Software Engineering in the Mariokart System

Wim Looman
wgl18@uclive.ac.nz

Coauthors: Simon Richards, Zachary Taylor and Henry Jenkins
scr52@uclive.ac.nz zjt14@uclive.ac.nz hvj10@uclive.ac.nz

Supervisor: Dr. Andrew Bainbridge-Smith
andrew.bainbridge-smith@canterbury.ac.nz

Department of Electrical and Computer Engineering
University of Canterbury
Christchurch, New Zealand

***Abstract*—Something amazing about engineering a software system.**

I. Introduction

A. Software Engineering

Since this report is aimed at an engineering audience most of you will believe that a description of Software Engineering is not really required. Unfortunately true Software Engineering is relatively unknown, especially in programming courses run in Electrical departments around the world. That is not to say that Computer Science departments do a better job of teaching it, in fact Software Engineering really should be taught as a subset of Engineering [1], just that the style of programming taught to Electrical students is generally light on following the engineering practices that the rest of their courses rely on.

So, what is Software Engineering? It is simply the application of standard Engineering practice to the development of software. However because of the nature of software as a much more fluid abstract thing than the normal circuits designed by Electrical Engineers the precise method of application has to be changed.

At the same time as being more abstract than a circuit software is also much more concrete; there are no (or at least very few) annoying real world effects directly on the software. Assuming the circuit a microprocessor is in has been designed well the Software Engineer can take it for granted that the digital I/O used by something like a Inter-Integrated Circuit (I^2C) is basically a perfect connection straight to the internals of another device. Internally if there are no weird defects in the microcontroller you can assume that a function like:

```
int return_three() {
    int three = 3;
    return three;
}
```

will always return exactly 3. Not 2 when the batteries start running low, not 4 when it is a particularly hot day, always exactly 3.

This exactness of software enables the use of a few techniques that are not normally available in most engineering professions. For example it is possible to perform exhaustive testing and/or modelling of the system within acceptable time.

The major components of software engineering that will be discussed in this report are: version control, unit testing and continuous integration. Version control is probably the aspect of software engineering that is best applied by current engineers, however most still use an old system such as Subversion despite their being much better alternatives like Git and Mercurial available. Unit testing is a developmental practice that has been seeing a major increase in use for traditional software in recent years. This is largely because of development processes such as Test-Driven and Behaviour-Driven Development evangelised by the Agile Software Development proponents. Continuous integration is a major aspect of these newer software development methods where quality control is continuously applied to the system while under development, normally utilising unit testing as the main quality assurance system.

A lot of this is standard practice in software development shops. Unfortunately despite the large amount of code written by other engineering disciplines the same level of engineering practice they apply in designing their circuit board, concrete floor or ethanol extractor doesn't get applied to the code they develop in pursuit of these goals. This is most relevant to embedded development where the entire range of software engineering practices can be applied; some take a bit more effort because of the lower abstraction level, but they are all applicable in some way. Parts of this are also relevant for the other engineering disciplines, Matlab may not be a real programming language, but when developing simulations in it proper software engineering practices should still be followed.

B. Mariokart

The system on which this report will base most of the examples was codenamed Mariokart. This was a final year project for the University of Canterbury's Bachelor of Engineering degree carried out by the authors. The overall goal of the project was to take one of the electric go-karts the department had and retrofit a drive-by-wire system on to it with an overall goal of having the kart autonomously drive around the campus. For the purposes of this report the main details of the system developed are:

- The overall design is a distributed system with 5 boards:
  - One for communication with a host laptop.
  - One for steering.
  - One for brakes.
  - One to interface to the motor controller.
  - One for collecting data from a variety of sensors.
- Each board is running an Atmel SAM7XC microprocessor.
- Communication between boards is carried over a Controller Area Network (CAN) bus.

For more details see *Embedded Hardware Design For Autonomous Electric Vehicle* by Henry Jenkins [2].

II. Software Engineering

A. Version Control

Out of all Software Engineering practices Version Control is definitely the most widely used by normal engineers. Unfortunately it is still not used everywhere. For example, you have just finished looking at a new power regulation system that you want to switch to for your next PCB revision; now you need to write a quick report on why it is so much better than the current system that you should spend all this time changing your design. What's the first thing you should do, open a new Microsoft Word document? Load up your LaTeX editor? *No*, you should initialise a new repository or ensure you have the projects documentation repository available and updated. Anything and everything that is more than a few lines long, or will ever be shared with a team member should be under version control.

This is very important for a multitude of reasons. Firstly it provides you with a time line of development activity. If you need to revisit a decision months later you can identify exactly when the initial review was made and when any revisions happened.

Secondly it provides you with a safety net. The following anecdote is a very good example of why this safety net is important.

> “A younger programmer asked an elder about his code and his coding style, and how the older programmer would do certain things. The older programmer said ‘Let's take a look at your code’, so the younger took out his laptop, opened his editor, and showed him.
>
> The older programmer looked at the code, thought about it for a bit, and then started editing it. He deleted the class internals, leaving only the structure, and then rearranged the structure, saying ‘Here's how I would do it to make it more efficient and readable’. After he was done, he saved the file and gave it back to the younger programmer, who was ashen-faced.
>
> ‘That... My code is gone!’ said the younger programmer. ‘But you have it in version control somewhere, right?’ asked the elder. ‘N.... no.’ was the reply. ‘Well then,’ said the older, ‘now you've learned two lessons.’”
>
> — *Dan Udey* [3].

Having this safety net is also a very good incentive to experiment. As mentioned in the anecdote the older programmer, assuming that the younger was using version control, felt free to delete most of the code and rearrange what remained. If the younger had been using version control then they could have simply committed this major change on a separate branch and checked out their prior work to compare the two.

B. Unit Testing

As mentioned earlier Unit Testing is currently seeing a major increase in use in forward-thinking software development companies, mainly because of evangelical Agile development proponents. Unfortunately despite this increase for traditional software development, the uptake in embedded development projects has been a lot slower.

The basic premise of unit testing is that if you verify that all parts of your system work as intended, then the system as a whole will work as intended. To do the verify you write a lot of small *unit* tests to verify minimal sections of your code. By ensuring that all code you write is tested by multiple unit tests you can be confident that the code performs as you expect.

Of course this isn't the same thing as being right, unit tests only verify that the code does what the test says it should. To ensure the code does what it should do you need to validate your tests. This is most commonly done informally, the test developers know the intended outcome and the tests with the codes purpose in mind. For more critical systems an external validation can be performed using a method such as modelling. Our attempts at modelling one of the critical sections of our system can be read about in *Safety by Design for the Mariokart System* by Simon Richards [4].

References

[1] D. L. Parnas, “Software engineering programs are not computer science programs,” *IEEE Software*, vol. 16, no. 6, pp. 19–30, 1999. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=805469

[2] H. V. Jenkins, "Embedded hardware design for autonomous electric vehicle," 2011. [Online]. Available: https://raw.github.com/team-ramrod/mariokart/master/Documentation/ScientificReport/Henry/report.pdf

[3] Dan Udey, 2008. [Online]. Available: http://stackoverflow.com/questions/132520/good-excuses-not-to-use-version-control/135002#135002

[4] S. Richards, "Safety by design for the mariokart system," 2011. [Online]. Available: https://raw.github.com/team-ramrod/mariokart/master/Documentation/ScientificReport/Simon/report.pdf