

Best Practices for Writing Samples



By Guy Smith
Programmer Writer
Steyer Associates

May 2011

Steyer
ASSOCIATES INC.

*Technical
Communications
Staffing*

I spend a lot of my time writing sample applications. It's one of the best ways to figure out how an API works. The exercise gives you a good sense of what sorts of issues you have to explain to your audience. Anything that you trip over in the process of writing your samples is probably something that you should explain to your audience.

As a side benefit, samples give you a source of excerpts for your documentation. However, there are some tricks to writing samples so that they work well in documentation, and some tricks to making effective use of code excerpts. Here's some of what I've learned in over fifteen years of writing samples for everything from device drivers to games and incorporating them into documentation.

What Is a Sample?

Production-level samples are best-practice examples of how to write real applications. They are typically large complex applications that carefully validate parameters, check return values, handle exceptions, use extensive modularization, implement a well-designed GUI, and so on. The upside of this approach is that it gives readers bullet-proof code that shows how to do things right. The downside is that these samples are usually difficult to read through. The programming logic can be difficult to follow and the API that you are trying to explain tends to get buried in a mass of relatively uninteresting or routine code.

Pedagogical samples are typically short simple applications that focus on a relatively limited aspect of an API. They have an easy-to-follow program flow with limited modularization, little or no routine error checking or parameter validation, and so on. The upside of these samples is code with simple straightforward programming logic that shows how the API works with a minimum of distractions. The downside is that pedagogical samples usually fall well short of the standards expected for shippable code.

It's nearly, if not completely, impossible to implement samples that serve both purposes. Because my primary concern is showing folks the ins and outs of how to use an API, I generally stick with pedagogical samples to back up my documentation. They are much easier to explain to the reader, and quicker and easier to implement. The programmers I write for can generally figure out how to adapt pedagogical samples for production-level purposes without too much difficulty. To alleviate concerns that I might be encouraging poor coding practices, I generally include an explicit warning in the accompanying documentation, something like "Routine error-correction code has been omitted for brevity and clarity."

The rest of this article contains some thoughts about how to write good pedagogical samples. It's based on my own experience and on working with samples written by some SDEs that I've worked with who really understand how to write samples. It also includes the best practices I've learned from struggling through more than a few poorly conceived and written samples.

1. Keep it Readable

A pedagogical sample is meant to be read, so don't make it harder than necessary. Here are some general ideas on how to write readable code. It's not necessarily the way you would write production-level code, but for documentation, readability is usually more important.

The C family of languages lets you implement a lot of functionality in one compact and thoroughly cryptic statement. Resist the temptation to show off your programming skills; write things out. For example, instead of implementing a long LINQ query as a single statement, break it down into a series of simple queries that you can explain one at a time.

A variable or method name should help reader easily remember its purpose and how it fits into the rest of the application. Calling a variable "w" or "sf" tells the reader little or nothing about its purpose, and by the time they've read a few more paragraphs, they'll have forgotten what it means. "weightVector" or "shellFolder" are much more useful to your readers.

Don't modularize more than necessary. It's hard to follow the programming logic if you have to jump in and out of a function every few lines. It's much easier to read through an application if most, if not all, of the code is contiguous. Modularization is often necessary and appropriate, but keep it to a minimum. If it's code that's used multiple times, it makes sense to put it in a separate method. If it's just a few lines of code that are used only once, don't put them in a separate method.

Limit the number of files in the project. It's easier to read through a single file. Don't make your reader jump from file to file more than necessary.

The sample should be completely self-contained. In particular, don't put a bunch of code in a library that is then used by multiple samples. It's a convenient and efficient way to implement applications, but it's also very hard for the reader to follow. One team I worked with implemented a set of samples this way, and then had to go back and reimplement them all after getting flak from their readers about how hard the samples were to follow.

Avoid routine error handling code, such as checking return values, handling exceptions, and validating parameters, and so on. It's important for production-level work, but most of it is basically boilerplate and buries the API that you are trying to demonstrate in a mass of uninteresting code. I generally write with the assumption that the API will work as usual; methods won't fail, return values will provide the expected data, and so on. I usually include error-handling code only if it addresses an important scenario for the API. For example, one sample I wrote depended on a web service that was often too busy to respond to requests. In that case, an error was expected and handling it correctly was a key to using the API.

Avoid using "var" unless it's necessary. It's a convenient shorthand, but it can leave the reader wondering what type it represents, especially if they are looking at the code in a document and don't have IntelliSense to fall back on. Better to use an explicit type. I'll often use fully qualified types, just to help the reader learn their way around the API.

2. Keep It Focused

Avoid the temptation to cram the whole API into a single sample. It's better to write several short samples, each one highlighting a relatively small well-defined aspect of the API. I generally try to limit myself to no more than a handful of closely related APIs—sometimes just a single API. The rest of the code is whatever is necessary to show how the API works in an application.

If an API is large and complex, consider writing a series of related samples. Start with the bare essentials and have successive samples build on the preceding ones until you arrive at a fairly full-featured application. That way, you are introducing only a manageable number of new concepts with each sample.

However, don't overdo the simplicity thing. APIs should be presented in a context that bears at least a vague resemblance to how they will be used in a real application. If the sample is too stripped down, you can end up with little more than a trivial example of API syntax, which isn't much more useful than the reference page, if that. If a group of APIs is typically used together, put them all in and show how they are typically used together.

3. Keep It Simple

Don't do more than is necessary to put the API through its paces. I usually implement samples as console applications, unless the API I'm working with is graphical. GUI applications require a lot of code just to implement and manage the UI, and if it doesn't really have anything to do with the API, it's just a distraction. Better to implement the sample as a console application, which is typically much shorter and requires less extraneous code.

If you do have to implement a GUI, resist the temptation to show off your design skills. Don't use three buttons if one will get the point across. It just clutters up the code with unnecessary event handlers. Keep it simple and straightforward, with as little extraneous code as possible.

4. Use Excerpts

Documentation almost always benefits from sample excerpts. Sometimes, the whole purpose of writing a sample is just to provide a set of excerpts. Here are some ideas on how to excerpt code.

First and foremost, sample excerpts included in documentation must be accurate. That means no misspelled API names, no missing commas or semi-colons, or any other easy-to-make mistakes. The easiest way to ensure that your samples are at least syntactically and grammatically correct is to take excerpts only from actual samples that compile and run. If you freehand your examples, it's too easy to make hard-to-catch mistakes that will, at best, annoy your readers and, at worst, seriously mislead them.

Keep the excerpts short and to the point. Ideally, an excerpt should be no more than a half screen long, preferably less. Instead of including an entire 30 or 40 line method, just extract the key code for your immediate purpose. I'll often work through a lengthy method in multiple small chunks, instead of presenting it as one large indigestible block of code. An alternative approach that's sometimes useful is to use a largish excerpt and insert numbered comments to identify the key sections. You can then work through the sections one at a time in the following text, using the numbers to help the reader find the associated code.

If the sample includes routine error correction code, edit it out unless it's truly important. Use an ellipsis (...) to represent the missing code and refer the reader to the sample for details.

Limit comments or omit them completely. They are useful in the sample, but I prefer to remove them from excerpts and put the information in the accompanying text.

5. Enjoy Breaking Things

Finally, have fun breaking stuff. Writing samples, even simple ones, often makes you an ex officio member of the test team. I've turned up a lot of bugs in the process of writing samples, some of them major. That's not meant to knock the test team. They can't think of everything and sometimes I'm working on parts of the API that they simply haven't gotten to. Testers usually appreciate the help, and smoking out a shipstopper before it is released, does wonders for your credibility with the development and test team.

About Steyer Associates

When you're looking for the best technical communications professionals for your next project, turn to the expert recruiters at Steyer Associates. Our clients repeatedly tell us that we're their first choice for technical communications staffing thanks to:

- **The exceptional quality of our talent**
- **The deep knowledge and experience of our recruiters**
- **The high level of customer service that we provide**

These factors help us earn almost 90 percent of our business from repeat clients and referrals. Unlike most other staffing agencies, we focus on the technical communications arena and have more than 12 years of experience helping clients meet their project goals.

Contact us today for an initial consultation, or to learn more about what makes Steyer the premier technical communications staffing agency in the Pacific Northwest.



*Technical
Communications
Staffing*

115 Hall Brothers Loop NW
Suite104
Bainbridge Island, WA 98110

206.780.3345 TEL
206.780.9083 FAX
info@steyer.net

 twitter.com/SteyerTalent

© 2011 Steyer Associates Inc. All rights reserved.