GO FIRST Source Code Conventions

Max Veit
GO FIRST Project Manager #2

Last Updated: August 1, 2012

0.   Contents

<div style="border: 1px solid black; padding: 10px;">

1. Purpose and Audience

</div>

The purpose of this document is to establish a uniform style of writing source code for all programs written for GO FIRST. This is intended to facilitate collaboration between programmers as well as maintainability of software, since a uniform style removes one of the main barriers to understanding the source code written by others (or even yourself!).

This document is aimed at anybody interested in writing programs that will become a part of or aid GO FIRST projects. It assumes only a basic, general knowledge of programming, including the existence of different programming languages, the purpose of a source code file, and the existence of a language-specific syntax for the text contained therein. However, the style guides in Section 3 do assume a basic familiarity with the specific syntax of the language being described. To better understand these guides it might be helpful to find a simple introductory tutorial in the language and its syntactic constructs (such as function definitions, "if," and "for" constructs).

<div style="border: 1px solid black; padding: 10px;">

2. Language Classes

</div>

This section lists the classes of languages that will be discussed in this document.

C-syntax (C, C++, Java and C#)
> This is a family of languages with differing programming paradigms. The one thing they have in common is the basic syntax derived from C, so they will generally be discussed as a group in this document. We will almost certainly be using one or more of these languages in most GO FIRST projects.

Python
> This language blurs the boundaries between programming and scripting; its rapid development cycle and interpreted nature make it ideal for high-level control of large, composite programs, an area known as the "glue layer." Its syntax is much simpler and more consistent than that of the C-syntax languages. Whether we use this language in GO FIRST projects remains to be seen.

LabVIEW
> The language designed for use with National Instruments (NI) hardware. Since we plan on using NI controllers in our larger robot projects we will almost certainly need to use LabVIEW at some point. What distinguishes it from most other languages is that it is a

graphical language; programs are created not by writing statements to a source code file but rather by visually connecting virtual blocks with wires. This graphical nature makes it difficult to set specific style guidelines for LabVIEW code. Some conventions are already imposed upon the programmer, but the visual layout of the blocks and wires themselves is completely up to the programmer. "Spaghetti code" that looks like a literal pile of spaghetti can quickly ensue if you are not careful to organize the blocks and wires in a logical, readable fashion. A general rule: If you cannot understand the code that you wrote yesterday, you need to put more effort into organizing your code layout.

3. References

So as not to reinvent the wheel, we will be holding ourselves to some already established and well-known standards. Also includes a few humorous general tips on programming.

- Sun Code Conventons for Java: http://www.oracle.com/technetwork/java/codeconventions-150003.pdf
  This is a good, but not perfect, introduction to basic style for Java source code. It provides a guide for the general "cosmetics" of source code files in C-like languages so programmers don't have to fight with something as trivial and infuriating as brace placement. We will be using it with a few modifications (detailed below) for all of our Java, C and C++ code because of the syntactic similarities between the languages. Please read this thoroughly if you have no experience with C syntax (come on, it's only 18 pages. A TL;DR response will disqualify you from our programming team). If you have programmed in the C-syntax languages before, at least skim the guide to see what specific conventions well be using.

- Style Guide for Python Code: http://www.python.org/dev/peps/pep-0008/
  A very good guide for Python code with general advice that applies to other programming languages as well. Please at least read the introduction, first section, and section on comments. If we do any coding in Python, the style should follow this guide. Comments in other languages should follow the advice in this guide; for all other areas the language's style guide naturally has precedence.

- Pasta Code: http://www.gnu.org/fun/jokes/pasta.code.html
  If youre doing object-oriented programming, please try for ravioli code. Canderli code and Ristto code should be avoided because of poor maintainability. Polenta code is nice, too, if it never needs to be modified (which is generally never the case, so watch out!)

- Good Code: http://xkcd.com/844/
  Please try to stay in the "Code Well" loop. Yes, requirements do change, and we've had to

throw out a few projects because of this, but trust me: this is not always the case!

4. Modifications to the Java Style Guide

There are a few topics where the Java style guide provides advice that is either unsuited to the GO FIRST software development model or actually hurts the readability of code.

5. Technical Notes

- Encoding: Please use UTF-8 for general symbol support and inter-system compatibility. Line-endings usually aren't a problem if your editor is not Notepad (if it is, shame on you. See Appendix A), but if we have problems with this we'll have to go with Windows line endings (CR LF).

- Line Length: 79 characters is the *maximum* amount of characters allowed in one line of source code. Don't go over it, even by one character. Doing so will annoy every other person who tries to read the code with an 80-character terminal, as the resulting line wraps visually disrupt and distort the text. Every decent language has a feature for long-line continuation; please use it.
  The only exception here is the source code for documentation to be typeset with TEX/ LATEX(like this document). Since such files contain long blocks of prose, it can often be desirable to let the editor handle the line wrapping.

- General Rules: We will presumably have programmers on all different systems, so if you have a choice between convenience and compatibility you should almost always choose compatibility.

6. Whitespace

Please read the Python style guide, sections "Code lay-out" and "Whitespace. . . " (you can ignore the "Imports" subsection since it is Python-specific). These rules apply well both to Python and C-syntax code, so please observe them.

Yes, Python is quite different from other languages in terms of handling whitespace, but I think imposing a standard on how whitespace can be used to emphasize code structure is a good thing in any programming language. For one, it quells the stupid "holy wars" on what style of indentation is preferable, or even superior. If you want a good reason to use spaces instead of tabs, consider the case of continuation of long lines of code. Clearly you want to align the continuation in some logical way with the preceding line, such as a continued argument list of a function that you want to align with the first argument in the previous line. Plain tabs are just a recipe for disaster (different editors will throw off the indentation wildly), and mixing spaces and tabs for indentation in one file is generally frowned upon.

So please, for the sake of consistency, set your editor to expand tabs to spaces up to the next column that is a multiple of four (most decent editors should be able to do this). Those who claim to have a real problem with this in terms of readability will have to deal with it, i.e. either get used to it or take the time to program/script a solution.

7. General Recommendations

- Just because a certain code construct or statement is redundant in a specific case does not mean you should leave it out, even if the two forms are equivalent *after* compilation. In some cases, leaving out such redundant statements can greatly hinder source code readability. For a simple example, take variable initialization. Java automatically initializes fields (variables belonging to a class) to a default value of `0`, `false`, or `null`, depending on the type. Do *not* rely on other people (or yourself, some time later) to recognize this fact immediately! Another programmer reading the code may be confused by the fact that a variable is used apparently without being initialized, an error in many other languages. Worse still, if this hypothetical programmer doesn't see the code that relies on this implicit assignment, they may change the variable's value elsewhere, creating an incredibly elusive bug. In addition, programmers with relatively little Java experience may not always remember the exact initial assigned value for a field. With integers and objects this is usually easy to figure out, but the boolean value `false` is not at all intuitively obvious.

- "Magic Numbers:" It's generally considered bad practice to embed numbers other than 0, 1, 2, or -1 directly into a program's control flow, rather than as constants at the global, file, or class level. Other numbers (integers!) that are tied closely with the structure of the surrounding code block (for instance, as an increment for a step of a specific algorithm) are exempt from this rule *if* this use is documented with a comment briefly explaining the significance of the value. Just writing a statement like `increment = sensorLevel * 67` (example in Java) will leave anyone else who reads the code scratching their head and trying to figure

out just what on Earth the value 67 even *means* in this context. Even worse, if the value 67 is used at multiple points throughout the code, and tests determine that a value of 72 gives more accurate results, replacing every single occurrence of the value would be a pain. And no, find-and-replace is often not an option if the example were using a more common value, such as 4, which you really wouldn't want to replace throughout your project. It would be much better to first declare 67 as a constant at the top of the class: `public static final int sensorScaling = 67`, then using `increment = sensorLevel * sensorScaling`. This way, the semantics of the sensor scaling value are made a bit more clear (and can be made even more clear by a comment next to the constant). Also, changing the scaling to 72 is nearly trivial, and tuning this value experimentally is much easier than it would be in the previous example.

A.   Editor and IDE recommendations

This section lists some text editors and Integrated Development Environments (IDEs) that members of GO FIRST have found to be useful and productive for programming diverse languages on the different platforms.

A.1   Text editors by platform

Windows

As mentioned above, you want to move away from Notepad. It's not even good for editing configuration files. One alternative is Notepad++ (http://notepad-plus-plus.org/).

Mac

The built-in text editor is usable, but unsuited to programming. Luckily, quite a few cross-platform editors integrate well with OSX. You could also try TextWrangler (http://www.barebones.com/products/TextWrangler/), although I have personally never used it.

GNU/Linux

The built-in editors GEdit (GNOME) and Kate (KDE) are both good, extensible, full-featured, and usable for programming in many languages. However, it might be worth it to take a look at some of the cross-platform editors as well.

Cross-platform

There are two classic editors for Unix-like systems that have been extensively ported:

- Vim, the "improved version" of Vi (`http://www.vim.org`). Max's editor of choice; many people claim it's by far the most efficient editor they've ever used. However, Vim is notorious for its learning curve. If you have a free weekend and feel like learning to use the editor efficiently, check out the program "vimtutor" which should be installed along with the editor (try typing `:help tutor` after starting Vim). Features include everything you'd expect from a programmer's text editor (syntax highlighting, search/replace with regexes, code folding, and diff to name a few). Built-in support for most known programming languages, often with extensions available for specific languages. Originally designed for use in a terminal, although good graphical versions are available on the download page.

- (x)Emacs (`http://www.xemacs.org`), perhaps the more full-featured (according to some, bloated) of the two classic editors. Some people (such as Max; now you know where he stands in the "editor wars") find it unusable, although if you're a boss with chained keyboard shortcuts in Word or whatever you might like this editor. Lots of programmer's features and extensions available. It even has a built-in psychologist (try `M-x doctor` or something like that). The graphical version is xEmacs.

In addition, jEdit (`http://www.jedit.org/`) is a full-featured, cross-platform text editor with support for many programming languages. It appears to to work well for editing Java code.

A.2   IDEs by language

Java

- Eclipse (`http://www.eclipse.org/`) is a popular cross-platform IDE for Java and C++ development. It has all the features you'd expect from a standard IDE, plus a (somewhat clumsy) plugin installation system. It can integrate with version control systems such as Git (which we will most likely be using) through the plugin framework. It's a large piece of software – it's several hundred megabytes and slow to start up on any system. However, many Java (and C++) programmers think the rich feature set, which includes integration with a sort of task-tracking software, is worth it.

- NetBeans (`http://netbeans.org/`) is definitely more lightweight than Eclipse but specializes on Java. It also has a plugin system with a large database of user-contributed plugins (Git support is naturally among them). Some programmers might

find the relatively minimalistic approach of NetBeans more appealing than the one-stop-shop philosophy of Eclipse.

C/C++

- As mentioned above, Eclipse also works well with C++.

- If you're on a Mac, you could check out XCode (it costs $ 5 on the Mac App Store). However, it's frustrating to use for development not involving a shiny Mac application.

- On Windows, Visual Studio is the standard all-in-one IDE available for CSE students for free at http://cselabs.umn.edu/software/msdnaa. Although designed primarily for Windows development, it's usable for general C/C++ program development.

C# (If we do end up using it):

- Those with Windows should use the Microsoft Visual Studio IDE.

- Those who don't have Windows (or don't want to install it) can try the Mono development environment, see http://www.monodevelop.org/.

Python
Again, we might not end up using Python at all, but if we do, here are my IDE recommendations: First, it's perfectly possible to develop Python with nothing but a text editor and terminal (that's what I do, and I prefer it to using an IDE). However, if you're used to using IDEs and want to use one for Python, there is one available: IDLE should be installed with the default Python installation (Linux users might need to install an additional package). It's about as minimalistic as IDEs get; nevertheless, it does simplify and integrate the Python development process. Users new to Python or lacking a decent command-line interface might find this helpful.

General
If you like lightweight IDEs, try Geany (http://www.geany.org/). It's not tied to a specific language, which is both an advantage and a disadvantage depending on your personal preferences. In any case, it's got pretty much everything you need to edit and build most types of source code effectively.

Others
Search the web and use your best judgement :)