returns a TestSuite. Here is AllTests for the Money example:

```
public class AllTests {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(AllTests.class);
    }

    public static Test suite() {
        TestSuite result= new TestSuite("TFD tests");
        result.addTestSuite(MoneyTest.class);
        result.addTestSuite(ExchangeTest.class);
        result.addTestSuite(IdentityRateTest.class);
        return result;
    }
}
```

As you see, can give it a main() method so the class can be run directly from the IDE.

Design Patterns

Null Object

How do you represent special cases using objects? Create an object representing the special case. Give it the same protocol as the regular objects.

```
java.io.File
    public boolean setReadOnly() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkWrite(path);
        }
        return fs.setReadOnly(this);
    }
```

There are 18 places where the same "security != null" check takes place in java.io.File. While I appreciate their diligence in making files safe for the world, I'm also a bit nervous. Are they careful to always check for a null as the result of getSecurityManager()?

The alternative is to create a new class, LaxSecurity, which doesn't throw exceptions ever.

```
java.io.LaxSecurity
    public void checkWrite(String path) {
    }
```

If someone asks for a SecurityManager and there isn't one available, we send back a LaxSecurity instead:

```
java.lang.SecurityManager
    public static SecurityManager getSecurityManager() {
        return security != null ? security : new LaxSecurity();
    }
```

Now we don't have to worry about someone forgetting to check for null. The original code cleans up considerably:

```
java.io.File
    public boolean setReadOnly() {
        SecurityManager security = System.getSecurityManager();
        security.checkWrite(path);
        return fs.setReadOnly(this);
    }
```

Erich Gamma and I once got in an argument at an OOPSLA tutorial about whether a Null Object was appropriate somewhere in JHotDraw. I was ahead on points when he pointed out the cost of introducing the Null Object was around 10 lines of code, for which we would get to eliminate 1 conditional. I hate those late round TKOs.

Command

Template Method

Composite

Pluggable Object

Collecting Parameter

How do you collect the results of an operation that is spread over several objects? Add a parameter to the operation in which the results will be collected.

A simple example is the java.io.Externalizable interface. The writeExternal method writes an object and all the objects it references. Since the objects all have to cooperate loosely to get written out, the method is passed a parameter, an ObjectOutput, as the collecting parameter:

```
java.io.Externalizable
public interface Externalizable extends java.io.Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
}
```

Value Object

How do you design objects that will be widely shared, but for whom identity is unimportant? Set their state when they are created and never change it. Operations on the object always return a new object.

Objects are wonderful. I can say that here, can't I? Objects are a great way to organize logic for later understanding and growth. However, there is one little problem (okay, more than one, but this one will do for now.)

Suppose I (an object) have a Rectangle. I compute some value based on the Rectangle, like its area. Later, someone politely asks me for my Rectangle, and I, not wanting to appear uncooperative, give it to them. Moments later, lo and behold, the Rectangle has been changed behind my back. The area I computed earlier is out of date, and there is no way for me to know.

This is the classic aliasing problem. If two objects share a reference to a third, if one object changes the referred object, the other object better not rely on the state of the shared object.

There are several ways out of the aliasing problem. One solution is never to give out the objects that you rely on, but instead to always make copies. This can get expensive in time and space, and ignores those times when you want to share changes to a shared object. Another solution is Observer, where you explicit register with objects on which you rely and expect to be notified when they change. Observer can make control flows difficult to follow, and the logic for setting up and removing the dependencies gets ugly.

Another solution is to treat the object as less than an object. Objects have state that change over time. We can, if we choose, eliminate the "that change over time". If I have an object and I know it won't change, I can pass around references to it all I want, knowing that aliasing won't be a problem. There can be no hidden changes to a shared object if there are no changes.

I remember puzzling over Integers when I was first learning Smalltalk. If I change bit 2 to a 1, why don't all 2's become 6's?

```
a := 2.
b := a.
a := a bitAt: 2 put: 1.
a => 6
b => 2
```

Because Integers are really values masquerading as objects. In Smalltalk this is literally true of small integers, and simulated in the case of integers that don't fit in a single machine word. When I set that bit, what I get back is a new object with the bit set, not the old one with the bit changed.

When implementing a Value Object, every operation has to return a fresh object, leaving the original unchanged. Users have to be aware they are using a Value Object and store the result (as in the example above.) All of these object allocations can create performance problems, which should be handled like all performance problems,

when you have realistic data sets, realistic usage patterns, profiling data, and complaints about performance.

I have a tendency to use Value Object whenever I have a situation that looks like algebra—geometric shapes being intersected and unioned, unit values where units are carried around with a number, symbolic arithmetic. Any time Value Object makes the least sense I try it, because it makes reading and debugging so much easier.

All Value Objects have to implement equality (and in many languages by implication they have to implement hashing.) If I have this Contract and that Contract and they aren't the same object, then they are different, not equal. However, if I have this five francs and that five francs, it doesn't matter if they are the same five francs, five francs are five francs and they should be equal.

Imposter

Refactoring

There is a brief description of how to accomplish each refactoring in small steps. More importantly, each refactoring discusses why you might want to use it.

In TDD we use "refactoring" in an interesting way. Usually, a refactoring cannot change the semantics of the program under any circumstances. In TDD, the circumstances we care about are the tests that are already passing. So, for example, we can replace constants with variables in TDD can call this operation, in good conscience, a refactoring, because it doesn't change the set of tests that pass. The only circumstance under which semantics are preserved may actually be our one test case. Any other test case that was passing would fail. However, we don't have those tests yet, so we don't worry about them.

This kind of weird "refactoring with respect to tests" places a burden on you to have enough tests that as far as you know, a refactoring with respect to the tests is the same as a refactoring with respect to all possible tests, at least by the time you're done. It's no excuse to say, "I knew there was a problem, but the tests all passed so I checked the code in." Write more tests.

Reconcile Differences

How do you unify two similar looking pieces of code? Gradually bring them closer. Unify them only when they are absolutely identical.

Refactoring can be a nerve wracking experience. The easy ones are obvious. If I extract a method, as long as I do it mechanically correctly, there is very little chance of changing the system's behavior. Some refactorings push you to examine the control flows and data values carefully. A long chain of reasoning leads you to believe that the change you are about to make won't change any answers. Those are the refactorings that enhance your hairline.

Such a leap of faith refactoring is exactly what we're trying to avoid with our strategy of small steps and concrete feedback. While you can't always avoid leapy refactorings, you can reduce their incidence.

For example, suppose we have code in two subclasses that we think does exactly the same thing:

I'm having trouble finding an example

We could just write a method in the superclass that we think implements the same algorithm. That would be a big step. However, if the two methods were exactly the same, we could put the method in the superclass without any worries.

The same general strategy works for all refactorings—if you want to make a big change, think about how the last step of the change could be trivial, then work backwards. For example, if you want to remove several subclasses, the trivial last step is if a subclass contains nothing. Then the subclass can be replaced by the superclass without changing the behavior of the system. One by one, empty out the subclasses and, when they are empty, replace references to them by references to the superclass.

Isolate Change

How do you change one part of a multi-part method or object? First, isolate the part that has to change.

If you want to be absolutely sure you aren't changing anything accidentally (not that you'll be right always), before you change it, make a place for it. We saw an example in working with Money. The conversion code said:

```
return new Note(source.amount / rate, currency)
```

We wanted to change rate lookup. First we extracted the access to the instance variable rate into its own method:

```
return new Note(source.amount / findRate(), currency)
private int findRate() {
    return rate;
}
```

Now we can work on findRate() in isolation, without having to worry about affecting the parts of convert() that have nothing to do with rate lookup.

The picture that comes to my mind is surgery, where all of the patient except the part to be operated on is draped. The draping leaves the surgeon with only a fixed set of variables. Now, we could have long arguments over whether this abstraction of a person to a lower left quadrant abdomen leads to good health care, but at the moment of surgery, I'm kind of glad the surgeon can focus.

You may find that once you've isolated the change and then made the change, that the result is so trivial that you can undo the isolation. If we found that really all we needed was to return the instance variable in findRate(), we should consider inlining findRate() everywhere it is used and deleting it. Don't make these changes automatically, however. Balance the cost of an additional method with the value of having an additional concept explicit in the code.

Some possible ways to Isolate Change are Extract Method (the most common), Extract Object, and Method Object.

Migrate Data

How do you move from one representation? Temporarily duplicate the data.

How:

Here is the internal-to-external version:

1. Add an instance variable in the new format
2. Set the new format variable everywhere you set the old format
3. Use the new format variable everywhere you use the old format
4. Delete the old format
5. Change the external interface to reflect the new format

Sometimes, though, you want to change the API first. Then you should:

1. Add a parameter in the new format
2. Translate from the new format parameter to the old format internal representation
3. Delete the old format parameter
4. Replace uses of the old format with the new format
5. Delete the old format

Why:

One to Many creates a data migration problem every time. Suppose we wanted to implement TestSuite using One to Many. We would start with:

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.run(self.result)
    assert("1 run, 0 failed" == self.result.summary())
```

Which is implemented (in the “One” part of One to Many) by:

```
class TestSuite:
    def add(self, test):
        self.test= test
    def run(self, result):
        self.test.run(result)
```

Now we begin duplicating data. First we initialize the collection of tests:

```
TestSuite
    def __init__(self):
        self.tests= []
```

Everywhere “test” is set, we add to the collection, too:

```
TestSuite
    def add(self, test):
        self.test= test
        self.tests.append(test)
```

Now we use the list of tests instead of the single test. For purposes of the current test cases this is a refactoring (it preserves semantics) because there is only ever one element in the collection.

```
TestSuite
    def run(self, result):
        for test in self.tests:
            test.run(result)
```

We delete the now-unused instance variable “test”:

```
TestSuite
    def add(self, test):
        self.tests.append(test)
```

You can also use stepwise data migration when moving between equivalent formats with different protocols, as in moving from Java’s Vector/Enumerator to Collection/Iterator. *Would this make a better example, since the above example is duplicated elsewhere in the text?*

Extract Method

Inline Method

Extract Interface

Move Method

Method Object

How do you represent a complicated method that requires several parameters and local variables? Make an object out of the method.

How:

1. Create an object with the same parameters as the method.
2. Make the local variables also instance variables of the object.
3. Create one method called "run()", whose body is the same as the body of the original method.
4. In the original method, create a new object and invoke run().

Why:

Method Objects are useful in preparation for adding a whole new kind of logic to the system. For example, you might have several methods involved in computing the cash flow from component cash flows. When you want to start computing the net present value of the cash flows, you can first create a Method Object out of the first style of computation. Then you can write the new style of computation with its own, smaller-scale, tests. Then plugging in the new style will be a single step.

Method Objects are also good for simplifying code that doesn't yield to Extract Method. Sometimes you'll find a block of code that has a bunch of temporary variables and parameters, and every time you try to extract a piece of it you have to carry along five or six temps and parameters. The resulting extracted method doesn't look any better than the original code, because the method signature is so long. Creating a Method Object gives you a new namespace in which you can extract methods without having to pass anything.

Add Parameter

How do you add a parameter to a method?

How:

1. If the method is in an interface, add the parameter to the interface first
2. Use the compiler errors to tell you what other code you need to change

Why:

Adding a parameter is often an extension step. You got the first test case running without needing the parameter, but in this new circumstance you have to take more information into account in order to compute correctly.

Adding a parameter can also be part of migrating from one data representation to another. First you add the parameter, then you delete all uses of the old parameter, then you delete the old parameter.

Method Parameter to Constructor Parameter

How do you move a parameter from a method or methods to the constructor?

How:

1. Add a parameter to the constructor
2. Add an instance variable with the same name as the parameter
3. Set the variable in the constructor
4. One by one, convert references to "parameter" to "this.parameter"
5. When no more references exist to the parameter, delete the parameter from the method and all caller
6. Remove the now-superfluous "this." from references
7. Rename the variable correctly

Why:

If you pass the same parameter to several different methods in the same object, you can simplify the API by passing the parameter once (eliminating duplication). You can run this refactoring in reverse if you find that an instance variable is only used in one method.

Mastering TDD

I hope to raise questions here for you to ponder as you integrate TDD into your own practice. Some of the questions are small, and some are large.

How large should your steps be?

There are really two questions lurking here:

- How much ground should each test cover?
- How many intermediate stages should you go through as you refactor?

You could write the tests so they each encouraged the addition of a single line of logic and a handful of refactorings. You could write the tests so they each encouraged the addition of hundreds of lines of logic and hours of refactoring. Which should you do?

Part of the answer is that you should be able to do either. The tendency of TDDers over time is clear, though—smaller steps. However, folks are experimenting with driving development from application-level tests, either alone or in conjunction with the programmer-level tests we've been writing.

At first when you refactor, you should be prepared to take lots of little tiny steps. Manual refactoring is prone to error, and the more errors you make and only catch later, the less likely you are to refactor. Once you've done a refactoring 20 times by hand in little tiny steps, experiment with leaving out some of the steps.

Automated refactoring accelerates refactoring enormously. What would have taken you 20 manual steps now becomes a single menu item. An order of magnitude change in quantity generally constitute a change in quality, and this is true of automated refactoring. When you know you are supported by an excellent tool, you become much more aggressive in your refactorings, trying many more experiments to see how the code wants to be structured.

The Refactoring Browser for Smalltalk is as I write still the best refactoring tool available. Java refactoring support is appearing in many Java IDEs, and refactoring support is sure to spread quickly to other languages and environments.

How much feedback do you need?

How many tests should you write? Here's a simple problem—given three integers representing the length of the sides of a triangle, return:

- 1 if the triangle is equilateral
- 2 if the triangle is isoceles
- 3 if the triangle is scalene

and throw an exception if the triangle is not well formed.

Go ahead, try the problem (my Smalltalk solution is listed at the end of this question).

I wrote 6 tests (kind of like Name That Tune, “I can code that problem in four tests.” “Code that problem.”) Bob Binder, in his comprehensive book *Testing Object-Oriented Software*, wrote 65 for the same problem. You’ll have to decide, from experience and reflection, about how many tests you want to write.

I think about Mean Time Between Failure (MTBF) when I think about how many tests to write. For example, Smalltalk integers act like integers, not like a 32-bit counter, so it doesn’t make sense to test MAXINT. Well, there is a MAXINT, but it has to do with how much memory you have. Do I need to write a test that fills up memory with extremely large integers? How will that affect the MTBF of my program? If I’m never going to get anywhere close to that size of triangle, my program is not measurably more robust with such a test than without it.

Whether a test makes sense to write depends on how carefully you measure MTBF. If you are trying to get from an MTBF of 10 years to an MTBF of 100 years in your pacemaker, it probably makes sense, unless you can demonstrate in some other way that you never get triangles that large.

TDD’s view of testing is pragmatic. The tests are a means to an end, the end being code in which we have great confidence. If our knowledge of the implementation gives us confidence in the absence of a test, we will not write that test. Black box testing, where we deliberately choose to ignore the implementation, has some advantages. By ignoring the code, it demonstrates a different value system—the tests are valuable alone. It’s an appropriate attitude to take in some circumstances. However, it is different than TDD.

```
TriangleTest
testEquilateral
    self assert: (self evaluate: 2 side: 2 side: 2) = 1

testIsoceles
    self assert: (self evaluate: 1 side: 2 side: 2) = 2

testScalene
    self assert: (self evaluate: 2 side: 3 side: 4) = 3

testIrrational
    [self evaluate: 1 side: 2 side: 3]
        on: Exception
        do: [:ex | ^self].
    self fail

testNegative
    [self evaluate: -1 side: 2 side: 2]
        on: Exception
        do: [:ex | ^self].
    self fail

testStrings
    [self evaluate: 'a' side: 'b' side: 'c']
        on: Exception
        do: [:ex | ^self].
    self fail

evaluate: aNumber1 side: aNumber2 side: aNumber3
    | sides |
    sides := SortedCollection
        with: aNumber1
        with: aNumber2
        with: aNumber3.
    sides first <= 0 ifTrue: [self fail].
    (sides at: 1) + (sides at: 2) <= (sides at: 3) ifTrue: [self fail].
    ^sides asSet size
```

When should you delete tests?

More tests is better, but if two tests are redundant with respect to each other, should you keep them both around?

How does the programming language and environment influence TDD?

Try TDD in Smalltalk with the Refactoring Browser. Try it in C++ with vi. How does your experience differ?

In programming languages and environments where cycles are harder to come by (TDD cycles—test/compile/run/refactor), you will likely be tempted to take larger steps:

- Cover more ground with each test
- Refactor with fewer intermediate steps

Does this make you go faster or slower.

In programming languages where cycles are plentiful, you will likely be tempted to try lots more experiments. Does this help you go faster or reach better solutions, or would you be better off institutionalizing some kind of time for pure reflection (reviews or literate programs)?

How can you use TDD to teach programming, design, and/or testing?

Should you use TDD to teach CS1? Should (or how should) experienced TDDers use it to learn new languages or environments? How should TDD be modified to maintain the current pedagogical separation between programming, testing, and design? How should the current pedagogical separation between programming, testing, and design be modified to take advantage of TDD? How does TDD change when pair programming? How about if the pair are both inexperienced? If one is inexperienced and one is experienced? If both are experienced? If both are experienced but in different domains?

Can you test-drive enormous systems?

Does TDD scale to extremely large systems? What new tests would you have to write? What new kinds of refactorings would you need?

Can you drive development with application-level tests?

The problem with driving development with small scale tests (I call them “unit tests”, but they don’t match the accepted definition of unit tests very well. *Why?)* is that you run the risk of implementing what you think a user wants, but having it turn out to be not what they wanted at all. What if we wrote the tests at the level of the application? Then the users (with some help) could write tests themselves for what exactly they wanted.

There is a technical problem—fixturing. How can you write and run a test for a feature that doesn’t exist yet. There always seems to be some way out of this problem, typically by introducing an interpreter which gracefully signals an error when it comes across a test that it doesn’t know how to interpret yet.

There is also a social problem with application test driven development. Writing tests is a new responsibility for users (by which I really mean a team that includes users), and that responsibility comes at a new place in the development cycle, namely before implementation begins. Organizations resist this kind of shift of responsibility. It will require concerted effort (that is, the effort of many people on the team working in concert) to get application tests written first.

TDD as described in this book is a technique that is entirely under your control. You can pick it up and start using it today if you so choose. Mixing up the rhythm of red/green/refactor, the technical issues of application fixturing, and the organizational change issues surrounding user-written tests is unlikely to be successful. The One Step Test rule applies. Get red/green/refactor going in your own practice, then spread the message.

Another aspected of ATDD is the length of the cycle between test and feedback. If a customer wrote a test and ten days later it finally worked, you would be staring at a red bar most of the time. I think I would want to still do programmer-level TDD so:

- I got immediate green bars
- I simplified the internal design

Is TDD sensitive to initial conditions?

Some orders in which you take the tests seem to work very smoothly. Red/green/refactor/red/green/refactor. You can take the same tests and implement them in a different order, and it seems like there is not a way to advance in small steps. Is it really true that one sequence of tests is an order of magnitude faster/easier to implement than another? Is this just because my implementation technique is not up to the challenge? Is there something about the tests that should tell me to tackle them in a certain order? If TDD is sensitive to initial conditions in the small, is it predictable in the large? (In the same way that little eddies in the Mississippi are unpredictable, but you can count on 2,000,000 cfs. more or less at the river mouth.)

Why does TDD work?

I saved the weirdest for last. Let’s assume for the moment that TDD helps teams productively build loosely coupled, highly cohesive systems with low defect rates and low cost maintenance profiles. (I’m claiming no such thing in general, just for myself, but I trust you to imagine impossible things.) How could such a thing happen?

An answer comes from the fevered imagination of complex systems. The inimitable Phlip says:

> Adopt programming practices that "attract" correct code as a limit function, not as an absolute value. If you write UnitTests for every feature, and if you Refactor to simplify code between each step, and if you add features one at a time and only after all the UnitTests pass, you will create what mathematicians call an "iterative dynamic attractor". This is a point in a state space that all flows converge on. Code is more likely to change for the better over time instead of for the worse; the attractor approaches correctness as a limit function.
>
> This is the "correctness" that nearly all programmers get by with (except, of course, for medical or aerospace software). But it's better to explicitly understand the attractor concept than deny it or disregard its importance.

I hope that gives you something to chew on.

Glossary

I tend to use technical terms without defining them. If we were talking together, that glazed look in your eyes would tell me I'd wandered off into jargon land. In the absence of your inestimable feedback, here are the terms reviewers have huh'd.

SBPP

value object

hashCode

inline

model code

method signature

Appendix 1: Influence Diagrams

You will find many examples of influence diagrams in the text. The idea of influence diagrams is taken from Gerald Weinberg's excellent Quality Software Management series, particularly book 1 *Systems Thinking* (Dorset House, 1992). The purpose of an influence diagram is to see how the elements of a system affect each other.

Influence diagrams have three elements:

- Activities, notated as a word or short phrase
- Positive connections, notated as a directed arrow between two activities, meaning that more of the source activity tends to create more of the destination activity or less of the source activity tends to create less of the destination activity
- Negative connections, notated as a directed arrow between two activities with a circle over it, meaning that more of the source activity tends to create less of the destination activity or less of the source activity tends to create more of the destination activity

That's lots of words for a simple concept. Here are some examples:

Circus attendance and hedge trimming

Figure N: Two seemingly unrelated activites

Eating positively connected to weight

Figure N: Positively connected activities

The more I eat, the more I weigh. The less I eat, the less I weigh. (Personal weight is a far more complicated system than this, of course. Influence diagrams are models to help you understand some aspect of the system, not understand and control it perfectly.)

Exercise negatively connected to weight

Figure N: Negatively connected activities

Feedback

Influence doesn't just work one way. Often the effects of an activity come back around to change the activity itself, either positively or negatively. For example:

Weight negatively connected to self esteem negatively connected to eating positively connected to weight.

Figure N: Feedback

If my weight rises, my self-esteem drops, which makes we want to eat more, which makes my weight rise, and so on. Any time you have a cycle in an influence diagram, you have feedback.

There are two kinds of feedback:

- Positive
- Negative

Positive feedback causes systems to encourage more and more of an activity. You can find positive feedback loops by counting the number of negative connections in a cycle. If there are an even number of negative connections, you have a positive feedback loop. The loop above is a positive feedback loop. It will cause you to keep gaining weight until the influence of some other activity kicks in.

Negative feedback damps or reduces an activity. Cycles with an odd number of negative connections are negative feedback loops.

The key to system design is

- Creating virtuous cycles, where positive feedback loops encourage the growth of good activities
- Avoiding death spirals, where positive feedback loops encourage the growth of unproductive or destructive activities
- Creating negative feedback cycles to prevent overuse of good activities

System Control

When choosing a system of software development practices, you'd like the practices to support each other so that you tend to do about the right amount of any activity, even under stress. Here's an example of a system of practices that leads to insufficient testing:

Time pressure neg testing neg errors pos time pressure

Figure N: Not enough time to test reduces the available time

Under the pressure of time, you reduce the amount of testing, which increases the number of errors, which increases the time pressure. Eventually some outside activity (like "Cash Flow Panic") steps in to ship the software regardless.

When you have a system that isn't behaving, you have a host of options:

- Drive a positive feedback loop the other directions. If you have a loop between tests and confidence, and tests have been failing thus reducing confidence, you can make more tests work to increase confidence in your ability to get more test working.
- Introduce a negative feedback loop to control an activity which has grown too large.
- Create or break connections to eliminate loops that are not helping.