

Version Control and Programming Documentation Lesson

Ethan Ruffing

December 16, 2014

Contents

1 Lesson Overview	5
Order of Topics	5
Getting Started	5
Complete Examples	6
2 Basic Git Commands	7
Committing	7
Branching	7
Logs	8
3 Gitflow	9
4 Project Design	11
Folders	11
File Names	11
Documentation	11
License	12
5 Markdown	13
GitHub-Flavored Markdown	13
6 Commenting	15
Introduction to Docblock Commenting	16

7 Basic Style Conventions	19
Program Order	19
Paragraphing	19
Naming	21
Braces	21

Chapter 1

Lesson Overview

By Ethan Ruffing

Order of Topics

1. Getting started
2. Git and version control
3. Proper project design
4. Proper documentation of programs
5. Style guide usage
6. A complete example

Getting Started

1. Install git from the [git-scm website](#).
2. Install SourceTree from [its website](#).
3. Create account on [GitHub](#) or [Bitbucket](#). (We will discuss the pros/cons of each hosting service.)
4. Create folder for local repositories and move into it:

```
mkdir git  
cd git
```

5. Clone this lesson repository:

```
git clone https://github.com/ruffinge/RoboLesson.git
```

6. Open this file.

7. Begin working through tutorial [here](#).

- Note that you should not set a global `user.name` or `user.email` because of the shared nature of your account on the computer. Instead, when when committing, do the following:

```
git commit -m "Commit message." --author="Ethan Ruffing <ruffinge@gmail.com>"
```

Complete Examples

The following two repositories show projects which I have completed. They demonstrate proper use of all concepts which are covered in this lesson.

Chess Game (in Java): <https://bitbucket.org/eruffing/chess.git>

Arduino Benchmarking System (in C): <https://bitbucket.org/eruffing/arduinobenchmark.git>

Chapter 2

Basic Git Commands

by Ethan Ruffing

This page is meant to be a *very* brief overview of some basic commands in git. You **will** need to use more commands than are displayed here. There are numerous suitable references available in multiple locations online.

Committing

To make a commit:

```
git commit -m "Commit message" --author "Ethan Ruffing <ruffinge@gmail.com>"
```

Branching

To create and checkout a new branch:

```
git checkout -b new-branch old-branch
```

To merge another branch into the current one:

```
git merge to-merge --commit -m "Merge branch 'to-merge' into current-branch"
```

Logs

To display a basic log of commits:

```
git log
```

To display a log in a graph view (to see the branching that has been used):

```
git log --graph
```

To display a log of all commits in a graph view (to see the branching that has been used):

```
git log --graph --all
```

To display a simple graph model of all commits (useful for viewing an overview of branching and commit history):

```
git log --graph --all --oneline
```

or

```
git log --graph --abbrev-commit --decorate --date=relative --format=format:'%C(bold blue)%h%C(reset) -
```

or

```
git log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) -
```


Chapter 3

Gitflow

by Ethan Ruffing

Git flow is a workflow for branching in the Git version control system. It was originally proposed by Vincent Driessen on his blog at [nvie](#).

The original post documenting this workflow can be found [here](#), and a practical introduction [is provided](#) in the [Atlassian tutorials](#) on using Git.

Either of these are a good place to start for learning the basics of this workflow; I personally prefer the original post.

It is worth noting that SourceTree, Atlassian's version control client, has built-in support for gitflow, allowing you to use this workflow easily. It will take care of the steps needed to perform merges and deletions of branches, letting you think of the branching system at a higher level, and allowing you to focus more on the work of programming.

Chapter 4

Project Design

by Ethan Ruffing

Folders

- All files related to a project should be placed in a folder whose name corresponds to that project.
- Subfolders should be used where appropriate.
 - Example: a folder for documentation, one for sources, one for binaries

File Names

- Files should be named according to what they represent.
- File names should *never* start with a number (written numbers are fine, digits are not).

Documentation

README File

There should be several levels of program documentation. The first should be a `README` file located in the top level folder of the project. This document should provide a brief overview of the purpose of the program, its author(s), and pertinent licensing/copyright information. It should also give a brief overview of where to look for further documentation on various aspects of the program.

A good way to develop the `README` file is to use Markdown, a language developed to be easily interpreted for HTML and PDF conversion, as well as to be easily readable and nicely formatted in its source code. If using markdown, name the `README` file `README.md`. See `Markdown.md` for more information.

API Documentation

The output of automatically-generated API documentation should be located in an easily accessible folder near the root of the project (for example, in a subfolder named `doc`).

Changelog

It is a good idea to include a changelog, where major changes are listed by version number. This should be included in a file named `CHANGELOG`.

License

If using a license, be sure to include the appropriate files as required by the license (usually included in a file named `LICENSE`).

Chapter 5

Markdown

An overview by Ethan Ruffing

Markdown is designed to be a universal markup language. One of the primary goals of Markdown is that the source files should be readable as-is. That is, they should appear well-formatted and cleanly laid out in their plain text version, in addition to being easily interpreted for translation to other formats (such as HTML, LaTeX, PDF, etc.).

Although Markdown can be interpreted from any plain-text file, it is best practice to use the suffix `.md`.

GitHub-Flavored Markdown

The most common variant of Markdown is GitHub-Flavored Markdown (GFM), which adds several features to the list of recognized layouts. For example, there is the ability to add syntax-highlighted code blocks, like this:

```
static void main()
{
    printf("Hello, World!\n");
}
```

A good tutorial on GitHub-Flavored Markdown is provided at the [GitHub website](#).

Chapter 6

Commenting

By Ethan Ruffing

You should have **DETAILED** comments ***EVERYWHERE!!!***

This:

```
void
moveFunc(int ch1, int ch2, int ch4)
{
    vexMotorSet(motBackRight,  ch1 - ch2 + ch4);
    vexMotorSet(motFrontRight, -ch1 - ch2 + ch4);
    vexMotorSet(motBackLeft,   -ch1 - ch2 - ch4);
    vexMotorSet(motFrontLeft,  ch1 - ch2 - ch4);
}
void
setDriveMotors(int speed)
{
    vexMotorSet(motBackRight,speed);
    vexMotorSet(motFrontRight,speed);
    vexMotorSet(motBackLeft,speed);
    vexMotorSet(motFrontLeft,speed);
}
void  extenderControl(int btn5U,int btn5D){
    vexMotorSet(motExtender, 96 * (btn5U - btn5D));
}
```

IS APPALLING!

Introduction to Docblock Commenting

Docblock comments are a system used to create API documentation for functions that you write.

Before each function, a “docblock” is inserted to describe its purpose and the way that it can be called and used by other functions. These blocks can be parsed by documentation generators, such as [Doxygen](#), to create nicely-formatted documentation of an entire program.

Each docblock should begin with a full description, using *complete sentences*. Grammatical style should follow Strunk & White’s *The Elements of Style*.

Descriptions of parameters and returned values need not be full sentences.

See more in the style guide.

Example:

```
/**
 * This function moves the robot according to joystick input.
 *
 * @author Ethan Ruffing <ruffinge@gmail.com>
 * @since 2014-09-07
 *
 * @param[in] ch1
 *     The Vex remote joystick channel corresponding to x-axis motion
 * @param[in] ch2
 *     The Vex remote joystick channel corresponding to y-axis motion
 * @param[in] ch4
 *     The Vex remote joystick channel corresponding to z-axis rotation
 */
void moveFunc(int ch1, int ch2, int ch4)
{
    vexMotorSet(motBackRight,  ch1 - ch2 + ch4);
    vexMotorSet(motFrontRight, -ch1 - ch2 + ch4);
    vexMotorSet(motBackLeft,   -ch1 - ch2 - ch4);
    vexMotorSet(motFrontLeft,  ch1 - ch2 - ch4);
}

/**
 * This function sets all drive motors to a specified speed.
 *
 * @author Ethan Ruffing <ruffinge@gmail.com>
 * @since 2014-09-07
 *
 * @param[in] speed
 *     The speed to apply to the drive motors
 */
```



```
    */
void setDriveMotors(int speed)
{
    vexMotorSet(motBackRight, speed);
    vexMotorSet(motFrontRight, speed);
    vexMotorSet(motBackLeft, speed);
    vexMotorSet(motFrontLeft, speed);
}

/**
 * This function will calculate the square of a decimal number.
 *
 * @author Ethan Ruffing
 * @since 2014-12-14
 *
 * @param[in] a
 *     The number to square
 * @return
 *     The square of the specified number
 */
double square(double a)
{
    // Declare variables
    double b;

    // Calculate square of input value
    b = a * a;

    // Return calculated value
    return b;
}
```

Notice also the comments within each function in the example above. These are essential for keeping your program understandable to others.

Chapter 7

Basic Style Conventions

By Ethan Ruffing

Program Order

Your program should follow a logical order. All functions should be *declared* before the `main` function, but none should be *defined* until after the `main` function (in your case, treat the `vexAutonomous` and `vexOperator` functions as the `main` function).

ALL preprocessor statements should be at the beginning of the file, preceded *only* by the file's header comment block.

```
#include <stdio.h>
#include "vex.h"

#define PI 3.14159
#define BACKRIGHT kVexMotor_1
```

Paragraphing

There should *always* be a blank line between *all* functions:

```
void vexAutonomous()
{
    // Stuff
}
```

```
int doStuff()
{
    // Stuff
}
```

There should not be a return within a function declaration/definition line.

Bad:

```
void
moveFunc(int ch1, int ch2, int ch4)
{
    vexMotorSet(motBackRight,  ch1 - ch2 + ch4);
    vexMotorSet(motFrontRight, -ch1 - ch2 + ch4);
    vexMotorSet(motBackLeft,   -ch1 - ch2 - ch4);
    vexMotorSet(motFrontLeft,  ch1 - ch2 - ch4);
}
```

Good:

```
void moveFunc(int ch1, int ch2, int ch4)
{
    vexMotorSet(motBackRight,  ch1 - ch2 + ch4);
    vexMotorSet(motFrontRight, -ch1 - ch2 + ch4);
    vexMotorSet(motBackLeft,   -ch1 - ch2 - ch4);
    vexMotorSet(motFrontLeft,  ch1 - ch2 - ch4);
}
```

Groups of code that logically go together should be grouped, with a blank line in between:

```
static void main()
{
    printf("Hello, World!\n");
    printf("This is a test.\n");
    printf("I'm still testing.\n");

    // Declare variables
    char[] st = char[100];

    printf("Now I want some input:\n");
    gets(st);
}
```

Naming

Names of variables, functions, and macros should all be logical and easy to interpret. They should properly reflect their purpose.

In addition, they should all comply with the following conventions:

- Variables should always be named in `headlessCamelCase`.
- Functions should always be named in `headlessCamelCase`.
- Macros and other preprocessor definitions should always be in `SCREAMING_SNAKE_CASE`

Braces

The cause of many a war.

By convention in C programming, opening braces should follow the “next-line” (as opposed to “end-of-line”) style. This means an opening brace should be placed on its own line.

Bad (but the style I prefer):

```
static void main() {  
    // Stuff  
}
```

Good:

```
static void main()  
{  
    // Stuff  
}
```

Note, however, that this applies only to functions and the like. In the case of loops and conditionals, the “end-of-line” convention should be followed:

```
int i, b = 0;  
  
while (something) {  
    if (somethingElse) {  
        return;  
    }  
}
```

```
for (i = 0; i < 5; i++) {  
    b = !b;  
}
```