Introduction

> style:1b. the shadow-producing pin of a sundial.
> 2c. the custom or plan followed in spelling, capitalization, punctuation, and typographic arrangement and display.
>
> —*Webster's New Collegiate Dictionary*

The syntax of a programming language tells you what code it is possible to write—what machines will understand. Style tells you what you ought to write—what humans reading the code will understand. Code written with a consistent, simple style is maintainable, robust, and contains fewer bugs. Code written with no regard to style contains more bugs, and may simply be thrown away and rewritten rather than maintained.

Attending to style is particularly important when developing as a team. Consistent style facilitates communication, because it enables team members to read and understand each other's work more easily. In our experience, the value of consistent programming style grows exponentially with the number of people working with the code.

Our favorite style guides are classics: Strunk and White's *The Elements of Style*[3] and Kernighan and Plauger's *The Elements of Programming Style*.[4] These small books work because they

[3] Strunk, William Jr., and E. B. White. *The Elements of Style, Fourth Edition.* (Allyn & Bacon, 2000).
[4] Kernighan, Brian and P. J. Plauger. *The Elements of Programming Style.* (New York: McGraw-Hill, 1988).

are simple: a list of rules, each containing a brief explanation and examples of correct, and sometimes incorrect, use. We followed the same pattern in this book. This simple treatment—a series of rules—enabled us to keep this book short and easy to understand.

Some of the advice that you read here may seem obvious to you, particularly if you've been writing code for a long time. Others may disagree with some of our specific suggestions about formatting or indentation. The most important thing is consistency. What we've tried to do here is distill many decades of development experience into an easily accessible set of heuristics that encourage consistent coding practice (and hopefully help you avoid some coding traps along the way). The idea is to provide a clear standard to follow so programmers can spend their time on solving the problems of their customers instead of worrying about things like naming conventions and formatting.

The guidelines in this book complement the official .NET design guidelines in the ECMA C# specification[5] and Krzysztof Cwalina and Brad Abrams' excellent *Framework Design Guidelines*.[6] This book extends those guidelines to internal implementation and coding style.

Disclaimer

We have dramatically simplified the code samples used in this book to highlight the concepts related to a particular rule. In many cases, these code fragments do not conform to

[5] ECMA International, Standard ECMA-334: "C# Language Specification." 3rd Edition, June 2005. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[6] Cwalina, Krzysztof and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley, 2005. ISBN 0321246756.

conventions described elsewhere in this book—they lack real documentation and fail to meet certain minimum declarative requirements. Do not treat these fragments as definitive examples of real code!

Acknowledgments

Books like these are necessarily a team effort. Major contributions came from the original authors of *The Elements of Java Style*: Al Vermeulen, Scott Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson, and the original authors of *The Elements of C++ Style*: Trevor Misfeldt, Greg Bumgardner, and Andrew Gray. Both of those books have some roots in "C++ Design, Implementation, and Style Guide," written by Tom Keffer, the "Rogue Wave Java Style Guide," and the "Ambysoft Inc. Coding Standards for Java," documents to which Jeremy Smith, Tom Keffer, Wayne Gramlich, Pete Handsman, and Cris Perdue all contributed.

Thanks also to the reviewers who provided valuable feedback on drafts of this book, particularly Brad Abrams, Krzysztof Cwalina, and Mark Vulfson of Microsoft Corporation; Mike Gunderloy of Larkware; and Michael Gerfen of Evolution Software Design.

This book would certainly never have happened without the help and encouragement of the folks at Cambridge University Press, particularly Jessica Farris and Lauren Cowles, who kept us on track throughout the writing and publication process.

1.

General Principles

While it is important to write software that performs well, many other issues should concern the professional developer. *Good* software gets the job done. But *great* software, written with a consistent style, is predictable, robust, maintainable, supportable, and extensible.

1. Adhere to the Style of the Original

When modifying existing software, your changes should follow the style of the original code.[7] Do not introduce a new coding style in a modification, and do not attempt to rewrite the old software just to make it match the new style. The use of different styles within a single source file produces code that is more difficult to read and comprehend. Rewriting old code simply to change its style may result in the introduction of costly yet avoidable defects.

2. Adhere to the Principle of Least Astonishment

The *Principle of Least Astonishment* suggests you should avoid doing things that would surprise other software developers. This implies that the means of interaction and the behavior exhibited by your software must be predictable and

[7] Jim Karabatsos. "When does this document apply?" In "Visual Basic Programming Standards." (GUI Computing Ltd., 22 March 1996).

consistent,[8] and, if not, the documentation must clearly identify and justify any unusual patterns of use or behavior.

To minimize the chances that anyone would encounter something surprising in your software, you should emphasize the following characteristics in the design, implementation, packaging, and documentation of your software:

Simplicity	Meet the expectations of your users with simple classes and simple methods.
Clarity	Ensure that each class, interface, method, variable, and object has a clear purpose. Explain where, when, why, and how to use each.
Completeness	Provide the minimum functionality that any reasonable user would expect to find and use. Create complete documentation; document all features and functionality.
Consistency	Similar entities should look and behave the same; dissimilar entities should look and behave differently. Create and apply consistent standards whenever possible.
Robustness	Provide predictable, documented behavior in response to errors and exceptions. Do not hide errors and do not force clients to detect errors.

3. Do It Right the First Time

Apply these rules to any code you write, not just code destined for production. More often than not, some piece of prototype

[8] George Brackett. "Class 6: Designing for Communication: Layout, Structure, Navigation for Nets and Webs." In "Course T525: Designing Educational Experiences for Networks and Webs." (Harvard Graduate School of Education, 26 August 1999).

or experimental code will make its way into a finished product, so you should anticipate this eventuality. Even if your code never makes it into production, someone else may still have to read it. Anyone who must look at your code will appreciate your professionalism and foresight at having consistently applied these rules from the start.

4. Document Any Deviations

No standard is perfect and no standard is universally applicable. Sometimes you will find yourself in a situation where you need to deviate from an established standard. Regardless, strive for clarity and consistency.

Before you decide to ignore a rule, you should first make sure you understand why the rule exists and what the consequences are if it is not applied. If you decide you must violate a rule, then document why you have done so.

This is the *prime directive*.

5. Consider Using a Code-Checking Tool to Enforce Coding Standards

A source code analysis tool enables you to check your code for compliance with coding standards and best practices. For example, FxCop[9] is a popular .NET code analysis tool that uses reflection, MSIL parsing, and callgraph analysis to check for conformance to the .NET Framework design guidelines. FxCop is extensible, and can thus incorporate the particular coding standards used by your own organization.

---

[9] http://www.gotdotnet.com/team/fxcop/.

2.

Formatting

2.1 White Space

6. Include White Space

White space is the area on a page devoid of visible characters. Code with too little white space is difficult to read and understand, so use plenty of white space to delineate methods, comments, code blocks, and expressions clearly.

Use a single space to separate the keywords, parentheses, and curly braces in control flow statements:

```
for·(...)
{
  //...
}

while·(...)
{
  //...
}

do
{
  //...
}·while·(...);

switch·(...)
{
  //...
}
```

```
if·(...)
{
  //...
}
else·if·(...)
{
  //...
}
else
{
  //...
}

try
{
  //...
}
catch·(Exception)
{
  //...
}
finally
{
  //...
}
```

Use a single space on either side of binary operators, except for the "." operator:

```
double length = Math.Sqrt(x * x + y * y);
double xNorm = (length > 0.0) ? (x / length) : x;
double currentTemperature =
    engineBlock.Temperature;
```

Use a single space after commas and semicolons:

```
Vector normalizedVector =
    NormalizeVector(x, y, z);

for (int i = 0; i < 100; i++)
{
  //...
}
```

Use a single space between the parentheses for the parameter list in a method declaration:

```
Vector NormalizeVector(double x, double y,
                       double z)
{
  //...
}
```

Use blank lines to separate each logical section of a method body:

```
public void HandleMessage(Message message)
{
  string content = message.ReadContent();
  switch (message.ErrorLevel)
  {
    case ErrorLevel.Warning:
      //... do some stuff here ...
      break;
    case ErrorLevel.Severe:
      //... do some stuff here ...
      break;
    default:
      //... do some stuff here ...
      break;
  }
}
```

Use blank lines to separate each method definition in a class:

```
public void SendEmail()
{
  //...
}

public void SendFax()
{
  //...
}
```

7. Use Indented Block Statements

One way to improve code readability is to group individual statements into block statements and uniformly indent the content of each block to set off its contents from the surrounding code.

If you generate code using an integrated development environment (such as Visual Studio), make sure that everyone on your team uses consistent indentation options. If you are generating the code by hand, use two spaces to ensure readability without taking up too much space (see Rule #9):

```
void PesterCustomer(Customer customer)
{
  customer.SendLetter();
  if (customer.HasEmailAddress())
  {
    customer.SendEmail();
    if (customer.IsForgetful())
    {
      customer.ScheduleReminderEmail();
    }
  }
  if (customer.HasFaxNumber())
  {
```