Software development plan - Electronics Mälardalen University Solar Team

Erik Kamph*,
School of Innovation, Design and Engineering,
Mälardalens University, Västerås, Sweden
Email: *ekh17001@student.mdh.se

I. Introduction

Whether you are new joining the team or already in the team, this is a plan describing how the software development process is done. It should give a brief introduction to git and version control, as well as give a plan for the version control in Solar Team. The document should also describe the development standard to some point and give a link to the development standard described. This is to be able to read more detailed information about the standard.

A. Changelog

1) Initial version
2) Version 2:
   - Doxygen documentation
   - Doxygen and epytext bug documentation
   - Link to doxygen preset settings
   - Testing standard for C was changed
   - Use-Case diagram was added under design standard

II. Git introduction

If you know Git, you can skip ahead to Section III (the next section), otherwise continue reading. Git is a program for version management in repositories whether local or online. If typing ”man git” into a terminal window you’ll probably see that it is described as a fast, scalable, distributed revision control system[1]. What it does is to keep track of any changes in the local or online distributed version control system. When it comes to daily usage, there are several commands in daily use, these are

- status - checks the current status of the repository
- pull - download an update of the online repository
- push - send the latest commit to the online repository
- commit - commit any changes in the local repository
- merge - Check the code for conflicts and choose how to merge the files
- remove - Removes a file from the revision control system
- init - Initializes a folder for revision control
- branch - switches to a specified branch that exists
- restore - undo changes

There also exists a cheat sheet that explains a lot of the commands that the git application has, the link to the cheat sheet is here.

III. Version Control Plan

The version control plan will include answers to following questions, as well as figures to help explaining.

a) Where and how should everyone work?
b) How and when to create and manage branches?
c) How and when to create a pull request?
d) How often should a push happen?

A. Where and how should everyone work?

The plan is that every member in the electronics domain should work in the same GitHub repository. There will be a set structure that will be linked to the branching system in the repository. The structure is found in figure 2 and will be explained below together with the gitflow found in figure 1.

1) Structure and branching: The repository root is the master branch, from where every subsystem branch will be created with its respective working folder. This folder and branch can be seen as the root and master branch for your subsystem. Each subsystem will hold two folders, one for software and one for hardware, if needed add more folders. Those folders do not need to be branched, as those are just to separate the software from the hardware configuration files if any exists. Then inside the software, each feature implemented should have its own folder as seen in figure 2 and branch which is represented by the green dot in figure 1. A feature branch will exist for each feature until its implemented, where you will have the responsibility of creating a pull request for a repository manager to review and accept.

B. How and when to create and manage branches?

Branches is created with ”git branch [name]”, where name is the name of the branch. For example if the branch is a feature called ”communication” in vehicle to vehicle communication, then you would run ”git branch communication”. To switch to a specific branch you will then have to run ”git checkout [name]” where name can be ”communication”. There is also an alternative command which both creates and switches to the branch, that is ”git checkout -b [name]” where the name could be ”communication” if that is what you are working on.

C. How and when to create a pull request?

Each time a branch is at the end of its lifetime, in other words you are done with a feature, you need to make sure that the branch is available online. To create a pull request

for a merge of two branches, you need to head to the repository online and go to ”Pull Requests”, and click ”New Pull Request” as seen in figure 3. From there you’ll have to select your branch and your feature branch that you want to merge, this will create a request to review the code. If needed, more information about pull requests can be found at pull-requests.

D. How often should a push happen?

A push shall happen when

- you create a branch to make sure it is available online,
- developing a feature, one or more times to create checkpoints in-case you need to roll back,
- a feature is fully implemented, tested and you know that it is working before creating a pull request,
- a configuration file or any other file is added to the subsystem branch.

IV. Software Development Standard

This section will cover software development and standards such as code style, design, testing and documentation. We will also cover when to create designs because some designs is not a must. Also we will cover the testing, both unit, each function by itself, but also component testing, all functions together as one component.

A. Code style standard

When it comes to standards for coding in different languages, there are a lot depending on what company and where you are looking. In our case for the C programming style we are looking to develop using the NASA C coding style found at ntrs.nasa.gov. However if we are to program in Python we shall follow their PEP-8 standard which can be found at python.org. Both of the standards covers coding style, comments, naming conventions and other things.

B. Design Standard

Recommended software for diagrams under this subsection is draw.io or lucidchart.com, both have support for flow chart diagrams and class diagrams. Standards for both types of diagrams are described below. Besides the standards described, we will point out that whatever you are using it shall use GitHub as you will place your file under your subsystem that you are developing.

1) Flow Chart Diagrams: There is no exact standard, but most developers give tips on how to create these diagrams. Following are recommended rules for either a flow chart diagram that uses ”swimlanes” or a flow chart diagram that goes from left-to-right[2]:

- Orientation of arrows and diagram(left-to-right or top-to-bottom)
- Begin with start and leave end out from the diagram until the diagram is finished
- Think of the color scheme, use at max 3 colors, if possible less.
- Using different shapes adds more information and makes it easier to read
- Have a name on the node and put all extra details and information needed to understand in a side note instead.

2) Class Diagrams: If applicable and if there is too many different files with classes and functions in a sub-directory, it will be necessary to make a class diagram. This diagram should hold all the classes in that sub-directory, as well as functions, variables, dependencies, etc. This is done in order to make it easier for another person to get an overview of whats in those directories. A guide for making class diagrams can be found at uml-diagrams.org, it describes all the different elements a class diagram can have, as well as how they are used. When creating a class diagram it is important to understand the different relationships, association, aggregation, composition and generalization in order to draw a correct class diagram.

C. Testing standard

This section covers standards for testing, as well as describes how to test single units e.g. functions and so on individually. It will also cover component testing, which is all of the functions together as one single unit e.g. all the functions developed during several feature implementations. A subsystem can be seen as a unit with multiple units inside. For testing we will follow the ”IEEE Standard for Software Unit Testing” [3], while documentation should follow the ”IEEE Standard for Software Test Documentation”[4] described under section IV-D ”Documentation standard”. The purpose of the standard is to help with planning, while it is not linked to any computer software for implementation of the unit tests. The standard can also be applied to any digital computer software or firmware either being developed or that is developed already. There are two types of tests that needs to be planned and executed, these are unit tests and component tests described below. Each test shall focus on one and only one method from the feature implementation. Furthermore when testing, there should be at least two people testing. The reason is that we can see the code from different eyes and fix small problems that might make a huge difference.

1) Unit testing: Unit testing shall focus on single functions, and each test case shall only focus with inputs on the function specified in the name. Inputs and outputs need to be selected in advance with the help of planning the testing from [3]. Unit tests can look different for different types of languages, Java has JUnit, C has Cunit, while Python has its ”unittest”-framework. A unit test function for python is visible in appendix D. Though that unit test is more of a component test than unit test, in other words it’s a test of functions working together. In appendix E we focus more on one function which is what unit testing is for, testing single methods or functions. A guide for writing unit tests for python is found at docs.python.org. Documentation for how to write unit tests in C can be found at cunit.sourceforge.net if using CUnit, other frameworks for C can be found at stackoverflow.com. The last

option is to use assert from the header file ”assert.h” which is standard and included in C.

2) Component testing: Component testing as seen under appendix D works as unit testing and are written in the same way as unit testing. However the scope of testing is different than unit testing. The scope of unit testing was to have a test function for each function implemented, while the scope for component testing is one unit test for each subsystem in this project. That means that each test will try how all functions inside one subsystem work together forming a component, therefore the name component testing.

D. Documentation standard

The documentation standard will describe standards for manuals, documentation of test results and documentation of known bugs. It will finally describe how to create a summary of all of these documents that are produced. The summary in general will include parts of documentation and manuals and discuss them.

1) Manuals: A general documentation standard for manuals will follow the ”IEEE Standard for Software User Documentation” which presents minimum requirements for structure, information content and format of user documentation. It provides this both for paper written documentation as well as electronically written documentations. According to the purpose of the standard ”The revised standard will address the interests of software acquirers, producers, and users in standards for consistent, complete, accurate, and usable documentation.”[5]

2) Documentation of test results: Documentation of test result should follow the ”IEEE Standard for Software Test Documentation”. The standard describes a style for documenting the tests and it is describing a test plan as well as an test incident report, both of which according to the standard can be used for code reviews. Those reports can also be used for design. Its applicability is not restricted by size, complexity or criticality of the produced software and therefore can be applied on anything. We are going to apply this on the documentation of testing for the software we product.

3) Documentation of known bugs: The table below holds the most important information for each bug that is found. Each bug found has to be documented and if fixed, describe the fix that solved the issue as well as what version that fixed it. The reason for this is in-case a similar bug is found later, we can easily fix that bug and solve the problem if not permanently, then temporarily.

Bug ID	A unique identifier for the bug detected.
Description	A short description about the bug
Status	The current state of the bug within the software.
Version Detected	What version of the software was it detected in
Version Fixed	If applicable include what version of the software it was fixed in
Fix Description	Give a fix description as well as a location of the fix in case any further problems occur
Severity	The highest failure impact that the bug have, as determined by the owner of the software
Type	What type does this bug fall into? E.g. Communication, Security or other

Table I
A STANDARD FOR HOW TO DESCRIBE A BUG THAT HAS BEEN DETECTED IN THE SOFTWARE WRITTEN, THIS STANDARD IS PRODUCED WITH THE HELP OF ”IEEE STANDARD CLASSIFICATION FOR SOFTWARE ANOMALIES”[6].

Each bug that we find in the software should follow the table design above and should be kept in a single file of bugs. The reason for this is that we can easily fix it or know if it is something that will happen during execution of the software. This file is for a subsystem and will be placed in same directory as your subsystem. It will also be included as a section as seen in figure 4. If the bug has been found inside a certain function, document it in the code as seen in an example in section IV-D4 for C and section IV-D5 for Python documentation. Don’t forget to not mix up details that describes the function with the bug command that holds the bug description.

4) Bug documentation in C: ’

```
/*!
 * \fn <function name>
 * \details <detailed description>
 * \bug <bug description>
 * \par Detected: <version detected>
 * \par Fixed: <version fixed>
 * \attention Severity: <severity>
 * \attention Type: <type>
 */
```

5) Bug documentation in Python:

```
"""
Description:
<short or long function description>

Bug description:
<bug description>

Version Detected: <version>
Version Fixed: <version>
Severity: <severity>
Type: <type>
"""
```

6) Summary: The summary comes after the table of contents in the big document according to figure 4. It should include following content:

a) System hardware breakdown structure: The breakdown structure should include sections of hardware existing in the subsystem. This should also be followed by a table describing extra parts such as screws, cables etc. and their

quantity. This is because any other person who reads the document shall easily see what is needed in order to swap component in the subsystem without problems.

b) Configuration: An overview of the configuration that is required for hardware. An example can be BLE, where you have to specify how to setup the devices in order to get a way of communicating data between the two devices.

c) Security: Describes mostly what security features are in use in the software, what it prevents.

d) Design overview: Any flowcharts, class diagrams or other design of the system that has been produced also needs to be in the summary.

Finally the summary should be at most two to three pages covering all of those four items. The reason for this is to make a quick introduction to the subsystem you are reading about and what you need if you need to fix anything.

Appendix A
Gitflow

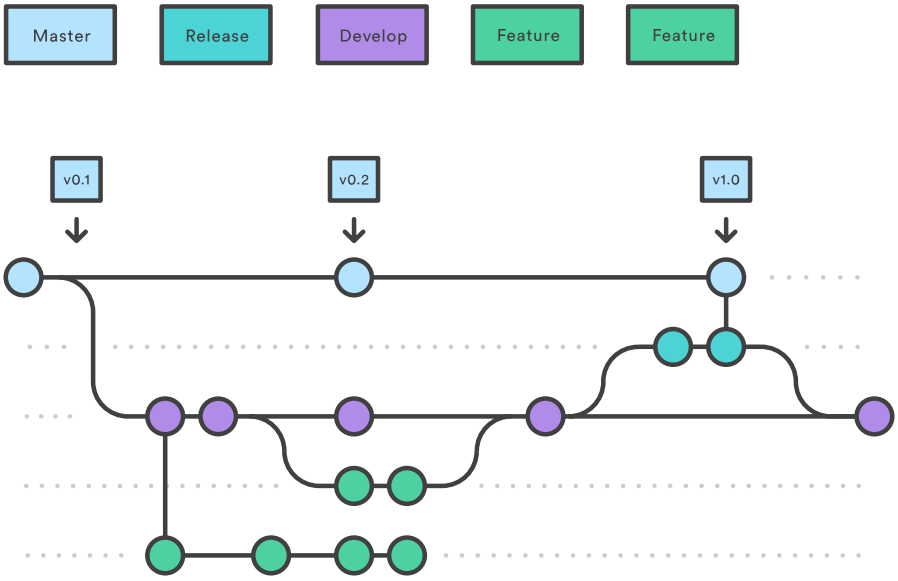

Figure 1. A figure showing versions, development branches, feature branches and releases on the different branches. It is used to explain the workflow and version control plan.

Appendix B
Structure

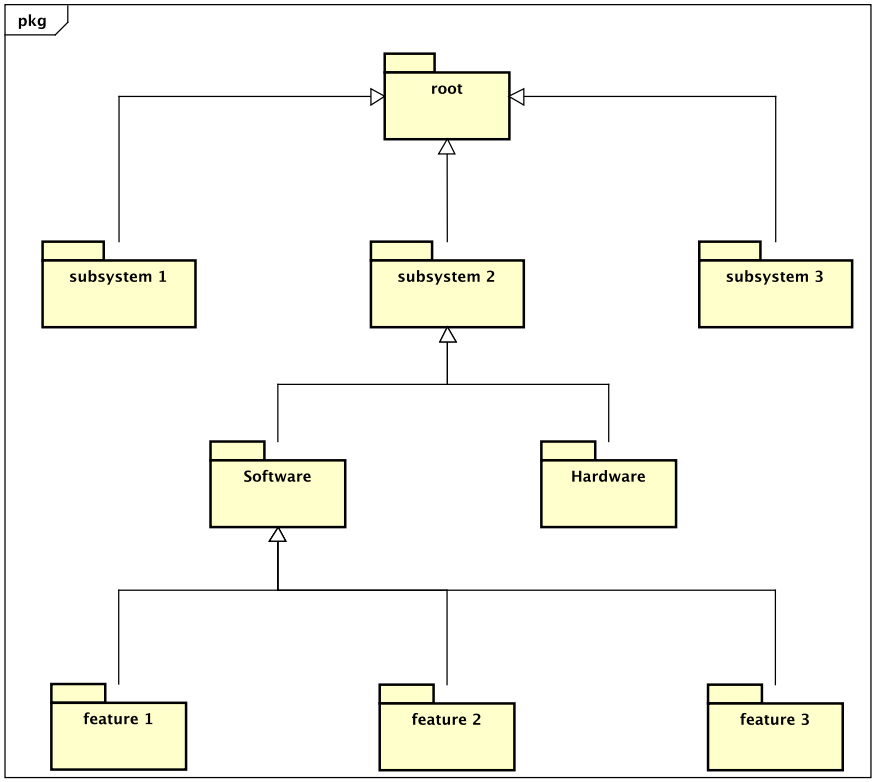

Figure 2. Recommended folder structure for git repository

APPENDIX C
PULL REQUEST - GITHUB

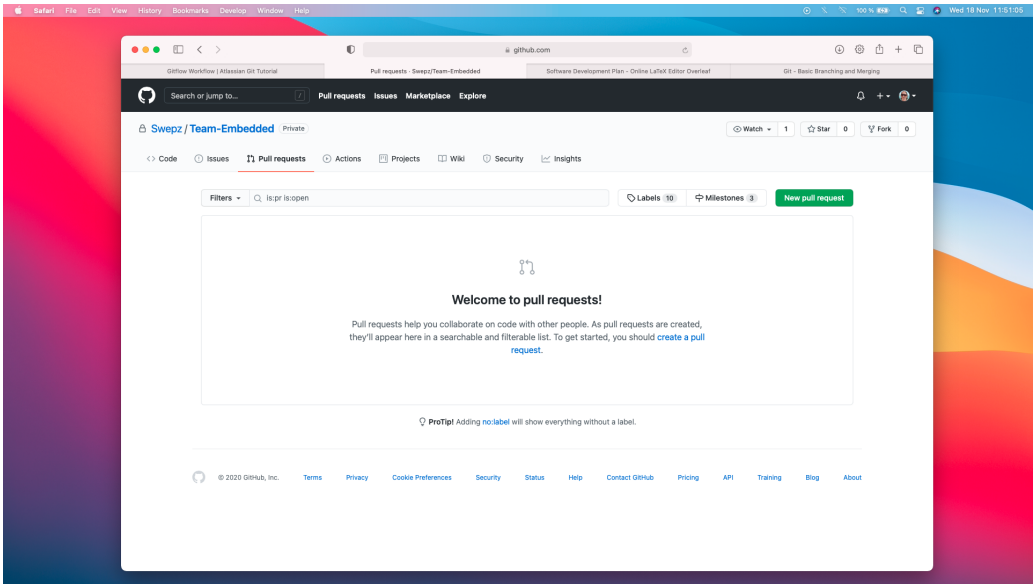

Figure 3. Pull request in a repository

APPENDIX D
PYTHON UNIT TEST - COMPONENT

```
def test_accuracy(self):
    self.test_forwardpropagation()
    self.test_find_index()
    df = pd.read_csv('assignment5.csv', nrows=1)
    net = nw.Neutral_Network(df, None, None, 48, 0.0001)
    target = df.values[0][0]
    values = dv.values[0][1:]
    output = net.forwardpropagation(nw.normalize(values))
    index = np.where(np.max(output) == output)[0][0]
    expected_accuracy = sum([1 if i == v else 0 for i, v in
            zip([index], [target])]) / len(df.values)
    accuracy = net.accuracy(df.values)
    self.assertTrue(expected_accuracy == accuracy,
            "The accuracy function is working!")
```

APPENDIX E
PYTHON UNIT TEST - SINGLE FUNCTION

```
def test_normalize(self):
    df = pd.read_csv('assignment5.csv', nrows=1)
    normal = nw.normalize(df.values)
    true = df.values / 255
    self.assertMatrixEquality(true[0], normal[0],
        "The normalization function doesnot normalize)
```

APPENDIX F
REPORT DOCUMENT OUTLINE

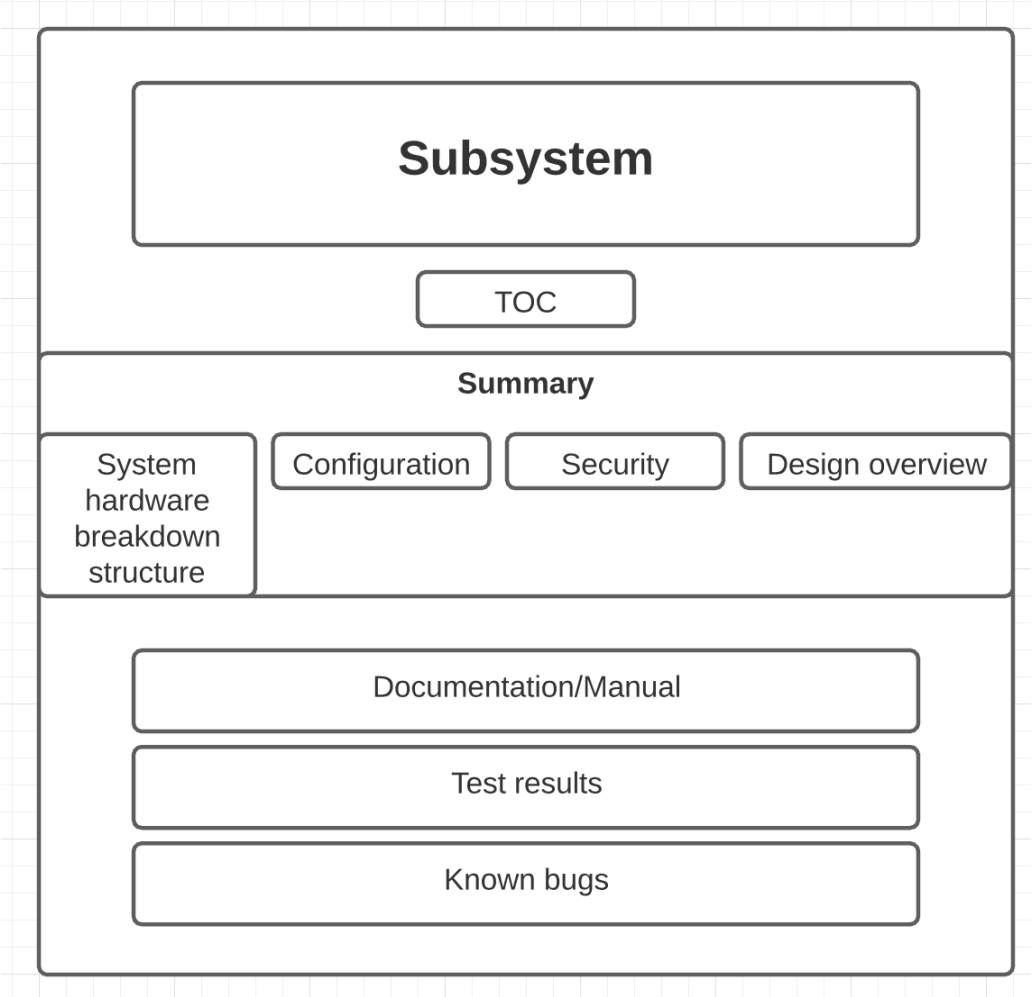

Figure 4. This is a document outline, showing how the completed document should look like. From top to bottom it is Subsystem name, Table of Contents, Summary, Documentation, Test results and lastly known bugs.

REFERENCES

[1] L. Torvalds and J. C. Hamano. (2020, Nov.) Ubuntu manpage: git - the stupid content tracker. [Online]. Available: http://manpages.ubuntu.com/manpages/precise/en/man1/git.1.html
[2] A. Mecham. (2020, Nov.) How to design a flowchart. [Online]. Available: https://www.lucidchart.com/blog/how-to-design-a-flowchart
[3] IEEE, "Ieee standard for software unit testing," *ANSI/IEEE Std 1008-1987*, pp. 1–28, 1986.
[4] ——, "Ieee standard for software test documentation," *1*, pp. 1–48, 1983.
[5] ——, "Ieee standard for software user documentat1ion," *IEEE Std 1063-2001*, pp. 1–24, 2001.
[6] ——, "Ieee standard classification for software anomalies," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23, 2010.