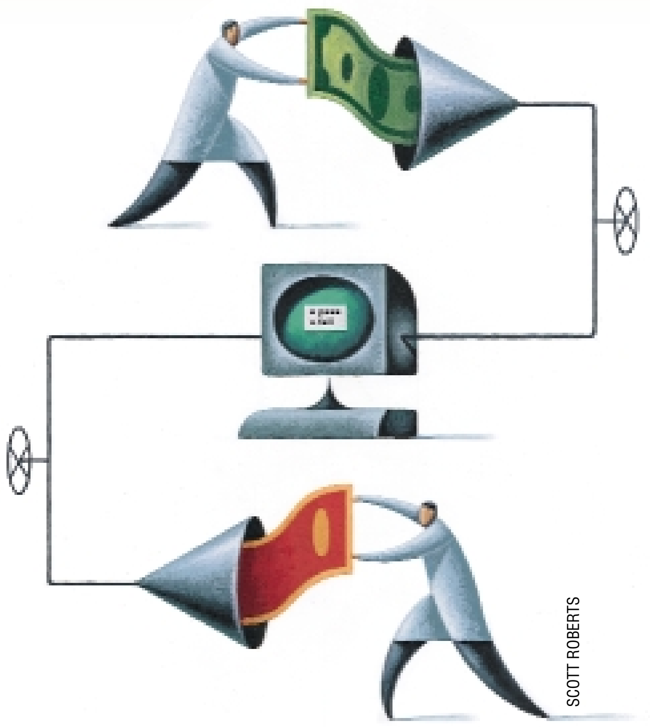
by Jeffreys Copeland and Haemer



Quality isn't really a free variable. The only possible values are "excellent" and "insanely excellent," depending on whether lives are at stake or not. Otherwise you don't enjoy your work, you don't work well, and the project goes down the drain.
– *extreme Programming explained*, Kent Beck

Animal testing is a terrible idea; they get all nervous and give the wrong answers.
–Stephen Fry

Testy, Aren't We?

Well buckaroos, we're back. Last month, we talked about eXtreme Programming (XP), particularly about eXtreme Testing (XT), –writing tests as you code, or even before, instead of after you're done.

This month, we want to drive the point home with an example.

Our first problem is how to find a good example.

We need a problem that we can explain, and write tests for before we start on the code. We also need to be able to show and explain both the tests and the code when we are done. All this needs to fit in our 2,000-word column limit.

Our solution? Theft.

After we gave him a copy of our last month's column, our friend, Dave Taenzer, pointed us at a wonderful article by Kent Beck and Erich Gamma, at `http://members.pingnet.ch/gamma/junit.htm`, which takes on this same chore in Java.

Perfect. We'll just solve the problem all over again in Perl.

Look at how much we get from this simple intellectual piracy: XP advocates writing tests before writing code. What tests should we write before we begin coding? Easy. The tests we want to pass are already specified in the Beck/Gamma article. We'll just translate them into Perl, using the `Test` and `Test::Harness` modules (which will give us a chance to talk about those modules, too).

Moreover, whenever we can't figure out how to explain something, we can just claim that the solution is trivial and refer you to the original article.

Useful tip: if you're going to steal, steal from people who know what they are doing.

• Kent Beck invented XP in the first place. Read *extreme Programming explained* [ISBN 201-61641-6]. (We said that last month, but a little repetition never hurts.)

• Erich Gamma is the first author of the popular Gang-of-Four book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, ISBN 0-201-63361-2).

We won't include any of their code here–we're pirates, not plagarists–but to make comparisons as easy as possible for those reading the articles side-by-side, we'll try to parallel the Beck and Gamma design as closely as we can.

Money, Money, Money...

First we'll sketch the problem. The goal is to create a class that stores and adds collections of money.

Beck and Gamma work in Switzerland, an international banking center, so the objects must be able to hold more than one kind of currency. We also need two basic operations: addition and a test for equality.

• If one object has 200 Kuwaiti dinars and a second has 12 dinars and 300 Romanian lei, their sum should have 212 Kuwaiti dinars and 300 Romanian lei.

• Two objects that each have exactly

Figure 1

```
 1 #!/usr/bin/perl -w
 2 # $Id: test_equals,v 1.5 2000/06/13 03:08:40 jsh Exp jsh $

 3 use strict;
 4 use Test;
 5 use Money_fixture;

 6 ok($f12chf != undef);
 7 ok($f12chf, $f12chf);
 8 ok($f12chf, new Money(CHF=>12));
 9 ok($f12chf ne $f14chf);

10 BEGIN { plan tests => 4 }
```

Figure 2

```
 1 #!/usr/bin/perl -w
 2 # $Id: Money_fixture.pm,v 1.2 2000/06/13 01:56:03 jsh Exp $

 3 package Money_fixture;

 4 use strict;
 5 use Exporter;
 6 use Money;

 7 our (@ISA, @EXPORT);
 8 @ISA = qw(Exporter);
 9 @EXPORT = qw($f12chf $f14chf $f26chf $f7usd $f21usd $fmb1 $fmb2);

10 our $f12chf = new Money(CHF=>12);
11 our $f14chf = new Money(CHF=>14);
12 our $f26chf = new Money(CHF=>26);
13 our $f7usd  = new Money(USD=>7);
14 our $f21usd = new Money(USD=>21);
15 our $fmb1   = new Money(CHF=>12, USD=>7);
16 our $fmb2   = new Money(CHF=>14, USD=>21);

17 1;
```

Figure 3

```
 1 #!/usr/bin/perl -w
 2 # $Id: test_bag_equals,v 1.4 2000/06/13 03:08:40 jsh Exp jsh $

 3 use strict;
 4 use Test;
 5 use Money_fixture;

 6 ok($fmb1 != undef);
 7 ok($fmb1, $fmb1);
 8 ok($fmb1 != $f12chf);
 9 ok($f12chf != $fmb1);
10 ok($f7usd != $fmb1);
11 ok($fmb1 != $fmb2);

12 BEGIN { plan tests => 6 }
```

12 dollars, 16 yen and 50 Swiss francs should have the same values, even if they are different objects.

The Tests

Tests first. The structure of all our tests are similar, so we'll go through one line-by-line to explain how they're laid out.

The first thing to test is whether we've implemented comparisons correctly. We know from the outset what tests we'll use: the very same ones Kent and Eric do. Here is our dramatic reading of the first set:

In Figure 1, Line 1 is a shebang line to invoke the perl interpreter, with the `-w` flag, to warn of possible errors. Line 2 is automatically generated, configuration-management information from RCS. Invoking the nit-picky strict pragma is Line 3. These three lines are our nod to the homily, "Cleanliness is next to Godliness."

Line 4 brings in the Test module, which defines some utility functions for testing. We'll be using `plan()` and `ok()`. As with all other things Perl, `Test.pm` is available from the CPAN, (`http://www.perl.com/CPAN/`).

Line 5 introduces our test fixture: a context to run the tests in. Typically, a test fixture is a set of routines, constants and variables to be shared among a set of tests. We'll show you ours in a minute; for now, we'll say that it creates things like `$f12chf`, a `Money` object containing a dozen Swiss francs.

Lines 6 through 9 use `ok()`, which tests an observed outcome against an expected one.

In `Test.pm` version 1.14, `ok()` accepts strings, numbers, regular expressions and even function pointers as arguments. Function pointers are dereferenced and evaluated, and regular expressions trigger a regex match, so you can look for patterns, not just exact matches. Invoked with a single argument, `ok()` just looks to see whether it's non-zero.

The first test asks whether `$f12chf` has a value. It had better. It also tests the use of `!=`. The second asks if `$f12chf` equals itself; the third, whether it has the same value as a newly created object containing 12 more Swiss francs; and the fourth, whether it has a different value from `$f14chf`, which contains 14 Swiss francs. The fourth also tests the use of `ne`.

The last line calls `plan()` to say we'll be running four tests. Putting the call inside a `BEGIN{}` provides this information at compile time. `Test.pm` forces us to do this and aborts the compilation if we don't.

Now's a good time to show our test fixture (Figure 2).

As advertised earlier, the module just creates a few `Money` objects. Some only have one currency type, others have two. Lines like our

```
$fmb1 = new MoF 0
ney(CHF=>12, USD=>7);
```

are what we think calls to our constructor should look like. Remember, we still haven't written any code; writing the tests is forcing us to define the interfaces.

By letting our constructor, which we call `new()` out of habit, take currency/amount pairs as arguments, it can also be an initializer. (Beck and Gamma use separate classes for objects that hold a single kind of currency and more than one kind. This seems like overkill for Perl's relaxed view of the world.)

The keyword `our` is new to Perl 5.6 and declares lexically scoped globals. Earlier Perls needed a `use vars` pragma.

These objects in hand, we can show our tranlation of Beck and Gamma's second bucket (Figure 3), which checks whether bags of mixed currency compare correctly.

Adding Addition

What should addition look like? The simplest design we can think of is this:

```
$a = new Money(USD=>10);
$b = new Money(USD=>20);
$c = $a + $b;
```

Let's assume we can do that. Figure 4 shows the Beck and Gamma simple test case, and Figure 5, their tests for mixed-currency addition.

Each of our tests is just a Perl program; we'll run individual tests by executing them from the command line.

Having written all the tests and designed the interfaces, all that's left is writing the class. And how will we know when we've written it?

The tests will pass.

The Code

Figure 6 shows our first attempt.

We won't do a full exegesis–our focus is the testing, not the code–but the perl is straightforward. Two noteworthy points:

- Our comparison functions overload both

Figure 4

```
#!/usr/bin/perl -w
# $Id: test_simple_add,v 1.4 2000/06/13 03:08:40 jsh Exp jsh $

use strict;
use Test;
use Money_fixture;

ok($f26chf, $f12chf + $f14chf);

BEGIN { plan tests => 1 }
```

Figure 5

```
#!/usr/bin/perl -w
# $Id: test_mixed_add,v 1.4 2000/06/13 03:08:40 jsh Exp jsh $

use strict;
use Test;
use Money_fixture;

ok($fmb1, $f12chf + $f7usd);
ok($fmb2, $f21usd + $f14chf);
ok($fmb1 + $fmb2, $f7usd + $f21usd + $f26chf);
ok($fmb1 + new Money(CHF=>-12), $f7usd);

BEGIN { plan tests => 4 }
```

Figure 6

```
#!/usr/bin/perl -w
# $Id: Money.orig.pm,v 1.1 2000/06/13 03:08:40 jsh Exp jsh $

use strict;
package Money;

sub new {
  my $class = shift;
  bless {@_}, $class;
}

sub equals {
  use overload ('==' => \&equals, 'eq' => \&equals );

  my ($s1, $s2) = @_;
  my @k1 = keys %$s1;
  my @k2 = keys %$s2;
  return 0 unless @k1 == @k2;
  foreach (@k1) {
   return 0 unless $s1->{$_} == $s2->{$_};
  }
  1;
}

sub not_equals {
  use overload ('!=' => \&not_equals, 'ne' => \&not_equals );
  !equals @_;
}

sub stringify {
  use overload('""' => \&stringify);
```

Continued on Page 42

```
  my $s = "";
  while (my ($k,$v) = each %{$_[0]}) {
    $s .= ", " if $s;
    $s .= "$k => $v";
   }
   $s;
  }

  sub plus {
   use overload('+' => \&plus);

   my ($m1, $m2) = @_;
   my %s = (%$m1, %$m2);
   my $s = {};
   foreach ( keys %s ) {
    $s->{$_} = (defined $m1->{$_} ? $m1->{$_} : 0) +
        (defined $m2->{$_} ? $m2->{$_} : 0);
   }
   $s;
 }

1;
```

==/!= and eq/neq because we're not sure which pair we'll want to use.

• We wrote a stringify function, overloading '""', to let print() pretty-print the contents of a Money object. Oh, but wait: we need to test stringify.

```
#!/usr/bin/perl -w
# $Id: test_printing,v 1.1 2000/06/13 03:08:40 jsh Exp jsh $

use strict;
use Test;
use Money_fixture;

ok("$f12chf", "CHF => 12");
ok("$fmb1", "CHF => 12, USD => 7");

BEGIN { plan tests => 2 }
```

Writing this new test reveals a design problem. A test requires predictable output, so stringify(), needs print hash elements in a predictable order. We fix stringify to look like this:

```
sub stringify {
  use overload('""' => \&stringify);

  my $m = shift;
  my $s;
  foreach (sort keys %$m) {
    $s .= $s ? ", $_ => $m->{$_}" : "$_ => $m->{$_}";
  }
  $s;
}
```

At last, we're ready to run the test suite. As soon as we do–voila! bugs. Comparison and simple addition seem to work fine, but our mixed addition fails.

```
1..4
ok 1
ok 2
not ok 3
# Test 3 got: 'HASH(0x80ea938)' (./test_mixed_add at line 12)
#  Expected: 'HASH(0x82339a0)'
not ok 4
# Test 4 got: 'HASH(0x8234f44)' (./test_mixed_add at line 13)
#  Expected: 'USD => 7'
```

In fact, our addition results aren't even printing in the error output. We already know that Money objects print correctly, so … addition must not be giving us Money objects.

Perl's lack of strict-typing has let us write a plus() that does not return the kind of object we want. Here's our next version.

```
sub plus {
 use overload('+' => \&plus);

 my ($m1, $m2) = @_;
 my %s = (%$m1, %$m2);
 my $s = new Money;
 foreach ( keys %s ) {
  $s->{$_} = (defined $m1->{$_} ? $m1->{$_} : 0) +
      (defined $m2->{$_} ? $m2->{$_} : 0);
 }
 $s;
}
```

Should we stop and write a test to check that plus() returns a Money object? No. We already have one. That's how we knew it didn't!

Okay, now the third test passes, but the fourth still doesn't, for the same reason that Beck and Gamma's first version didn't pass their tests: Our code thinks that CHF=>0, USD=>7 is different from USD=>7. Paralleling the original paper, we'll let plus() eliminate '0' values before returning.

```
 my $sum = (defined $m1->{$_} ? $m1->{$_} : 0) +
   (defined $m2->{$_} ? $m2->{$_} : 0);
 $s->{$_} = $sum if $sum;
```

With this change, everything passes and we're done.

Test Harnesses

Having illustrated unit tests, Beck and Gamma move up a level to show a test harness: a program that runs a suite of tests, and reports the results in a stereotyped way.

Beck and Gamma develop a TestSuite object and a TestRunner tool to drive and oversee the Java test in their paper.

Since Perl provides a Test::Harness module, Figure 7, built to work with Test, we will use it to write our very own tiny harness. Here's what its output looks like:

```
test_bag_equal......ok
test_equal..........ok
test_mixed_ad.......ok
test_printin........ok
test_simple_ad......ok
test_subclassin.....ok
All tests successful.
Files=6, Tests=21,
   3 wallclock secs
  ( 2.65 cusr + 0.19 csys = 2.84 CPU)
```

Nearly all the work is done by line six, which runs every test specified on the command line (or all tests whose name starts with `test_`, by default). `runtest()` normally prints its timing and statistics summary (the last two lines of our sample output) to `STDERR`. By enclosing our call in an `eval()`, the string is captured in `$@`.

Our call to `catch()` prints this to `STDOUT`, but if we set up a `cron` job to invoke our harness with a `-m` flag, `catch()` will use `mail_me ()` to mail us the summary. This lets us promote conformance tests into regularly scheduled regression tests.

Figure 7

```
 1 #!/usr/bin/perl -w -s
 2 # $Id: money_suite,v 1.5 2000/06/13 03:08:40 jsh Exp jsh $

 3 use strict;
 4 use Test::Harness;
 5 our $m;# -m switch for "mail me"

 6 eval { runtests(@ARGV ? @ARGV : <test_*>); };
 7 catch($@);

 8 sub catch {
 9   my $results = shift;
10   print $results;
11   mail_me ($results) if $m;
12 }

13 sub mail_me {
14   my $message = shift;
15   my $author = (getpwuid($<))[0];
16   open(MAIL, "|/usr/lib/sendmail -oi -t")
17    or die "Can't fork sendmail: $!\n";

18 print MAIL <<"EOF";
19 From: $0 (Your Friendly Test Harness)
20 To: $author;
21 Subject: Test summary

22 $message

23 EOF
24   close(MAIL) or die "Argh!  Can't close(MAIL)\n: $!\n";
25 }
```

We Finish With Testing (for Now)

Okay. We're done.

Dave Taenzer, who started us on this journey by giving us a book, says software development methodologies are like fad diets: no matter how many haven't worked, people are always eager to try another. "The high-carbohydrate-low-fat diet? No, no. You want the grapefruit-and-protein diet." "You've been using strict typing? No wonder you've been having problems! You want typeless languages."

Despite this mild cynicism, Dave likes both *extreme Programming explained* and *Design Patterns: Elements of Reusable Object-Oriented Software*. In our experience, Dave's recommendations are worth listening to.

We are also longtime fans and practitioners of eXtreme Testing: developing conformance tests and test harnesses from the beginning, and then writing code to pass to the conformance tests, instead of testing after the fact. We always like anything that plays to our prejudices.

We hope we've helped some readers see that good testing needn't mean armies of low-level, "keyboard monkeys" walking through testing scripts. Good testing is fun, intellectually challenging and technical.

We've skipped some things that deserve mentioning. We'll mention three and shut up:

• Perhaps the best-known software test harness is Cygnus's DejaGnu, originally built by Rob Savoye, on top of Expect to oversee tests of Cygnus's suite of GNU utilities. DejaGnu is very mature and can handle jobs as sophisticated as driving tests from one machine that cross-compile on a second machine and run on a third. If you need a really sophisticated test harness, instead of something simple and roll-your-own, consider DejaGnu (`http://dejagnu.sourceforge.net/`).

• The simple `Test` and `Test::Harness` modules worked fine for us. As they say in the Perl world, though, There Is More Than One Way To Do It (pronounced "tim-toady"). `http://c2.com/cgi/wiki?PerlUnit`, has pointers to several projects to develop testing modules more like `junit`, including an ongoing Open Source project that you can join and help develop.

• Kent Beck has written us to correct our extreme capitalization: "… the capitalization is only funny on the book cover. XP is spelled out 'Extreme Programming'." Thanks, Kent.

Until next time, happy trails.

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer (`jsh@usenix.org`) *works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.