Your coding conventions are hurting you

If you take a walk in my hometown, and head for the center, you may stumble into a building like this:

from a distance, it's a relatively pompous edifice, but as you get closer, you realize that the columns, the ornaments, everything is fake, carefully painted to look like the real thing. Even shadows have been painted to deceive the eye. Here is another picture from the same neighbor: from this angle, it's easier to see that everything has just been painted on a flat surface:

Curiously enough, as I wander through the architecture and code of many systems and libraries, I sometimes get the same feeling. From a distance, everything is object oriented, extra-cool, modern-flexible-etc, but as you get closer, you realize it's just a thin veneer over procedural thinking (and don't even get me started about being "modern").

Objects, as they were meant to be

Unlike other disciplines, software development shows little interest for classics. Most people are more attracted by recent works. Who cares about some 20 years old paper when you can play with Node.js? Still, if you never took the time to read The Early History of Smalltalk, I'd urge you to. Besides the chronicles of some of the most interesting times in hw/sw design ever, you'll get little gems like this: *"The basic principal of recursive design is to make the parts have the same power as the whole." For the first time I thought of the whole as the entire compute and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers, as time sharing was starting to? But not in dozens. Why not thousands of them, each simulating a useful structure?*

That's brilliant. It's a vision of objects like little virtual machines, offering specialized services. Objects were meant to be smart. Hide data, expose behavior. It's more than that: Alan is very explicit about the idea of methods as **goals**, something you want to happen, unconcerned about how it is going to happen.

Now, I'd be very tempted to write: "unfortunately, most so-called object-oriented code is not written that way", but I don't have to :-), because I can just quote Alan, from the same paper: *The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as sites of higher level behaviors more appropriate for use as dynamic components. [...]It is unfortunate that much of what is called “object-oriented programming” today is simply old style programming with fancier constructs. Many programs are loaded with “assignment-style” operations now done by more expensive attached procedures.*

That was 1993. Things haven't changed much since then, if not for the worse :-). Lot of programmers have learnt the *mechanics* of objects, and forgot (or ignored) the underlying *concepts*. As you get closer to their code, you'll see procedural thinking oozing out. In many cases, you can see that just by looking at class names.

Names are a thinking device

Software development is about discovering and encoding knowledge. Now, humans have relatively few ways to encode knowledge: a fundamental strategy is to **name** things and concepts. Coming up with a good name is hard, yet programming requires us to devise names for:

- components
- namespaces / packages
- classes
- members (data and functions)
- parameters
- local variables
- etc

People are basically lazy, and in the end the compiler/interpreter doesn't care about our beautiful names, so why bother? Because finding good names is a journey of discovery. The names we choose shape the dictionary we use to talk and think about our software. If we can't find a good name, we obviously don't know enough about either the problem domain or the solution domain. Our code (or our model) is telling us something is wrong. Perhaps the metaphor we choose is not properly aligned with the problem we're trying to solve. Perhaps there are just a few misleading abstractions, leading us astray. Still, we better listen, because we are doing it wrong.

As usual, balance is key, and focus is a necessity, because clock is ticking as we're thinking. I would suggest that you focus on class/interface names first. If you can't find a proper name for the class, try naming functions. Look at those functions. What is keeping them together? You can apply them to... that's the class name :-). Can't find it? Are you sure those functions belong together? Are you thinking in concepts or just slapping an implementation together? What is that code **really** doing? Think method-as-a-goal, class-as-a-virtual-machine. Sometimes, I've found useful to think about the opposite concept, and move from there.

Fake OO names and harmful conventions

That's rather bread-and-butter, yet it's way more difficult than it seems, so people often tend to give up, think procedurally, and use procedural names as well. Unfortunately, procedural names have been institutionalized in patterns, libraries and coding conventions, therefore turning them into a major issue.

In practice, a few widely used conventions can seriously stifle object thinking:
- the -er suffix
- the -able suffix
- the -Object suffix
- the I- prefix

of these, the I- prefix could be the most harmless in theory, except that in practice, it's not :-). Tons of ink has been wasted on the -er suffix, so I'll cover that part quickly, and move to the rest, with a few examples from widely used libraries.

Manager, Helper, Handler...
Good ol' Peter Coad used to say: *Challenge any class name that ends in "-er" (e.g. Manager or Controller). If it has no parts, change the name of the class to what each object is managing. If it has parts, put as much work in the parts that the parts know enough to do themselves* (that was the "er-er Principle"). That's central to object thinking, because when you need a Manager, it's often a sign that the Managed are just plain old data structures, and that the Manager is the smart procedure doing the real work.

When Peter wrote that (1993), the idea of an Helper class was mostly unheard of. But as more people got into the OOP bandwagon, they started creating larger and larger, uncohesive classes. The proper OO thing to do, of course, is to find the right cooperating, cohesive concepts. The lazy, fake OO thing to do is to take a bunch of methods, move them outside the overblown class X, and group them in XHelper. While doing so, you often have to weaken encapsulation in some way, because XHelper needs a privileged access to X. Ouch. That's just painting an OO picture of classes over old-style coding. Sadly enough, in a wikipedia article that I won't honor with a link, you'll read that "Helper Class is one of the basic programming techniques in object-oriented programming". My hope for humanity is only restored by the fact that the article is an orphan :-).

I'm not going to say much about Controller, because it's so popular today (MVC rulez :-) that it would take forever to clean this mess. Sad sad sad.

Handler, again, is an obvious resurrection of procedural thinking. What is an handler if not a damn procedure? Why do something need to be "handled" in the first place? Oh, I know, you're thinking of events, but even in that case, EventTarget, or even plain Target, is a much better abstraction than EventHandler.

Something-able
Set your time machine to 1995, and witness the (imaginary) conversation between naïve OO Developer #1, who for some reason has been assigned to design the library of the new wonderful-language-to-be, and naïve OO Developer #2, who's pairing with him along the way.
N1: So, I've got this Thread class, and it has a run() function that it's being executed into that thread... you just have to override run()...
N2: So to execute a function in a new thread you have to extend Thread? That's bad design! Remember we can only extend one class!
N1: Right, so I'll use the Strategy Pattern here... move the run() to the strategy, and execute the strategy in the new thread.
N2: That's cool... how do you wanna call the strategy interface?
N1: Let's see... there is only one method... run()... hmmm
N2: Let's call it Runnable then!
N1: Yes! Runnable it is!

And so it began (no, I'm not serious; I don't know how it all began with the -able suffix; I'm making this up). Still, at some point people thought it was fine to look at an interface (or at a class), see if there was some kind of "main method" or "main responsibility" (which is kinda obvious if you only have one) and name the class after that. Which is a very simple way to avoid thinking, but it's hardly a good idea. It's like calling a nail "Hammerable", because you known, that's what you do with a nail, you hammer it. It encourages procedural thinking, and leads to ineffective abstractions.

Let's pair our naïve OO Developer #1 with an Object Thinker and replay the conversation:

N1: So, I've got this Thread class, and it has a run() function that it's being executed into that thread... you just have to override run()...
OT: So to execute a function in a new thread you have to extend Thread? That's bad design! Remember we can only extend one class!
N1: Right, so I'll use the Strategy Pattern here... move the run() to the strategy, and execute the strategy in the new thread.
OT: OK, so what is the real abstraction behind the strategy? Don't just think about the mechanics of the pattern, think of what it really represents...
N1: Hmm, it's something that can be run...
OT: Or executed, or performed independently...
N1: Yes
OT: Like an Activity, with an Execute method, what do you think? [was our OT aware of the UML 0.8 draft? I still have a paper version :-)]
N1: But I don't see the relationship with a thread...
OT: And rightly so! Thread depends on Activity, but Activity is independent of Thread. It just represents something that can be executed. I may even have an ActivitySequence and it would just execute them all, sequentially. We could even add concepts like entering/exiting the Activity...
(rest of the conversation elided – it would point toward a better timeline :-)

Admittedly, some interfaces are hard to name. That's usually a sign that we don't really know what we're doing, and we're just coding our way out of a problem. Still, some others (like Runnable) are improperly named just because of bad habits and conventions. Watch out.

Something-Object
This is similar to the above: when you don't know how to name something, pick some dominant trait and add Object to the end. Again, the problem is that the "dominant trait" is moving us away from the concept of an object as a virtual machine, and toward the object as a procedure. In other cases, Object is dropped in just to avoid more careful thinking about the underlying concept.

Just like the -able suffix, sometimes it's easy to fix, sometimes is not. Let's try something not trivial, where the real concept gets obscured by adopting a bad naming convention. There are quite a few cases in the .NET framework, so I'll pick **MarshalByRefObject**.

If you don't use .NET, here is what the documentation has to say:
Enables access to objects across application domain boundaries in applications that support remoting
What?? Well, if you go down to the Remarks section, you get a better explanation:
Objects that do not inherit from MarshalByRefObject are implicitly marshal by value. […] The first time [...] a remote application domain accesses a MarshalByRefObject, a proxy is passed to the remote application
Whoa, that's a little better, except that it's "are marshaled by value", not "are marshal by value" but then, again, the name should be MarshaledByRefObject, not MarshalByRefObject. Well, *all your base are belong to us* :-)

Now, we could just drop the Object part, fix the grammar, and call it MarshaledByReference, which is readable enough. A reasonable alternative could be MarshaledByProxy (Vs. MarshaledByCopy, which would be the default).
Still, we're talking more about implementation than about concepts. It's not that I want my object marshaled by proxy; actually, I don't care about marshaling at all. What I want is to keep a single object identity across appdomains, whereas with copy I would end up with distinct objects. So, a proper one-sentence definition could be:

Preserve object identity when passed between appdomains [by being proxied instead of copied]
or
Guarantees that methods invoked in remote appdomains are served in the original appdomain

Because if you pass such an instance across appdomains, any method call would be served by the original object in the original appdomain. Hmm, guess what, we already have similar concepts. For instance, we have objects

that, after being created in a thread, must have their methods executed only inside that thread. A process can be set to run only on one CPU/core. We can configure a load balancer so that if you land on a specific server first, you'll stay on that server for the rest of your session. We call that concept **affinity**.

So, a MarshalByRefObject is something with an **appdomain affinity**. The marshaling thing is just an implementation detail that makes that happen. **AppDomainAffine**, therefore, would be a more appropriate name. Unusual perhaps, but that's because of the common drift toward the *mechanics* of things and away from *concepts* (because the mechanics are usually much easier to get for techies). And yes, it takes more clarity of thoughts to come up with the notion of appdomain affinity than just slapping an Object at the end of an implementation detail. However, clarity of thoughts is exactly what I would expect from framework designers. While we are at it, I could also add that AppDomainAffine should be an attribute, not a concrete class without methods and fields (!) like MarshalByRefObject. Perhaps I'm asking too much from those guys.

ISomething
I think this convention was somewhat concocted during the early COM days, when Hungarian was widely adopted inside Microsoft, and having ugly names was therefore the norm. Somehow, it was the only convention that survived the general clean up from COM to .NET. Pity :-).

Now, the problem is not that you have to type an I in front of things. It's not even that it makes names harder to read. And yes, I'll even concede that after a while, you'll find it useful, because it's easy to spot an interface just by looking at its name (which, of course, is a relevant information when your platform doesn't allow multiple inheritance). The problem is that it's too easy to fall into the trap, and just take a concrete class name, put an I in front of it, and lo and behold!, you got an interface name. Sort of calling a concept IDollar instead of Currency.

Case in point: say that you are looking for an abstraction of a container. It's not just a container, it's a special container. What makes it special is that you can access items by index (a more limited container would only allow sequential access). Well, here is the (imaginary :-) conversation between naïve OO Developer #1, who for unknown reasons has been assigned to the design of the base library for the newfangled language of the largest software vendor on the planet, and himself, because even naïve OO Developer #2 would have made things better:

N1: So I have this class, it's called List... it's pretty cool, because I just made it a generic, it's List<T> now!
N1: Hmm, the other container classes all derive from an interface... with wonderful names like IEnumerable (back to that in a moment). I need an interface for my list too! How do I call it?
N1: IListable is too long (thanks, really :-)))). What about IList? That's cool!
N1: Let me add an XML comment so that we can generate the help file...

IList Interface
Represents a collection of objects that can be individually accessed by index.

So, say that you have another class, let's call it Array. Perhaps a SparseArray too. They both can be accessed by index. So Array IS-A IList, right? C'mon.

Replay the conversation, drop in our Object Thinker:

N1: So I have this class, it's called List... it's pretty cool, because I just made it a generic, it's List<T> now!
N1: Hmm, the other container classes all derive from an interface... with wonderful names like IEnumerable. I need an interface for my list too! How do I call it?
OT: What's special about List? What does it really add to the concept of enumeration? (I'll keep the illusion that "enumeration" is the right name so that N1's mind won't blow away)
N1: Well, the fundamental idea is that you can access items by index... or ask about the index of an element... or remove the element at a given index...
OT: so instead of *sequential access* you now have *random access*, as it's usually called in computer science?
N1: Yes...
OT: How about we call it RandomAccessContainer? It reads pretty well, like: a List IS-A RandomAccessContainer, an Array IS-A RandomAccessContainer, etc.

N1: Cool... except... can we put an I in front of it?
OT: Over my dead body.... hmm I mean, OK, but you know, in computer science, a List is usually thought of as a sequential access container, not a random access container. So I'll settle for the I prefix if you change List to something else.
N1: yeah, it used to be called ArrayList in the non-generic version...
OT: kiddo, do you think there was a reason for that?
N1: Oh... (spark of light)

Yes, give me the worst of both worlds!
Of course, given enough time, people will combine those two brilliant ideas, turn off their brain entirely, and create wonderful names like IEnumerable. Most .NET collections implement IEnumerable, or its generic descendant IEnumerable<T>: when a class implements IEnumerable, its instances can be used inside a foreach statement.

Indeed, IEnumerable is a perfect example of how bad naming habits thwart object thinking, and more in general, abstraction. Here is what the official documentation says about IEnumerable<T>:
Exposes the enumerator, which supports a simple iteration over a collection of a specified type.
So, if you subscribe to the idea that the main responsibility gives the -able part, and that the I prefix is mandatory, it's pretty obvious that IEnumerable is the name to choose.

Except that's just wrong. The right abstraction, the one that is completely hidden under the IEnumerable/IEnumerator pair, is the sequential access collection, or even better, a Sequence (a Sequence is more abstract than a Collection, as a Sequence can be calculated and not stored, see yield, continuations, etc). Think about what makes more sense when you read it out loud:

A List is an IEnumerable (what??)
A List is a Sequence (well, all right!)

Now, a Sequence (IEnumerable) in .NET is traversed ("enumerated") through an iterator-like class called an IEnumerator ("iterator" like in the iterator pattern, not like that thing they called iterator in .NET). Simple exercise: what is a better name than IEnumerator for something that can only move forward over a Sequence?.

Is that all you got?
No, that's not enough! Given a little more time, someone is bound to come up with the worst of **all** worlds! What about an interface with:
an I prefix
an -able suffix
an Object somewhere in between

That's a challenging task, and strictly speaking, they failed it. They had to put -able in between, and Object at the end. But it's a pretty amazing name, fresh from the .NET Framework 4.0: IValidatableObject (I wouldn't be surprised to discover, inside their code, an IValidatableObjectManager to, you know, *manage* :-> those damn stupid validatable objects; that would really close the circle :-).

We can read the documentation for some hilarious time:
IValidatableObject Interface - Provides a way for an object to be invalidated.

Yes! That's what my objects really want! To be **invalidated**! C'mon :-)). I'll spare you the imaginary conversation and go straight to the point. Objects don't want to be invalidated. That's procedural thinking: "validation". Objects may be subject to **constraints**. Yahoo! **Constraints**. What about a Constraint class (to replace the Validation/Validator stuff, which is so damn **procedural**). What about a Constrained (or, if you can't help it, IConstrained) interface, to replace IValidatableObject?

Microsoft (and everyone else, for that matter): what about having a few more Object Thinkers in your class library teams? Oh, while we are at it, why don't you guys consider that we may want to check constraints in any given place, not just in the front end? Why not moving all the constraint checking away from UI or Service layers and make the whole stuff available *everywhere*? It's pretty simple, trust me :-).

Bonus exercise: once you have Constraint and IConstrained, you need to check all those constraints when some event happens (like you receive a message on your service layer). Come up with a better name than ConstraintChecker, that is, something that is not ending in -er.

And miles to go...
There would be so much more to say about programming practices that hinder object thinking. Properties, for instance, are usually misused to expose internal state ("expressed figuratively" or not), instead of being just zero-th methods (henceforth, goals!). Maybe I'll cover that in another post.

An interesting question, for which I don't really have a good answer, is: could some coding convention **promote** object thinking? Saying "don't use the -er suffix" is not quite the same as saying "do this and that". Are your conventions moving you toward object thinking?

Note: for more on the -er suffix, see: Life without a controller, case 1.

If you read so far, you should follow me on twitter.