
Self-Testing Code

David Thureson

EC Wise, Inc.

dthureson@ecwise.com

© 2001, EC Wise, Inc., All rights reserved

Self-Testing Code	2
Benefits	2
Responsible Code.....	2
Defensive Code.....	3
Confidence.....	4
Self-Documenting Code.....	4
Specification.....	5
Complete Code.....	5
Ease of Integration.....	5
Debugging.....	5
Refactoring.....	6
Better Estimation.....	6
Simplicity.....	6
Support for Change.....	7
Faster Coding.....	7
Techniques	8
Assertions.....	8
Assertions vs. Exceptions.....	9
Invariants.....	10
Explicit States.....	11
Embedded Unit Tests.....	12
Displays.....	13
Sample Instances.....	13
Coverage Tools.....	14
Unfrozen Caveman Coding vs. Cleverness.....	15
Practices, Summarized	16

Self-Testing Code

The English Positivist philosophers Locke, Berkely, and Hume said that anything that can't be measured doesn't exist. When it comes to code, I agree with them completely. Software features that can't be demonstrated by automated tests simply don't exist. I am good at fooling myself into believing that what I wrote is what I meant. I am also good at fooling myself into believing that what I meant is what I should have meant. So I don't trust anything I write until I have tests for it. The tests give me a chance to think about what I want independent of how it will be implemented. Then the tests tell me if I implemented what I thought I implemented.

-- Kent Beck, *Extreme Programming*, p. 45

Benefits

This white paper presents an overview of the benefits of self-testing code, without specifics. For a detailed presentation or collaboration on applying the techniques described herein, please contact EC Wise, Inc.

Responsible Code

Many developers felt no shame if bugs were found in their code after the product had shipped. They'd ask indignantly, "Why didn't Testing find that bug before we shipped?" Testing should have responded, "Why did you put that bug in the product in the first place?"

-- David M. Moore, *Director of Development, Microsoft*

If I write a bug into a class, and then check that in to source control for other people to use, I'm potentially wasting not only my own time, but that of:

Other programmers

Testers

Managers

Customers

Sales people

Support people

This is time they could have spent making the product better, or shipping it early, or maybe having a nice meal or a good workout, or going to their son's ballet recital or their daughter's little league game. Whatever they would have done with that time, dealing with my bug was probably very low on their priorities list. Yet I have stolen a small chunk of their lives from them, a chunk they can never get back. I should be ashamed of myself.

Now, granted nobody's perfect, and there are all sorts of errors which are not addressed by unit testing, such as miscommunications over requirements, but certain kinds of errors are common, and simple to test for:

Forgetting to initialize an object

Forgetting to negate a conditional

Returning true when I meant false

Using the wrong variable (common after cut-and-paste)

Failing to handle edge cases (zero, negative numbers, nulls, empty strings, etc.)

I don't want someone staying up all night pulling their hair out because I forgot to type a "!" when I should have.

Defensive Code

Garbage in does not mean garbage out. A good program never puts out garbage, regardless of what it takes in. A good program uses "garbage in, nothing out"; "garbage in, error message out"; or "no garbage allowed in" instead. By today's standards, "garbage in, garbage out" is the mark of a sloppy program".

-- Steve McConnell, Code Complete, p. 97

It can be sufficient [in an example case] to make it clear to clients that passing a null pointer will result in undefined behavior, backing that up with an assert statement at the beginning of the function implementation... If someone misuses the function, it is their own fault – and they'll find out about it soon enough!

-- John Lakos, Large-Scale C++ Software Design

The converse of responsibility is defensiveness. I can't get much done if I'm spending a lot of time looking through code for which I'm responsible, trying to find a bug, when really the bug is elsewhere. Defensive coding practices allow me to build a wall around one module and know that it is working fine, and if some other code has a problem, the cause is outside of the walls.

Defensive code is also responsible code, because it puts me in a much better position to save time for whoever is calling my code. We don't have to have conversations like:

"Your code isn't working"

"Really? I haven't changed anything. Are you calling it right?"

"Yeah, I mean it worked yesterday."

"Well, I haven't changed it. At least, I'm pretty sure it was nothing that would have affected what you're doing."

"Hmmm, I'll go back and see if I can figure out what I did. Man, I've been trying to track this down for two hours. Maybe I'll just head to lunch and hit it this afternoon."

This is how it often goes with non-testing code. It's exhausting, painful, seemingly endless. Things take all day which should take a couple of minutes.

And conversations like this are endless as well, because neither party knows what we mean by "calling it right". We can trace through the program with a debugger, but that only goes so far. Maybe the problem was with the hundredth time his code called mine, and we'd never get there in a debugger unless we knew what we were looking for.

Consider how much better it would have gone if the first time the code was run after the bug was introduced in the calling module, a message popped up saying, in effect "Hey, you can't call method xyz() with an inactive

Company as a parameter". The bug would likely have been obvious, since it was just introduced minutes earlier, and the programmer would have solved his own problem in minutes, and not even bothered me with it. By coding defensively, I've saved both of us a lot of aggravation.

Confidence

Every time a programmer writes some code, they think it's going to work. So every time they think some code is going to work, they take that confidence out of the ether and turn it into an artifact that goes into the program. The confidence is there for their own use. And because it is there in the program, everyone else can use that confidence, too.

-- Kent Beck, Extreme Programming, p. 115

I was able to enjoy this confidence during a bug that came up while I was editing this document. Someone came up to me and asked about a class that seemed not to be. I had a lot of confidence that it was working fine, since I knew that it was self-testing, and that I had run all of the tests before checking it in. The other programmer and I were able to hit a button and run the class test and both see that the code worked fine. This saved me from having to spend time wondering if my code was breaking, and trying to think of test cases or stepping through it line by line from the call that was breaking.

It's not a matter of escaping blame, or claiming infallibility. Confidence in your code means simply that you know what it can and can't do, and you can demonstrate this in a few seconds, and nobody wastes time looking in the wrong place when something goes wrong. This is a favor to everybody.

Self-Documenting Code

Within every method are assumptions about the data, such as that a certain amount is positive, or that a referenced object is not null.

The programmer knows these assumptions when writing the code, and they are implicit somewhere in the code, if you look hard enough. The code will fail eventually, somewhere, if the assumptions turn out to be incorrect, if that number happens to be negative or that object turns out to be null.

But someone trying to figure out what went wrong will not necessarily know what the assumptions were, and may think along the lines of "I need to add some code here to handle negative values, which the original programmer for some reason didn't think of", and may introduce new bugs in the attempt, when the real problem was somewhere else in the code, in that the number was negative when it shouldn't have been.

The assumptions may be stated in comments, but there is no guarantee that they are correct at runtime, and the comments and the code may get out of sync. When the assumptions are stated unambiguously in executable code, they work as both comments and code.

Another way in which self-testing code is also self-documenting is in that it provides examples of how to use the class. It's common to deliver a class for use by someone else and have that person then ask how it should be used. It's not always clear just from the constructor and the public methods, because they may provide much more functionality than is needed in the usual case. You could write some sample usage code in a document, but there is no guarantee that it actually runs, and a simple typo or a missing step here would lead to more confusion. If the code is self-testing, examples of usage are built in. They're guaranteed to run correctly, and they will never be out of date when the class changes.

The test code also provides context for using the class. For instance, the constructor may require a connection to a certain database, and maybe the programmer who will use your class doesn't know how to get that connection (which drivers, passwords, etc.), which leads to another round of questions. If the code has built-in tests, they wouldn't have run at all unless there was code within them to create that database connection, so the user of the class is provided with a fully working example of usage.

Specification

One method of clarifying the intent of the requested class, and mapping it to requirements, is to specify some of the test cases. These will flow naturally from use cases and scenarios. Once they're specified, all parties know that the class must fulfill, as part of its contract, at least the given test cases. It can be delivered with the confidence that it supports the known scenarios.

Complete Code

If you write the test code before writing the class code, you know in advance what the class code will have to do. Since the test code focuses mainly on the external interface of the class, it shows what public methods you need, and what calculations must be done internally.

It's easy to write the whole class, confirm that it all compiles and makes sense, even run through a code review and have everyone agree that it looks fine, and check it in to the master sources, all without realizing that it's missing a certain public method. The class "works" fine, it just doesn't fulfill its contract with the outside world. But if there were test cases which needed that method, its absence would be apparent on the first compile.

Kent Beck says, simply, "When you can't think of any tests to write that might break, you are completely done." (p. 45). Of course you also have to step back and see if you're meeting the requirements, but the initial set of test cases hopefully already covers that.

We need to be able to walk away from code, to check it in and call it done, as long as:

it's reasonably clear

it performs well enough

it fulfills the requirements

it tests out perfectly

Unless we can define these four criteria, and know when we've hit them, we're never done with anything.

Ease of Integration

If we integrate early and often, we won't have major surprises close to the release date. OK, this is obvious, but how do you do this without spending all of your time integrating? Self-testing code can help to quickly identify and narrow down any problems during integration. If one class breaks its contract with another, we know instantly.

Debugging

I find that with self-testing code (including the whole toolkit of assertions, invariants, displays, and class tests), I rarely need to do step-by-step debugging. Problems tend to leap out at me the moment they're introduced. Debugging consists mostly of looking at a call stack printout and seeing where something went wrong, rather than where in hundreds of other calls something went wrong.

Refactoring

If you want to refactor, the essential precondition is having solid tests... I don't see this as a disadvantage. I've found that writing good tests greatly speeds my programming, even if I'm not refactoring. This was a surprise to me, and it's counterintuitive for many programmers...

-- Martin Fowler, Refactoring, p. 89

We should take advantage of opportunities to clean up the code, make it simpler, more extendable, less fragile. But we can't refactor unless we know that each refactoring has not broken the code. Built-in unit testing allows us to do a refactoring, then call it good only when we can prove that the code still works as it did before.

Better Estimation

Self-testing code puts more of the burden of construction up front. Without it, it's relatively quick and easy to add functionality to code, at least at first. Because the cost of debugging the code is deferred, it seems like no big deal to add a feature. When you test before or during the initial cut at the code, though, the true cost is much more apparent, because more of it must be paid at the time.

Since I've started doing extensive built-in testing, I've had many internal dialogues along the lines of "I could add feature X in only a few dozen lines of code. But how much work would it be to test it well, and know that it's done? Do we really need feature X? Nahh, forget it." Multiply this by a couple hundred and I can save a lot of time on a project simply by having a better sense of the true cost of a feature.

And the dialogue is not just internal. Very few programmers give estimates that include communication and debugging time, except as a very rough fudge factor. If I didn't do self-testing code, and a given feature might take half a day to code, I'd estimate it as half a day to whoever was making the decision, and this person would probably assume that once this half day was over, that was it. But this doesn't include the three or four hours of communication to the users of this feature, the debugging time that might hit me at random for several hours some time in the next month, the testers and coders and managers and users who might be idled waiting for the fix, etc. The true cost of this feature might really be a week or more. But the decision to go ahead with it was based on half a day.

On the other hand, if by estimating the code I mean estimating the time to have it fully written and documented and tested, a feature that I could walk away from with confidence that it will work fine and I am not likely to hear from it again, that might be two days for the same feature. Not only is this lower than the total time without testing, but it's a much more accurate number. It's the real cost. From this, one can decide whether the feature is justified.

So we win either way. Either the feature is worth the time, and it gets done well and causes no problems for anybody, or it's not worth the time and I can do other tasks. We remain dedicated to living in reality.

Simplicity

Self-testing code will tend toward simplicity of function, for two reasons.

First, by writing the test cases first, and if necessary clarifying the requirements to do so, we know what the class is supposed to do, and we can code to requirements and nothing more.

Second, because much of the cost of developing a class is paid up front, in a manageable block rather than scattered over several people and times, its more apparent when a requested change will be too costly. It's easier to decide not to do the change, or to find a simpler alternative, when the costs are clear.

Support for Change

Several factors come out of the story about what made our code easy to modify, even after years of production:

- A simple design, with no extra design elements – no ideas that weren't used yet but were expected to be used in the future.

- Automated tests, so we had confidence we would know if we accidentally changed the existing behavior of the system.

- Lots of practice in modifying the design, so when the time came to change the system, we weren't too afraid to change it.

With this shift in assumption about the cost of change comes the opportunity to take an entirely different approach to software development. It is every bit as disciplined as other approaches, but it is disciplined along other dimensions. Instead of being careful to make big decisions early and little decisions later, we can create an approach to software development that makes each decision quickly, but backs each decision with automated tests, and that prepares you to improve the design of the software when you learn a better way to design it.

-- Kent Beck, Extreme Programming, pp 24-25

What he said.

Faster Coding

Whether self-testing code is faster or slower to write depends on where you measure from. The initial coding may take longer, but there is less time spent tracking down problems. OK, this makes sense in the long term, but are there ways that self-testing code will make the initial coding itself faster? Are there immediate returns in speed?

I'd say no. In my experience it takes about 3 times longer to code a class using all of the techniques described here. However, the benefits in:

Simplicity

Self-documentation

Clearer specification

Defensiveness

Easier debugging

mean that the first cut of code *that actually does what it's supposed to do and from which I can walk away with confidence* should come about faster. I don't really care how fast I could code something that may or may not work. That's not a legitimate finish line.

There is a way that some of these techniques will speed up initial coding, though. Steve McConnell makes the distinction between bugs that we create and find and fix before ever checking in the code, and those that get in to the master sources and cause problems later. Even in the initial coding of a class, there is a certain amount of time spent tracking down bugs. And these bugs still take time, even if we find them and fix them during initial coding, and they never get into the master sources, and they never appear on a bug list and nobody else ever knows about them. Many categories of these bugs can be caught faster with self-testing code.

Techniques

Assertions

If it can't happen, use assertions to ensure that it won't. Whenever you find yourself thinking "but of course that could never happen," add code to check it. The easiest way to do this is with assertions.

-- Andrew Hunt and David Thomas, The Pragmatic Programmer

An assertion is a statement that I as the programmer expect to be true. If it's false, there's a bug somewhere (or the assertion itself is wrong). I want to know as soon as possible when I introduce a bug that causes the assertion no longer to be true. So, for instance, after a call like:

```
Carrier carrier = benefitPlan.getCarrier();
```

it might be fine for the returned value to be null, indicating that this benefit plan has no assigned carrier yet, or it could be that it should never be null, and if it's null there's a bug in BenefitPlan (or something that it calls) that allows carrier to be null.

If I'm going to do something with carrier right away, I'll know soon enough if it's null:

```
Carrier carrier = benefitPlan.getCarrier();  
String url = carrier.getUrl();
```

I'll get a null pointer exception on the second line. So far so good. But later the code might read:

```
Carrier carrier = benefitPlan.getCarrier();  
String url = UriMaker.makeURL(carrier);
```

Now my code runs fine with a null carrier and happily passes it on to another class. The error won't be spotted until something uses carrier, which may be several calls later. Then when the null pointer exception finally hits, we'll have several classes that could be at fault. Also, the UriMaker.makeURL() code may not use the carrier parameter yet, such as if it were initially written as a stub with a hard coded return value. So it might be weeks until the bug in BenefitPlan is discovered.

If instead I have an assertion:

```
Carrier carrier = benefitPlan.getCarrier();  
Assert.check(carrier != null);  
String url = UriMaker.makeURL(carrier);
```

then I will discover the bug much closer to its source, both in code and in time.

We don't assert against conditions that could legitimately arise at runtime due to the environment. So if BenefitPlan.getCarrier() were something that made a remote HTTP request, and it's reasonable to expect that sometimes those will fail, the method will probably throw an exception, and it's up to the code in Enrollment to either do something to recover from the exception, or to pass it on. Handling this case with an assertion would fail for two reasons, because:

Assertions don't know how to do anything constructive. They just stop the program.

Assertions disappear in the production code.

Likewise, if benefitPlan.getCarrier() may legitimately return null, then we deal with the null case with a simple conditional rather than an assertion, something like:

```
Carrier carrier = benefitPlan.getCarrier();
```

```
if (carrier == null) {  
    return "No Carrier specified;";  
} else {  
    return carrier.getName();  
}
```

The specific form of the assertions is geared to make them:

Easy to write

Easy to read

Nonexistent in the production code

They take the form:

```
Assert.check(Assert.ENABLED && (oneCheck()));  
Assert.check(Assert.ENABLED && (!anotherCheck()));  
Assert.check(Assert.ENABLED && (thing1 != thing2));  
Assert.check(Assert.ENABLED && (thing1 != null));
```

The only thing of interest, the only thing that varies, is the part between the inner parentheses after the &&. It's very quick to copy and paste and modify to get the assertions we need.

Since Java compilers generally short-circuit conditionals, the actual test part of the assertion is not executed if Assert.ENABLED is false. A statement like:

```
Assert.check(Assert.ENABLED && (oneCheck()));
```

becomes, more or less:

```
Assert.check(false);
```

and the Assert.check() method itself returns immediately if Assert.ENABLED is false, regardless of the parameter passed into it. Since the value of Assert.ENABLED is known at compile time, the compiler can optimize the entire call out of the code. So in production it's as if the assertion is not there at all.

Assertions vs. Exceptions

Every Java book has a chapter on Exceptions, so I won't go into them here. But we should look at the decision between assertions and exceptions for handling a particular error. Generally exceptions are better suited for things that could legitimately go wrong at runtime, such as:

Unable to make a database connection

Database connection is lost

Unable to connect to an HTTP server

File not found

File is read only

None of these is necessarily caused by a coding error, but could happen due to problems in the external environment. There will never be a time when one can say that the code doesn't have to worry about these conditions at runtime. Even so, for many of these cases, it often makes sense to figure out the problem with a conditional rather than an exception. For instance, if the file may not be found, this can be treated as a normal occurrence, and checked for with normal (non-assert, non-exception) code. But if you're in the middle of reading from a file and it suddenly disappears, that's a good candidate for an exception.

Because final production code needs to handle these conditions, exceptions provide a careful structured way to surface problems to the calling code, where they can be dealt with or passed further up the calling chain. This means that the calling code must think about the exceptions and handle them.

Assertions, on the other hand, are all about catching coding and design errors. They should be quick and cheap and obvious when they fail. No recovery is necessary. An assertion failure means something has gone wrong in the logic of a particular class, or in its contract with other classes, and the first priority is finding and fixing the problem. Therefore their favored action upon failing an assertion is to terminate the program on the spot.

Exceptions are designed to allow recovery from an unexpected failure. Recovery is great in a production system, but costly during development, because recovery can hide errors.

Invariants

An invariant is a set of conditions that must be true for an object at all times from the perspective of the outside world. That is, I should be able to check that the object is in a legal state at the beginning and end of any public method. This reassurance of a legal state helps to avoid several kinds of errors. For instance, if an Employee always has a pointer to a default Beneficiary and a secondary Beneficiary, and at least one Phone, the invariant might be:

```
private boolean invariant() {  
  
    // There is always a primary and a secondary beneficiary.  
    Assert.check(primaryBeneficiary != null);  
    Assert.check(secondaryBeneficiary != null);  
    Assert.check(primaryBeneficiary != secondaryBeneficiary);  
  
    // There must always be one Phone.  
    Assert.check(phones != null);  
    Assert.check(phones.size() > 0);  
    Assert.check(AssertLib.checkListItemTypes(phones, Phone.class));  
  
    return true;  
}
```

Everything that I believe to be true about the object is stated explicitly in code in the invariant. Any time I introduce a bug that allows the object to be in an illegal state, the invariant catches it immediately.

The `AssertLib.checkListItemTypes()` call checks the type of every item in a Collection, and checks that every item is not null. It's meant to catch common copy-paste coding errors. For instance, I might start with:

```
public void addPhone(Phone newPhone) {  
    phones.add(newPhone);  
}
```

and then copy and modify to get the similar code for benefit plans:

```
public void addPlan(Plan newPlan) {  
    phones.add(newPlan);  
}
```

This will compile and run just fine, and there's no problem with it except that it's completely wrong. We'll end up with the phones collection containing all of the Phones and all of the Plans. The line:

```
Assert.check(AssertLib.checkListItemTypes(phones, Phone.class));
```

would catch this, though.

In the middle of a method, it's OK for the object temporarily to be in an invalid state.

In the Employee example, if I needed a method to swap the two beneficiaries, in the most obvious way:

```
public void swapBeneficiaries() {
    Beneficiary hold = primaryBeneficiary;
    primaryBeneficiary = secondaryBeneficiary;
    secondaryBeneficiary = hold;
}
```

there would be a moment where the invariant would fail if it were to be called. But I can call it at the beginning and end of the method and everything works, so we maintain the rule that the invariant could be called at any time before or after a public method.

Invariants can enforce expectations in a class hierarchy as well. Suppose Plan has a pointer to a Carrier that must always be valid, and MedicalPlan has, in addition, a pointer to a PCPList that must always be valid. The invariant for Plan might be:

```
protected boolean invariant() {
    Assert(carrier != null);
}
```

and that for MedicalPlan might be:

```
private boolean invariant() {
    super.invariant();
    Assert(pcpList != null);
    return true;
}
```

Later we might add additional checks in Plan, and they'll be called from Medical Plan every time. This has the additional benefit of pointing out where the assumptions of the class hierarchy are wrong. For instance, later we might add a company-administered SpendingPlan that doesn't have a carrier, so its constructor doesn't take a carrier, and the superclass' setCarrier() is never called. The first time SpendingPlan's invariant is run, and calls super.invariant(), it fails because carrier is null. A mismatch like this, between the expectations of the class and its superclass, might otherwise go undetected for months until some code change caused a method to be called in Plan that relied on a carrier. Instead we know on the first compile that there is a mismatch. At this point we have to decide between:

It's really a requirement that all Plans have a carrier, in which case either we figure out what to use for the carrier in a SpendingPlan, or we say that Spending Plan is not an extension of Plan.

Plans don't necessarily have a carrier, in which case we drop the test from the Plan invariant and change any methods in plan that refer to carrier, so that they properly handle the null case.

The invariants force the design question to be raised and resolved very early in the process of coding.

Explicit States

One wrinkle on invariants is that the legal condition of an object may depend on its overall state. We can have classes with multiple defined states, but for our purposes we've defined two, complete and incomplete. Complete means the object is ready to be aggregated by other objects, and otherwise used as a legal object.

Most classes will be complete as soon as the constructor is done. For instance, the Enrollment class may define isComplete() as:

```
public boolean isComplete() {
    return true;
}
```

but objects that use Enrollment, say by aggregating a list of Enrollments, don't care whether Enrollment is *always* complete, just that when they use one, it is complete.

The typical invariant() method takes a boolean parameter that shows whether the object is supposed to be complete at the time of the call. For something like Enrollment, this parameter is ignored, since the legal conditions of the object are always the same:

```
public boolean invariant(boolean assertComplete) {
    if (invariantActive) {
        return true;
    }
    invariantActive = true;

    Assert.check(enrolee != null);
    Assert.check(benefitPlan != null);
    invariantActive = false;

    return true;
}
```

Embedded Unit Tests

Each class has a public static classTest() method which runs all of the self-testing code. Typically it would look like this:

```
public final static void classTest() {
    System.out.print(" PickList ... ");
    int count = 0;
    count += testNominal();
    count += testCacheing();
    count += testDatabase();
    System.out.println("passed" + " (" + count + ")");
}
```

When every class' classTest() is called in sequence, I get a listing, one line per class, showing the number of assertions that fired each time:

```
Company ... passed (12342)
DeductionStrategy ... passed (878)
Beneficiary ... passed (450)
etc.
```

The number of assertions is not terribly important, but it helps me in two cases:

If it's zero, I know that either something is commented out that shouldn't be, or the ENABLED is false in the Assert class for this package, and thus no tests are running.

If it's something in the order of several million assertions, I might want to slim it down a bit so that the test code doesn't make it impossible to run the code at all.

The first line just lets you know that the class is running its tests, so that in case something goes wrong, I have some idea where the problem occurred, unlike if I had the display happening completely at the end. Consider the difference between:

```
Company ... passed (12342)
DeductionStrategy ... passed (878)
(hang or crash)
```

and:

```
Company ... passed (12342)
DeductionStrategy ... passed (878)
Beneficiary ...
(hang or crash)
```

At least in the second example I know it failed in Beneficiary.classTest().

Displays

I want every object to be able to tell me its internal state, whenever asked, something like:

```
public void display(int indent) {
    display(indent, null);
}

public void display(int indent, String message) {
    if (indent == 0) {
        System.out.println();
    }
    System.out.println(StringLib.indent(indent) + toString());
    if (message != null) {
        System.out.println(StringLib.indent(indent + 1) + message);
    }
    Iterator i = items.iterator();
    while (i.hasNext()) {
        PickListItem item = (PickListItem)i.next();
        item.display(indent + 1);
    }
}
```

With this code in PickList, and similar code in PickListItem, I can call:

```
pickList.display(0, "After load from DB");
and get:
```

```
com.projectx.metadata.PickList - 234 - (3 items)
com.projectx.metadata.PickListItem - 289 - "United States"
com.projectx.metadata.PickListItem - 290 - "Mexico"
com.projectx.metadata.PickListItem - 291 - "Canada"
```

I would never have to trace through code looking at variables, or write custom debugging code to find out the state of an object, as long as the objects show themselves and the objects that they contain.

Each line in the sample display comes from the toString() of each class. Sometimes a display will take more than one line per class. This is fine, as long as the indenting clearly shows nesting relationships.

Sample Instances

Every class has a getSampleInstance() method, which greatly eases the process of writing test code in *other* classes. For instance, say I have a CoverageTierList class which contains a collection of CoverageTiers. To properly test CoverageTierList, I need to create a bunch of CoverageTiers. Without a getSampleInstance() method in CoverageTier, I'd have to write something like:

```
private final static int testNominal() {

    Assert.resetCount();

    DependentTypeList dtl = new DependentTypeList();
    dtl.addType(DependentType.TYPE_CHILD);
    dtl.addType(DependentType.TYPE_PARENT);

    CoverageTier ctA = new CoverageTier("Coverage Tier A", true, false, dtl);
    CoverageTier ctB = new CoverageTier("Coverage Tier B", true, true, dtl);
    CoverageTier ctC = new CoverageTier("Coverage Tier C", false, false, dtl);

    CoverageTierList ctList = new CoverageTierList();
    Assert.check(ctList.getSize() == 0);
    Assert.check(!ctList.hasCoverageTier(ctA));
}
```

```
ctList.addCoverageTier(ctA);
Assert.check(ctList.getSize() == 1);
Assert.check(ctList.hasCoverageTier(ctA));
```

etc.

If CoverageTier is written with a getSampleInstance, I can say:

```
private final static int testNominal() {

    Assert.resetCount();

    CoverageTier ctA = CoverageTier.getSampleInstance();
    CoverageTier ctB = CoverageTier.getSampleInstance();
    CoverageTier ctC = CoverageTier.getSampleInstance();

    CoverageTierList ctList = new CoverageTierList();
    Assert.check(ctList.getSize() == 0);
    Assert.check(!ctList.hasCoverageTier(ctA));

    ctList.addCoverageTier(ctA);
    Assert.check(ctList.getSize() == 1);
    Assert.check(ctList.hasCoverageTier(ctA));

    etc.
```

This has the advantage of making it easier to write the test code for CoverageTierList, but more importantly it decouples the two classes to some degree. If there's no real (non-testing) code in CoverageTierList which constructs CoverageTiers, or cares about their internal structure or public interface, then there's no reason the test code should care either. With the second method, it's OK for CoverageTier to change its constructor, and all of the test code in CoverageTierList still runs, as it should.

Coverage Tools

Often, testing turns out to cover only part of the code. I don't know how many times I've looked at a piece of code that had supposedly been working for some time and was now acting strangely, and thought "Wait a minute, there's no way this ever would have worked". Particularly with overloaded methods and "standard" getters and setters that are often put in the code without an urgent reason, it's easy to have lurking bugs that would be immediately obvious if all of the code were executed at least once.

Consider this bug:

```
public boolean checkBeneficiaries() {
    Iterator i = getActiveBeneficiaries();
    while (i.hasNext()) {
        Beneficiary beneficiary = (Beneficiary)i.next();
        if (beneficiary.isValid()) {
            return false;
        }
    }
    return true;
}
```

I could run test cases against this method all day, thinking that they are executing the loop, and getting a return value of true. But there could be a problem in my data or in getActiveBeneficiaries() which means that the Iterator is empty and the inside of the loop never executes. And I wouldn't see that the line:

```
        if (beneficiary.isValid()) {
should have been:
```

```
if (!beneficiary.isValid()) {
```

which is a subtle logic bug, easy to do and hard to spot. Especially hard to spot if I never execute the code. But with a code coverage tool I would immediately know that the inside of the loop never ran, than I could figure out why, and once I did, the logic bug would be apparent.

The other way this bug might have been spotted would be to make sure there was a test case for `checkBeneficiaries()` which had an intentionally invalid `Beneficiary`, and an expectation that the method would return `false`. In this case it would return `true`, and we'd know that something was wrong.

Unfrozen Caveman Coding vs. Cleverness

"I'm just an unfrozen caveman who became a lawyer. Your modern world frightens and amazes me."

-- The Unfrozen Caveman Lawyer

Suppose the following line is failing with a null pointer exception:

```
urlList += plan.getCarrier().getUrl().toLowerCase();
```

The line seems harmless enough, but it has four things that could be null:

`urlList`

`plan`

`plan.getCarrier()`

`plan.getCarrier.getUrl()`

I could always track down the error by stepping through the code and painstakingly following all of the calls that this line sets into motion. Assuming that the bug would happen on the first call. It might be only on the 1000th call, in which case I'd have to check the call stack and figure out how to break under the right conditions. This is not a huge deal, but it adds up, when it should have taken zero time to find the problem:

1. `Carrier carrier = plan.getCarrier();`
2. `Assert.check(Assert.ENABLED && (carrier != null));`
3. `String carrierUrl = carrier.getUrl();`
4. `Assert.check(Assert.ENABLED && (carrierUrl != null));`
5. `urlList += carrierUrl.toLowerCase() + " - ";`

This code is equivalent to the single line above, but it will error out on line 1 if `plan` is null, line 2 if `carrier` is null, line 4 if `carrierUrl` is null, or line 5 if `urlList` is null. So I will never have to trace through this code to find the problem. And even if I do, having each operation on its own line allows me to check the values at each step, rather than tracing into each method call.

It's also easier to understand the code when each line does one or two operations. Consider:

```
DeductionCalendar dc = new DeductionCalendar(company.getPlanYear(),  
        employee.getPayroll().findPayFrequency());
```

vs.

```
PlanYear planYear = company.getPlanYear();  
Payroll payroll = employee.getPayroll();  
PayFrequency payFrequency = payroll.findPayrollFrequency();  
DeductionCalendar dc = new DeductionCalendar(planYear, payFrequency);
```

Or perhaps:

```
int premium = (plan.getCost() > 0 ? plan.getCost() : plan.getSecondCost());
```

vs.

```
int premium;  
int planCost = plan.getCost();  
if (planCost > 0) {  
    premium = planCost;  
} else {  
    premium = plan.getSecondCost();  
}
```

In either case, writing the code over multiple lines allows me to:

Read it more easily

Spot errors immediately while writing the code

Add assertions

Add comments

Make changes easily

Debug through the code more easily

Add code to display interim values when necessary

The Unfrozen Caveman Lawyer character gained an advantage in the courtroom by pretending to be less clever than he really was. This is a strategy that serves me well in programming. I don't want my code to be a testament to how many things I can keep in my head at once, or how much I can squeeze into one line or one method or one class and still make it work. It's not a contest. My code is not a puzzle to be solved, it's an artifact that many people will have to live with for a long time, and if someone looks at it and declares it boring and obvious, and not a bit clever, then I've done my job well.

Practices, Summarized

For every class:

Implement the Testable interface.

Write an invariant() method.

Assert the invariant() at the end of the constructor, at the beginning of every public method, and at the end of any public method that changes something in the object.

Assert all assumptions about parameters and where appropriate assert expected states of variables.

Do one interesting thing per line.

Write an isComplete() method that normally returns true except in special cases.

Write a toString() method that gives a single line with the name of the class and some relevant information about its state, such as the values of the most important attributes.

Write a classTest() method that runs testNominal() and any other test...() methods.

Write a testNominal() method that exercises all of the public interface and many edge cases, combinations, different sequences, etc.

Write a test...() method for every other aspect of the class not covered in testNominal(), such as loading from a database, converting to XML, etc.

Write a display() method that shows the toString(), followed by any other necessary information, followed by the aggregated contents of the class, such as by calling the display() method for each contained object in a list.

Write a displayStatic() method if necessary to show the contents of a static cache of objects.

Write a getSampleInstance() method to aid other classes in testing using this class.

Check for near-100% code coverage if you're using a coverage tool.

Check in the code and go home without a care in the world.