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/

Test-driven development Copyright © 2002 Kent Beck, All Rights Reserved 3/8/02, 132 Pages

Test-driven development 2 of 132

**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?

Test-driven development 3 of 133

**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

Test-driven development 4 of 133

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.

Test-driven development 5 of 133

**Contents**

**TO DO 2**

**RANDOM THOUGHTS 3**

**CONTENTS 5**

**PREFACE 9**

**Fear 10**

**Acknowledgements 11**

**STORY TIME 13**

**SECTION I: MONEY EXAMPLE 15**

**MONEY EXAMPLE 16**

**DEGENERATE OBJECTS 22**

**EQUALITY FOR ALL 24**

**PRIVACY 27**

**FRANC-LY SPEAKING 29**

**EQUALITY FOR ALL, REDUX 32**

**APPLES AND ORANGES 35**

**MAKIN’ OBJECTS 37**

**TIMES WE’RE LIVIN’ IN 40**

**THE ROOT OF ALL EVIL 45**

**ADDITION, FINALLY 51**

Test-driven development 6 of 134

**MAKE IT 55**

**CHANGE 60**

**MIXED CURRENCIES 64**

**ABSTRACTION, FINALLY 68**

**MONEY RETROSPECTIVE 72**

**Metaphor 72**

**JUnit Usage 73**

**Code Statistics 73**

**Process 74**

**Test Quality 75**

**EXAMPLE: XUNIT 76**

**XUNIT TEST-FIRST 78**

**SET THE TABLE 82**

**COUNTING 87**

**HOW SUITE IT IS 90**

**XUNIT RETROSPECTIVE 94**

**SECTION III: PATTERNS 95**

**PATTERNS FOR TEST-DRIVEN DEVELOPMENT 96** Test *n.* 96 Isolated Test 96 Test List 97 Test-First 98 Assert First 99 Test Data 100 Evident Data 101

**Implementation Strategies 102**

Test-driven development 7 of 135

Fake It (‘Til You Make It) 102 Triangulate 103 Obvious Implementation 104 One to Many 104

**Process 106** One Step Test 106 Starter Test 107 Explanation Test 108 Another Test 108 Regression Test 108 Break 109 Do Over 110 Cheap Desk, Nice Chair 110

**Testing Techniques 111** Child Test 111 Mock Object 111 Self Shunt 112 Log String 114 Crash Test Dummy 114 Broken Test 115 Clean Check-in 116

**Using xUnit 116** Assertion 116 Fixture 117 External Fixture 118 Test Method 119 Exception Test 119 AllTests 119

**Design Patterns 120** Null Object 120 Command 121 Template Method 121 Composite 121 Pluggable Object 121 Collecting Parameter 121 Value Object 122 Imposter 123

**Refactoring 123** Reconcile Differences 123 Isolate Change 124 Migrate Data 125 Extract Method 126

Test-driven development 8 of 136

Inline Method 126 Extract Interface 126 Move Method 126 Method Object 126 Add Parameter 127 Method Parameter to Constructor Parameter 127

**MASTERING TDD 129**

**How large should your steps be? 129**

**How much feedback do you need? 129**

**When should you delete tests? 131**

**How does the programming language and environment influence TDD? 132**

**How can you use TDD to teach programming, design, and/or testing? 132**

**Can you test-drive enormous systems? 132**

**Can you drive development with application-level tests? 133**

**Is TDD sensitive to initial conditions? 133**

**Why does TDD work? 134**

**GLOSSARY 135**

**APPENDIX 1: INFLUENCE DIAGRAMS 136**

**Feedback 136**

**System Control 137**

Test-driven development 9 of 133

**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.

Test-driven development 10 of 133

***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.

Test-driven development 11 of 133

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

Test-driven development 12 of 133

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.

Test-driven development 13 of 133

**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

Test-driven development 14 of 133

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.

Test-driven development 15 of 133

**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

Test-driven development 16 of 133

**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

Test-driven development 17 of 133

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:

DollarDollar classclass Dollar Now we need the constructor, but it doesn’t have to do anything just to get the test to compile:

DollarDollar

Dollar(intint amount) { } We need a stub implementation of times(). Again we’ll do the least work possible just to get the test to compile:

DollarDollar

voidvoid times(intint multiplier) { } Finally, we need an amount field:

DollarDollar

intint amount; Now we can run the test and watch it fail.

Test-driven development 18 of 133

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:

DollarDollar

intint amount= 10; Now we get the green bar, fabled in song and story.

Test-driven development 19 of 133

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.

Test-driven development 20 of 133

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?

DollarDollar

intint 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?

DollarDollar

intint amount;

voidvoid times(intint 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:

Test-driven development 21 of 133

DollarDollar

Dollar(intint amount) {

thisthis.amount= amount; } we can use it in times():

DollarDollar

voidvoid times(intint multiplier) {

amount= amount \* 2; } The value of the parameter “multiplier” is 2, so we can substitute the parameter for the constant:

DollarDollar

voidvoid times(intint 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):

DollarDollar

voidvoid times(intint 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

Test-driven development 22 of 133

**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:

publicpublic voidvoid testMultiplication() {

Dollar five= newnew 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.

publicpublic voidvoid testMultiplication() {

Dollar five= newnew 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():

DollarDollar

Dollar times(intint multiplier) {

amount \*= multiplier; return return nullnull; } Now the test compiles, but it doesn’t run. Progress! Making it run requires that we return a new Dollar with the correct amount:

DollarDollar

Dollar times(intint multiplier) {

return return newnew 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:

Test-driven development 23 of 133

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.

Test-driven development 24 of 133

**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:

Test-driven development 25 of 133

DollarDollar publicpublic boolean boolean equals(Object object) {

return return truetrue; } 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:

DollarDollar publicpublic boolean boolean equals(Object object) {

Dollar dollar= (Dollar) object; return 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.)

Test-driven development 26 of 133

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

Test-driven development 27 of 133

**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:

DollarDollar private private intint amount; And we can cross another item off the list:

Test-driven development 28 of 133

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

Test-driven development 29 of 133

**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

Test-driven development 30 of 133

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.)

FrancFranc classclass Franc {

private private intint amount;

Franc(intint amount) {

thisthis.amount= amount; }

Franc times(intint multiplier) {

return return newnew Franc(amount \* multiplier); } publicpublic boolean boolean equals(Object object) {

Franc franc= (Franc) object; return 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:

Test-driven development 31 of 133

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

Test-driven development 32 of 133

**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: MoneyMoney classclass 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.

DollarDollar classclass Dollar extends extends Money {

private private intint amount; } Can it? No, the tests still all run. Now we can move the “amount” instance variable up to Money:

MoneyMoney classclass Money {

protected int protected int amount; amount; } DollarDollar classclass Dollar extends 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:

Test-driven development 33 of 133

DollarDollar publicpublic boolean boolean equals(Object object) {

Money dollar= (Dollar) object; Money dollar= (Dollar) object; return return amount == dollar.amount; } All the tests still run. Now we change the cast:

DollarDollar publicpublic boolean boolean equals(Object object) {

Money dollar= (Money) object; Money dollar= (Money) object; return return amount == dollar.amount; } To be communicative, we should also change the name of the temporary variable:

DollarDollar publicpublic boolean boolean equals(Object object) {

Money moneymoney= (Money) object; return return amount == moneymoney.amount; } Now we can move it from Dollar to Money:

MoneyMoney

publicpublic boolean boolean equals(Object object) { Money money= (Money) object; return 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.

Test-driven development 34 of 133

Tests in place, we can have Franc extend Money:

FrancFranc classclass Franc extends extends Money {

private private intint amount; } We can delete Franc’s field “amount” in favor of the one in Money:

FrancFranc classclass Franc extends 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:

FrancFranc publicpublic boolean boolean equals(Object object) {

Money franc= (Franc) object; return return amount == franc.amount; } Then we change the cast:

FrancFranc publicpublic boolean boolean equals(Object object) {

Money franc= (Money) object; return 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:

FrancFranc publicpublic boolean boolean equals(Object object) {

Money money= (Money) object; return 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

Test-driven development 35 of 133

**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.

MoneyMoney

publicpublic boolean boolean equals(Object object) {

Money money = (Money) object; return 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.

Test-driven development 36 of 133

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

Test-driven development 37 of 133

**Makin’ Objects**

The two implementations of times() are remarkably similar:

FrancFranc

Franc times(intint multiplier) {

return return newnew Franc(amount \* multiplier); } DollarDollar

Dollar times(intint multiplier) {

return return newnew Dollar(amount \* multiplier); } We can take a step towards reconciling them by making them both return a Money:

FrancFranc

Money times(intint multiplier) {

return return newnew Franc(amount \* multiplier); } DollarDollar

Money times(intint multiplier) {

return return newnew 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: publicpublic voidvoid testMultiplication() {

Dollar five = Money.dollar(5); Money.dollar(5); assertEquals(new Dollar(10), five.times(2)); assertEquals(new Dollar(15), five.times(3)); } The implementation creates and returns a Dollar:

MoneyMoney

staticstatic Dollar dollar(intint amount) {

return return newnew Dollar(amount); } But we want references to Dollars to disappear, so we need to change the declaration in the test:

Test-driven development 38 of 133

publicpublic voidvoid testMultiplication() {

MoneyMoney 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():

MoneyMoney abstract abstract class Money

abstract abstract MoneyMoney times( times(int int multiplier); ultiplier); Now we can change the declaration of the factory method:

MoneyMoney

staticstatic MoneyMoney dollar(intint amount) {

return return newnew 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:

publicpublic voidvoid testMultiplication() {

Money five = Money.dollar(5); assertEquals(Money.dollar Money.dollar(10), five.times(2)); assertEquals(Money.dollar Money.dollar(15), five.times(3)); } publicpublic voidvoid testEquality() {

assertTrue(Money.dollar Money.dollar(5).equals(Money.dollar Money.dollar(5))); assertFalse(Money.dollar Money.dollar(5).equals(Money.dollar Money.dollar(6))); assertTrue(newnew Franc(5).equals(newnew Franc(5))); assertFalse(newnew Franc(5).equals(newnew Franc(6))); assertFalse(newnew Franc(5).equals(Money.dollar 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.

Test-driven development 39 of 133

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():

MoneyMoney

staticstatic Money franc(intint amount) {

return return newnew 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

Test-driven development 40 of 133

**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:

publicpublic voidvoid testCurrency() {

assertEquals("USD", Money.dollar(1).currency()); assertEquals("CHF", Money.franc(1).currency()); } First we declare currency() in Money:

MoneyMoney

abstract abstract String currency(); Then we implement it in both subclasses:

FrancFranc

String currency() {

return return return return "CHF"; } DollarDollar

String currency() {

return return 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

Test-driven development 41 of 133

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.)

FrancFranc private private String currency; Franc(intint amount) {

thisthis.amount = amount; currency = currency = "CHF""CHF"; } String currency() {

return currency; return currency; } We can do the same with Dollar:

DollarDollar private private String currency; Dollar(intint amount) {

thisthis.amount = amount; currency = currency = "USD""USD"; } String currency() {

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

MoneMoney

protected protected String currency; String currency() {

return 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:

FrancFranc

Franc(intint amount, String currency String currency) {

thisthis.amount = amount; thisthis.currency = "CHF"; } This breaks the two callers of the constructor:

Test-driven development 42 of 133

MoneyMoney

staticstatic Money franc(intint amount) {

return return newnew Franc(amount, nullnull); } FrancFranc

Money times(intint multiplier) {

return return newnew Franc(amount \* multiplier, nullnull); } 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:

FrancFranc

Money times(intint multiplier) {

return return Money.franc(amount \* multiplier); } Now the factory method can pass “CHF”:

MoneyMoney

staticstatic Money franc(intint amount) {

return return newnew Franc(amount, "CHF""CHF"); } And finally we can assign the parameter to the instance variable:

FrancFranc

Franc(intint amount, String currency) {

thisthis.amount = amount; thisthis.currency = 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:

Test-driven development 43 of 133

MoneyMoney

staticstatic Money dollar(intint amount) {

return return newnew Dollar(amount, "USD""USD"); } DollarDollar

Dollar(intint amount, String currency) {

thisthis.amount = amount; thisthis.currency = currency currency; } Money times(intint multiplier) {

return return Money.dollar(amount \* multiplier); 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:

Test-driven development 44 of 133

MoneyMoney

Money(intint amount, String currency) {

thisthis.amount = amount; thisthis.currency = currency; } FrancFranc

Franc(intint amount, String currency) { super(amount, currency); super(amount, currency); } DollarDollar

Dollar(intint amount, String currency) { super(amount, 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

Test-driven development 45 of 133

**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.

FrancFranc

Money times(intint multiplier) {

return return Money.franc(amount \* multiplier); } DoDollarllar

Money times(intint multiplier) {

return 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?)

FrancFranc

Money times(intint multiplier) {

return return newnew Franc(amount \* multiplier, "CHF"); } DollarDollar

Money times(intint multiplier) {

return return newnew Dollar(amount \* multiplier, "USD"); } In Franc, though, we know that the currency instance variable is always “CHF”, so we can write: FrancFranc

Money times(intint multiplier) {

return return newnew Franc(amount \* multiplier, currency currency); } That works. The same trick words in Dollar:

DollarDollar

Money times(intint multiplier) {

return return newnew Dollar(amount \* multiplier, currency 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.

Test-driven development 46 of 133

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:

FrancFranc

Money times(intint multiplier) {

return return newnew MoneyMoney(amount \* multiplier, currency); } The compiler tells us that Money must be a concrete class:

MoneyMoney classclass Money

Money times(intint amount) {

return return nullnull; } 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:

MoneyMoney

publicpublic String toString() {

return 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():

MoneyMoney

publicpublic boolean boolean equals(Object object) {

Money money = (Money) object; return 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

Test-driven development 47 of 133

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.)

FrancFranc

Money times(intint multiplier) {

return return newnew FrancFranc(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: publicpublic voidvoid testDifferentClassEquality() {

assertTrue(newnew Money(10, "CHF").equals(newnew Franc(10, "CHF"))); } It fails, as expected. The equals() code should compare currencies, not classes:

MoneyMoney

publicpublic boolean boolean equals(Object object) {

Money money = (Money) object; return return amount == money.amount && currenc currency().equals(money.currency() y().equals(money.currency()); } Now we can return a Money from Franc.times() and still pass the tests:

FrancFranc

Money times(intint multiplier) {

return return newnew Money(amount \* multiplier, currency); } Will the same will work for Dollar.times()?

DollarDollar

Money times(intint multiplier) {

return return newnew MoneyMoney(amount \* multiplier, currency); } Yes! Now the two implementations are identical, so we can push them up.

Test-driven development 48 of 133

MoneyMoney

Money times(intint multiplier) {

return return newnew 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:

FrancFranc staticstatic Money franc(intint amount) {

return return newnew MoneMoney(amount, "CHF"); } Then Dollar:

DollarDollar

staticstatic Money dollar(intint amount) {

return return newnew MoneyMoney(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))); }

Test-driven development 49 of 133

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

Test-driven development 50 of 133

• 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

Test-driven development 51 of 133

**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

publicpublic voidvoid 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:

MoneyMoney

Money plus(Money addend) {

return return newnew 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.

Test-driven development 52 of 133

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:

publicpublic voidvoid 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:

publicpublic voidvoid 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:

publicpublic voidvoid testSimpleAddition() {

... Bank bank= newnew Bank(); Money reduced= bank.reduce(sum, "USD"); assertEquals(Money.dollar(10), reduced); } The sum of two Moneys should be an Expression:

Test-driven development 53 of 133

publicpublic voidvoid testSimpleAddition() {

... Expression sum= five.plus(five); Bank bank= newnew Bank(); Money reduced= bank.reduce(sum, "USD"); assertEquals(Money.dollar(10), reduced); } At least we know for sure how to get five dollars:

publicpublic voidvoid testSimpleAddition() {

Money five= Money.dollar(5); Expression sum= five.plus(five); Bank bank= newnew 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 Expression interface interface Expression Money.plus() needs to return an Expression:

MoneyMoney

Expression plus(Money addend) {

return return newnew Money(amount + addend.amount, currency); } Which means that Money has to implement Expression (which is easy, since there are no operations yet):

MoneyMoney classclass Money implements implements Expression We need an empty Bank class:

BankBank classclass Bank Which stubs out reduce():

BankBank

Money reduce(Expression source, String to) {

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

BankBank

Money reduce(Expression source, String to) {

return return Money.dollar(10); } We’re back to a green bar, and ready to refactor. First, reviewing, we:

Test-driven development 54 of 133

• 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

Test-driven development 55 of 133

**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:

BankBank

Money reduce(Expression source, String to) {

return return Money.dollar(10) Money.dollar(10); } is really the same as the “$5 + $5” in the test:

publicpublic voidvoid testSimpleAddition() {

Money five= Money.dollar(5); Expression sum= five.plus(five) five.plus(five); Bank bank= newnew 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:

publicpublic voidvoid 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:

Test-driven development 56 of 133

SumSum classclass Sum {

Money augend; Money addend; } This gives us a ClassCastException, because Money.plus() is returning a Money, not a Sum: MoneyMoney

Expression plus(Money addend) {

return return newnew Sum(thisthis, addend); } Sum needs a constructor:

SumSum

Sum(Money augend, Money addend) { } And Sum needs to be a kind of Expression:

SumSum classclass Sum implements 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):

SumSum

Sum(Money augend, Money addend) {

ththisis.augend= augend; thisthis.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:

publicpublic voidvoid testReduceSum() {

Expression sum= newnew Sum(Money.dollar(3), Money.dollar(4)); Bank bank= newnew 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.

Test-driven development 57 of 133

BankBank

Money reduce(Expression source, String to) {

Sum sum= (Sum) source; intint amount= sum.augend.amount + sum.addend.amount; return return newnew 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.)

BankBank

Money reduce(Expression source, String to) {

Sum sum= (Sum) source; return return sum.reduce(to); } SumSum publicpublic Money reduce(String to) {

intint amount= augend.amount + addend.amount; return return newnew 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:

Test-driven development 58 of 133

publicpublic voidvoid testReduceMoney() { Bank bank= newnew Bank(); Money result= bank.reduce(Money.dollar(1), "USD"); assertEquals(Money.dollar(1), result); } BankBank

Money reduce(Expression source, String to) {

if (source instanceof Money) return (Money) source; if (source instanceof Money) return (Money) source; Sum sum= (Sum) source; return 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.

BankBank

Money reduce(Expression source, String to) {

ifif (source instanceof instanceof Money) return return (Money) source.reduce(to); Sum sum= (Sum) source; return return sum.reduce(to); } MoneyMoney

publicpublic Money reduce(String to) {

return return thisthis; } If we add reduce(String) to the Expression interface:

Expression Expression

Money reduce(String to); We can eliminate all those ugly casts and class checks:

Test-driven development 59 of 133

BankBank

Money reduce(Expression source, String to) {

return 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

Test-driven development 60 of 133

**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:

publicpublic voidvoid testReduceMoneyDifferentCurrency() {

Bank bank= newnew 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:

MoneyMoney

publicpublic Money reduce(String to) {

intint rate = (currency.equals("CHF") & to.equals("USD"))

? 2 : 1; return return newnew 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: BankBank

Money reduce(Expression source, String to) {

return return source.reduce(thisthis, to); } Then the implementors: