# Writing Good Software
## Development and Coding Standards

by alan partis
thundernet development group, inc.
boynton beach, florida

**Revision #4**
**16-Jun-02**

This document specifies suggested rules and guidelines for writing software source code. While the rules are considered required practices, the guidelines are more like strong suggestions – deviate from the guidelines only with specific reason and need.

These standards are based loosely on "Java Software Coding Standards Guide" as published by Netscape Communications, Inc., the GNU Coding Standards, and the personal experiences of the author. This document constitutes a proposed standard for Thundernet Development Group members. Comments for changes are welcome and encouraged.

## Table of Contents

# 1. ABSTRACT

## 1.1 Goals

The objective of this document is to establish a standard within the Thundernet Development Group for the successful development of professional software components and projects. Success in this instance is not limited only to whether a particular module behaves properly, but is also easy for debugging (in the rare instance that a bug exists) and especially easy to maintain.

The number one cost in any software project over its lifetime is the cost of maintenance. With this in mind, it is easy to see that the easiest place to attain cost savings is to improve the maintainability of code.

Thundernet's goal is to produce software that is highly reliable, maintainable, extensible, and where possible, elegant. While flashy code may impress others with our knowledge and skills, it is boring and *"simple"* code that is easiest to test and debug and therefore wins contracts and brings home the bacon.

Using the rules and guidelines that follow should result in:

- code that is easy to read, follow, and understand
- documentation that is accurate and helpful
- reusable objects that will reduce costs of future projects

It is also the desire on the part of Thundernet to establish these rules and guidelines so as to shorten the learning curve of new members, and more efficiently bring people into a project later in the project lifecycle. People familiar with the standard will have a much easier time learning and understanding the coding algorithms and therefore be more readily helpful during a project's *"crunch time."* Also, the quality of source code that is delivered to a client as part of a project's production reflects on Thundernet's abilities and increases our credibility

## 1.2 Scope

This document covers only the rules and guidelines for the development of software source code. It is not concerned with object architecture, algorithm development, or usage of object libraries. While C is the default language for the application of these standards/guidelines, they are generally applicable to many other languages. Some sections are also provided explicitly for other languages such as C++ and/or Java so as to note certain variations or situations unique to those languages.

# 2. GENERAL RULES

## *2.1 When In Rome …*

While it is the purpose of this document to establish standards for good coding practices, it is good to keep in mind that there are always caveats. The most notable guiding principle is the "When in Rome, do as the Romans do" rule.

Always implement these standards when writing new code, but there are times when these standards should be ignored. Here are the rules regarding when NOT to impose these standards:

- Always try to follow the style of existing code in a project. Mixing styles in a single source file makes it very difficult for future programmers to debug or make enhancements.
- Do not impose these standards on existing code while debugging. If only minor modifications are being made to resolve a bug, then to make wholesale style changes raises the risk of introducing new bugs. Further, the more changes made, the more difficult it is to use tools (such as diff for example) to compare revisions of a single file from a repository. This type of situation can dramatically add to the time necessary to make future modifications.
- Do not apply these standards to code that is to be maintained by another group, if that group has established their own rules and guidelines and they differ from these.

# 3. TOOLS AND ENVIRONMENTS

Every project has various requirements regarding the usage of development tools. Some projects may require the use of a particular tool or code editor, while others have no restrictions whatsoever. In general, all Thundernet projects, to the extent possible, should rely on the established toolsets commonly utilized in the open source development community. In all cases, however, the acceptable tools for a given project should be established at the outset. There are advantages and disadvantages to every tool and these should be properly evaluated to determine those most appropriate for the given project.

## *3.1 Editors*

Over the years, countless programmer's editors have been produced and distributed. Some have earned wide acclaim and loyal followings while others have truly forgetable. Much like other religious battles, the arguments as to which editors are the best are generally pointless and not worth pursuing – everyone has their favorite. However, since one of the keys to success in the delivery of a software project is a programmer's comfort in the environment, it is best to simply specify the rules and guidelines for using these tools and let individual programmers make their own decisions.

Programmer's editors provide the most flexible development environment and are also the most portable and ubiquitous. Because of this, they are likely to be used somewhere along the line in almost every development project. As such, it is necessary to establish standards regarding their usage.

### *3.1.1 Tabs*

In all instances, pressing the <tab> key on the keyboard should insert an appropriate number of spaces. <Tab> characters should never be emedded in the source code. <Tabs> are treated differently in different environments and cause unanticipated problems. When a source file is saved, any existing <tab> characters should be converted to the appropriate number of <space> characters.

### *3.1.2 Spaces on Ends of Lines*

Whenever possible, trailing blank spaces at the end lines should be truncated. Most good editors have the ability to filter unwanted "white space" characters after the last non-blank character.

### *3.1.3 End of File*

No special characters should be embedded as the **end-of-file** marker. Many versions of DOS and Unix required files to end with a special character (entered with Ctrl-Z in many cases), but this practice can be problematic for some of today's tools.

## *3.2 IDEs*

Integrated Development Environments (IDEs) are generally GUI-based coding tools that try to simplify the programming experience, or provide additional functionality that can make software development more efficient. At the very least, most IDEs provide an editor, an integrated JVM (for Java IDE's), and a run-time and debug facility. Some Java IDEs provide additional functionality for generating Java Beans and other graphical components and applications.

At one end of the spectrum, IDEs can dramatically decrease the amount of time necessary to deliver projects. At the other end, they can cause no end of frustration and increase the time necessary to

deliver a project. As with anything, careful evaluation should be made of any canidate IDEs before accepting their use.

In general, IDEs should be avoided excpet for development of GUI applets or applications. For development of server-side components, system services, Java servlets or other "back end" business objects, IDEs do not often provide the best solution. Some of the issues are detailed here.

### *3.2.1 Automatic Java Code Generation*

Most recent Java IDEs (Visual Age, Visual Café, etc.) have the ability to automatically generate Java code based on some bit of input from the programmer. At first glance this seems to be a welcome assistance since it is unlikely to encounter syntax or logic errors in this code. It can, however, cause significant problems if there is reason to believe that any programmer modifications will be necessary to the generated code. It's a safe bet that the generated code will not conform to these Java coding standards; some more than others. The greater the deviation, the greater the amount of time required to make subsequent modifications or customizations.

Code generation features should be used quite sparingly, and only if the programmer has few other attractive options.

### *3.2.2 Integrated Editors*

In almost all cases, IDEs provide their own code editing environment. Unfortunately, these editors are rarely as configurable and customizable as most programmer's editors. As a result, more often than not, a programmer's productivity is reduced by having to learn a whole new editor, and one that is likely to have fewer and less powerful features. It seems that the ideal situation would be one where a programmer could integrate the editor of their own choosing into the IDE.

When using an integrated editor, try to adhere to as many of the rules layed out for programmer's editors i.e. spaces instead of tabs, etc. Be vigilant to the troubles that the development team may encounter if some programmers are using stand alone editors while others are using those from an IDE. In the most extreme cases, the two won't be able to work with source files previously edited in the the other environment. In the case of IBM's Visual Age for Java, the code editing environment almost completely removes the concept of a source file by only displaying code from one method at a time. Even though Visual Age allows for the importing and exporting of source code so it can be modified in a separate environment, the changes that Visual Age imposes can be rather annoying.

Use of the editing environment provided by an IDE is up to the programmer's discretion, but should be approached with caution.

### *3.2.3 Integrated Repositories*

To provide the most potability and flexibility, the source files for most projects are stored in a directory tree that reflects the modules being built and further maintained in a versioning repository such as CVS. However, in the case of Visual Age for Java, a source repository is built into the IDE. This is great if the project team has chosen Visual Age for Java as the development tool of choice, but if not, then it should not be used. Since it is unlikely that Thundernet will adopt the IBM repository tool in Visual Age for Java as the standard repository across all projects, usage of IBM's repository can only complicate the development, versioning, and backup processes already in place.

Proprietary repositories contained in IDEs should be avoided. However, if an IDE can easily integrate with the standard repository for Thundernet projects, then this should be considered a good thing.

### *3.2.4 Limitations in Meeting Java 2 Specs*

When considering the viability of using a Java IDE, any limitations caused by the IDE should be investigated thoroughly. For example, IBM's Visual Age for Java, v2.0 does not support inner classes. This is a conflict with the latest GUI programming techniques as specified by Sun for Java 2. This also prevents the methods for unit testing specified elsewhere in this document.

Visual Age for Java v2.0 should not be used. Other IDE's with similar limitations should also not be used.

## *3.3 Debuggers*

The most essential tools in any programmer's toolbox are the debuggers. Most good debuggers support source level debugging which can save hours or days in the process of seeking out bugs and their solutions. For the Java platform, debugging hooks are provided directly in the JVM and exposed for debugger manufacturers through the JPDA (Java Platform Debugger Architecture). Debuggers are often part of an IDE and can be difficult to find as independent products.

Choosing a debugger is largely a matter of programmer comfort and familiarity as most high-level debuggers operate similarly. For that reason, there is little to stipulate about which debugger to use or how to use them. Suffice it to say that good software engineers should be familiar with a few different ones and use them as necessary to avoid lengthy 'debugging by printf()' sessions.

# 4. GENERAL CODING RULES

## 4.1 Naming Conventions

One large source of frustration for programmers that are tasked with debugging or maintaining someone else's code, whether during the initial development or ongoing maintenance phases of a project, is inconsistent naming conventions. Consistency in naming items in code can go a long way toward making code more readable and understandable.

The use of the so-called "reverse Hungarian" naming convention is at the discretion of the programmer.

The following rules govern the conventions for naming various items in Java source code.

### 4.1.1 Constants

Constants, whether for strings or numerics, should always be in all caps and use underscores to separate words.

For example:

```
#define       MONTHS_IN_YEAR    12
#define       TOKEN_STRING      "abc"
```

in Java:

```
public static final int     MONTHS_IN_YEAR = 12;
public static final String  TOKEN_STRING   = "abc";
```

### 4.1.2 Global Variables

Global variables should be used sparingly, and only if the architecture specifies it. When global variables are used, they are named using concatentated words in initial caps.

For example:

```
int     UserId;
```

in Java:

```
public Int  UserId;
```

### 4.1.3 Functions/Methods and their Arguments

Function/method names should be composed of descriptive concatenated words using initial caps, but should always begin with a lower case letter.

Arguments to functions may be prefixed with "arg", especially in large functions for greater visability, but are otherwised derived from descriptive words concatenated together in initial caps.

For example:

```
void myAction(int argUserChoice, SomeType argTargetObject)
{
} /* end of method myAction() */
```

Functions that compose the exposed public API for a library (whether static or dynamic/shared) should all share a common lower case prefix to provide consistency in naming and usage as well as help avoid naming collisions.

### 4.1.4 Local Variables

Local variables are composed of descriptive words (or initials for convention items) concatenated with underscores and in all lower case.

For example:

```
char   *login_name;        // retrieved from User object
long    mean_temperature;  // calculated by formula
int     rc;                // return code
```

### 4.1.5 Structures, Types, and Classes

Classes in C++ and Java are functionally very similar to `struct`, `typedef`, and `union` in C. All of these constructs should use descriptive names (composed of multiple words if necessary). The "initial caps" convention should be used for all words, including the first word.

Names of Java *Interface* classes should end with "able".

Names of abstract classes should end be prefixed with a capital "A".

For example:

```
struct CarEngine {
      int    cylinders;
      int    max_horse_power;
};

typedef _Car {
      char              *color;
      int                doors;
      struct CarEngine   engine;
} Car;
```

in Java:

| Class Type | Acceptable | Unacceptable |
|---|---|---|
| Concrete | `GoodClassName` | `bad_class_name` |
| Interface | `Sortable` | `Isort` |
| Abstract | `AHighLevelClass` | `HighLevelClass` |

### 4.1.6 Java Packages

All packages produced by Thundernet development teams should begin with ***com.thundernet.*** Packages should be further deliniated with references to the client and project. Package names should be entirely in lower case letters and should be single words (no hypens or underscores).

The pattern to follow is:

package com.thundernet.<client>.<project>;

For example:

package com.thundernet.sony.la;

This refers to the package of classes that may be produced on a hypothetical project for the Latin America division of Sony.

## *4.2 Comments*

## *4.3 Source File Layout*

functions in alphabetical order

## *4.4 Unit Testing*

# 5. SYNTAX STYLE GUIDELINES

## 5.1 Grammar

### 5.1.1 Spacing and Indentation

space after language control statements

space after commas

3 blank lines between functions

4 space indentation

### 5.1.2 Variable Declarations

at the top of methods, not embedded in code blocks

listed in alphabetical order

### 5.1.3 Braces

K&R brace style

### 5.1.4 if … else

```
if (expression) {
    statement1;
    statement2;
} else if (another_expression) {
    statement3;
    statement4;
} else {
    statement5;
    statement6;
}
```

### 5.1.5 for

```
for (i = 0; i < 10; i++) {
    statement1;
}
```

### 5.1.6 while

```
while (rc == SUCCESS) {
    rc = apiFunction();
}
```

### 5.1.7 do … while

```
do {
    rc = apiFunction();
} while (rc != NULL);
```

### 5.1.8 switch

```
switch (c) {
case 'a':
    x = funcA();
    break;
case 'b':
    x = funcB();
    break;
default:
    x = default_value;
}
```

### 5.1.9 try … catch … finally (Java / C++)

```
try {
    exceptionalFunc();
    result = doSomething();
} catch (...) {
    printf("exception detected and handled");
}
```