



Agile Testing In Embedded Systems to Improve Product Quality

Betty.K.John and Tintu Roshni
Network Systems and Technologies Pvt (Itd),
A-3, Periyar, Technopark, Trivandrum, Kerala, India
{betty.john, tintu.roshni}@nestgroup.net

Abstract:

A good software development process should optimize testing and improve feedback from reality. Testing is a disciplined process of evaluating an application's behavior, performance, and robustness -- usually against expected criteria. Testing from the beginning of the project and continually testing throughout the project lifecycle is the basis of agile testing.

One of the main implicit criteria is to make the product as defect-free as possible. If a defect is caught within twenty four hours, the cost of fixing the bug is negligible when compared with the cost of fixing it later, after more code has been written on top of it. Agile Testing is not the answer for all projects. Agile manifesto values individuals and interactions over processes and tools. However testers must use a risk-based approach, grounded in both the system's architectural reality and the attacker's mindset, to assure a bug free product to the customer.

Keywords:

Agile methodologies, Extreme Programming, Embedded Development.

1 Introduction

Agile methodologies have captured the interest of academia and practitioners alike in the past few years. Embedded software is a unique specialty within the broader software field. Effective management of software development risks is very much essential for producing software with high quality and with optimal cost. Embedded systems can be termed as resource constraint systems. So the quality in embedded systems is centered towards platform specific testing tools that are geared towards debugging.

Extreme Programming is almost unheard in the world of embedded technology. Possibly this is due to the little guidance and knowledge available on agile testing, in the embedded domain. A set of powerful test techniques and good knowledge in them is well enough to perform agile testing to improve the quality of a product. Agile Software development is first of all a people centered activity and only secondarily a technical discipline.

Agile Testing is working with the customer to help him specify his requirements in terms of tests, in turn which makes them completely unambiguous. The tests either pass or they don't. If coders only write code to pass tests, we can be sure of one hundred percent test coverage. If we keep our testers, developers and customers (or customer representatives) in constant face-to-face communication with each other, we can eradicate most of the

errors caused by wrong inference from documents. Breaking the projects into smaller chunks of work and iterating them will give us frequent feedback on the current state of the project

The testing of embedded software has its deep roots in the corresponding hardware. Multiple test strategies help agile methods work well with evolving hardwares. Thus the quality of high reliability systems which has embedded software as its heart can be improved.

Application of agile testing methodology in embedded projects reduces, defect count, increases product quality and also working on a clean and highly stable code base makes new development faster and optimizes the cost of production. Besides this strategy helps to avoid unpleasant customer negotiations. One goal with this paper is to help the project team working on embedded technology to get onboard with agile testing techniques. Real teamwork is the greatest necessity to get the most from the techniques described in this paper.

2 Challenges and Strategies

The goal was to build an infotainment system that can be plugged into the Car for getting GPS based route map information. The Audio and video systems in Navigation system serves the entertainment purpose. There were a number of hidden technological risks that we need to undertake in this effort. Our company had expertise but cataloguing the products requirements was found difficult. Frequent requirement changes from the customer made it a tedious task to create a complete set of requirements and a traceability matrix.

It was very important to see what the data would look like and how the algorithms would perform using it. The strategy to keep things moving was to identify a set of deliverables for each release and to freeze the requirement set for the particular release – an agile strategy. The design was kept intact and simply rearranged the design for the corresponding releases with new requirements. Also it was taken care that these changes do not affect the quality of the product being developed.

Another difficulty was to make the staffing estimates as a detailed set of requirements could not be obtained at the start of the project itself. From past experiences in the embedded domain, the team based the estimate on the present set of requirements and assumptions. Metrics from previous projects was used as an input for staffing estimations. Another hurdle came when some of the team members lacked important background. Trainings on the required domain were initiated to efficiently spread domain knowledge around the team.

Team made the best use of agile testing technique as the team members could easily step through the code with a debugger to see the detailed workings. If someone makes a mistake, and cannot get the code working again, they can back off to the point where all tests ran before their changes. The team had put together the best practices used effectively before to get through the hurdles.



3 Evolution of Hardware Platform

Since hardware is required for the running of a software application, one can compare its impact to the construction materials have in traditional engineering disciplines. Hardware always changes in embedded development and these hardware changes needs to be supported by the software also. Embedded software development strategy deals with changing hardware and hardware sets the tight requirements for the software in the embedded domain.

Embedded Testing Techniques

Trouble Log Mechanism

One of the oldest defense mechanism against bugs is to code a mechanism for displaying brief messages to indicate that some portion of the code was reached or a variable has an invalid value. These methods can enable the code to execute differently at times while attempting to trouble shoot a problem. Bugs may be due to hardware, software or both. Embedded software relies on specialized debugging than on high level software. Thus finding the source of unintended behavior generally requires more effort than in high level software. After a bug is found and corrected there is nothing in place to watch that same code and to point out undesired code interaction in future. Relying heavily on debugging rarely enforces good coding practices. A different kind of trouble log was used within the project to establish the trouble log mechanism. The “assert” macro can be used to state an assumption.

Example of assert statement used in code:

```
ASSERT(response == CR_SUCCESS_SEND_CONTROL_HANDSHAKE);  
ASSERT(dwRet == ERROR_SUCCESS) ;
```

Example of function for testing delete API for non existing file using automated test application.

```
void XFILEDeleteTest::testNonExistingFile()  
{  
#ifdef XFILE_USB  
    TEST_ASSERT_EQUALS(  
"TC 119",XFILETestObj.Delete(NON_EXISTING_FILE),false);  
    TEST_ASSERT_EQUALS(  
"TC 7",XFILETestObj.OpeXFILE(NON_EXISTING_FILE,READ_MODE),false);  
    TEST_ASSERT_EQUALS(  
"TC 57",XFILETestObj.IsOpen(),false);  
    TEST_ASSERT_EQUALS(  
"TC 63",XFILETestObj.CloseFile(),false);  
  
    TEST_ASSERT_EQUALS(  
"TC 120",XFILETestObj.Delete(""),false);  
    TEST_ASSERT_EQUALS(  
"TC 57",XFILETestObj.IsOpen(),false);  
    TEST_ASSERT_EQUALS(  
"TC 63",XFILETestObj.CloseFile(),false);  
#endif  
}
```



```
TEST_ASSERT_EQUALS(  
"TC 121",XFILETestObj.Delete(FILENAME_LENGTH_GREATER_256BYTES),false);  
TEST_ASSERT_EQUALS(  
"TC  
121",XFILETestObj.Delete(FILENAME_LENGTH_GREATER_260BYTES_PATH),false);  
TEST_ASSERT_EQUALS(  
"TC 57",XFILETestObj.IsOpen(),false);  
TEST_ASSERT_EQUALS(  
"TC 63",XFILETestObj.CloseFile(),false);  
#endif  
#ifdef XFILE_HDD  
TEST_ASSERT_EQUALS(  
"TC 119",XFILETestObj.Delete(NON_EXISTING_FILE_HDD),false);  
TEST_ASSERT_EQUALS(  
"TC 7",XFILETestObj.OpeXFILE(NON_EXISTING_FILE_HDD,READ_MODE),false);  
TEST_ASSERT_EQUALS(  
"TC 57",XFILETestObj.IsOpen(),false);  
TEST_ASSERT_EQUALS(  
"TC 63",XFILETestObj.CloseFile(),false);  
TEST_ASSERT_EQUALS(  
"TC 120",XFILETestObj.Delete(""),false);  
TEST_ASSERT_EQUALS(  
"TC 57",XFILETestObj.IsOpen(),false);  
TEST_ASSERT_EQUALS(  
"TC 63",XFILETestObj.CloseFile(),false);  
TEST_ASSERT_EQUALS(  
"TC  
121",XFILETestObj.Delete(FILENAME_LENGTH_GREATER_256BYTES_HDD),false);  
TEST_ASSERT_EQUALS(  
"TC  
121",XFILETestObj.Delete(FILENAME_LENGTH_GREATER_260BYTES_PATH_HDD),fal  
se);  
TEST_ASSERT_EQUALS(  
"TC 57",XFILETestObj.IsOpen(),false);  
TEST_ASSERT_EQUALS(  
"TC 63",XFILETestObj.CloseFile(),false);  
#endif  
}
```

4.1.1 Trouble Log Output

Trouble log output provides a comprehensive record of all trouble log entries. The output enables the development team to analyze the error generated by the trouble log mechanism and necessary steps to debug the same can be initiated. The project team used it throughout all of the code to validate assumptions and to trace the execution of the code that is difficult to trace in any other way. The trouble log mechanism helped to reduce the bug count throughout the execution of the project. The trouble log output helps to improve the quality of the code. Severity levels can also coded into each trouble log call to govern the behavior of the system. The trouble log mechanism is an agile methodology



to balance the resource use with the big advantage that we can troubleshoot a bug without altering the way the code executes by enabling the logging system. The state of the product can be easily identified by analyzing the trouble log at the time of field test. This technique depends on the team to put enough calls into the code at the right places. The severity also needs to be understood. The team relied on thorough code reviews to achieve thorough and consistent use of the trouble log calls in the code.

Example of Sample trouble log output.

```
XFILEDeleteTest  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 33  
TestCase Number -> TC 1  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 34  
TestCase Number -> TC 55  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 35  
TestCase Number -> TC 123  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 131  
TestCase Number -> TC 1  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 132  
TestCase Number -> TC 81  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 133  
TestCase Number -> TC 59  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 134  
TestCase Number -> TC 1  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE  
XYZ TestApp\XFILEDeleteTest.cpp at 135  
TestCase Number -> TC 78  
  
Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG  
TestApp\XFILE
```



*XYZ TestApp\XFILEDeleteTest.cpp at 136
Test Case Number -> TC 78*

*Assertion Failed at C:\ZZZ0016\ZZZ\CPU25\Application\src\EFG
TestApp\XFILE
XYZ TestApp\XFILEDeleteTest.cpp at 137
TestCase Number -> TC 59*

4.2 Dual Targeting

The practice of test driven development allows you to concurrently develop automated tests and the working code that satisfies those tests. Rather than coding followed by debugging, this approach gives you feedback as code and automated tests are concurrently developed. The practice results in higher quality software and reduced schedule uncertainty. In the embedded environment, this approach encourages dual targeting where code and tests are designed, built and run on a host development system before they are run on the target. If you can make your module testable on the development system, you have successfully managed your hardware dependency. The Hardware always represents an extra dimension in the embedded domain and this concept needs to be very well addressed in the testing strategy also. We built the code as dual target software. The software could be run on Windows as well as on the target through the use of compile time switches. This strategy helps the team to test the same software first on Windows and then on the target where the hardware was stable. Through this methodology, the team could exercise the logic very well. Dual targeting is viewed as a simple test framework where tests are designed to run on both platforms and was used throughout the execution of the project.

4.3 Hardware Driver Unit Tests

In dual targeting strategy, there is no explicit test code for the parts of the software that drove the hardware. There can be classes that touch the hardware and that do not. A clear distinction between components that are driven by the environment and those that drive the hardware needs to be made. For the former, unit testing strategy can be identified in terms of stubs. However for the latter, unit tests that can be run only on target hardware has to be identified.

Building of embedded systems involves the production of the hardware, the development of an API that defines how to access that hardware, and software that drives usage of that hardware. The team performed a significant amount of testing from the windows so that hardware testing need only verify the interfaces between the software and the hardware, and confirm that the functionality that was verified on the windows still operates as expected on the target device. Functionalities like initialization, read, write, mount, security, performance etc were tested for hard disk drive (HDD) before going for system integration testing. The hardware unit test code satisfied the entire unit tests. The entire hardware unit tests could be put together to run as a final test before the product was released. Because the very same code was driving the hardware in the application, the



team never had to debug the code at that level once it correctly drove the hardware. This helped to increase the confidence within the team.

```
bReturn = DSK_Init();/* Hard Disk initialisation
pDisk = (PDISK)(pDiskObject);
if(pDisk == 0 || (bReturn == FALSE))
{
    DbgPrintf("Initialisation Failed");
    DbgPrintf("Event is id %d\n",pEventInfo->event);
}
```

4.4 Domain Level Tests

A specific problem for the team was the serial communications domain was having unexpected delays in receiving a command and then responding. System test could bring this out. Unit tests could not address the timing issue. Domain level tests helps to identify whether a problem is within a domain or is somewhere else in the system. In embedded systems timing constraints are more common. The team built an image of the actual target containing only the serial communication feature, load that to target hardware and send it to test commands over the serial link. Timing problem was detected with just the serial communications feature running stand alone and also with the full system running. Thus the team members were able to conclude that the timing issue was with the serial communication feature. Domain level testing can be viewed as a tool for isolating the problems observed at the system level, mostly timing issues in embedded systems. Domain level tests helped the team to identify the problems very quickly. Domain level tests were also done to ensure throughput, Performance, data Integrity etc.

4.5 Special Test Mode for Volatile domain

The algorithm to be implemented in our software was in a preliminary state. The team used Matlab to run the test data through its logic. We could understand that some changes need to be implemented in the written code. The team decided to have a check on the computations against results obtained from Matlab. Test data was prepared for use in Matlab. The data path for our product originated in sensor hardware and went through a variable number of computation stages before a final result was obtained. If the developed software could inject the same initial test data and compare the end result with Matlab result, then testing would be much easier. We can easily identify a mistake in the product software if the two end results are different. Our code had a variable to enable this test mode and a set of files to pull the Matlab data for comparison. This technique allowed the team to easily implement and test the changes that occurred often. This is a full system test on target hardware and hence all possible trouble sources could be checked quickly and easily. This technique helped to perform integration test as well as unit test in a simultaneous manner. Situations where a domain is expected to undergo many changes throughout development, the changes at whatever level of maturity the

remainder of the system may be in can be monitored. Iterative releases that were planned initially helped the team to check the intermediate stage results.

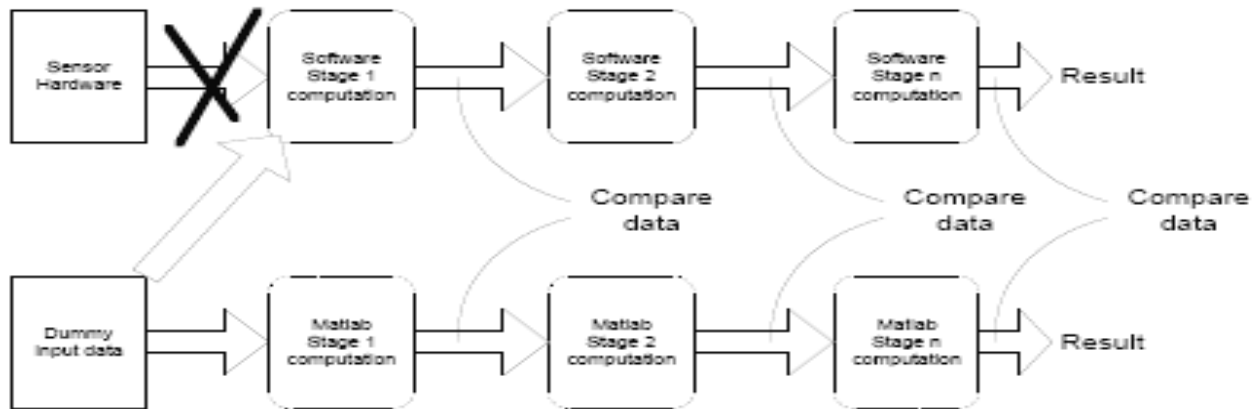


Figure 1. Execution flow in Matlab

4.6 Bug Tracking

Bugs were tracked at each testing point using our bug tracking tool (Mantis). Any undesired or unexpected behavior in the software was logged as a bug into the tool. Root cause analysis was done at the required points depending upon the criticality of the bugs, which helped the team to reduce the bug count to considerable levels. Bugs that got past the teams unit testing had an even chance of getting to ITT (Independent Testing Team) further. A frequent root cause that we observed as in every other project was that code review was not thorough enough. Detailed bug statistics on the number of bugs caught in testing of each module were analyzed against industry baselines at every milestone for achieving a reduced bug count.

5 Results

The biggest result of using these testing techniques is that we had a very low bug rate. Most of the bugs were caught at the unit level testing itself. Every bug that was reported was closely analyzed. The team could utilize their time adding value instead of scrambling to fix defects. The relationship, the team had with the hardware group was something different from that of non agile projects. The team could easily isolate the bug in the software by parting the software into different chunks. The hardware was also checked along with the software before concluding that there was a software problem. Finally the team could prove that software was the most stable part of the system.

6 Conclusion

The Agile testing methodology in embedded systems explained above takes the project team one step forward by bringing in improvement within the project management, execution and testing strategies. Bug rate can be reduced by adopting the agile testing methodology within the embedded project development. Bugs are captured at the unit level testing itself which helps us to learn about the possible unintended defects at every stage of the project execution. Above all this methodology helps to improve the relationship between the development team and the testing team than what we observe in non agile projects.

Acknowledgement

The Authors are thankful to Mr. N. Jehangir, Managing Director, NeST Group and Mr. Sasikumar, President of NeST, TVM for their support in carrying out this work. We are grateful to Mr. Sebastian Antony Ukken, QTG Head and Balamurali.L, SQA Manager for their useful discussions and suggestions at different stages of this work. The continual motivation and guidance from Mr. Rajagopal K, Senior Test Manager NeST and Annu George Associate SQA Manager, is highly acknowledged.

Citations

- **Journal Articles**

Matt Fletcher, William Bereza, Mike Karlesky and Greg Williams, Atomic Object,LLC: Evolving into Embedded Development.

- **Book**

Deming. W. Edwards, The new economics for the Industry, Government and Education 2nd Edition,MIT Press, Cambridge, MA, 1993.

M. Dowty, “ Test Driven development of Embedded Systems Using existing software test infra structure“.

References

[1] J.Grenning, 'Extreme Programming and Embedded Software development.'
www.objectmentor.com/resources/articles/EmbeddedXp.pdf.

[2] L.Crispin and T.House, Testing Extreme Programming,Addison-Wesley,2002

Biography



Betty.K.John is working as Quality Assistant at NeST– A SEI CMMI Level 5 Company, Thiruvananthapuram. She did her MCA at Electronic Research and Development Center of India (ER&DCI), Thiruvananthapuram. She has an experience of 2+ years in the software industry and has worked in quality activities of the organization. She is a Certified Internal Software Quality Auditor by STQC, Ministry for Information Technology; Govt. of India She may be contacted at betty.john@nestgroup.net



Tintu Roshni is working as Engineer-QA at NeST– A SEI CMMI Level 5 Company, Thiruvananthapuram. She did her B.Tech at Calicut University Institute of Engineering and Technology. She has an experience of 1 year in the software industry and has worked in quality activities of the organization. She may be contacted at tintu.roshni@nestgroup.net