# JOEL ON SOFTWARE

I'm Joel Spolsky, a software developer in New York City. More about me.

MAY 11, 2005 *by* JOEL SPOLSKY

# Making Wrong Code Look Wrong

ROCK STAR DEVELOPER, NEWS

Way back in September 1983, I started my first real job, working at Oranim, a big bread factory in Israel that made something like 100,000 loaves of bread every night in six giant ovens the size of aircraft carriers.

The first time I walked into the bakery I couldn't believe what a mess it was. The sides of the ovens were yellowing, machines were rusting, there was grease everywhere.

"Is it always this messy?" I asked.

"What? What are you talking about?" the manager said. "We just finished cleaning. This is the cleanest it's been in weeks."

Oh boy.

It took me a couple of months of cleaning the bakery every morning before I realized what they meant. In the bakery, clean meant no dough on the machines. Clean meant no fermenting dough in the trash. Clean meant no dough on the floors.

Clean did not mean the paint on the ovens was nice and white. Painting the ovens was something you did every decade, not every day. Clean did not mean no grease. In fact there were a lot of machines that needed to be greased or oiled regularly and a thin layer of clean oil was usually a sign of a machine that had just been cleaned.

The whole concept of clean in the bakery was something you had to learn. To an outsider, it was impossible to walk in and judge whether the place was clean or not. An outsider would never think of looking at the inside surfaces of the dough rounder (a machine that rolls square blocks of dough into balls, shown in the picture at right) to see if they had been scraped clean. An outsider would obsess over the fact that the old oven had discolored panels, because those panels were *huge*. But a baker couldn't care less whether the paint on the outside of their oven was starting to turn a little yellow. The bread still tasted just as good.

After two months in the bakery, you learned how to "see" clean.

Code is the same way.

When you start out as a beginning programmer or you try to read code in a new language it all looks equally inscrutable. Until you understand the programming language itself you can't even see obvious syntactic errors.

During the first phase of learning, you start to recognize the things that we usually refer to as "coding style." So you start to notice code that doesn't conform to indentation standards and Oddly-Capitalized variables.

It's at this point you typically say, "Blistering Barnacles, we've *got* to get some consistent coding conventions around here!" and you spend the next day writing up coding conventions for your team and the next six days arguing about the One True Brace Style and the next three weeks rewriting old code to conform to the One True Brace Style until a manager catches you and screams at you for wasting time on something that can never make money, and you decide that it's not really a bad thing to only reformat code when you revisit it, so you have about half of a True Brace Style and pretty soon you forget all about that

and then you can start obsessing about something else irrelevant to making money like replacing one kind of string class with another kind of string class.

As you get more proficient at writing code in a particular environment, you start to learn to see other things. Things that may be perfectly legal and perfectly OK according to the coding convention, but which make you worry.

For example, in C:

```
char* dest, src;
```

This is legal code; it may conform to your coding convention, and it may even be what was intended, but when you've had enough experience writing C code, you'll notice that this declares **`dest`** as a **`char`** *pointer* while declaring **`src`** as merely a **`char`**, and even if this *might* be what you wanted, it probably isn't. That code smells a little bit dirty.

Even more subtle:

```
if (i != 0)
    foo(i);
```

In this case the code is 100% correct; it conforms to most coding conventions and there's nothing wrong with it, but the fact that the single-statement body of the `if`statement is not enclosed in braces may be bugging you, because you might be thinking in the back of your head, gosh, somebody might insert another line of code there

```
if (i != 0)
    bar(i);
    foo(i);
```

… and forget to add the braces, and thus accidentally make **`foo(i)`**unconditional! So when you see blocks of code that aren't in braces, you might sense just a tiny, wee, soupçon of uncleanliness which makes you uneasy.

OK, so far I've mentioned three levels of achievement as a programmer:

1. You don't know clean from unclean.

2. You have a superficial idea of cleanliness, mostly at the level of conformance to coding conventions.

3. You start to smell subtle hints of uncleanliness beneath the surface and they bug you enough to reach out and fix the code.

There's an even higher level, though, which is what I really want to talk about:

4. You deliberately architect your code in such a way that your nose for uncleanliness makes your code more likely to be correct.

This is the real art: making robust code by literally *inventing conventions* that make errors stand out on the screen.

So now I'll walk you through a little example, and then I'll show you a general rule you can use for inventing these code-robustness conventions, and in the end it will lead to a defense of a certain type of Hungarian Notation, probably not the type that makes people carsick, though, and a criticism of exceptions in certain circumstances, though probably not the kind of circumstances you find yourself in most of the time.

But if you're so convinced that Hungarian Notation is a Bad Thing and that exceptions are the best invention since the chocolate milkshake and you don't even want to hear any other opinions, well, head on over to Rory's and read the excellent comix instead; you probably won't be missing much here anyway; in fact in a minute I'm going to have actual code samples which are likely to put you to sleep even before they get a chance to make you angry. Yep. I think the plan will be to lull you almost completely to sleep and then to sneak the Hungarian=good, Exceptions=bad thing on you when you're sleepy and not really putting up much of a fight.

**An Example**

Right. On with the example. Let's pretend that you're building some kind of a web-based application, since those seem to be all the rage with the kids these days.

Now, there's a security vulnerability called the Cross Site Scripting Vulnerability, a.k.a. XSS. I won't go into the details here: all you have to know is that when you build a web application you have to be careful never to repeat back any strings that the user types into forms.

So for example if you have a web page that says "What is your name?" with an edit box and then submitting that page takes you to another page that says, Hello, Elmer! (assuming the user's name is Elmer), well, that's a security vulnerability, because the user could type in all kinds of weird HTML and JavaScript instead of "Elmer" and their weird JavaScript could do narsty things, and now those narsty things appear to come from you, so for example they can read cookies that you put there and forward them on to Dr. Evil's evil site.

Let's put it in pseudocode. Imagine that

```
s = Request("name")
```

reads input (a POST argument) from the HTML form. If you ever write this code:

```
Write "Hello, " & Request("name")
```

your site is already vulnerable to XSS attacks. That's all it takes.

Instead you have to encode it before you copy it back into the HTML. Encoding it means replacing " with `&quot;`, replacing > with `&gt;`, and so forth. So

```
Write "Hello, " & Encode(Request("name"))
```

is perfectly safe.

All strings that originate from the user are *unsafe*. Any unsafe string must not be output without encoding it.

Let's try to come up with a coding convention that will ensure that if you ever make this mistake, the code will just *look* wrong. If wrong code, at least, *looks* wrong, then it has a fighting chance of getting caught by someone working on that code or reviewing that code.

**Possible Solution #1**

One solution is to encode all strings right away, the minute they come in from the user:

```
s = Encode(Request("name"))
```

So our convention says this: if you ever see **`Request`** that is not surrounded by **`Encode`**, the code must be wrong.

You start to train your eyes to look for naked **`Request`**s, because they violate the convention.

That works, in the sense that if you follow this convention you'll never have a XSS bug, but that's not necessarily the best architecture. For example maybe you want to store these user strings in a database somewhere, and it doesn't make sense to have them stored HTML-encoded in the database, because they might have to go somewhere that is not an HTML page, like to a credit card processing application that will get confused if they are HTML-encoded. Most web applications are developed under the principle that all strings internally are *not* encoded until the *very last moment* before they are sent to an HTML page, and that's probably the right architecture.

We really need to be able to keep things around in unsafe format for a while.

OK. I'll try again.

**Possible Solution #2**

What if we made a coding convention that said that when you *write out* any string you have to encode it?

```
s = Request("name")


// much later:
Write Encode(s)
```

Now whenever you see a naked **`Write`** without the **`Encode`** you know something is amiss.

Well, that doesn't quite work… sometimes you have little bits of HTML around

in your code and you *can't* encode them:

```
If mode = "linebreak" Then prefix = "<br>"

// much later:
Write prefix
```

This looks wrong according to our convention, which requires us to encode strings on the way out:

```
Write Encode(prefix)
```

But now the **"<br>"**, which is supposed to start a new line, gets encoded to **<br>** and appears to the user as a literal **< b r >**. That's not right either.

So, sometimes you can't encode a string when you read it in, and sometimes you can't encode it when you write it out, so neither of these proposals works. And without a convention, we're still running the risk that you do this:

```
s = Request("name")

...pages later...
name = s

...pages later...
recordset("name") = name // store name in db in a column "name"

...days later...
theName = recordset("name")

...pages or even months later...
Write theName
```

Did we remember to encode the string? There's no single place where you can look to see the bug. There's no place to sniff. If you have a lot of code like this, it takes a ton of detective work to trace the origin of every string that is ever written out to make sure it has been encoded.

### The Real Solution

So let me suggest a coding convention that works. We'll have just one rule:

All strings that come from the user must be stored in variables (or database columns) with a name starting with the prefix "us" (for Unsafe String). All strings that have been HTML encoded or which came from a known-safe location must be stored in variables with a name starting with the prefix "s" (for Safe string).

Let me rewrite that same code, changing nothing but the variable names to match our new convention.

```
us = Request("name")

...pages later...
usName = us

...pages later...
recordset("usName") = usName

...days later...
sName = Encode(recordset("usName"))

...pages or even months later...
Write sName
```

The thing I want you to notice about the new convention is that now, if you make a mistake with an unsafe string, *you can always see it on some single line of code*, as long as the coding convention is adhered to:

```
s = Request("name")
```

is a priori wrong, because you see the result of `Request` being assigned to a variable whose name begins with `s`, which is against the rules. The result of `Request` is always unsafe so it must always be assigned to a variable whose name begins with "us".

```
us = Request("name")
```

is always OK.

```
usName = us
```

is always OK.

```
sName = us
```

is certainly wrong.

```
sName = Encode(us)
```

is certainly correct.

```
Write usName
```

is certainly wrong.

```
Write sName
```

is OK, as is

```
Write Encode(usName)
```

Every line of code can be inspected *by itself*, and if every line of code is correct, the entire body of code is correct.

Eventually, with this coding convention, your eyes learn to see the `Write usXXX` and know that it's wrong, and you instantly know how to fix it, too. I know, it's a little bit hard to see the wrong code at first, but do this for three weeks, and your eyes will adapt, just like the bakery workers who learned to look at a giant bread factory and instantly say, "jay-zuss, nobody cleaned insahd rounduh fo-ah! What the hayl kine a opparashun y'awls runnin' heey-uh?"

In fact we can extend the rule a bit, and rename (or wrap) the `Request` and `Encode`functions to be `UsRequest` and `SEncode`… in other words, functions that return an unsafe string or a safe string will start with `Us` and `S`, just like variables. Now look at the code:

```
us = UsRequest("name")
usName = us
recordset("usName") = usName
sName = SEncode(recordset("usName"))
Write sName
```

See what I did? Now you can look to see that both sides of the equal sign start with the same prefix to see mistakes.

```
us = UsRequest("name") // ok, both sides start with US
s = UsRequest("name") // bug
usName = us // ok
sName = us // certainly wrong.
sName = SEncode(us) // certainly correct.
```

Heck, I can take it one step further, by naming **Write** to **WriteS** and renaming **SEncode** to SFromUs:

```
us = UsRequest("name")
usName = us
recordset("usName") = usName
sName = SFromUs(recordset("usName"))
WriteS sName
```

This makes mistakes even *more* visible. Your eyes will learn to "see" smelly code, and this will help you find obscure security bugs just through the normal process of writing code and reading code.

Making wrong code look wrong is nice, but it's not necessarily the best possible solution to every security problem. It doesn't catch every possible bug or mistake, because you might not look at every line of code. But it's sure a heck of a lot better than nothing, and I'd much rather have a coding convention where wrong code, at least, looked wrong. You instantly gain the incremental benefit that every time a programmer's eyes pass over a line of code, that particular bug is checked for and prevented.

### A General Rule

This business of making wrong code look wrong depends on getting the right things close together in one place on the screen. When I'm looking at a string, in order to get the code right, I need to know, everywhere I see that string, whether it's safe or unsafe. I don't want that information to be in another file or on another page that I would have to scroll to. I have to be able to see it *right there* and that means a variable naming convention.

There are a lot of other examples where you can improve code by moving things next to each other. Most coding conventions include rules like:

- Keep functions short.
- Declare your variables as close as possible to the place where you will use them.
- Don't use macros to create your own personal programming language.
- Don't use `goto`.
- Don't put closing braces more than one screen away from the matching opening brace.

What all these rules have in common is that they are trying to get the relevant information about what a line of code really does physically as close together as possible. This improves the chances that your eyeballs will be able to figure out everything that's going on.

In general, I have to admit that I'm a little bit scared of language features that hide things. When you see the code

```
i = j * 5;
```

… in C you know, at least, that `j` is being multiplied by five and the results stored in `i`.

But if you see that same snippet of code in C++, you don't know anything. Nothing. The only way to know what's really happening in C++ is to find out what types `i` and `j` are, something which might be declared somewhere altogether else. That's because `j` might be of a type that has `operator*` overloaded and it does something terribly witty when you try to multiply it. And

`i` might be of a type that has `operator=` overloaded, and the types might not be compatible so an automatic type coercion function might end up being called. And the only way to find out is not only to check the type of the variables, but to find the code that implements that type, and God help you if there's inheritance somewhere, because now you have to traipse all the way up the class hierarchy all by yourself trying to find where that code really *is*, and if there's polymorphism somewhere, you're *really* in trouble because it's not enough to know what type `i` and `j` are *declared*, you have to know what type they are *right now*, which might involve inspecting an arbitrary amount of code and you can never really be sure if you've looked everywhere thanks to the halting problem (phew!).

When you see `i=j*5` in C++ you are really on your own, bubby, and that, in my mind, reduces the ability to detect possible problems just by looking at code.

None of this was supposed to matter, of course. When you do clever-schoolboy things like override `operator*`, this is meant to be to help you provide a nice waterproof abstraction. Golly, `j` is a Unicode String type, and multiplying a Unicode String by an integer is *obviously* a good abstraction for converting Traditional Chinese to Standard Chinese, right?

The trouble is, of course, that waterproof abstractions aren't. I've already talked about this extensively in The Law of Leaky Abstractions so I won't repeat myself here.

Scott Meyers has made a whole career out of showing you all the ways they fail and bite you, in C++ at least. (By the way, the third edition of Scott's book Effective C++ just came out; it's completely rewritten; get your copy today!)

Okay.

I'm losing track. I better summarize The Story Until Now:

> Look for coding conventions that make wrong code look wrong. Getting the right information collocated all together in the same place on screen in your code lets you see certain types of problems and fix them right away.

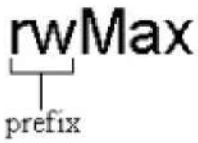
**I'm Hungary**

So now we get back to the infamous Hungarian notation.

Hungarian notation was invented by Microsoft programmer Charles Simonyi. One of the major projects Simonyi worked on at Microsoft was Word; in fact he led the project to create the world's first WYSIWYG word processor, something called Bravo at Xerox Parc.

In WYSIWYG word processing, you have scrollable windows, so every coordinate has to be interpreted as either relative to the window or relative to the page, and that makes a big difference, and keeping them straight is pretty important.

Which, I surmise, is one of the many good reasons Simonyi started using something that came to be called Hungarian notation. It looked like Hungarian, and Simonyi was from Hungary, thus the name. In Simonyi's version of Hungarian notation, every variable was prefixed with a lower case tag that indicated the kind of thing that the variable contained.




I'm using the word *kind* on purpose, there, because Simonyi mistakenly used the word *type* in his paper, and generations of programmers misunderstood what he meant.

If you read Simonyi's paper closely, what he was getting at was the same kind of naming convention as I used in my example above where we decided that `us` meant "unsafe string" and `s` meant "safe string." They're both of type `string`. The compiler won't help you if you assign one to the other and Intellisense won't tell you bupkis. But they are semantically different; they need to be interpreted differently and treated differently and some kind of conversion function will need to be called if you assign one to the other or you will have a *runtime* bug. *If* you're lucky.

Simonyi's original concept for Hungarian notation was called, inside Microsoft,

Apps Hungarian, because it was used in the Applications Division, to wit, Word and Excel. In Excel's source code you see a lot of **`rw`** and **`col`** and when you see those you know that they refer to rows and columns. Yep, they're both integers, but it never makes sense to assign between them. In Word, I'm told, you see a lot of **`xl`** and **`xw`**, where **`xl`** means "horizontal coordinates relative to the layout" and **`xw`** means "horizontal coordinates relative to the window." Both ints. Not interchangeable. In both apps you see a lot of `cb` meaning "count of bytes." Yep, it's an int again, but you know so much more about it just by looking at the variable name. It's a count of bytes: a buffer size. And if you see **`xl = cb`**, well, blow the Bad Code Whistle, that is obviously wrong code, because even though **`xl`** and **`cb`** are both integers, it's completely crazy to set a horizontal offset in pixels to a count of bytes.

In Apps Hungarian prefixes are used for functions, as well as variables. So, to tell you the truth, I've never seen the Word source code, but I'll bet you dollars to donuts there's a function called **`YlFromYw`** which converts from vertical window coordinates to vertical layout coordinates. Apps Hungarian requires the notation **`TypeFromType`** instead of the more traditional **`TypeToType`** so that every function name could begin with the type of thing that it was returning, just like I did earlier in the example when I renamed `Encode` `SFromUs`. In fact in proper Apps Hungarian the Encode function would *have* to be named **`SFromUs`**. Apps Hungarian wouldn't really give you a choice in how to name this function. That's a good thing, because it's one less thing you need to remember, and you don't have to wonder what kind of encoding is being referred to by the word `Encode`: you have something much more precise.

Apps Hungarian was extremely valuable, especially in the days of C programming where the compiler didn't provide a very useful type system.

But then something kind of wrong happened.

The dark side took over Hungarian Notation.

Nobody seems to know why or how, but it appears that the documentation writers on the Windows team inadvertently invented what came to be known as Systems Hungarian.

Somebody, somewhere, read Simonyi's paper, where he used the word "type,"

and thought he meant type, like class, like in a type system, like the type checking that the compiler does. He did not. He explained very carefully exactly what he meant by the word "type," but it didn't help. The damage was done.

Apps Hungarian had very useful, meaningful prefixes like "ix" to mean an index into an array, "c" to mean a count, "d" to mean the difference between two numbers (for example "dx" meant "width"), and so forth.

Systems Hungarian had far less useful prefixes like "l" for long and "ul" for "unsigned long" and "dw" for double word, which is, actually, uh, an unsigned long. In Systems Hungarian, the only thing that the prefix told you was the actual data type of the variable.

This was a subtle but complete misunderstanding of Simonyi's intention and practice, and it just goes to show you that if you write convoluted, dense academic prose nobody will understand it and your ideas will be misinterpreted and then the misinterpreted ideas will be ridiculed even when they weren't your ideas. So in Systems Hungarian you got a lot of `dwFoo` meaning "double word foo," and doggone it, the fact that a variable is a double word tells you darn near nothing useful at all. So it's no wonder people rebelled against Systems Hungarian.

Systems Hungarian was promulgated far and wide; it is the standard throughout the Windows programming documentation; it was spread extensively by books like Charles Petzold's Programming Windows, the bible for learning Windows programming, and it rapidly became the dominant form of Hungarian, even inside Microsoft, where very few programmers outside the Word and Excel teams understood just what a mistake they had made.

And then came The Great Rebellion. Eventually, programmers who never understood Hungarian in the first place noticed that the misunderstood subset they were using was Pretty Dang Annoying and Well-Nigh Useless, and they revolted against it. Now, there are still some nice qualities in Systems Hungarian, which help you see bugs. At the very least, if you use Systems Hungarian, you'll know the type of a variable at the spot where you're using it. But it's not nearly as valuable as Apps Hungarian.

The Great Rebellion hit its peak with the first release of .NET. Microsoft finally started telling people, "Hungarian Notation Is Not Recommended." There was

started telling people, "Hungarian Notation is Not Recommended." There was much rejoicing. I don't even think they bothered saying why. They just went through the naming guidelines section of the document and wrote, "Do Not Use Hungarian Notation" in every entry. Hungarian Notation was so doggone unpopular by this point that nobody really complained, and everybody in the world outside of Excel and Word were relieved at no longer having to use an awkward naming convention that, they thought, was unnecessary in the days of strong type checking and Intellisense.

But there's still a tremendous amount of value to Apps Hungarian, in that it increases collocation in code, which makes the code easier to read, write, debug, and maintain, and, most importantly, it makes wrong code look wrong.

Before we go, there's one more thing I promised to do, which is to bash exceptions one more time. The last time I did that I got in a lot of trouble. In an off-the-cuff remark on the Joel on Software homepage, I wrote that I don't like exceptions because they are, effectively, an invisible goto, which, I reasoned, is even worse than a goto you can see. Of course millions of people jumped down my throat. The only person in the world who leapt to my defense was, of course, Raymond Chen, who is, by the way, the best programmer in the world, so that has to say something, right?

Here's the thing with exceptions, in the context of this article. Your eyes learn to see wrong things, as long as there is something to see, and this prevents bugs. In order to make code really, really robust, when you code-review it, you need to have coding conventions that allow collocation. In other words, the more information about what code is doing is located right in front of your eyes, the better a job you'll do at finding the mistakes. When you have code that says

```
dosomething();
cleanup();
```

… your eyes tell you, what's wrong with that? We always clean up! But the possibility that **`dosomething`** might throw an exception means that **`cleanup`**might not get called. And that's easily fixable, using **`finally`** or whatnot, but that's not my point: my point is that the only way to know that **`cleanup`** is definitely called is to investigate the entire call tree of **`dosomething`** to see if there's anything in there, anywhere, which can throw an exception, and that's ok, and there are

things like checked exceptions to make it less painful, but the real point is that exceptions eliminate collocation. You have to look *somewhere else* to answer a question of whether code is doing the right thing, so you're not able to take advantage of your eye's built-in ability to learn to see wrong code, because there's nothing to see.

Now, when I'm writing a dinky script to gather up a bunch of data and print it once a day, heck yeah, exceptions are great. I like nothing more than to ignore all possible wrong things that can happen and just wrap up the whole damn program in a big ol' try/catch that emails me if anything ever goes wrong. Exceptions are fine for quick-and-dirty code, for scripts, and for code that is neither mission critical nor life-sustaining. But if you're writing an operating system, or a nuclear power plant, or the software to control a high speed circular saw used in open heart surgery, exceptions are extremely dangerous.

I know people will assume that I'm a lame programmer for failing to understand exceptions properly and failing to understand all the ways they can improve my life if only I was willing to let exceptions into my heart, but, too bad. The way to write really reliable code is to try to use simple tools that take into account typical human frailty, not complex tools with hidden side effects and leaky abstractions that assume an infallible programmer.

**More Reading**

If you're still all gung-ho about exceptions, read Raymond Chen's essay Cleaner, more elegant, and harder to recognize. "It is extraordinarily difficult to see the difference between bad exception-based code and not-bad exception-based code… exceptions are too hard and I'm not smart enough to handle them."

Raymond's rant about Death by Macros, A rant against flow control macros, is about another case where failing to get information all in the same place makes code unmaintainable. "When you see code that uses [macros], you have to go dig through header files to figure out what they do."

For background on the history of Hungarian notation, start with Simonyi's original paper, Hungarian Notation. Doug Klunder introduced this to the Excel team in a somewhat clearer paper. For more stories about Hungarian and how it got ruined by documentation writers, read Larry Osterman's post, especially Scott Ludwig's comment, or Rick Schaut's post.

Scott Ludwig's comment, or Rick Schaut's post.

## WANT TO KNOW MORE?

You're reading Joel on Software, stuffed with years and years of completely raving mad articles about software development, managing software teams, designing user interfaces, running successful software companies, and rubber duckies.

## ABOUT THE AUTHOR.

I'm Joel Spolsky, co-founder of Trello and Fog Creek Software, and CEO of Stack Overflow. More about me.

← PREVIOUS POST
News

NEXT POST →
News