PROPUESTA METODOLÓGICA PARA VALIDAR LA FUNCIONALIDAD DE SOFTWARE EN SISTEMAS EMBEBIDOS

José Guillermo Fierro Mendoza
Tecnológico Nacional de México/Instituto Tecnológico de Celaya
guillermo.fierro@itcelaya.edu.mx

Julio Armando Asato España
Tecnológico Nacional de México/Instituto Tecnológico de Celaya
julio.asato@itcelaya.edu.mx

José Benigno Molina Castro
Tecnológico Nacional de México/Instituto Tecnológico de Celaya
benigno.molina@itcelaya.edu.mx

Jonathan Giovanni Delgado Núñez
Tecnológico Nacional de México/Instituto Tecnológico de Celaya
yon-a123@hotmail.com

Emmanuel Noriega Vaca
Tecnológico Nacional de México/Instituto Tecnológico de Celaya
vakemi94@gmail.com

Resumen

En la actualidad la innovación tecnológica ha permitido incorporar diversos dispositivos electrónicos, que realizan tareas automatizadas en aparatos o equipos industriales, o de uso en hogares u oficinas. En general, estos dispositivos son computadoras de propósito especial dentro de un sistema mecánico o eléctrico, que están diseñados para satisfacer las necesidades específicas, conocidos como sistemas embebidos. Estos sistemas los encontramos en los automóviles, en el control industrial, en aparatos del hogar como refrigeradores, lavadoras, entre

otros. Estos sistemas incluyen un microprocesador que internamente mantiene algoritmos computacionales que leen datos provenientes de sensores, los procesan y producen una acción de retroalimentación. La complejidad del sistema se refleja en el manejo de señales recibidas y en el procesamiento de la relación entre ellas y el estado de las variables de control para generar una respuesta. La programación de la lógica de operación, anteriormente se hacía en lenguaje ensamblador, pero actualmente se realiza en lenguajes de alto nivel, como son Java y C, teniendo especial cuidado en aquellos sistemas en que el tiempo de respuesta es crítica, como lo establece Fariña (2015). El incremento de la complejidad de los algoritmos que realizan los dispositivos adiciona la posibilidad de generar fallos en el sistema, aunado con la necesidad de mejorar la productividad del software intentando construir programas altamente confiables en periodos de tiempo cada vez más cortos. Este artículo es una propuesta metodológica para incorporar mecanismos que mejoren la calidad del producto de software, aplicando reglas de codificación en el desarrollo, utilizar herramientas que permitan cuantificar su complejidad y analizar los modos en los que puede fallar e incorporando la solución.

Palabras Clave: Prueba de software, pruebas unitarias, software embebido, validación, verificación.

Abstract

Today technological innovation has allowed incorporating various electronic devices that perform automated industrial equipment or use equipment in homes or offices tasks. In general, these devices are special purpose computers within a mechanical or electrical system, which are designed to do specific task, known as embedded systems. These systems are found in automobiles, industrial control, in household appliances such as refrigerators, washing machines, among others. These systems include a microprocessor internally maintains computational algorithms that read data from sensors, process and produce a feedback action. System complexity is reflected in handling received signals and processing the relationship between them and the state of the control variables to generate a

response. These systems include a microprocessor inside it has computational algorithms that read data from sensors, process and produce a feedback action. System complexity is reflected in handling received signals and processing the relationship between them and the state of the control variables to generate a response. The programming was did previously in assembly language, but currently it done in high level languages as C and Java, taking special care to those systems where response time is critical, Fariña (2015). The increasing complexity of the algorithms that perform operation of device can produce system failure coupled with the need to improve the software productivity trying to build software highly realibility in less time. This article is a proposal to incorporate mechanisms to improve product quality software, introducing coding rules applying in the development, use tools to quantify their complexity and analyze the ways that can fail.

Keywords: *Test software, unit testing, embedded software, validation, verification.*

1. Introducción

Un par de pregunta para iniciar, ¿Qué falla más, el Hardware o el Software?, ¿Hay aplicaciones más críticas que otras? De acuerdo a nuestra experiencia, en la respuesta a la primera pregunta, las personas involucradas en la construcción de productos que implican hardware y software, se inclinan más a responder que es el software el que falla más. Otros, un poco más experimentados responderán que depende del producto, por ejemplo en una computadora y sobre todo si tiene cierto sistema operativo, es notorio que falla más el software. Sin embargo, en sistemas especializados, en los que el software está embebido, como las computadoras de vuelo de un avión, la ECU (*Engine Control Unit*) de un automóvil (Yoshida, 1996), dispositivos de telemedicina que monitorean remotamente las condiciones de salud de los pacientes, o incluso el módulo de control automático de nuestras lavadoras, una vez en operación es poco frecuente que fallen debido a una causa de software, generalmente las fallas están relacionadas con el tiempo de vida del hardware del producto. Por otra parte, también se dice que entre más complejo es el software tiene más posibilidad de fallar, de hecho se establece

como la causa número uno de fallas del software (Yoshida, 1996). Es entonces importante, reflexionar sobre la metodología que se utiliza para construir software que no falle y más en los tiempos actuales en que el auge del internet de las cosas, está causando una verdadera revolución tecnológica, en el intento de conectar todos los dispositivos electrónicos de uso diario al internet. En este trabajo, proponemos una metodología para hacer software embebido de calidad utilizando esquemas de trabajo ágiles. Seguramente el lector estará familiarizado con el concepto de caída de sistemas controlados por software que provocan muchos inconvenientes, sin embargo, existen algunos sistemas donde las caídas pueden provocar una pérdida económica importante, daños físicos o amenazas a la vida (Sommerville, 2011). Estos sistemas se conocen como sistemas críticos (Fariña, 2015). Un atributo esencial de los sistemas críticos es la confiabilidad, concepto que involucra los temas de disponibilidad, fiabilidad, seguridad y protección. Desde esta perspectiva y en respuesta a la segunda pregunta, es importante aclarar que efectivamente los sistemas críticos se clasifican por su tipo en sistemas de seguridad críticos, sistemas de misión críticos y sistemas de negocios críticos. Esta consideración es importante al momento de construir y liberar un producto de software. En la búsqueda de caracterizar los procesos de construcción de productos de software para que sean confiables, se han diseñado marcos metodológicos, similares a los que se emplean en el proceso de hardware, en los que el producto de software se prueba exhaustivamente de tal manera que se asegura que trabajará adecuadamente dentro de los límites operacionales para las que fue diseñado y en condiciones de seguridad. En este artículo se describe el marco metodológico para desarrollar software embebido confiable, producto de los trabajos de investigación relacionados y nuestra experiencia en el área. En la actualidad se puede apreciar un incremento notable en el grado de complejidad en el software de aplicación específico, debido al desarrollo tecnológico e innovación. Cada día se incorporan nuevas funciones en ellos, por ejemplo el caso de las computadoras de las computadoras de automóviles (ECUs) que cada vez incorporan nuevas señales de sensores y eso hace que los sistemas de control o la lógica de procesamiento sea vuelva más compleja. Scrum es un marco de

trabajo que ha sido muy bien aceptado e incorporado en la industrias de desarrollo de software y por eso lo hemos contemplado a ser un factor importante para guiar el proceso de software. Proponemos vincular el modelo en V que se utiliza en la producción de software embebido con Scrum e introducimos algunas técnicas que en particular nos ha dado buen resultado para la construcción de software libre de fallas, es decir, que la tasa de errores sea los más baja posible.

2. Método

Para fundamentar la propuesta se realizó una investigación documental sobre el estado del arte actual en el desarrollo de productos de software embebido. Se encontró que si bien el modelo en V, es una herramienta de bastante utilidad para guiar el desarrollo, es posible adicionar algunos aspectos en los que se puede precisar con mayor nivel de detalle las herramientas a utilizar para garantizar que en cada fase el producto obtenido sea de calidad y que el tiempo de desarrollo esté dentro de los márgenes aceptables, y por otra parte, se incorporan los elementos de Scrum, marco de referencia muy consolidado actualmente para replantear las etapas del proceso de desarrollo.

Incrementar la complejidad de un programa, implica un aumento de requerimientos que se deben validar y verificar para garantizar la calidad del producto de software y que en sistemas de la naturaleza que hemos indicado es sumamente estricta. Por lo tanto es importante analizar y contemplar en la propuesta, técnicas y herramientas reconocidas por sus estándares y la garantía que ofrecen para el análisis de pruebas estáticas. Se hace una breve descripción de herramientas como MISRA-C y el impacto en la disminución de errores por codificación.

Durante el proceso de desarrollo de software, se va generando código gradualmente y de la misma manera se va verificando su funcionalidad, sin embargo, en la mayor parte de las ocasiones, se construye nuevo código que afecta o que tiene relación con el código anterior o que simplemente requiere cambios a lo ya construido, lo que origina muchas veces se tengan que realizar pruebas regresivas al producto en cuestión, afectando el tiempo de desarrollo y

sobre todo, puede propiciar omisiones al dar por hecho que el sistema no se ve afectado con los cambios implementados. La incorporación de pruebas automatizadas es una alternativa de solución que se contempla en la metodología. La incorporación de pruebas unitarias en la fase de la codificación es una solución para esta situación. Para ello, es deseable dominar el desarrollo orientado por las pruebas TDD (*Test Driven Development*) como estrategia de desarrollo.

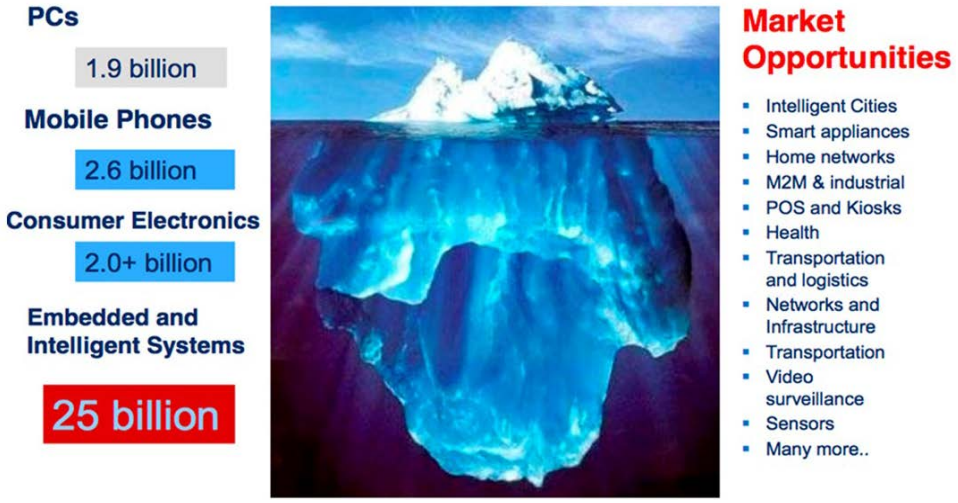
Como afirman Yoshida y Tada (1996), el software que desarrollamos puede fallar por diversas circunstancias, entre otras porque el hardware falla, por problemas con los recursos, como memoria o disco. Por otra parte, debemos recordar que el software es un producto hecho por humanos y que puede acarrear errores involuntarios debido a problemas de comunicación, documentación o combinación de ellos, los motivos pueden ser de varios tipos. Se pueden generar diversos errores pero como afirma Cortés (2007), si el software presenta una falla, sin duda, es por falta de rigurosidad en el análisis de requerimientos, o porque nadie conocía las condiciones que provocaron el fallo, se dice que el único "bug aceptable" es aquel que revela algo que no sabíamos del sistema, todos los demás se deben prevenir. Durante la etapa de desarrollo, se pueden ir detectando errores, también conocidos como *bugs*, que deberán corregirse sin afectar la funcionalidad que ya estaba probada. En este caso debemos realizar las pruebas que validaban el código antes de la modificación y agregar nuevos test que nos ayuden a asegurar que si se corrigió el bug no dejemos sin funcionar algo que estaba bien, conocido como pruebas de regresión. Está comprobado que no es la cantidad de pruebas que hagamos al código, sino la calidad de dichas pruebas lo que nos producirá código menos propenso a fallas. Podemos citar algunos accidentes debido a fallas de programas informáticos que provocaron catástrofes ya sea físicas o económicas, como el caso del fallo del software para controlar el convertidor "*boost*" en un módulo que forma parte del sistema híbrido del modelo Prius de Toyota, que según el portal Infotechnology (2014) puede afectar millones de estos vehículos. El producto de software que elaboremos puede fallar si las especificaciones no son las adecuadas o si no realizamos las pruebas correctas. Para el diseño de pruebas funcionales debemos utilizar técnicas pertinentes

aplicándolas a pequeñas unidades de código y diseñando un número de pruebas que garantice que se cubra todo el dominio de operación del software en proceso. Una vez que el software se ha probado funcionalmente podemos aplicar las pruebas no funcionales enfocadas más con la operación del producto total. Realizar pruebas de Verificación del producto implica confrontar lo que hace el producto con respecto a las especificaciones y las pruebas de validación están enfocadas a corroborar que el software cumple con las expectativas del cliente, Nijhawan (2016). De acuerdo con Fariña (2015) el crecimiento en la complejidad del software embebido y la necesidad de construir el software más rápido, originan la necesidad de incorporar esquemas de detección temprana en el desarrollo de software, las metodologías actuales se han vuelto ineficientes y es necesario incorporar nuevos esquemas o marcos de desarrollo que consideren esta complejidad.

Un sistema embebido consiste de una aplicación de propósito específico alojada en un microprocesador, en un microcontrolador o en un dispositivo programable para procesar información generada por sensores y, mediante señales de salida, retroalimentar la posición de actuadores para realizar una acción de control o simplemente hacer el despliegue de información para cuestiones de monitoreo. Muchos sistemas de control computarizados están integrados en otros dispositivos que se utilizan en comunicaciones, en sistemas de transporte (automóviles, trenes, aviones), en medicina, en robótica, en domótica e incluso en el hogar, para desarrollar una tarea específica. Los sistemas embebidos permiten realizar nuevos esquemas de control. Ejemplos de ellos los tenemos en sistemas de alarmas antirrobo, control de aire acondicionado y calefacción, un refrigerador industrial, el horno de microondas, el control del sistema de combustión del automóvil, las máquinas despachadoras de combustible para automóviles, e incluso los celulares, entre otros. Una de las partes más importantes de un sistema embebido es un microprocesador que incluye interfaces de entrada/salida en el mismo chip. Estos sistemas se programan directamente en lenguaje ensamblador o también mediante compiladores específicos que utilizan C, C++ o java. El software para estos sistemas se caracteriza por: 1. operar en tiempo real, 2.requerir código

óptimo que utilice al máximo los recursos del microprocesador, 3. Disponer de un sistema de desarrollo específico para cada familia de microprocesadores que se utilice. El caso del refrigerador industrial, es un ejemplo de sistema embebido, el sistema puede interactuar con un usuario a través de un selector para posicionar la temperatura a la cual desea mantener los alimentos, un sensor de temperatura en el compartimiento del refrigerador registra la temperatura y la envía como dato a un microprocesador, donde se encuentra el software embebido y que de acuerdo a la temperatura indicada por el sensor, produce una señal para el control del compresor con la finalidad de mantenerlo en operación hasta obtener la temperatura deseada. También enviará el dato de temperatura real a un "display" del refrigerador y podrá incorporar algunos esquemas de alarma, como puerta abierta. A futuro podríamos pensar en registrar las temperaturas de operación del refrigerador en una plataforma web a través de un sensor GPRS para tener reportes diarios y compartir información o simplemente habilitar notificaciones a un dispositivo celular del usuario advirtiendo que la temperatura está muy alta o fuera de rango. En el ejemplo anterior se puede extender y convertir en un sistema inteligente para la administración de un los alimentos en un refrigerador donde se puede apreciar que a través de sensores RFID y un sistema lector que se comunica a través de la red inalámbrica con una aplicación que se ejecuta en una tableta, se podría facilitar la administración de los alimentos ubicados en los compartimientos del refrigerador, aumentando también la funcionalidad y complejidad del sistema.

Una estimación del crecimiento de los sistemas embebidos, muestra que el incremento será mucho mayor que el mercado actual de las computadoras personales y teléfonos móviles en los siguientes años (Solis, 2016), situación que debe ser motivante para desarrollar de tecnologías y marcos de trabajo que permitan el desarrollo de soluciones en menor tiempo y de alta calidad. En la figura 1 se muestra la expectativa del mercado de sistemas embebidos, el cual será mucho mayor que el mercado actual de computadoras personales y teléfonos móviles en los siguientes años.



Fuente: https://goo.gl/OsyfbR

Figura 1 Expectativa del Mercado de los dispositivos embebidos al 2020.

En la construcción de un producto de software y en la búsqueda de garantizar que esté libre de errores o fallos, el desarrollador de software embebido, debe ser capaz de aplicar las reglas de codificación en el desarrollo y revisión de un programa en el lenguaje utilizado, anteriormente, el que más se utilizaba anteriormente era el lenguaje C, con el auge ya comentado del internet de las cosas (IoT), han surgido nuevas tendencias en cuanto a lenguajes a utilizar.

Aún es muy importante elegir primero el hardware y luego seleccionar el lenguaje, Podemos optar por lenguajes clásicos que dominamos y obtienen buenos resultados o por lenguajes algo más específicos. Los más utilizados son C y C++, Java, JavaScript, Python, Go/Rust, Forth, Node-RED, OpenHAB, Google AppScript, ThinkSpeak. Las alternativas son muchas y algunos se están posicionando fuertemente en el mundo de Internet de las Cosas, Nijhawan (2016). Será importante evaluar los estándares o herramientas que se desarrollan en paralelo y que aportan valor agregado para garantizar que el código sea óptimo y libre de errores. Como el caso de los estándares MISRA para Cy C++ y la facilidad para definir pruebas unitarias en los ambientes de programación para la validación dinámica del software desarrollado.

Cada vez se liberan más normas de regulación de operación de los sistemas críticos. Aunque muchos de estos estándares están para satisfacer las necesidades de las industrias específicas, requieren procesos de desarrollo

similares para demostrar su cumplimiento. En la guía de Quality Metrix (2015) para medir la calidad del software, se establece lo siguiente:

El sistema embebido integrado, que incluye el hardware y el software debe cumplir ciertos estándares de acuerdo al tipo de aplicación, ya sea automotriz, industrial, militar o aeroespacial y también garantizar un índice de protección, por ejemplo IP-67, IP-68 o IP-69K, que tiene que ver con las condiciones físicas de operación de polvo, fluido u operación bajo agua. La norma IPC-2221B-1.6.2 clasifica a los sistemas embebidos en tres clases; productos electrónicos de uso general, productos electrónicos de servicio ininterrumpido y productos electrónicos con alto grado de confiabilidad.

La seguridad funcional está referida a procesar las normas de certificación de seguridad que se aplican a los sistemas embebidos para la validación. Por ejemplo, la IEC 61508 es un estándar que se ha adaptado a diferentes industrias, como la automotriz (ISO 26262) y los sistemas médicos (IEC 60601) y comparte similitudes a los estándares en la industria aeroespacial (DO-178B y DO-254). Además de los procesos del ciclo de vida que son mandatorios para obtener la certificación.

El proceso de pruebas de software (*testing*) para sistemas embebidos es más riguroso que para sistemas tradicionales, debido a la proximidad con el hardware (Kumar, 2012). Por este motivo la certificación del producto implica que se lleve a cabo bajo una metodología o ciclo de vida que garantice el cumplimiento de requerimientos.

A continuación se describe la propuesta para integrar algunas herramientas en la metodología para la construcción, validación y verificación de la funcionalidad de un programa para sistemas embebidos basada en buenas prácticas documentadas en el desarrollo de sistemas en diversos proyectos.

Para la gestión del proceso y para dar practicidad al desarrollo de software de sistemas embebidos y con la finalidad de minimizar los riesgos de implementar un producto que no corresponda con los requerimientos utilizamos Scrum para la gestión del proceso. Usaremos la definición de los tres roles, los dos artefactos y para los Sprints, hacemos una adaptación, ya que el proceso de desarrollo lo

haremos basados en el modelo en V para desarrollo de software. En realidad, el proceso en V contiene las etapas definidas en Scrum: análisis, diseño, código, integración y pruebas, con la ventaja de que las pruebas de verificación y validación toman un papel importante en cada una de las etapas del proceso. En la etapa de implementación usaremos TDD, o sea el desarrollo orientado mediante pruebas. Aplicamos los demás elementos del Sprint, la reunión de planeación (*sprint planning meeting*), las reuniones diarias para el seguimiento (*daily scrum meeting*) y la reunión para realizar la evaluación del sprint (*sprint retrospective*).

Etapa 0 (Establecimiento de roles). En esta etapa establecemos las personas que desempeñarán los roles de Scrum master (encargado de la comunicación y enfoque del equipo de trabajo), de desarrolladores (*team developers*) y el de *product owner* (el encargado de definir las funcionalidades del producto).

Etapa 1 (Planeación). Scrum propone como etapa inicial, llevar a cabo la reunión de planeación, con el propósito de tener un primer acercamiento con el cliente o con el encargado. El resultado de esta reunión será analizar que va a hacer el sistema, mostrar toda una lista de requerimientos y que le equipo de trabajo determine y priorice los requerimientos. Considerando que en sistemas embebidos se pueden utilizar diversos lenguajes y plataformas, esta etapa inicial la usaremos para establecer el entorno de desarrollo, definir el *backlog* en base al establecimiento de requerimientos, realizar la distribución de las historias de usuario convertidos a puntos de función, definir la arquitectura del sistema y Establecer el número de iteraciones para la realización total del producto. Esta etapa algunos autores la denominan "sprint cero".

Es recomendable que los equipos cuenten con una herramienta colaborativa en línea para la comunicación y administración de requerimientos e historias de usuarios, planificación y administración de Sprints, por ejemplo Jira. Esta herramienta permite la definición de *epics* o historias de usuario, mismas que se pueden descomponer en un conjunto de historias más pequeñas y derivar en puntos de función que se convertirán en tareas para los desarrolladores.

Etapa 2 (Realización de los "sprints" de incremento). Realizaremos tantos sprints como requerimientos funcionales tengamos que implementar, cada sprint iniciará

con la reunión de planeación (Scrum planning) cuyo objetivo será la priorización de los requerimientos que se realizará en el sprint en cuestión, la estimación de tiempos y la autoasignación para su desarrollo. Cuando se tienen personas con diferentes niveles de dominio de programación se recomienda la asignación por pares, ya que ha dado buenos resultados. La duración del sprint los define el equipo, pero en este tipo de proyectos de una a dos semanas es adecuado.

Análisis y definición de requerimientos. En esta etapa, el equipo de desarrollo, realiza para cada requerimiento asignado el diagrama de componentes, el diagrama de clases o de estructura modular y el diagrama de interfaz de comunicación o interfaz gráfica GUI y la definición de pruebas de aceptación, en la figura 2, se muestra una propuesta con los elementos que debe incluir una prueba de este tipo. Estableceremos primero, un escenario de inicio para la realización de la prueba y los cambios o perturbaciones al sistema indicando los resultados o comportamiento esperado. La definición de las pruebas de aceptación la llevan a cabo el encargado de realizar el requerimiento y el *product owner*. El conjunto de pruebas resultante constituirá el plan de prueba de aceptación.

Test Name: Test Case 3: Display list of all TICRS documents.
Description: Case Details screen tab will provide access to users to view or display list of all TCRS documents.
Requirement(s): TICRS-6.
Prerequisites: The user is logged on as an LIE/SLIE/BA.
Setup: Tester must point to Mock P drive in test environment. Verify that the most recent file is currently in Trademarks\FAST 2.1\BIN\Fast Application.exe of the current CM Build. Create a desktop shortcut of the FAST executable file from Mock P and launch the FAST exe.
Map to the TICRS drive(s):

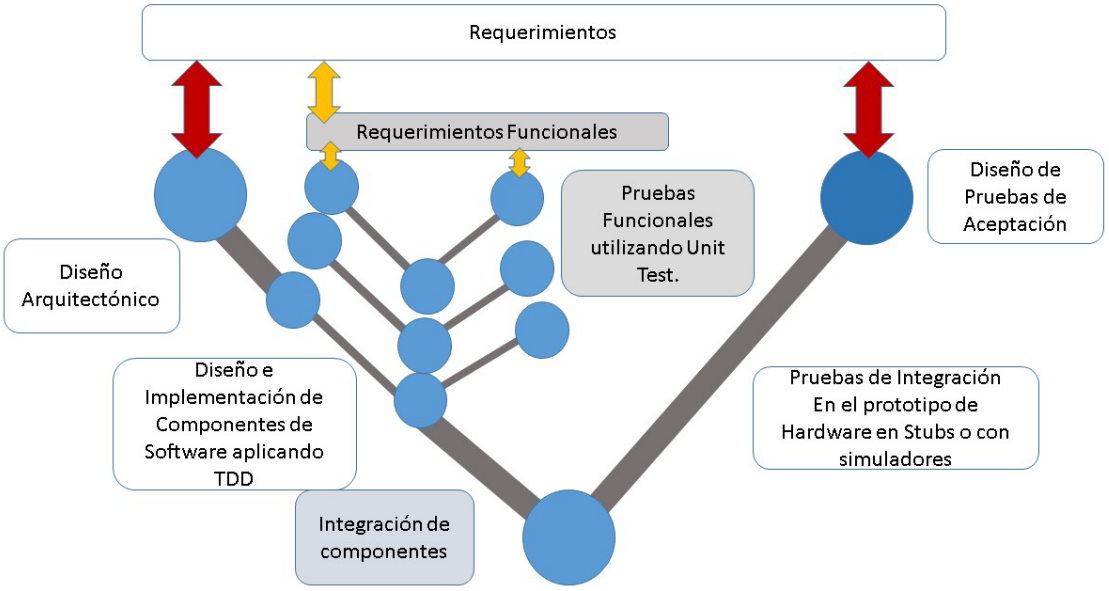
Step	Operator Action	Expected Results	Observed Results	Pass/ Fail
1.	Logon to system as an SLIE user with valid user ID.	User should be able to log in the system and Workload screen appears for SLIE.		
2.	Click on the 'Work List' tab.	The 'Work List' screen will be appear, also the Suspensions cases category page display by default.		
3.				
4.				
5.				

Fuente: https://goo.gl/ft6UW3

Figura 2 Ejemplo de estándar para documentar pruebas de aceptación.

En el modelo en V, iteramos entre el análisis, diseño y la prueba, como se muestra en la figura 3, para que nuestro software vaya evolucionando incrementalmente. En esta etapa incorporamos más nivel de detalle a los elementos del análisis,

podemos incluir de diagramas E-R en caso de requerir manejo de archivos o acceso a base de datos, así como diagramas de estados y transiciones. La comunicación entre los diversos módulos del sistema, por ejemplo un módulo de comunicación que recibe los datos de un sensor y los envía al módulo de procesamiento, seguramente los estarán realizando diferentes desarrolladores, por lo que es necesario establecer un plan de pruebas de integración. Aquí radica la importancia de establecer en forma documental los parámetros de entrada y los efectos esperados o respuestas de cada componente de software.



Fuente: basado en Selin y Sandmann, 2011

Figura 3 Proceso en V para desarrollo de Software.

Etapa 3 (Diseño de pruebas unitarias). En esta etapa, muchos están pensando en la codificación de funciones o de métodos si usamos programación orientada a objetos. Antes de ellos, debemos diseñar las pruebas que guiarán el desarrollo. Es importante utilizar técnicas de caja blanca, caja negra o ambas conocidas como pruebas de caja gris para diseñar el número de pruebas necesarias para garantizar la operación correcta de las funciones o métodos en el dominio del problema. Todo el código que implementa una funcionalidad debe tener una prueba unitaria, por lo tanto es fundamental iniciar los programas con un enfoque de TDD, esta estrategia de las pruebas unitarias conocidas como RED (fallo) -

GREEN (éxito), es especialmente útil en equipos de desarrollo ágil. Para implementarlo se siguen los pasos mostrados en la figura 4. El conjunto de pruebas unitarias resultante queda en la plataforma de desarrollo para ser ejecutadas automáticamente cuando se hagan modificaciones al código o lo que llamamos refactorización de código, agilizando las pruebas de regresión.

1. Escribir el código del test (**Stub**) para que compile (pase de **RED** a **GEEN)**
   a. Inicialmente la compilación fallará **RED** debido a que falta código.
   b. Implementar sólo el código necesario para que compile **GREEN** (aún no hay implementación real).
2. Escribir el código del test para que se **ejecute** (pase de **RED** a **GREEN)**. Inicialmente el test fallará **RED** ya que no existe funcionalidad.
   a. Implementar la funcionalidad que se va a probar hasta que se ejecute adecuadamente y pase a **GREEN.**

b. Refactorizar el test y el código una vez que este todo en **GREEN** y así la solución se va optimizando.

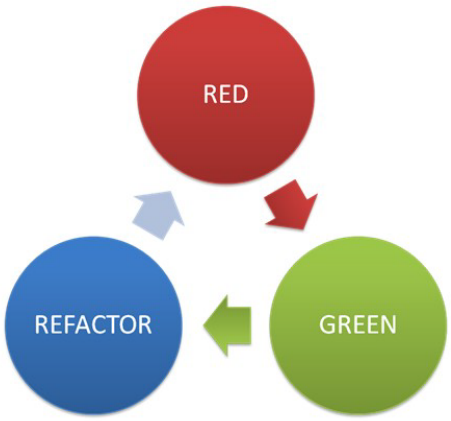


Figura 4 Paradigma de Desarrollo Orientadas por Pruebas (TDD).

Etapa 4 (Implementación o codificación). En esta etapa se codifican las funciones o métodos en el lenguaje elegido. Se recomienda utilizar una plataforma que permita la incorporación de las pruebas unitarias como parte del proyecto de desarrollo. En plataformas como Eclipse y Microsoft Visual Studio existen *pluggins* para permitir el desarrollo de programas en C/C++ para dispositivos embebidos. Una buena herramienta que podría dar muy buenos resultados es PolarSys, esta herramienta es soportada por un grupo de trabajo para la industria de Eclipse, para colaborar en la creación y el apoyo de herramientas de código abierto para el desarrollo de sistemas embebidos. Netbeans también ofrece buenas facilidades para implementar TDD, tanto en Java como en C y C++. En la figura 5, se presenta un ejemplo para definir una prueba unitaria en Netbeans, para el caso de la funcionalidad cuyo objetivo es mapear una señal de voltaje a su valor de temperatura y en la figura 6, se muestra la ejecución de la prueba unitaria en su

etapa inicial y final donde se aprecia que el código implementado pasa todas las pruebas.

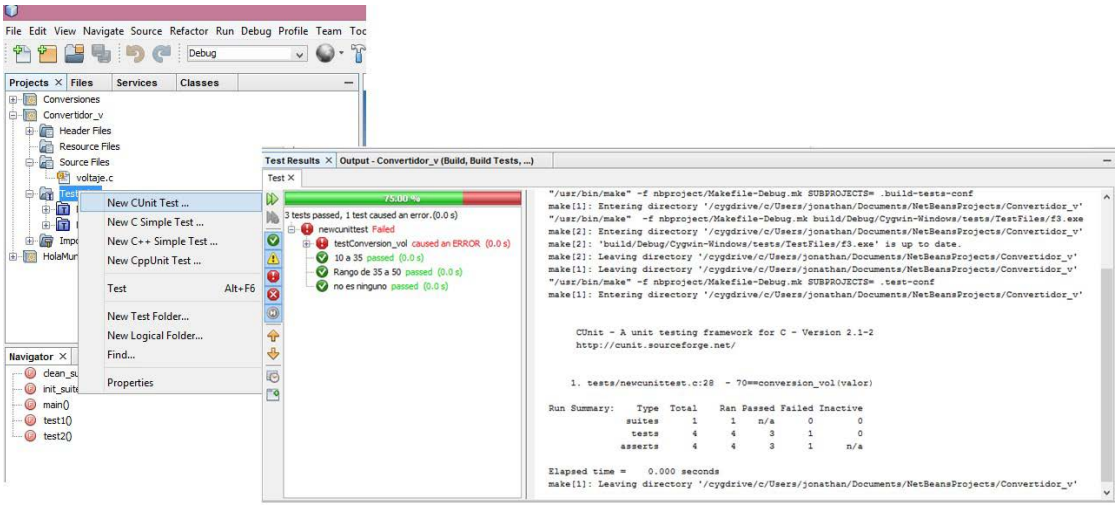


Figura 5 Ejemplo de fase inicial para crear prueba unitaria en Netbeans.

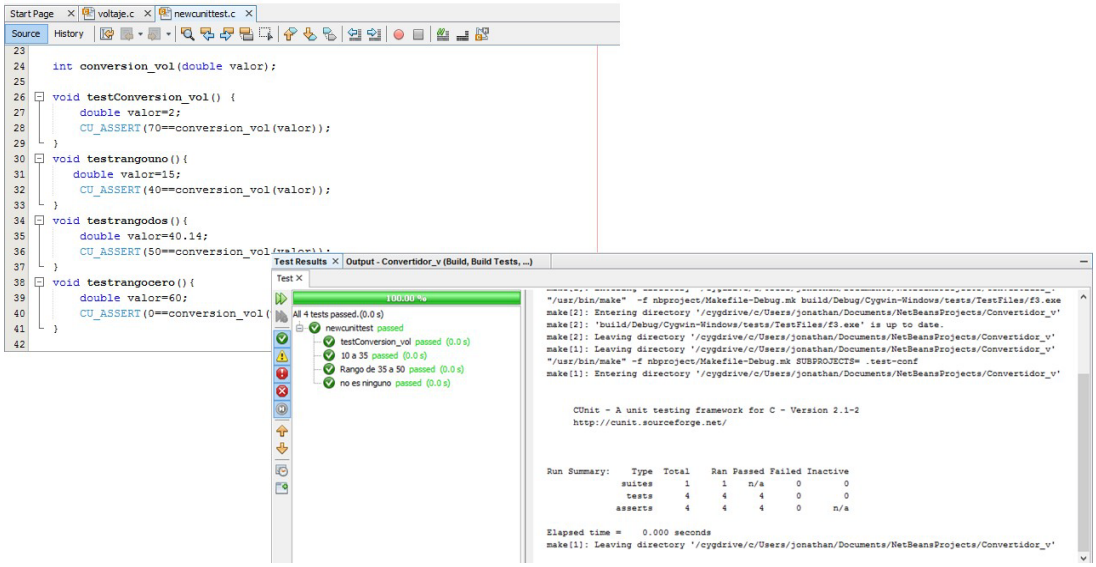


Figura 6 Ejemplo de etapa final de implementación de prueba unitaria en Netbeans.

Pruebas estáticas. En cualquier momento de la codificación se pueden estar ejecutando pruebas de caja negra o *Black Box*. Se recomienda hacerlo al término de la codificación de cada función o método para garantizar que se respetan las buenas prácticas y estándares de programación, si es lenguaje C o C++, recomendamos usar la herramienta MISRA-C de las siglas en inglés (*Motor*

Industry Software Realiability Association) .MISRA-C, incorpora una serie de estándares de codificación, para sistemas de software construidos usando lenguaje C y C++. Introduce una serie de buenas prácticas de diseño y reglas de codificación. MISRA, la primera versión fue publicado en 1998 y las versiones posteriores en 2004 y fue escrito para sistemas de seguridad críticos. Algunas de las reglas de MISRA las verifica el mismo compilador, pero otras hacen un análisis más preciso de situaciones en donde se especifica que ciertas situaciones no están definidas o son comportamientos no contemplados en las reglas de sintaxis del lenguaje (Davis, 2016). En la figura 7, se muestran algunas características de esta herramienta.

Fuente: https://www.misra.org.uk

Figura 7 El estándar MISRA-C.

Etapa 5 (Sprint de liberación o reléase). Se realizarán pruebas de Integración con otros módulos de software de acuerdo al comportamiento descrito en el documento de especificación de pruebas entre módulos. Se pone a operar el sistema, en un emulador, se realizan pruebas de acuerdo a los escenarios previstos y si todo va bien registra el cumplimiento de la prueba. En caso contrario, se levanta un "*bug*", conocida así como una situación de prueba, donde se registran las condiciones y se inserta como un ticket en el sistema utilizado para la gestión del desarrollo, que debe ser atendido por el desarrollador que programó la funcionalidad que presenta la falla.

Pruebas de Aceptación del software. Estas pruebas se consideran cuando probamos el software completo en el Hardware del sistema embebido o en emulador del sistema real. Se usan diferentes maneras de probar un sistema embebido, dependiendo de las situaciones particulares del contexto donde operará. La técnica de la simulación, implica generar modelos matemáticos del sistema real y crear un escenario en el que el sistema embebido interactuará. Otra técnica es la generación de prototipos, en la que se diseña un prototipo y se lleva al escenario real y en forma iterativa se va ajustando hasta obtener el comportamiento deseado. En algunos casos se usa la construcción de prototipos mediante el uso de plataformas para simular el comportamiento del sistema embebido.

Varias empresas de diferentes sectores para probar software para productos embebidos utilizan simuladores o modelos que representen el sistema real donde va a operar el software y se incluyen también simuladas las posibles fallas del sistema para analizar el comportamiento del software en estas condiciones. La mayoría de las empresas que desarrollan software para autos usa este tipo de tecnología. Los simuladores son útiles para probar el software en condiciones de riesgo en los que difícilmente se puede poner a funcionar el producto final y sobre todo ahorran dinero (Davis, 2016). La liberación o reléase se completa cuando el sistema ha pasado todas las pruebas de aceptación se libera el producto de software y entra en la etapa de producción y posteriormente mantenimiento. De esta manera, estaremos construyendo software embebido confiable. El establecimiento de este proceso de software de desarrollo con esquemas riguroso de *testing* (pruebas) y revisiones sobre la especificación de requerimientos nos garantiza un producto confiable.

3. Resultados

La metodología se validó mediante la construcción de un software para un sistema automático para la regulación de calidad del aire en un sistema colector de polvo. Este sistema es un prototipo para la industria para mejorar la calidad del aire en una cámara. A través de sensores se genera una señal de calidad de aire

del aire en la cámara y en función de ciertos parámetros se acciona un ventilador para limpieza de filtros. También registra en tiempo real una bitácora en web el estado de la variable controlada. Para el desarrollo se formaron dos equipos de trabajo de cinco personas cada uno, un *Scrum Master* y un *Product Owner*.

El proyecto inicia con la reunión preliminar con el *Product Owner*. Esta persona es parte del equipo de trabajo, pero es el que conoce más el proceso industrial y el hardware en el que se va a implementar. Se dio prioridad a la elaboración del módulo de comunicación de control-redundancia-cíclica (CRC) y extrajeron los requerimientos de las historias del usuario. Para el CRC, se generó una lista de cuatro requerimientos. Mismos que se distribuyeron entre los integrantes del primer equipo. Durante el diseño se definieron 21 pruebas unitarias, para verificar la transmisión de datos. A continuación se muestran ejemplo de requerimientos:

- REQ_01. Procesar la señal del sensor de calidad del aire e interpretarla y encender un led dependiendo del estado de la calidad del aire, alta (rojo) o normal (verde) de acuerdo a un parámetros establecido en la configuración del sistema.
- REQ_02. El programa para la comunicación entre los dispositivos el protocolo debe verificar la transmisión y recepción de datos, para garantizar la confiabilidad. Debe detectar posibles errores en el mensaje enviado y reenviar en caso necesario.

Para cumplir el REQ_02 el equipo decide implementar cuatro funciones: Send, Send_IB, Sen_var y Crc_cal_value, para calcular el CRC basado en polinomio generador de grado r, que se implementa tanto en el receptor como en el transmisor.

Sprint 0. Se definió el ambiente de programación y se configuró. Se utilizó Netbeans 8.0, para C /C++ y, debido a que tanto el CUnit como el plugin de C/C++ corren en un ambiente de Linux se utilizó Cygwin. También se requirió la versión de GNU de la utilería Make.

Sprint 1. Análisis y Diseño de Pruebas Unitarias Para cada componente. En esta etapa se establecieron con claridad la operación de cada función y las pruebas

unitarias necesarias. Por ejemplo para la función Send() se definieron las siguientes casos de prueba:

- Habilitar el modo de recepción de datos en el puerto G.
- Desactivar las interrupciones de la UART 2.
- Activar las interrupciones del Timer 2.
- Obtener el valor de la variable y escribirlo en el puerto 2.
- Producir un retardo de 1.3 milisegundos.
- Enviar el CRC, deshabilitando el modo de recepción de datos en el puerto G.
- Terminado el envío del CRC, activar las interrupciones de la UART.

En la misma etapa, de acuerdo al modelo en V y en paralelo con la actividad anterior, los desarrolladores diseñaron las pruebas unitarias para verificar el código.

Para la función Send se diseñaron siete pruebas unitarias. En la etapa de implementación los desarrolladores se enfocan a implementar en el lenguaje C la funcionalidad y en paralelo se implementan las funciones “stub” que simulan el efecto o la acción del microprocesador y la codificación de pruebas Unitarias que servirán para hacer la validación. Un ejemplo de diseño e implementación de la prueba unitaria se muestra en la figura 8. En la etapa de Ejecución de Pruebas se realizó lo siguiente:

- Ejecución de pruebas dinámicas o pruebas unitaria. Una vez implementado el código, se valida que el sistema pasa las 22 pruebas unitarias. Para ello se construye una tabla con el requerimiento y el conjunto de pruebas que validan que el programa realiza lo que debe hacer.
- Se analizaron los archivos de código utilizando el estándar de MISRA C 2004. Se analizaron las fallas al estándar y se hicieron las adecuaciones necesarias.

Sprint de Integración de código. Se integró el código probado en un ambiente diferente a Netbeans que se utilizó por la facilidad de realizar las pruebas

unitarias. Se migró el código al ambiente de programación para el microprocesador CCS PIC Compiler, para habilitar las funciones propias del PIC. Se generó el archivo hexadecimal y como los requerimientos se verificaron con las pruebas unitarias realizadas, el esfuerzo se enfocó en integrar el código y compilarlo. Es relevante destacar que la elección de un ambiente de programación que integre las librerías y funciones del microprocesador en cuestión es altamente deseable. Queda en puerta la posibilidad de probarlo con ambientes como Eclipse Embedded.

Id	Descripción	Entrada	Resultado Esperado
Data_value_puerto2	Valida el valor de la variable data_value y que se escriba ene l puerto 2	*CRC_Sent_UG_1 =0	*CRC_Sent_UG_1 = 0xAAAA *Puerto_UG_1 = 0x02

```
Data_value[0] = 0xFFFF;
CRC_Send_UG-1 = 0;
Puerto_UG_2 = 0;
Send();
If (/* check result */)
{
    CU_ASSERT(CRC_Sent_UG_2 == 0xAAAA)
    CU_ASSERT(CRC_UG_2 ==0x02 )
}
```

Figura 8 Ejemplo de diseño e implementación de prueba unitaria en el caso de aplicación.

4. Discusión

El desarrollo de software para sistemas embebidos está aumentando considerablemente y contar con una metodología para la validación y pruebas durante el proceso de desarrollo de software nos permite por un lado, garantizar que el producto es confiable, y por otro al establecer como marco de desarrollo SCRUM, nos permite agilizar el desarrollo, obteniendo el producto mejor enfocado a satisfacer la experiencia del usuario. Utilizar herramientas de estandarización internacional, nos permite pensar en la posibilidad de certificar el producto realizado, lo que otorga un valor agregado.

En esta investigación se hizo un análisis de diversas referencias bibliografías en el tema, se encontraron técnicas y herramientas muy útiles para aplicar en la verificación del código fuente de un producto de software. Hemos elaborado un

marco metodológico que arroja buenos resultados en cuanto al tiempo de producción del software y sobre todo puede acarrear menos problemas al momento de la integración del código elaborado por los elementos de un equipo de trabajo, impactando positivamente en la confiabilidad del producto de software.

El software testing no debe ser una fase separada sino integrarse en el desarrollo al igual que la programación. Están apareciendo nuevos lenguajes de programación, pero la elección del lenguaje habrá que tomarlo con precaución y probarlos exhaustivamente antes de usarlos para aplicaciones críticas. Algunos factores importantes en la elección del software son, el que podamos ejecutar pruebas unitarias para la automatización de pruebas que apoyen cuando se hace refactorización de código o se implementan nuevas funcionalidades que implican realizar pruebas de regresión ya que esto nos ahorra una gran cantidad de tiempo y nos evita introducir errores en código ya probado, que en la vida real es muy común.

En el caso práctico que utilizamos para probar la metodología, fue fundamental el compromiso el desempeño de los involucrados de acuerdo al rol establecido para gestionar el producto con SCRUM y el proceso en V para la construcción del software, así como el desarrollo guiado por pruebas lo que nos llevó a obtener un software con un alto grado de calidad verificado al momento de integrarlo con los demás componentes.

Una excelente alternativa para probar productos de software es la construcción de modelos que representen o modelen el sistema real donde va a operar el software y se incluyan también modeladas las posibles fallas del sistema para analizar el comportamiento del software en estas condiciones. La mayoría de los desarrolladores de software para autos usa este tipo de tecnología.

5. Bibliografía y Referencias

[1] Charette, R. N. (2005). Why Software Fails. Institute of Electrical and Electronics Engineers (IEEE). https://goo.gl/jaOn.

[2] Cortés, P. (20/ago/2007). La Naturaleza del Software. http://www.lnds.net /blog/2007/08/c2bfpor-que-falla-el-software.html.

[3] Davis, G., Green, H. (2013, marzo). Software, build secure and reliable embedded systems with MISRA C/C++. www.goo.gl/wv7UZT.

[4] Fariña, A. (2015). Nuevas técnicas de inyección de fallos en sistemas embebidos mediante el uso de modelos virtuales descritos en el nivel de transacción. Universidad de Alcalá. Departamento de Automática. http://hdl.handle.net/10017/22748

[5] Infotechnology (2014, 13 de febrero). Por fallas de software, Toyota llama a Revisión a 1,9 Millones de Vehículo híbridos. www.goo.gl/14Xr26.

[6] Kumar, S. (2012, abril). "Software Testing for Embedded Systems". En International Journal of Computer Applications 0975-8887), Vol. 43. No.17.

[7] Nijhawan, R. (2016, 16 noviembre). Explain Non Functional Testing With Example. https://goo.gl/xJ8rTx .

[8] Quality Metrics (2015, 4 de noviembre). A guide to measuring software quality. https://goo.gl/aqzxWW.

[9] Selin, D. & Sandmann, G. (2011). Model-Based Design based on AUTOSAR in an Electrical Systems Engineering Environment at Volvo Cars. https://goo.gl/K7flMr.

[10] Sommerville, I. (2011). Ingeniería de Software. 9ª ed. México: Pearson.

[11] Srinivas N. & Jagruthi D. (2012, junio). Black Box and White Box Testing techniques - a Literature Review. International Journal of Embedded Systems and Applications (IJESA) Vol.2, No.2.

[12] Yoshida, M., Tada, O. (1996). Programming of the Engine Control Unit by the C Language. SAE Technical Paper 960047.