GVSU Computer Engineering Program

Review, Critique, and Proposal

Written By:	Joe Gibson	gibsjose@mail.gvsu.edu
Comments and Review By:	Jesse Millwood	millwooj@mail.gvsu.edu
	Kurt VonEhr	vonehrk@mail.gvsu.edu

Date: July 2015

Executive Summary

This document attempts a review and critique of the Computer Engineering program at Grand Valley State University in the form of a proposal. Both the strengths and weaknesses of the program will be discussed, and additional topics will be proposed.

The goal of this document is to provide feedback from a former student who believes that the program is in a position to greatly enhance an already favorable curriculum. By reviewing this document and taking its contents to heart, the university will be demonstrating its commitment to student feedback and its ability to adapt to changing educational and industrial landscapes.

Grand Valley State University

Contents

1 Introduction

This document describes proposed changes to the Computer Engineering undergraduate program at Grand Valley State University. These changes are suggested based on suggestions drawn from the inferences and experiences of a recent graduate of the program.

Specifically, this proposal will discuss a particular set of courses related to the areas of **Programming Skills** and **Circuit Design and Analysis**, which are the two pillars of Computer Engineering.

To that end, this proposal is intended to achieve four things:

1. To give perspective on the usefulness and practicality of the existing curriculum
2. To critique certain aspects of the current curriculum that require review and careful attention
3. To emphasize the importance of certain topics that are currently **not** part of the curriculum
4. To suggest a possible idealistic outline of a modified curriculum to better suit the current environment of computer engineering, including ideas for new courses or projects

Please note that this document conveys my personal opinion on the different aspects of the curriculum, and while I believe the vast majority of students, and even faculty, would agree on most of the points, there may be ideas and topics discussed or criticisms presented that do not align with the views of others or are lacking empirical support in one way or another.

That being said, this document is really intended to give my perspective as a former student in the program. I believe the engineering department values the opinions of both former and current students, who have a unique perspective on the content and impact of the courses in the current curriculum. This particular perspective puts us in a spot to give relevant and significant feedback on the program and the ways in which it could be improved.

2 Existing curriculum

As it stands, the following Electrical Engineering and Computer Science courses are required for Computer Engineering graduates, and are the courses focused on **Programming Skills** and **Circuit Design and Analysis**:

2.1 Engineering Courses:

1. EGR 261 - Structured Programming in C
2. EGR 214 - Circuit Analysis I
3. EGR 226 - Introduction to Digital Systems
4. EGR 280 - Probability and Signal Analysis
5. EGR 314 - Circuit Analysis II
6. EGR 315 - Electronic Circuits I
7. EGR 326 - Embedded System Design

2.2 Computer Science Courses:

1. CIS 162 - Computer Science I (Java)
2. CIS 163 - Computer Science II (Java)
3. CIS 263 - Data Structures and Algorithms
4. CIS 350 - Introduction to Software Engineering
5. CIS 361 - System Programming
6. CIS 452 - Operating Systems Concepts

2.3 Computer Engineering Electives:

1. EGR 424 - Design of Microcontroller Applications
2. EGR 426 - Integrated Circuit System Design
3. CIS 457 - Data Communications
4. CIS 451 - Computer Architecture

3 Benefits of Existing Curriculum

There are clearly benefits to the existing curriculum, and the vast majority of the courses are well suited and in the proper order.

For example, the progression from **EGR 214** to **EGR 314** and **EGR 315** is logical and smooth, and the latter half of the CE curriculum, including **CIS 452**, **CIS 457**, and **EGR 424** is both challenging and incredibly insightful.

3.1 Preparation and Practicality

In the end, the major benefit of Grand Valley's engineering program is the raw practicality of the degree. I truly believe that I received a better engineering education at GVSU than I would have at even larger schools, including the University of Michigan. Our program is incredibly strong when it comes to getting students in a position to contribute in the workforce. While we may lag behind in research, our graduates feel more comfortable using multimeters, breadboarding circuits, and creating products from scratch. This puts us ahead of the curve in many ways, and it is clear that employers are recognizing this with full force.

3.2 Faculty Dedication and Knowledge

The faculty is another strong aspect of the program. Rather than having courses taught by graduate students, the professors are dedicated to their classes and their students, and are extremely helpful both in and out of class. Many of the faculty members here are very accomplished in their field and they bring that knowledge and dedication to their classes. Labs are often well defined by professors, and include substantial practical training that supplements the course lectures.

3.3 Cooperative Education Program

Finally, one of the greatest benefits of GVSU's engineering program is the Cooperative Education program. There is no doubt that our co-op program is both unique and effective at preparing students to succeed after graduation. More than just an opportunity for networking and a steady paycheck, students learn valuable technical and social skills during their co-op experience, and many discover the direction they do (or do *not*) want to take with their career once they graduate. The magnitude of the benefit given by the co-op program is difficult to measure, and is certainly one of the strongest areas of the entire engineering program.

3.4 Looking Further

While there is much more I could say about the strengths of the great program we have here at GVSU, the main focus of this document is intended to present both a review of changes that could be made as well as a supplimental look at what **other** things we can be doing to prepare our students even more, and to ensure that the education they receive, as it specifically relates to computer engineering, is as comprehensive and useful as it can possibly be.

4 Downfalls of Existing Curriculum

4.1 Computer Engineering: A "Frankensteining" of Computer Science and Electrical Engineering

The field of computer engineering is historically a combination of electrical engineering and computer science, and while the term "Frankensteining" may have a negative connotation to some, in some ways it is a benefit; students participate in two related fields, getting the best of both worlds. The CE program is an example of this, in which we receive vast electrical engineering education as well as important computer science education.

That being said, the computer engineering program has grown in recent years, and it may be time to reevaluate the curriculum to more appropriately address the needs of computer engineers directly, rather than continuing to do so vicariously through the EE and CS departments. Computer engineering is a vastly growing field; it has separated itself from being simply a combination of electrical engineering and computer science, and the computer engineering program at Grand Valley should reflect that.

This is not to say that any *major* program revisions are necessary, as clearly CEs should be taking the majority of their courses in the CS and EE disciplines, but rather that certain courses and course content which have been historically used at GVSU as part of the CE curriculum are not well suited for CE students, and perhaps specific CE courses should be created which are similar in the end goal of a course (such as teaching object oriented programming), but differ in their approach (such as doing so in C++ rather than Java), as will be described below.

4.2 Java I and II

Object Oriented Programming is without a doubt a fundamental aspect of computer science and thus computer engineering. To graduate with a degree in Computer Engineering and not have a basic understanding of OOP is not a favorable achievement.

That being said, Java, the choice of language in our introductory OOP courses (CIS 162 and 163), is not well suited for Computer Engineering students. While it is a useful language, the domain for Java development can be limited to specific areas and specific companies. For example, Java development is highly concentrated in areas like Android development, Enterprise systems, web development, and databases. All of these areas are important, but for Computer Engineers, contrary to Computer Scientists, these areas align less with both the rest of our curriculum and potential job opportunities. Given that CEs do not take any courses in mobile development, databases, or web development, the use of Java after CIS 162 and CIS 163 for CEs is precisely nill, save for a very limited number of local companies that may teach it during co-op.

Given the time spent learning OOP, it occurs that instead of teaching this concept in a language seldom used thereafter by CEs, a much more logical approach would be to teach OOP using C++.

In addition to the benefits of C++ over Java as an educational language, namely the advancement of the concepts of pointers, memory management, and other important aspects which are 'hidden' by Java, C++ actually *is* used in subsequent CE courses:

1. CIS 263 attempts a very quick and limited introduction to C++, while the main focus of the course is Data Structures and Algorithms. Depending on the professor, students may have little experience actually *coding* in C++ during the course, and many students complain about the lack of any real introduction to the language.

2. C++ can greatly simplify projects in courses like CIS 457 and CIS 452, where access to data structures such as maps, trees, sets, and even simply class-based organization itself are less of a luxury and more of a necessity.

3. Even on a microcontroller, C++ can be useful. For example, I completed my EGR 326 project using C++ rather than C on an ATmega328P, and found this experience to be both technically rewarding as well as a more efficient use of my coding time.

4. Looking back at all of the projects I completed in my upper-level CS courses, nearly all of them were in C++. A few of my project members and I were fortunate enough to have learned C++ either on our co-ops or on our own accord, but we had no formal training in C++, which could have served us well when it came to these senior courses. To be clear, these projects were not "C++ projects"; rather, C++ was an option given to those students who happened to know it.

Returning to the benefits of C++ as an educational tool for OOP, C++ offers more in the way of intellectual obligation than Java, and C++ is arguably a more logical progression from C. Although both languages are both based off C and yet very far from C in many ways, C++ shares, in addition to many more, the following attributes with it's predecessor:

1. The development environment for C++ is typically identical to that of C. With an install of `gcc`, the C++ compiler `g++` is already available and is nearly identical in use and function as the corresponding C compiler.

2. Memory management, and specifically the existence of pointers is a key aspect of both C and C++, whereas Java shields the programmer from both the struggle and *understanding* of pointers.

3. The syntax of C++ is much closer to C than Java is to C. Take, as a very simple and admittedly contrived example, the following three declarations:

```
char array[256];     //C
```

```
char array[256];     //C++
```

```
char[] array = new char[256];    //Java
```

   Clearly, in this simple scenario, Java's syntax is very much different than C, and can even be confusing (i.e. where to place the `[ ]`), which brings us to the next point:

4. C++ supports inline C. This means that when developing in C++, you could write your entire program in C and compile it with a C++ compiler, and the program will function the same as if you had done it with `gcc`. In fact, in the previous example, the array declaration **was** just plain C.

All of these attributes make C++ a more logical progression for the foundational OOP Computer Engineering course(s) than using Java.

C++ is also more widely used than Java in the field of computer **engineering**. I will not attempt an argument at programming language popularity in general (see statistics at (`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`), but clearly Java's use in Android development is a vast porportion of it's popularity, and it is clear that C and C++ are the most widely used languages in computer engineering. In my personal experience, through all three of my co-ops I used C once on a piece of avionics software, and C++ on both of the other two: once on a sub-orbital balloon satellite at NASA and again on a large particle physics project at CERN. At both NASA and CERN, C++ was by far the most prevalent language used by the engineers outside of web developers.

4.3 System Programming

CIS 361 is split into two distinct halves:

1. System programming in C in a Linux environment
2. Shell programming in a Linux environment

Both of these topics are extremely important and must be part of the CE curriculum.

However, the problem lies in the way the first half is typically instructed (at no fault to the instructor); since CIS 361 is offered as an elective for CS students (it is required for CEs), the first half of the course inevitably begins as a **"Java to C"** course, rather than a course on system programming.

Since CS students have little to no introduction to C, they often struggle, and thus the instructor must start with a **very** basic introduction to C: how to declare variables, how arrays work, what a string is in C, and an overview of pointers, which ultimately baffles many CS students who have never encountered the topic before. Thus, for two weeks the CS students struggle to catch up, while the CE students do their best to remain attentive while they review the most basic of C topics. Clearly there exists a better solution.

To be sure, the latter half of the course is both useful and engaging for both CS and CE students, but more on scripting will be discussed in the next section, **Extracurricular Topics**.

4.4 Hardware Design Skills, or, "Arduino Dependence"

EGR 214, 226, 314, 315, and 316 offer a broad range of both analog and digital circuit design and analysis concepts. Since CEs are often more concerned with the digital side of circuit design, so too will be this suggested improvement to the existing courses.

Out of the 30+ EE and CE students (a number which increases annualy), by the senior year, few if any have real, practical PCB design skills. To be sure, PCB design topics are *discussed* in EGR 326, and some courses even require designing small boards or 'shields', but the reliance of students

on the **Arduino** platform mitigates any opportunity for real practical experience when it comes to learning the ins and outs of PCB design. Mention the words 'gerber files', and many EE and CE students will cringe, or even worse, return a blank stare.

Additionally, the lack of preparation for students when it comes to board design leads professors to be reluctant towards surface mount components and non-breadboardable chips. These types of packages are not only the most common in any real circuit design, but surface mount soldering techniques, including the use of solder paste, are a basic element of being an electrical or computer engineer.

I feel there is a significant opportunity for students to improve their PCB design skills by foregoing the Arduino platform and having students design their **own** development board, based around the ATmega328P (the same microcontroller found on the Arduino), sometime during their sophomore or junior year.

For example, during EGR 214 or EGR 226, students could design the schematic and layout their own ATmega328P-based development board to use in EGR 226 and other courses (EGR 326). In all truth, designing a development board around the ATmega328P is very simple. To get a basic functional environment, the following components are necessary:

1. An ATmega328P
2. A couple of capacitors
3. A couple of resistors
4. A few LEDs
5. A 5V regulator (Like an LM7805)
6. A reset button
7. A fuse
8. A barrel connector
9. A six-pin ICSP header for programming the chip

And optionally:

1. An external oscillator (with two caps)
2. A USB connector
3. An FTDI chip to program it over USB instead of ICSP
4. Tx and Rx LEDs

This could even be breadboarded with minimal complications as a pre-lab, and then converted into a schematic and ultimately manufactured. To save time, instructors could prepare their own dev board schematics and have many extra PCBs printed, both so students can reference their designs as well as in case students are unable to complete the board design themselves, so they are prepared for future courses which will use the board (of course, you don't have to tell the students that there is a plan B if they don't finish...).

In addition to hardware experience, this would introduce students to open source software such as `avrdude` and `avr-gcc`, rather than allowing them to use the Arduino IDE and it's vast libraries

which, although helpful, do not facilitate learning how things work 'behind the scenes'. Students would relinquish their reliance on the Arduino IDE and the Arduino library environment, and instead it would force them to code in a traditional text-editor environment, use command line tools like makefiles and debuggers, program their microcontroller using the ICSP interface and `avrdude` directly, and communicate with the board using open source command line UART programs like `minicom`, all of which are incredibly valuable skills to have.

A logical next step for this would be when EEs and CEs take EGR 326. In EGR 326 we learn about regulator design, how to protect digital I/O, and in general how to make your embedded design more robust. What better scenario could there be to make a **rev 2** of the student's development board, in which they re-design the regulator (maybe use two LM317s in parallel with ballast resistors or something), implement safety features like using zener diodes and resettable fuses on digital pins, and learn the importance of flyback diodes and other useful components.

Additionally, EGR 326 does scratch the surface of *theoretical* board layout, spotlighting the importance of, for example, putting your filtering caps as close as you can to certain pins, or ensuring matched impedence on high-speed transmission lines. This could all be practically expressed during the design of the **rev 2** board, wherein students apply these techniques to build a much more robust development board; think 'Ruggeduino'. This would be the main project throughout EGR 326, culminating in the manufacturing of their board, which could be used in a very **simple** project at the end of the semester. As it were, the EGR 326 project we completed was heavily comprised of topics that were not the main focus of EGR 326, such as infrared diodes and mechanical design including very heavy reliance on 3D printing. Although these things are important, the design of a robust development board aligns more closely with the content of EGR 326, and in the end is something that students can be proud of and keep using on different projects long after they have graduated.

> **Side note**: Perhaps it makes sense to use an Arduino for EGR 226 and then graduate to their own development board in EGR 326, but in any case it is extremely important to have a practical experience with board design and digital systems.

These finished boards do much more than simply serve as a development board for them in future courses; the board will be a source of pride and individuality for students, who may even become excited about board design and find that they quite enjoy it.

Even more paramount is the ability for students to showcase their board during interviews. They will have at least one board (possibly a second revision) to physically hand prospective employers, while they explain the steps they took to design and enhance the board, and could even show the documentation they have prepared surrounding the board, such as schematics and calculations. This gives students an opportunity to empirically show that they can design a printed circuit board, and indeed a complete embedded system, to a specification and that they have improved upon it with documentation and critical thinking.

In addition to the hardware aspect, there exists an opportunity for students to interact with full-featured, open source software, and for partnerships to be made with alternative board manufacturers.

The fully open-source schematic and PCB design tool KiCad (www.kicad-pcb.org) is produced by a collaboration at CERN (the European Center for Nuclear Research), and is constantly undergoing bug fixes and feature enhancements. The software is fully cross-platform, with builds for OS X, Windows, and Linux. There are nightly builds and ongoing development on the tool, which is used around the globe due to it's 100% free open source commitment; widely available documentation and support community; and features like push-and-shove routing, which are traditionally found on only extremely expensive industry tools.

> **Side note**: KiCad is due for an official 'stable' release within the next couple of months. The current builds are 'stable' in the sense that they function well, but an official stable build is coming soon.

OSH Park (www.oshpark.com) is a community PCB manufacturing platform. They are based in the US (in Oregon) and offer both 2-layer and 4-layer boards for incredible prices. They charge just **$5 per square inch** for 2-layer boards, and they are guaranteed to arrive within two weeks. In addition, the price always includes **three** copies of the board, not one. We have used OSH Park for many of our projects and have had very good service, including being upgraded to a much faster turnaround time and shipping due to being repeat customers.

Their community model is based on the premise that they will fill up an entire panel with boards from different orders, and then print and ship the boards when a panel is full. Thus, there exists an opportunity for the SoE to collaborate with OSH Park for printing the boards of all students in any given course, who's boards would likely fill up a significant portion of a panel, increasing the turnaround time for board production.

Regardless of whether KiCad or Eagle is chosen for the design tool, and regardless of the board manufacturer, the ability for both EE and CE students to design schematics and physical boards is a **fundamental area of knowledge and skills**, and separates purely theoretically-focused engineers from those who have applicable skills desired by industry, research, and major technology companies alike.

4.5 Computer Engineering Electives

The current offering for CE electives basically boils down to the following options:

1. EGR 424 - Design of Microcontroller Applications
2. EGR 426 - Integrated Circuit System Design
3. CIS 457 - Data Communications
4. CIS 451 - Computer Architecture
5. One or two EE or biomedical courses

Of the four CS/CE electives, I would argue that two are in fact **fundamental** topics, and should be **required** by all CE students. These two courses are **EGR 424** and **CIS 457**.

EGR 424 is a very fundamental look at how a microcontroller functions and the intricate relationship between the code, the compiler, the linker, and the target architecture. **If any class**

is truly 'computer engineering', it is EGR 424. In addition to being arguably the most fundamental of all CE courses, it is also a wonderful companion to courses like CIS 452, which *is* a required course. Having taken EGR 424 before CIS 452, there were a generous helping of topics which were made absolutely more clear by having actually implemented a multi-threaded kernel in C.

CIS 457 introduces the concepts of packet-switched networking; network programming (sockets, etc.); the internet and TCP/IP; and a basic introduction to network security. These concepts are also fundamental to computer engineering, and are without a doubt the most well connected to the growing presence of internet-connected devices and security concerns today. The field of computer engineering has been moving towards connected devices and the 'internet of things' for quite some time, and a basic understanding of computer networking is the foundation of this movement.

To receive a degree in Computer Engineering without EGR 424 and CIS 457 is, in my opinion, a huge setback, and the consequence is a lack of preparation necessary to succeed in the field.

There are also some EE electives, namely the new **Embedded Systems Interface** course by Dr. Bossemeyer and Dr. Jiao that would be of great value to CEs as well as EEs.

4.6 Software Engineering

CIS 350 is intended to be a course in software engineering, which demonstrates the lifecycle of software projects and different project management schemes. This is an important area of computer engineering, and is a necessary part of the curriculum. However, in practicality, CIS 350 fails to prepare students for any meaningful concept of software engineering and the project management process. Due mainly to a lack of any practical examples or hands-on learning, CIS 350 is merely an apathetic theoretical discussion of Scrum and Agile development with a forced SWS component. The end result is that students are left wondering what, if anything, they received from the course, and why it is even a course at all.

A major improvement to CIS 350 would be to begin the course with a fake project proposal/set of specifications by a fake company, and to follow the development of that software project throughout the semester with practical examples as the different topics are introduced and discussed. The entire course could be engaged in demonstrating the benefits of certain aspects of software engineering in a **practical** manner, as it applies to a specific project example.

A theoretical look at software engineering is not only boring and not intellectually stimulating, but due to the number of different tools and methods available, the instructor can, at best, give a very limited view of any given methodology. Thus, I would argue that it would better suit the students to take a much broader view of software engineering and discuss only a select number of topics in greater detail. To complement this, real examples of UML charts, state machine diagrams, etc. would be created by the professor and students in concert along the way, and the instructor could address the different ways of surmounting a large software project with real, physical examples.

5 Extracurricular Topics

There are a vast number of topics that *could* be covered in a CE curriculum, but due to staffing and financial constraints, it is understandable that only a small subset of these extracurricular topics can be implemented into a program (probably as electives).

However, the following topics, not necessarily courses of their own, are highly valuable and are in growing demand for today's computer engineering graduates:

1. Computer security
2. Connected devices
3. Scripting
4. Linux development
5. Version control systems

5.1 Computer Security

Computer security is a field that is growing immensely, and there should be a focus at GVSU to take advantage of an emerging field.

For CS students, there is an option to take CIS 458, which gives at least an overview of security and modern cryptography concepts. In an ideal world, there would instead by two courses: Network and System Security, and Cryptography. Both of these courses would be offered to CS and CE students.

Additionally, for CE students there could also exist an FPGA Cryptography course, since FPGAs are often used in encryption/decryption schemes due to their high-performance computing abilities. I believe Dr. Parikh has done research in this area.

My job after graduation will be in the field of computer security. While I did gain some knowledge from CIS 457 (Data Communications), which briefly discusses it, I learned quite a bit of what I know from open-source online courses from other universities, and almost all of my knowledge will be gained on the job. This is an example of where (a) course(s) in computer security could prove to benefit many students in their job search, giving them a definite advantage over other students who lack such knowledge.

I know many students are very interested in the field, as it is one of the more provocative areas of computer science. There is no doubt that CIS 458 fills up quickly whenever it is offered by Dr. Kalafut.

5.2 Connected Devices

> **Side note**: EGR 436 - Embedded Systems Interface already exists and covers many of these points, but it is not available for CEs to take as an elective as of writing this document.

The so-called 'Internet-of-Things' is also an emerging field, and one that is easily intermingled with existing topics, such as embedded system design. Such a course in connected devices could build on the advanced embedded system design class, and projects could use bluetooth or WiFi connectivity by way of designing small boards to interface with an existing development board, or using existing development boards that include wireless interfaces (both Atmel and TI have many).

5.3 Scripting

Scripting is an important aspect of computer science and engineering, and is a basic skill that should be held by anyone graduating with such a degree. For example, Python is a highly useful language with applications ranging from web design to data processing. Python is incredibly easy to learn, and is very powerful for certain areas of application. Perhaps this would not be suited for it's own course, but Python scripting could be included in courses like CIS 361.

5.4 Linux Development

There is a focus on Linux development in CIS 361, CIS 452, and CIS 457. However, there should be a focus on open-source Linux development in many more areas of the CE curriculum. The benefits of open source tools and software are becoming more clear as time goes on, and a computer engineer without a solid knowledge of Linux is not a computer engineer that is highly desired by many companies. An emphasis on using open source tools and an emphasis on command line development is essential for the appreciation of large IDEs that take much of the education out of programming. While IDEs are very useful and will surely be used once engineers graduate and obtain jobs, the whole point of an IDE is that it can be easily learned and companies don't expect prospective employees to be fluent in their exact IDE or toolchain environment; what they do expect is that engineers are knowledgeable in the fundamentals of the coding process and environment, and developing using traditional text editors and command line tools is the only way to obtain such knowledge.

An additional enhancement in terms of Linux development would be to create a `gvsu-egr` package repository that could be installed easily using any Linux package manager: on Ubuntu and Debian systems using `apt-get install gvsu-egr` or other systems with tools like `yum` or `zypper`. This would allow professors to add software required for their class to the central repository and then students could easily install and upgrade this software without wasting time on a complex installation. While there is educational merit in learning to install tools on a Linux system, it is likely true that student's time is better spent learning to use the tools rather than debugging their faulty installation.

5.5 Version Control Systems

The education and use of version control systems, like `git`, in my opinion, **may be the most important of all the points made in this entire proposal**. Version control systems are so foundational to software development and software engineering that they are used by nearly all creditable companies worldwide.

Not only that, but version control offers incalculable benefits to **students**:

1. Students using version control will not mysteriously lose their projects; it is less chaos for both students *and* professors.
2. Students using version control are able to more quickly recover from mistakes made during the development process by rolling back changes to their code or taking advantage of branching schemes.
3. Version control systems make **team** development *possible*; After using `git` for team projects, it is extremely difficult to imagine doing so without it.
4. The popular `git` repository hosting website, `GitHub` (www.github.com) is now viewed by many companies not as a *suppliment* to a traditional resume, but rather as a nearly-obligatory resource through which they can view a candidate's work and really see if they produce quality code.

All of these benefits of version control systems, and more specifically of `git`, one of the most popular and easiest to use, make it an absolutely **essential** aspect of a modern computer science or computer engineering degree. I truly cannot imagine working on a team or any sort of coding project now without `git`, and my `GitHub` account link is the only other hyperlink on my resume other than my own personal website. `git` and `GitHub`, along with any version control system, are so prolific and unavoidable in today's software world that they simply **must** be a part of any engineering student's education.

Version control systems are a perfect fit for courses like CIS 350 - Software Engineering, if taught properly. The only trouble with `git` (and any version control software) is that if learned incorrectly, or not learned thoroughly enough, students can make mistakes that affect their entire project team. With version control, however, these mistakes are easily recovered from, and the entire history of their project is backed up in well-defined stages.

`git` is easy to learn and has a **ton** of wonderful documentation behind it (check out `https://help.github.com/`). This interactive website (`https://try.github.io/levels/1/challenges/1`) is a fantastic example of how easy it is to get started using `git`.

It is not only for the benefit of students that they use version control systems; professors will also benefit greatly. Instead of a chaotic mess of emails with 40 files named `helloworld.c` in their downloads folder, professors can simply receive a link to a student's repository. Even submitting their code can become an easy process. If students take advantage of `git` and `GitHub`'s very simple release functionality, submitting an assignment can be as simple as running `git tag -a g̈ibson-v1.0¨`, which automatically creates a zip folder of the student's project that they can download and submit. For group projects, professors can even see detailed statistics and graphs of who is contributing what to any given project, if they so choose to do so. To avoid plagiarism between students in classes, projects could be hosted on other sites such as Gitlab (www.gitlab.com) or Bitbucket (www.bitbucket.org), both of which offer free private repositories.

I propose that instructors in CS and CE courses actively promote using `git` and using version control schemes such as this incredibly powerful and easy to use branching model. (`http://nvie.com/posts/a-successful-git-branching-model/`). In the end, using `git` will benefit

both students and instructors and will result in every student who graduates from GVSU having their own personal portfolio of the code they have written readily available to show off to their potential employers.

> **Side note**: Everyone has their own favorite version control system, and while `git` is certainly one of the more popular options, others great options do exist. However, to keep things consistent, a single option should be chosen. `git` is a perfect choice for this scenario, because it is open source, distributed, very easy to learn, has a mountain of great documentation behind it, and is exclusively used by the most popular public repository site, GitHub.