design

Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

When to Make a Type

Martin Fowler

When I started programming computers, I began with fairly primitive languages, such as Fortran 4 and various early flavors of Basic. One of the first things you learn using such languages—indeed, even using more up-to-date languages—is which types your language supports. Being oriented toward number crunching, Fortran supported integer and real types, with the interesting rule that any variable whose name started with the letters I through N was an integer, and all other variables were floats. I'm glad that convention hasn't caught on, although Perl is close. Furthermore, using object-oriented languages, you can define your own types and in the best languages, they act just as well as built-in ones.

Defining types

I first started playing with computers in my math classes, where we were all frustrated by the fact that these oh-so-numerate computers didn't understand fractions (and our math teachers took a dim view of floating-point approximations). I was thus delighted to learn that Smalltalk supported fractions naturally—if you divided 1 by 3 you got a third, not some irritating long-running float.

When people talk about design, they often don't talk about little objects such as fractions, presumably because many architects consider such details unworthy of their attention. However, defining such types often makes life easier.

My favorite example is money. A lot of computer horsepower is dedicated to manipulating money, accounting, billing, trading, and so forth—few things burn more cycles. Despite all this attention, no mainstream language has a built-in type for money. Such a type could reduce errors by being currency aware, helping us, for example, avoid embarrassing moments of adding our dollars to our yen. It can also avoid more insidious rounding errors. It would not only remove the temptation to use floats for money (never, ever do that) but also help us deal with tricky problems such as how to split $10 equally between three people. In addition, it could simplify a lot of printing and parsing code. For more on this (why write the column if I can't plug my books?), see *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002).

The nice thing about OO programs is that you can easily define a type like this if the language and libraries don't include it. Other such low-level types I've often written are ranges, because I'm sick of writing `if (x <= top && x >= bottom)`, quantities (to handle things such as "6 feet"), and dates (at least most languages include them now, but they are usually incorrect).

Once you start writing these kinds of fundamental objects, you begin to ask yourself where to stop. For example, one person recently asked me whether he should make a type for currency, even though the only data was the international three-letter code? Another person asked about a person class with an age attribute and whether he should return an integer or define an age type and return that.

0740-7459/03/$19.00 © 2003 IEEE

When is it worth your while?

When should you make your own type? To begin with, make a type if it will have some special behavior in its operations that the base type doesn't have. The attention to rounding in money offers a good example: rather than have every user of a number class remember to round, provide the methods in a money class so you can program the rounding once.

Special types are handy when you have bits of data that often go together. A range object that brings together top and bottom values is a good example. You might argue that `if a.includes(x)` isn't much better than `if(x >= min && x <= max)`, but using a range communicates that the two values fit together. Indeed, communication is one of the biggest reasons to use a type. If you have a method that expects to take a currency parameter, you can communicate this more clearly by having a currency type and using it in the method declaration. This is valuable even if you don't have static type checking, although such checking is yet another reason to make a type.

Some types don't want to use their full capabilities. Often you'll find things such as product codes that are numeric in form. However, even though they look like a number, they don't behave like one. Nobody needs to do arithmetic on product codes—with a special type you can avoid bugs, such as using someone's personnel number in the arithmetic for their paycheck.

That's a common thing to watch for. Even if a currency code looks like a string, if it doesn't behave like one, it should get a different type. Look at the string's interface and ask how much of it applies to a currency code? If most of it doesn't, then that's a good argument for a new type.

Validation constraints are another factor to consider. We might want to throw an exception if someone sets my age to –1 or 300. Some languages have defined types purely on the basis of ranges. Currencies often will have a fixed list that can be defined in terms of enums or symbolic constants. The value of this depends on how widely people create the objects. If creating an age only occurs in the person class, then there's less value in making age a class than there would be if ages were created all over the code base.

One of the interesting variations to consider is system evolution. Today, an integral age might make sense, but suppose in six months you need to deal with someone's age in years and months? Here you consider the effect of the type on refactoring to make the change. With your own type, changing the data structure is much easier, just add an extra field to the age type. If you return an integer, you have a more involved refactoring, but you can do it steadily in stages. First, keep the method that returns an integer, probably renaming it for clarity, and provide a new method that returns the age type. That way, older code doesn't have to change, but newer code can use the new capability. Then, look at all the uses of the integral age and, over time, alter them to use ages rather than integers. This could take a matter of minutes or months. Once you alter them all, you can remove the integral age method.

On the whole, I'm inclined to say that when in doubt, make a new type. It usually requires little effort but often provides surprising results. For a currency, even if only a three-letter string, I'd make the type for communication reasons. For an age returned by a person, I'd be more inclined to stick with the integer—providing it wasn't widely used and everyone treated it as a number. Refactoring from one type to another isn't a huge deal—particularly if you make the change as soon as you realize the type is going to be used widely.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.

IEEE Software

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access http://computer.org/software/author.htm.

Letters to the Editor
Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web
Access http://computer.org/software for information about *IEEE Software.*

Subscribe
Visit http://computer.org/subscribe.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software.*

Membership Change of Address
Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.