Mbuel Capstone LabVIEW Automated Test Refactor

Morris E. Buel

Rev 03/21/2018

Table of Contents

Summary

This project dealt with improving the LabVIEW based test system that was deployed to help with assisted manual test. The program communicates to the device under test, and automatically captures readings from bench top equipment and the device under test. The production floor was not satisfied with the reliability and from the prior projectmany of the problems in the software were not being captured through the companies bug tracking system. They were just working around the broken system by testing boards with manual test. After doing a code review it was revealed the prior project code-base was literally spaghetti code. (This is the nature of an unplanned LabVIEW project) Prior to project completion it took about a month just to get the existing system stable where the production floor felt more comfortable using it. To address these issues and help increase productivity in the department the code was refactored to make it more maintainable, extensible and reliable.

The new system was designed from the ground up with a software development plan. The This has helped create a stable code-base. There is now unit testing implemented so bug fixes cause no new bugs. The completed project now has an easy way to add new products as the system is now built to be modular and extensible from the ground up. When adding most new products to the system, it can be completed in less than a day with no changes to the code-base.

The success of this refactor project has dramatically increased productivity in the production department, as the existing tests are completed faster andmore reliably. The ability to add tests quicker has also had a measurable affect on productivity. The test department met their productivity goals over a three month period for the first time in measured history. The more robust construction also makes it easier to adapt when needed. If a product does require a test case that does not currently exist, it can be added within about a day or two including unit and

product testing. After completion that addition can be used with all future tests.

The refactored LabVIEW implementation had a few problems that came up, dealt with during the project. After the second sprint work was completed on the most problematic VI in the code base it was released to Production while I continued working on the next VI to refactor. During this time there were a few high severity bugs caught that did not have unit tests to catch them. To ensure that production was not line down a solution was come up with to deal with the transition slightly differently. The company decided that to avoid stoppages to production we should switch to a parallel deployment. That until the new product met the stability requirements, they would still have the old application available for use. This was set up for them, and work continued on the last sprint with the added coverage of fixing those high severity bugs. After release there were still a few bugs to burn down, and unit tests added to catch those problems – until the product got to less than one bug reported on average per week. The switch over to the new system at that point was completed, and the old system removed from the network.

Review of Other Work

To meet production demands, it is important to test boards in a timely manner. The current project did not and this was improved to improve the production process. The current application is now more useful in moving us toward improving this metric and is no longer laden with project problems that need to be overcome. To overcome the problem the project entered a refactor project which helped refocus the prior project to meet the original goals of that were missed in the prior project. In their video on refactoring LabVIEW code, LabVIEW advantage (2016) stated; “the refactor does the same thing in a much more savvy way.” The goal of the refactor should not be to rewrite unless necessary, it should be to do the same thing in a way that is easier to read and takes up less code space on screen. It will provide the same functionality in less screen space. This helped drive the project to meeting the goals of being more maintainable, reliable and extensible. (LabVIEW Advantage, 2016)

The application's no longer a complex web of interleaved code screens even with over 450 VI connected to the project. While this was daunting, through a process of decomposition, complex items were broken down into smaller and more workable pieces of code. This is true also with the LabVIEW development environment. In their brief at *dev days* in 2011 National Instruments presented a document on dealing with inheriting and maintaining LabVIEW code. There were some great takeaways in this lecture, including the goal of any refactor should be; "more readable and maintainable so that cost does not increase over time." This metric was tracked in the outcome, by keeping track of the number of reported bugs per week in the bug reporting system the company uses. They also discuss an important distinction that was made during the first sprint. Sometimes the existing code-base was too far from the desired result. When the existing VI did not meet the required functionality, the developers decided to; "change the internal structure of a VI to make it more readable and maintainable." This of course required a system to make sure the rewritten and refactored VI meet the required functionality. (National Instruments, 2011)

In order to make the refactor successful it was important to understand why projects can fail, or why they are released and then fail to meet the original requirements. According to the article put out by ALE System Integration (2008) there are four reasons why bad code happens. Those reasons; "1. Novice programmer 2. Rushed Development 3. Prototype became final application 4. Experimenting of new algorithms or design patterns." (2008) any of these can cause a project deployment to fail, this project had all four of them as the root cause. It was created by a new developer, it's development was rushed, the prototype shown off to management became the final product and there was a lot of experimentation in the code-base with new algorithms and design patterns. This could have been an overwhelming burden to climb to bring this project to a maintainable, reliable and extensible state, but through the agile development process, by taking the functional patterns of the software and building on them, to release code that met the original design specifications. The article also has an excellent point on when to rewrite and when to refactor. If the VI works but; "just needs a feature added to a VI"(2008) then a refactor is appropriate. If the VI; "does not function"(2008) then it should be rewritten. Using this criteria helped during the first sprint when the schedule was laid out for the rest of the project. This method was used while documenting the existing code, to determine whether the VI met the functional requirements of that VI, or did not function as it should. (Terry Stratoudakis, PE, 2008)

To help ensure this project was a success where the first project failed was to properly implement test driven development. To really get to the heart of what we aimed for with test driven development, Roy Osherove through His book *the art of unit testing* (2009) covers this topic in great detail. The inclusion of unit testing alone did not mean this project would succeed. If the tests were poorly designed, implemented or notated they also could have created code that is not reliable, not maintainable and not extensible. According to Osherove when discussing inheriting code with poorly written tests (2009); "the tests were so brittle that any little change in our code broke them!". To accomplish this it's important that the unit tests were also documented, and kept simple. Just as with any other language, the entire function (VI) should be visible on one screen without any scrolling. They should have clear variable names, that make it clear what is being done, which also makes it clear what is being tested. Test driven development as with standard development, should aim to be reliable, extensible and maintainable. Every unit test was written with this idea and laid out to be reusable elsewhere, Osherove (2009); "If you don't devise ways to reuse parts of your tests…. If you don't you'll end up with test code that's either not maintainable or hard to understand." In order to accomplish this goal, we made sure that tests are as automated as possible within the IDE. The goal of this project of a more maintainable base, was accomplished through this process.

To make sure we used the JKI unit testing tools correctly, the team decided to watch a training video online. Unit testing requires code blocks to be decomposed into functional units of work. This was done during the second sprint as the first release, caused problems in production. Delivery of code, LabVIEW Architects Forum (2015) "is not just writing some code, there has to be documentation, testing, buiding, functional testing, reviewed and approved by internal and external customers." To make sure future releases did not cause production to go line down during the project, the release was shifted to parallel deployment. This left the old system on the network while the new system was being worked on. After the new system met all of the internal customer's goals, the project was considered complete.

During the refactor of one of the central VI it was determined that data abstraction was necessary. To help determine the best way to accomplish this, research was done on using the class objects inside of LabVIEW. It was determined that a class object provided a better way to pass all of the data related to a test between VI, instead of having an individual wire for each data point. The class object was constructed representing the test data as it passed through the system. Similar to other object oriented programming languages, classes can have abstract, private and public member variables. Since with private variables Tomi P. Maila (2007) "only methods that belong to the class can access this data" it was determined that all the variables within this holder class would be declared as public variables.

It was also determined during the project that the front panel (the main user interface) needed to be rewritten as the existing form did not meet the project requirements and was causing issues on the production floor. While researching solutions it was revealed the best replacement was a third party drop in state machine created by JKI. This “JKI state machine” is event driven instead of queue driven, this means that no matter what event was processed before, you can process every event next. The prior state machine could only process certain events in the queue structure. Ideally when a JKI (2008) “button is pressed it’ll actually execute those two states in sequence.” which makes every state in the program more responsive, along with increasing maintainability, stability and reliability.

Changes to the Project Environment

The current environment still uses LabVIEW which is a GUI driven IDE. This has an upside of allowing those who have little to no programming experience to get the prior project up and running quickly. The downside was it allowed those with little to no programming experience to get the prior project up and running quickly. The prior project had no focus on proper programming techniques it allowed the project to quickly and literally turn into spaghetti code.

This project environment made it easy to implement refactoring on a class by class basis. The nature of LabVIEW made each VI (or class in traditional programming) polymorphic by nature. Each VI has to function on it’s own, if it can’t nothing else will work. This is where tools were searched for to replace what was built inefficiently and where necessary replaced with off the shelf solution(s). To do this we used the VI package manager which is a free tool maintained by JKI(2015) software. This tool allows you to easily add third party VI(s) (classes) that accomplish goals that were commonly required, that the NI tool chain doesn’t do, or doesn’t do as well.

The development environment was improved through the following things:

1) Implemented unit testing. JKI has a unit testing package that can be installed from the package manager. To start with, unit testing was only completed on refactored or added packages.

2) Ensured the project was worked on from the local drive, and was executable and able to compile from that local drive. The prior project was run and compiled from diverse network locations creating dependency problems when build or run attempts were made.

3) Made sure the project is properly inserted into subversion nightly. The prior project was not in subversion, so if someone else was eventually added to the project there would have been problems. This also helped during development, when a mistake was made we could rollback to a working release.

4) The production environment now works from the companies internal auto updater. Previously they had to manually copy updates to each employee’s production machine, taking more of their time away from testing and delivering product.

These changes improved the development environment which made it easier to implement the improvements.

Methodology

This project used the agile methodology with sprints to focus on a VI then release the code to production. This allowed for test driven iterative deployment that was also implemented by adding unit testing to refactored VI. At the beginning of a sprint it was decided what would be worked on by the weight of it's difficulty, assuming that one man hour can get a difficulty total of 40 done per week. This difficulty was determined by the development team, voting on the cards. A sprint included the specified labor on those cards, to finish the work required and at the end of that sprint a usable product was ready for the production floor to use - that functionally works the same but is now more reliable.

A second advantage of the agile methodology was the quicker feedback from the internal customers on the production floor. After every deployment we were be able to use the bug tracking system to make sure bugs are actually decreasing with each iterative release. The first release had more problems than anticipated causing a shift in the release structure from transparent to parallel with cut over.

Sprint 1: Moved the project structure to the local working drive, uploaded the current version to source control, then installed the unit test platform for LabVIEW. (JKI Unit Test) Reviewed the bug list to focus on the most problematic section(s) of code to refactor first.

Sprint 2: After documenting the VI with the most bugs, work began on the refactor and rewrite (where necessary) of that the VI that was voted to be in sprint 2. The end result are VI that meet the required results with greater reliability, maintainability and extensibility.

Sprint 3: This was the most difficult sprint, as there was also priority one bugs that had to be completed during this sprint. It also did require six more VI to be refactored and rewritten. This sprint should have been broken down into another sprint, as the amount of labor was greater than the time available. was Through the plan however, this sprint did build upon what was learned during sprints 1 and 2, with unit testing and refactoring and rewriting where necessary. The second, third and fourth refactor or rewrite were faster than the first.

Project Goals and Objectives

	Goal	**Supporting objectives**	**Deliverables enabling the project objectives**	Met/Unmet
1	Review existing code to document what is being done – get codebase in SVN	1.a. Document existing code-base	1.a.i. Reviewed bugs to determine which Sub VI should be focused on first	Met
			1.a.ii. Documented functionality and what is trying to be accomplished in that VI.	Met
			1.a.iii. Created the schedule for phase 2 depending on the complexity and amount of bugs in each VI	Met
		1.b. Upload current code-base to subversion	1.b.i. Copied the existing code-base to the local working directory, make sure it runs and compiles without errors.	Met
			1.b.ii. Uploaded codebase to subversion.	Met
2	Refactor selected VI, Unit Test those VI release new code-base to Production at end of sprint	2.a. Begin refactor work on the VI selected in Phase 1.	2.a.i. Work schedule of what work needs to be done	Met
			2.a.ii. Verified refactored VI that functionally work the same way.	Met
		2.b. Unit test the	2.b.i. Built VI to test	Met

		rewritten, and refactored VI	refactored and rewritten VI.	
			2.b.ii. Delivered tested code to Production, get their feedback.	Met
		2.c Build schedule for last sprint	2.c.i Working schedule for the last sprint to deliver the final product.	Met
3	Impleme nt extensibl e code base	3.a work through schedule created in last sprint	3.a.i Refactored VI case structure, where modularity will be inserted. (rewritten VI)	Met
			3.a.ii Unit Tests on refactored and rewritten VI – all VI pass.	Met
		3.b ability to add tests from outside without any code changes.	3.b.i Ability to create a new test from outside the IDE	Met
			3.b.ii Ability to tun through simple test to verify that a test can be added without touching the code-base.	Met

The goal of this project was to refactor and/or rewrite the existing test environment in LabVIEW to make it more maintainable, reliable and extensible. To accomplish this goal, the goals were decomposed into smaller units of work, and accomplished by working through the project timeline and adapting it where necessary. This started in the first sprint by gathering up cards representing all of the existing bugs in the project. The scrum team then scored each one using a Fibonacci sequence number, where 21 represents the highest difficulty, and 1 represents the lowest difficulty. The reporting customer reported the severity on a scale of 1 to 4, with 1 being the most sever, and 4 being the least severe. After completing this the team had the cards sorted by priority and difficulty. The next deliverable was documenting the functionality of the highest priority card from the prior objective. This required a code review to meet and discuss the required functionality for that VI, and determine if the existing code met that requirement. The deliverable from this was documentation on the functionality for the proposed VI to work on.

The last deliverable for sprint 1.a was to create the list of VI that will be worked on for sprint 2, with the appropriate documentation coupled to that requirement. To make sure that work went without a hitch, the last objective for sprint 1 was to copy all the code to the local drive and upload it to the SVN server. This was done by first copying the code-base to the local drive, then making sure it compiled and ran. Once that was addressed, the code-base was uploaded to the SVN server to make sure there was a roll back available if something went wrong with the project.

The second sprint started the work on the VI to refactor. It was decided to focus primarily on one, and try to do another easy one if time allowed. The decided upon VI was the base serial driver. The prior serial driver VI had six different iterations, violating the rules for maintainable, and extensible and reliable code. The first thing the team did was to build a stand alone serial application with LabVIEW with unit testing, to determine what was needed in the code. This encapsulation and polymorphism led to more reliable code. After it passed Unit Testing, it was implemented to replace all six iterations of the prior serial driver with the new serial application. This took up most of the time, the product was compiled, function tested then delivered to production. After completing the unit testing and delivering the product, the schedule was laid out for the last sprint.

During the last sprint, it was revealed that the delivered product was not working as reliably as it needed to. It was determined after a quick scrum to switch the deployment to a parallel deployment, keeping the old version available until the new version met requirements. Before starting the remainder of the work, the high priority fly in was completed and testing added from what was learned in Production. After it was completed work continued on the refactor project. It took the remainder of the two weeks to finish the necessary work to allow creating tests outside of the LabVIEW development environment. The final deliverable was met, a test was added without touching the code-base at all.

The project did end up taking longer than the projected schedule which will be covered in the timeline. In order to meet the requirements set up by the parallel deployment shift, it took two more weeks to drive down bugs to release the stable code-base. The reason for this success was the agile methodology is open to changes where necessary. It created a flexible framework allowing for the necessary adjustments to the release and the schedule.

Project Timeline

Milestone or deliverable	Planned Duration (hours or days)	Actual Duration (hours or days)	Projected start date	Anticipated end date	Actual Start Date	Actual end date
Pre-Sprint 1 meeting to sort bugs by severity and difficulty	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Move project to local drive	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Verify Project still runs from local drive with no dependency errors	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Build project from local drive, verify no dependency errors	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Upload project to subversion	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Install JKI Unit Test platform	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Create schedule for Sprint 2	1 hour	1 hour	3/6/17	3/6/17	3/6/17	3/6/17
Run through training examples for successful unit	4 days	4 days	3/7/17	3/10/17	3/7/17	3/10/17

testing in LabVIEW with JKI (YouTube)						
Start sprint 2- start work on first VI from schedule.	2 days	2 days	3/13/17	3/14/17	3/13/17	3/14/17
Run testing on first VI and any created Sub VI	1 day	1 day	3/15/17	3/15/17	3/15/17	3/15/17
Refactor next VI that can be done in 1 day of labor, with unit testing	1 day	1 day	3/16/17	3/16/17	3/16/17	3/16/17
Release Production code, determine impact	1 day	1 day	3/17/17	3/17/17	3/17/17	3/17/17
Start sprint 3 – refactor next VI in schedule that will drive toward modularity. While developing, create and run unit tests in parallel.	2 days	2 days	3/20/17	3/21/17	3/20/17	3/21/17
Release code to production	1 day	1 day	3/22/17	3/22/17	3/22/17	3/22/17
Determine if sprint 2 release has any show stoppers that need worked on, if not work on next VI and it's unit tests	2 days	2 days	3/23/17	3/24/17	3/23/17	3/24/17
Work on high priority bugs from Sprint 2 (inserted)	1 day	1 day			3/27/17	3/27/17
Review prior release bugs, release new changes to Production.	1 day	1 day	3/26/17	3/26/17	3/28/17	3/28/17
Work on remaining VI to implement modularity in parallel with unit testing	3 days	3 days	3/27/17	3/29/17	3/29/17	4/3/17

Test modularity – release code to production	1 day	1 day	3/30/17	3/30/17	4/4/17	4/4/17
Fourth sprint: Bug burn down	1 week	1 week			4/5/17	4/12/17

The first sprint started and finished on schedule with no issues. The second sprint also finished on schedule, and the new code was released to Production at the end of the second sprint. During the third sprint however a problem cropped up in Production that did cause the project to run longer than anticipated. It also required changing the release method from cut over to parallel. After all the scheduled work was completed slightly later than anticipated it was decided to do a fourth week long sprint, to burn through the bugs in the released code. After this was completed the parallel deployment was made the main deployment.

Unanticipated Requirements

During the third sprint it was brought to the team's attention that there were several high priority bugs keeping the team from continuing their work. To deal with this a quick scrum was held to decide what to do. It was decided to switch the release to a parallel deployment until the internal customer was satisfied with the performance of the new code-base. Doing this allowed limited testing on the manufacturing floor to help drive down bugs more. The team also decided to add a fourth sprint to burn down the remainder of bugs that prevented the project from being accepted and replacing the older code-base. This ensured project completion at the cost of an extra seven days of labor.

Conclusions

The outcome of this project was code that reduces the amount of time the developer is spending on bugs, and increases the amount of time the production floor is able to use the application without a line down bug. This improvement was measured against two metrics. The first metric is the number of reported bugs per week. Previously that was 10 to 15 bugs per week, counting duplicates as the issues are not getting fixed. After completing the project the number of reported bugs per week average was measured at 3 per week. The initial expectation of the first new release having new bugs was correct. The plan should have been more conservative in this regard, planning for the necessity to burn those bugs down. At the end of that two week period after completion, we are only seeing 3 bugs reported per week.

The second improvement was an increased speed in adding new products to the test environment. This improvement allows rapid development of new test platforms, without touching the IDE code to add new tests. The project structure has created a blueprint for dealing with future problems that come up, driving the program to become even more maintainable, reliable and extensible. With the prior application it took over a month to add a new product and due to the fragile nature of the code-base that introduced dozens of new bugs that were not being burned down fast enough during bug sprints. Now that the project is completed, it takes less than a week to add a new product to the code-base.

Project Deliverables

Appendix A includes the chart of the average bugs reported per week, before and after project completion. The data was gathered using the TRAC software the company uses to report and track the bugs reported at the company. The before data set was an average of one month before, the after data set is an average of one month after project completion. Before the project was completed there were 8 bugs reported per week on average. (counting duplicate bugs) After the project that number is down to 3 per week.

Appendix B is a chart comparing how many products were added to the system per month before and after project completion. Before the project was completed I was averaging less than 1 product added per month over a six month period. Tracking since project completion (April 2017) on average, I've been able to add three new products per month.

Appendix C shows what the Measure Step by Step looked like before and after. Before is barely visible at this resolution and it required scrolling horizontally for two full screens. It also didn't meet the current functional requirements and for this reason underwent a complete rewrite. This require converting the data bundle that is running across the screen and being unbundled before being sent into the next sub VI about ½ through the VI. This bundle of data was converted to a class object, that only retrieves data as it needs it. It was also determined a lot of what was being done within this VI was redundant and was being done either earlier or later on. (the left side of the VI) The work done on the right was determined to need it's own encapsulation, and a sub VI was created to do that work. After the measurement is completed, it compares the measurement to the expected values, and writes the results to the data-stream class object.

Appendix D is a comparison between the old front panel and new front panel. This UI redesign was completed while rewriting the front panel and the team decided to make the UI match the other in-house programs which gives the program a more polished, consistent look. Underneath the new UI has the JKI event driven state machine that is more responsive. In the before state, several of the state buttons are grayed out as they are unavailable in the current state. This includes the "bug" and "settings" button. They were only available during one specific state, of the program. With the new program the buttons that are grayed out are grayed out because of process flow, not because of inability to be driven. You can change the settings at any time, and file a bug at any time.

References

LabVIEW Advantage (2016). *How to refactor code - LabVIEW*. Retrieved from https://www.youtube.com/watch?v=nV32Hf1TM1c

National Instruments. (2011). *Inheriting and maintaining labview code.* Retrieved from ftp://ftp.ni.com/pub/branches/uk/devdays_2011/Inheriting_and_Maintaining_LabVIEW_Code.pdf

ALE System Integration (2008). *Refactoring LabVIEW code*. Retrieved from https://ieee.li/pdf/viewgraphs/refactoring_labview_code.pdf

Osherove, Roy (2009). *The art of unit testing*. Physical copy

JKI (2015). *Caraya - A new take on LabVIEW Unit Testing.* Retrieved from http://blog.jki.net/community/caraya-a-new-take-on-labview-unit-testing

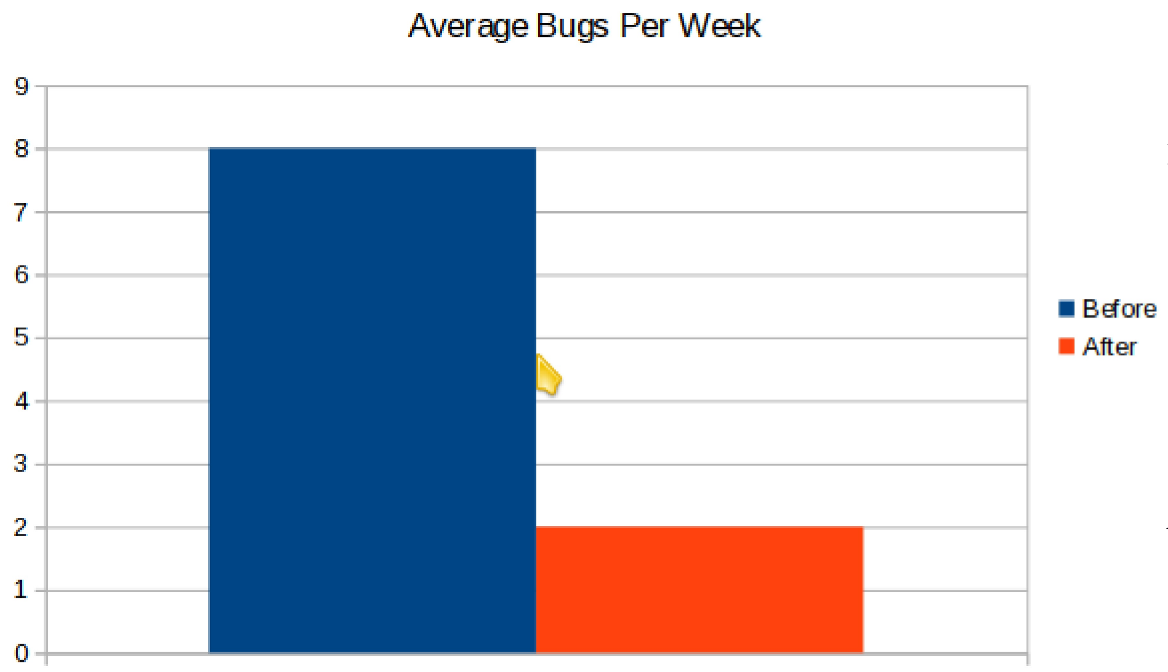
LabVIEW Architects Forum (2015) LAF Q2 2015 – *Unit Testing with the JKI VI Tester (Casey Lamers)* Retrieved from https://www.youtube.com/watch?v=AFNbdF7ZU6s

Maila, Tomi P. (2007) LabVIEW Object Oriented Programming Walkthrough. Retrieved from https://www.youtube.com/watch?v=pomEr5vQpxM&t=424s

JKI (2008) *JKI State Machine Basic Introduction*. Retrieved from https://www.youtube.com/watch?v=XJFujhIuZdU

Appendix A

Average rate of bugs reported per week



A

pp

en

di

x

B

Av

er

ag

e rate of products added per month

Ap

pen

dix

C

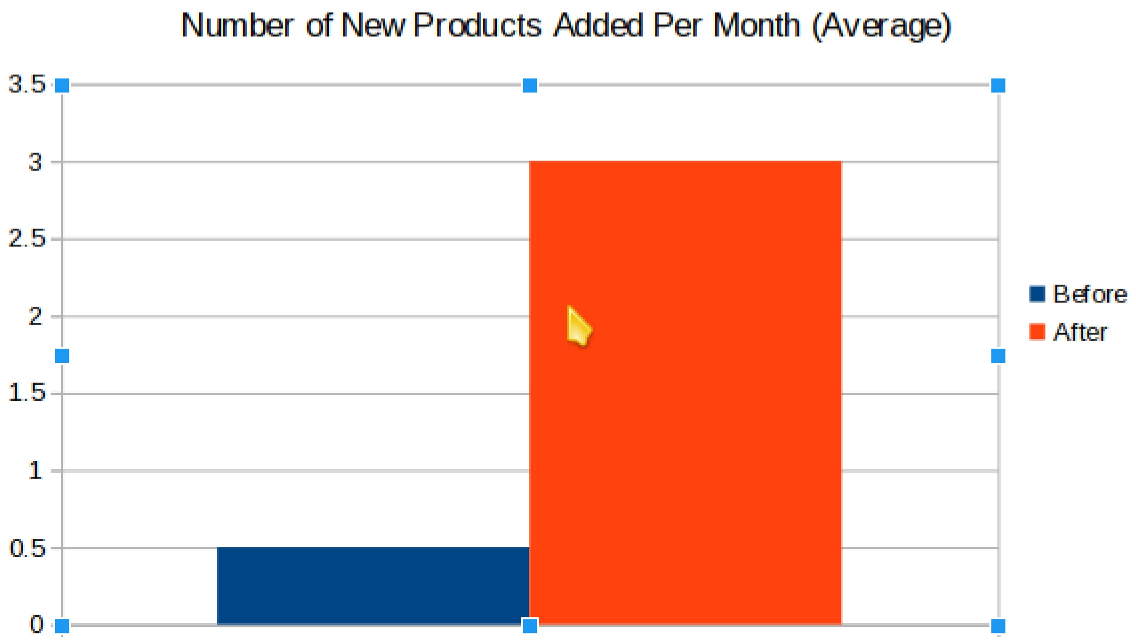
Me

asu

re

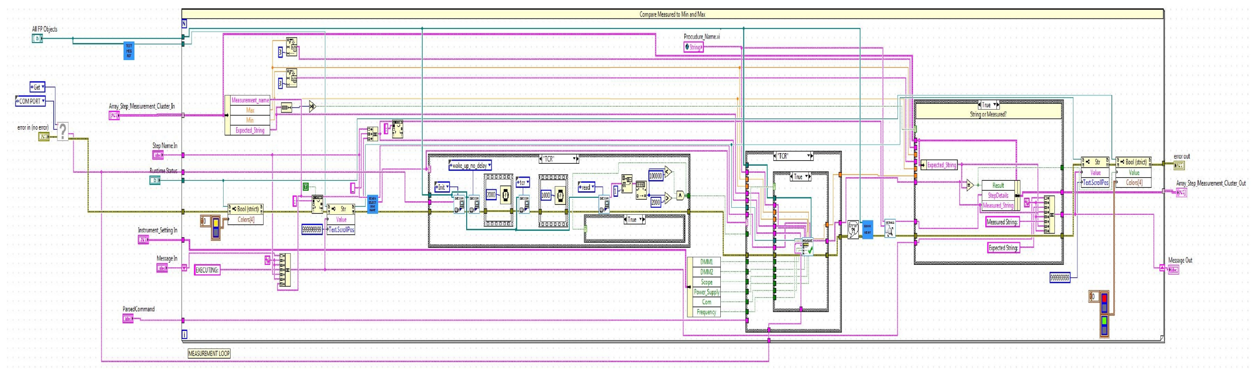
Ste

p

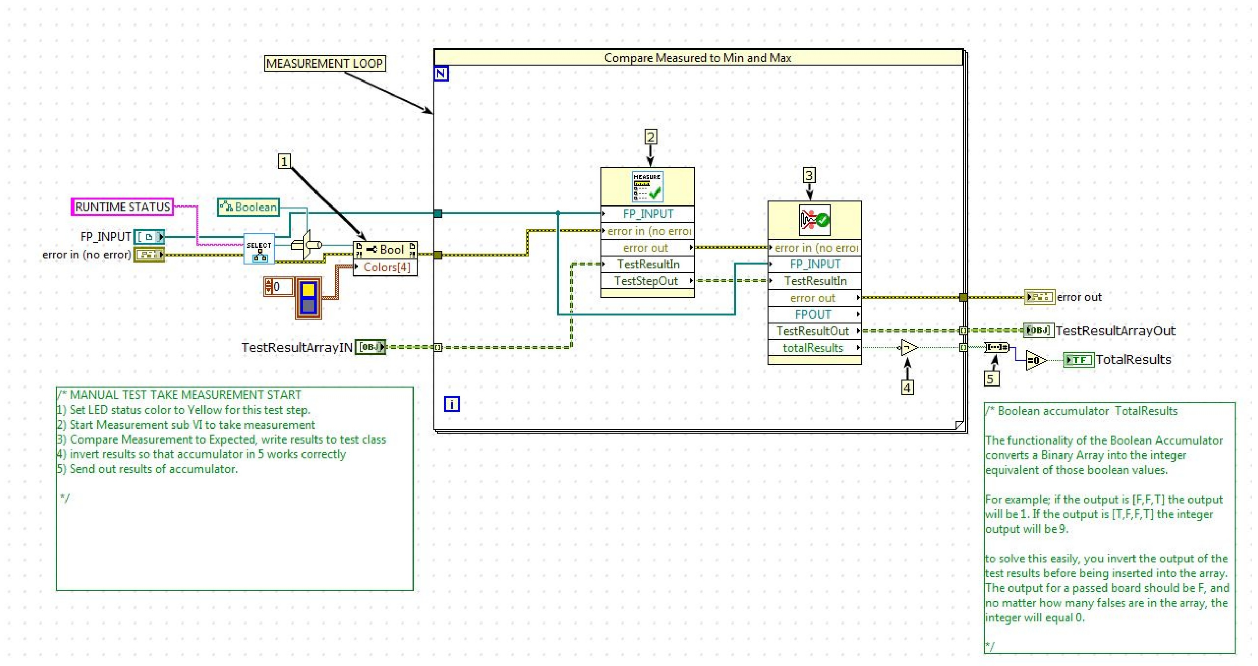


by Step (VI) before and after

Before:



After:



/* MANUAL TEST TAKE MEASUREMENT START
1) Set LED status color to Yellow for this test step.
2) Start Measurement sub VI to take measurement
3) Compare Measurement to Expected, write results to test class
4) invert results so that accumulator in 5 works correctly
5) Send out results of accumulator.

*/

/* Boolean accumulator TotalResults

The functionality of the Boolean Accumulator converts a Binary Array into the integer equivalent of those boolean values.

For example; if the output is [F,F,T] the output will be 1. If the output is [T,F,F,T] the integer output will be 9.

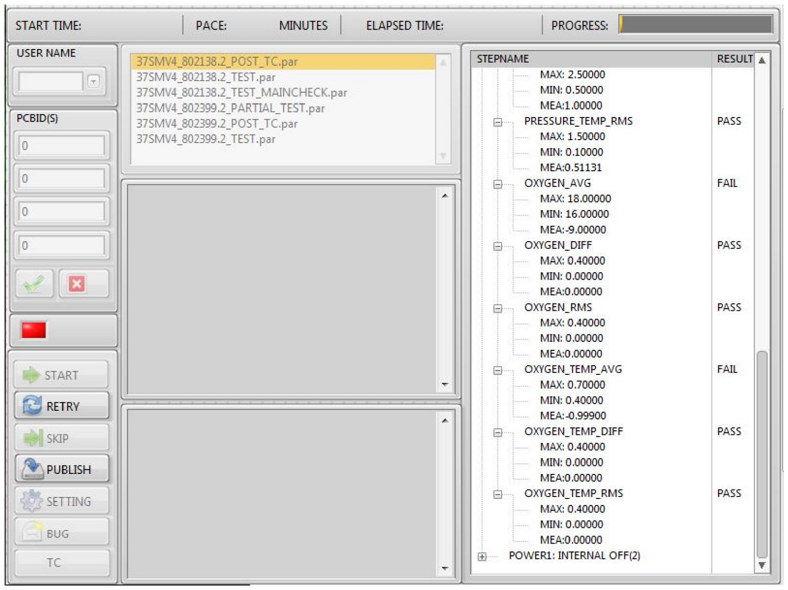
to solve this easily, you invert the output of the test results before being inserted into the array. The output for a passed board should be F, and no matter how many falses are in the array, the integer will equal 0.

*/

Appendix D

Front Panel Refactor before and after

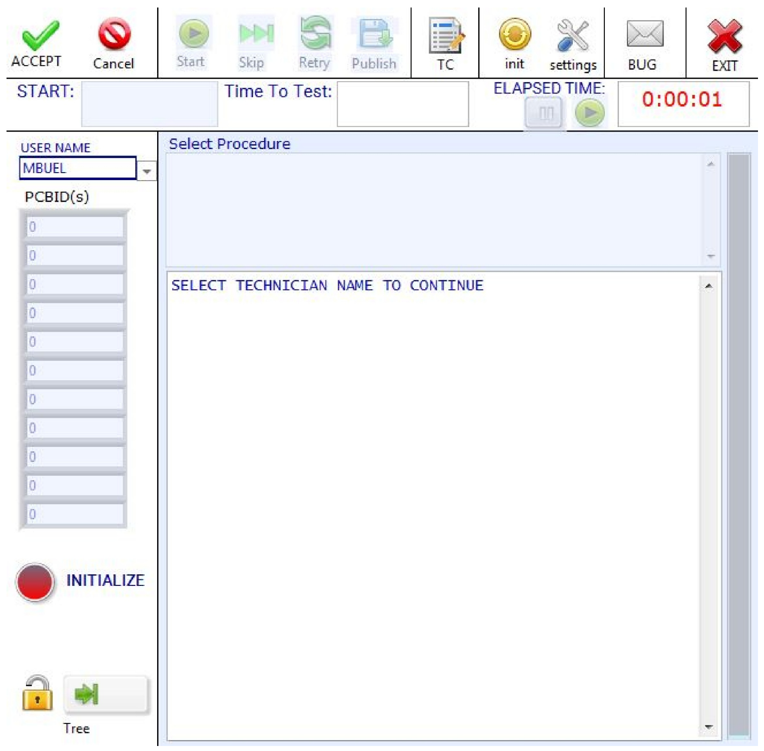
Before:



```
37SMV4_802138.2_POST_TC.par
37SMV4_802138.2_TEST.par
37SMV4_802138.2_TEST_MAINCHECK.par
37SMV4_802399.2_PARTIAL_TEST.par
37SMV4_802399.2_POST_TC.par
37SMV4_802399.2_TEST.par
```

```
STEPNAME                          RESULT
        MAX: 2.50000
        MIN: 0.50000
        MEA:1.00000
    PRESSURE_TEMP_RMS             PASS
        MAX: 1.50000
        MIN: 0.10000
        MEA:0.51131
    OXYGEN_AVG                    FAIL
        MAX: 18.00000
        MIN: 16.00000
        MEA:-9.00000
    OXYGEN_DIFF                   PASS
        MAX: 0.40000
        MIN: 0.00000
        MEA:0.00000
    OXYGEN_RMS                    PASS
        MAX: 0.40000
        MIN: 0.00000
        MEA:0.00000
    OXYGEN_TEMP_AVG               FAIL
        MAX: 0.70000
        MIN: 0.40000
        MEA:-0.99900
    OXYGEN_TEMP_DIFF              PASS
        MAX: 0.40000
        MIN: 0.00000
        MEA:0.00000
    OXYGEN_TEMP_RMS               PASS
        MAX: 0.40000
        MIN: 0.00000
        MEA:0.00000
  POWER1: INTERNAL OFF(2)
```

After:



USER NAME: MBUEL

ELAPSED TIME: 0:00:01

```
SELECT TECHNICIAN NAME TO CONTINUE
```