International Seminar on Civil Engineering an Academic Research Training and Internationalization of Engineering

María Leisy González Carrera[1], Cesar González Hernández[1], Yurenia Hernández Blanco.[1]
[1] Universidad de las Ciencias Informáticas, Cuba, mlcarrera@uci.cu, cgonzalezh@uci.cu, yhblanco@uci.cu

***Abstract*– Software testing is a critical element for ensuring software quality and customer satisfaction. These require a set of activities that are planned and carried out systematically. The Free Software Center of the University of Information Science has as its main tasks to develop the Cuban distribution of GNU / Linux Nova and advising companies and institutions during their process of migration to free software and open source. To support these processes is performed hmast development tool, which facilitates the management of telematics services. This paper describes the tests applied to the tool, to detect vulnerabilities and errors introduced in implementation.**

***Keywords*—quality assurance, software testing, free software**

Digital Object Identifier (DOI): http://dx.doi.org/10.18687/LACCEI2015.1.1.172
ISBN: 13 978-0-9822896-8-6
ISSN: 2414-6668

Experiencias en la aplicación de pruebas del software a la herramienta HMAST

María Leisy González Carrera[1], Cesar González Hernández[2], Yurenia Hernández Blanco[1]
[1]*Centro de Software Libre. Universidad de las Ciencias Informáticas, Carretera a San Antonio de los Baños, km 21/2, Torrens, Boyeros, La Habana, Cuba. CP.: 19370. mlcarrera@uci.cu*
[2]*Dirección de Informatización. Universidad de las Ciencias Informáticas, Carretera a San Antonio de los Baños, km 21/2, Torrens, Boyeros, La Habana, Cuba. CP.: 19370*

Las pruebas del software son un elemento crítico para el aseguramiento de la calidad del software y la satisfacción del cliente. Estas requieren de un conjunto de actividades que se planifican y se llevan a cabo sistemáticamente. El Centro de Software Libre de la Universidad de las Ciencias Informáticas tiene entre sus principales misiones desarrollar la distribución cubana de GNU/Linux Nova y asesorar a empresas e instituciones durante sus procesos de migración a software libre y código abierto. Para apoyar estos procesos se lleva a cabo el desarrollo de la herramienta HMAST, la cual facilita la administración de los servicios telemáticos. La presente investigación describe las pruebas aplicadas a la herramienta, para detectar vulnerabilidades y errores introducidos en la implementación.

Palabras clave: aseguramiento de la calidad, pruebas del software, software libre

I. Introducción

La prueba del *software* representa la visión final de las especificaciones, del diseño y de la codificación. Esta es conocida como verificación y validación. La verificación se refiere al conjunto de actividades que aseguran que el *software* implementa correctamente una función específica. La validación se refiere a un conjunto diferente de actividades que aseguran que el *software* construido se ajusta a los requisitos del cliente [1].

La Herramienta para la Migración y Administración de Servicios Telemáticos (HMAST) está constituida por un sistema base y los módulos DHCP, MySQL, SSH, Apache y Bacula. El sistema base se centra en dos funcionalidades principales: la gestión de los usuarios del sistema y la gestión de los servidores con los módulos a administrar. El módulo *DHCP* permite la administración de este servicio desde la herramienta, mediante la administración de: las configuraciones generales, los ámbitos dinámicos y las asignaciones estáticas. También permite visualizar los registros de las asignaciones estáticas a los clientes. El módulo *MySQL* ofrece acceso a las principales funcionalidades dirigidas a la administración de este servicio, como son: las configuraciones generales y avanzadas, la gestión de registros y la configuración de los parámetros de seguridad. El módulo *SSH* permite las siguientes funcionalidades: gestionar la autenticación y conexión del servicio, la administración las llaves públicas, la gestión de los usuarios y grupos permitidos a conectarse por dicho servicio y gestionar las configuraciones avanzadas.

El módulo *Apache* gestiona los servidores virtuales, el servidor principal y las conexiones del servicio. Además permite gestionar los módulos de multiprocesamiento. El módulo *Bacula* permite gestionar la configuración general del servicio, muestra y edita las configuraciones del gestor de copias, permite instalar y desinstalar el servicio en los clientes. También posibilita al administrador las gestiones de: los conjuntos de archivos, las programaciones de copias de seguridad, las copias de seguridad, las agrupaciones de volúmenes, los dispositivos y configuraciones de almacenamiento. Otras funcionalidades del módulo son: ejecutar y detener las copias de seguridad, mostrar estado y reporte de las copias de seguridad, mostrar qué trabajos están corriendo o finalizados en el gestor de copias de seguridad, y listar los archivos y directorios salvados en determinada copia de seguridad.

Todos los módulos implementan las funcionalidades de instalar, desinstalar, iniciar, reiniciar y detener el servicio. Otras funcionales son: mostrar el estado del servicio y aplicar, comprobar y descartar los cambios en las configuraciones de los servicios. El módulo *SSH* no realiza las funcionalidades de instalar, iniciar y desinstalar el servicio porque para poder conectarse a una PC el servicio SSH instalado, de no ser así no se realizará la conexión.

Debido a la importancia que tiene la herramienta HMAST en la ejecución de los procesos de migración, la presente investigación tiene como objetivo principal: describir mediante ejemplos el proceso de pruebas aplicado a la herramienta HMAST, de acuerdo a las estrategias de pruebas del *software* definidas por Roger Pressman.

II. Descripción Del Proceso De Pruebas

Las pruebas del *software* componen el conjunto de actividades que originan los procesos capaces de detectar los errores en un *software*. Todo sistema que se construye necesita de las pruebas del *software* aplicándose por cada nivel de las mismas para su correcta ejecución. Cada nivel de prueba contiene métodos y técnicas las cuales están diseñadas con el objetivo de encontrar posibles errores del *software* que no han sido descubiertos hasta entonces.

13th LACCEI Annual International Conference: "Engineering Education Facing the Grand Challenges, What Are We Doing?"
July 29-31, 2015, Santo Domingo, Dominican Republic **ISBN:** 13 978-0-9822896-8-6 **ISSN:** 2414-6668
DOI: http://dx.doi.org/10.18687/LACCEI2015.1.1.172

Válido aclarar que el proceso de pruebas se realizó teniendo en cuenta los criterios y conceptos definidos por Roger S. Pressman. A continuación se describen las técnicas, los métodos, las estrategias y herramientas utilizadas en el proceso de ejecución de pruebas a la herramienta HMAST.

A. *Técnicas de prueba del software*

El diseño de pruebas para el *software* requiere tanto esfuerzo, como el propio diseño inicial del producto, pero se debe diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible. Las técnicas de prueba del *software* son las pruebas de caja blanca y las pruebas de caja negra.

1) Prueba de Caja Blanca

La prueba de caja blanca, denominada a veces prueba de caja de cristal, es un método de diseño de casos de prueba. Mediante este método se puede obtener casos de pruebas que garanticen que se ejercite por lo menos una vez todos los caminos independientes de cada módulo, que se ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa, que se ejecuten todos los bucles en sus límites y con sus límites operacionales y que se ejerciten la estructuras internas de datos para asegurar su validez [2].

2) Prueba de Caja Negra

La prueba de caja negra, denominada a veces prueba de comportamiento, se centra en los requisitos funcionales del *software*. Esta prueba intenta encontrar errores de las siguientes categorías: funciones incorrectas o ausentes, errores de interfaz, errores de estructura de datos o en accesos a bases de datos externas, errores de rendimiento y errores de inicialización y de terminación [3].

B. *Métodos de diseños de casos de pruebas*

Toda técnica de diseño de casos de pruebas contempla métodos que apoyan el diseño de cada caso de prueba. Los métodos utilizados en la aplicación de las pruebas a la herramienta HMAST son los que se detallan a continuación:

1) Prueba del Camino Básico

La prueba del camino básico es una técnica de prueba de caja blanca. Este método permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución [2].

2) Partición Equivalente

La prueba de partición equivalente es un método de prueba de caja negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Esta prueba se basa en una evaluación de las clases de equivalencia para una condición de entrada. Una clase equivalente representa un conjunto de estados válidos o no válidos para condiciones de entrada. Típicamente, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica [4].

3) Análisis de Valores Límite

La prueba de análisis de valores límite es una técnica de prueba de caja negra que complementa a la técnica de prueba de partición equivalencia. Esta prueba lleva a la elección de casos de prueba en los extremos de la clase [4].

C. *Estrategias de prueba del software*

Una estrategia de prueba del *software* integra las técnicas de diseño de los casos de prueba en una serie de pasos bien planificados que dan como resultado una correcta construcción del *software*. Cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de las pruebas y la evaluación de los datos resultantes. Además, esta debe incluir pruebas de bajo nivel que verifiquen que todos los pequeños segmentos de código fuente se han implementado correctamente, así como pruebas de alto nivel que validen las principales funciones del sistema frente a los requisitos del cliente.

1) Prueba de Unidad

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del *software*: el componente *software* o módulo. Durante la prueba de unidad, la comprobación selectiva de los caminos de ejecución es la tarea esencial. Esta hace uso intensivo de las técnicas de prueba de caja blanca [5].

2) Prueba de Integración

La prueba de integración es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados con la interacción. El objetivo es coger los módulos probados mediante la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño. Durante la integración, las técnicas que más prevalecen son las de diseños de casos de prueba de caja negra, aunque se pueden llevar a cabo algunas pruebas de caja blanca con el fin de asegurar que cubren los principales caminos de control.

2.1) Integración Descendente

La prueba de integración descendente es un planteamiento incremental a la construcción de la estructura de programas. Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal. Los módulos subordinados al módulo de control principal se van incorporando en la estructura, bien de forma primero en profundidad, o bien de forma primero en anchura [6].

2.2) Prueba de Regresión

La prueba de regresión comienza cada vez que se añade un nuevo módulo como parte de la prueba de integración, el *software* cambia. Se establecen nuevos caminos de flujo de datos, pueden ocurrir nuevas entradas y salidas, y se invoca una nueva lógica de control. Estos cambios pueden causar problemas con funciones que antes trabajaban perfectamente. El objetivo es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo

anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados [6].

3) Prueba de Validación

La prueba de validación del *software* se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos.

3.1) Prueba de Aceptación

La prueba de aceptación se realiza cuando se construye un *software* a medida para un cliente, permitiendo al mismo validar todos los requisitos. La realiza el usuario final en lugar del responsable del desarrollo del sistema [7].

4) Prueba del Sistema

La prueba del sistema está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Todas estas pruebas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

4.1) Prueba de Seguridad

La prueba de seguridad intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho y de accesos impropios. El responsable de la prueba debe intentar conseguir las claves de acceso por cualquier medio, puede atacar al sistema con *software* a medida diseñado para romper cualquier defensa que se haya construido. También debe bloquear el sistema negando el servicio a otras personas, debe producir a propósito errores del sistema intentando acceder durante la recuperación o debe curiosear en los datos sin protección intentando encontrar la clave de acceso al sistema [8].

4.2) Prueba de Resistencia

La prueba de resistencia ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. El responsable de la prueba intenta romper el programa [8].

4.3) Prueba de Rendimiento

La prueba de rendimiento está diseñada para probar el rendimiento del *software* en tiempo de ejecución dentro el contexto de un sistema integrado. Esta prueba, a menudo, va emparejada con las pruebas de resistencia y, frecuentemente, requieren instrumentación tanto de *software* como de hardware [8].

D. Herramientas utilizadas

Las herramientas empleadas en el proceso de ejecución de las pruebas fueron:

JMeter para analizar las prestaciones de los recursos dinámicos y estáticos, tales como objetos Java, bases de datos y para emular grandes niveles de concurrencia sobre el servidor [9].

JUnit para asegurar que los métodos individuales del código funcionen correctamente. El soporte integrado del IDE NetBeans para la unidad de JUnit framework de pruebas, permite rápida y fácilmente crear pruebas JUnit y suites de prueba [10].

Nikto es un escáner de vulnerabilidades web, de tipo Open Source, con licencia GPL. Entre otras características: soporta SSL y una configuración para usarlo a través de un proxy. Comprueba la existencia de elementos desfasados en un servidor. Tiene la posibilidad de exportar un informe en formato csv, htm, xml. Permite escanear los puertos de un servidor a través de Nmap[1]. El tipo de escaneado se puede modificar o "tunear" para excluir las vulnerabilidades que no interesen [11].

III. Aplicación y Evaluación de los Resultados

Luego de haber descrito las técnicas, los métodos, los niveles de pruebas y las herramientas a utilizar para la ejecución del proceso de pruebas, en este epígrafe se abordará la aplicación de las pruebas y la evaluación de los resultados.

A. Prueba de Unidad

Las pruebas de unidad se realizaron a la herramienta HMAST utilizando la técnica prueba de caja blanca y el método de prueba del camino básico. Además, la herramienta JUnit del IDE NetBeans permitió la realización de las pruebas automáticas y además luego de haber diseñado los casos de prueba se pudo ejecutar con apoyo de JUnit las pruebas unitarias. A continuación se describe un ejemplo de un método reutilizable en todos los módulos de la aplicación, teniendo en cuenta los siguientes pasos:

1) Crear el grafo de flujo

Para crear el grafo de flujo del método se necesita asignar números enteros por cada sentencia del código (ver Fig. 1). Luego se dibuja el grafo de flujo (ver Fig. 2) del procedimiento de acuerdo a cada condición con los números resultantes.

```
public static boolean isBetweenMaxMinInteger
            (String value,String min, String max) {
  1 if(!StringValidator.isEmpty(value) 2
        3 && !StringValidator.isEmpty(min)
        4 && !StringValidator.isEmpty(max)){
        if(NumberValidator.isAInteger(value)5
            6 && NumberValidator.isAInteger(min)
            7 && NumberValidator.isAInteger(max)){ 8
            if( Integer.parseInt(value) >= Integer.parseInt(min)
                9 &&  Integer.parseInt(value) <= Integer.parseInt(max)){
                return true; 10
            }
        }
    }
    return false; 11
} 12
```

Fig. 1 Código del método

[1] *Nmap* ("mapeador de redes") es una herramienta de código abierto para exploración de red y auditoría de seguridad.

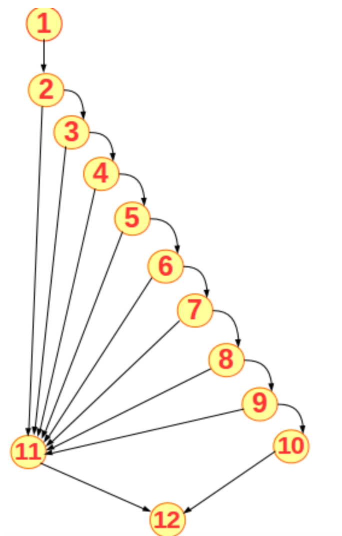

Fig. 2 Grafo de flujo del método

2) Determinar la complejidad ciclomática del grafo de flujo resultante

La complejidad ciclomática del grafo de flujo puede ser calculada de tres formas [1]:

- La complejidad ciclomática, V (G), de un grafo de flujo G se define como V (G) = R, donde R es el número de áreas delimitadas por los nodos y las aristas.
- La complejidad ciclomática, V (G), de un grafo de flujo G se define como V (G) = A – N + 2, donde A es el número de aristas del grafo flujo y N es el número de nodos del mismo. Los círculos del grafo se denominan nodos y las flechas del grafo se denominan las aristas.
- La complejidad ciclomática, V (G), de un grafo de flujo G se define como V (G) = P + 1, donde P es el número de nodos predicados contenidos en el grafo de flujo G. Cada nodo que contiene una condición se denomina nodo predicado y está caracterizado porque dos o más aristas emergen de él.

El grafo de flujo (ver Fig. 2) está constituido por doce nodos, diecinueve aristas, ocho nodos predicado y nueve regiones delimitadas por los nodos y las aristas. Con los datos anteriores se puede obtener por las tres formas la complejidad ciclomática del grafo. Sustituyendo los datos en las tres fórmulas, se concluye que la complejidad ciclomática del grafo de flujo es nueve (V (G) = 9).

3) Determinar el número de caminos linealmente independientemente

El valor resultante de la complejidad ciclomática indica la cantidad de caminos linealmente independientemente. Los caminos son:

Camino 1: 1-2-11-12
Camino 2: 1-2-3-11-12
Camino 3: 1-2-3-4-11-12
Camino 4: 1-2-3-4-5-11-12
Camino 5: 1-2-3-4-5-6-11-12
Camino 6: 1-2-3-4-5-6-7-11-12
Camino 7: 1-2-3-4-5-6-7-8-11-12
Camino 8: 1-2-3-4-5-6-7-8-9-11-12
Camino 9: 1-2-3-4-5-6-7-8-9-10-12

4) Preparar los casos de prueba de cada camino resultante

Se recogieron los datos, de forma que las condiciones de los nodos predicado están adecuadamente establecidas, con el fin de probar cada camino.

Caso de prueba del camino 1:

Entrada inválida: value = null
Entrada inválida: min = null
Entrada inválida: max = null
Resultado esperado: falso
Resultado obtenido: falso

En este caso de prueba *value* no puede ser nulo, por lo cual no se sigue ejecutando la sentencia de la condición (primer if) y el método devuelve falso.

Caso de prueba del camino 2:

Entrada válida: value = "Hola"
Entrada inválida: min = null
Entrada inválida: max = null
Resultado esperado: falso
Resultado obtenido: falso

En este caso de prueba el *value* es una cadena de caracteres, por lo cual se sigue ejecutando la sentencia de la condición. Cuando se verifica la segunda condición (primer if) *max* es nulo, por lo cual no se sigue ejecutando la condición y el método devuelve falso.

Caso de prueba del camino 3:

Entrada válida: value = "Hola"
Entrada válida: min = null
Entrada inválida: max = "Mundo"
Resultado esperado: falso
Resultado obtenido falso

En este caso de prueba *value* y *max* contienen una cadena de caracteres, por lo cual se sigue ejecutando la sentencia de la condición. Cuando se verifica la tercera condición (primer if) *min* es nulo, por lo cual no se sigue ejecutando la condición y el método devuelve falso.

Caso de prueba del camino 4:

Entrada inválida: value = "Hola"
Entrada válida: min = "Mundo"
Entrada válida: max = "Prueba"
Resultado esperado: falso
Resultado obtenido: falso

En este caso de prueba *value* no puede ser una cadena de caracteres constituida por letras, por lo cual no se sigue ejecutando la sentencia de la condición (segundo if) y el método devuelve falso.

Caso de prueba del camino 5:

Entrada válida: value = 1
Entrada inválida: min = "Mundo"
Entrada inválida: max = "Prueba"
Resultado esperado: falso
Resultado obtenido: falso

En este caso de prueba *value* es un número entero, por lo cual se sigue ejecutando la sentencia de la condición. Cuando se verifica la segunda condición (segundo if) *max* es una cadena de caracteres constituida por letras, por lo cual no se sigue ejecutando la condición y el método devuelve falso.

Caso de prueba del camino 6:
Entrada válida: value = 1
Entrada inválida: min = "Mundo"
Entrada válida: max = 2
Resultado esperado: falso
Resultado obtenido falso

En este caso de prueba *value* y *max* contienen números enteros, por lo cual se sigue ejecutando la sentencia de la condición. Cuando se verifica la tercera condición (segundo if) *min* es una cadena de caracteres constituida por letras, por lo cual no se sigue ejecutando la condición y el método devuelve falso.

Caso de prueba del camino 7:
Entrada válida: value = 1
Entrada válida: min = 2
Entrada válida: max = 0
Resultado esperado: falso
Resultado obtenido: falso

En este caso de prueba *value* es menor que *min*, por lo cual no se sigue ejecutando la sentencia de la condición (tercer if) y el método devuelve falso.

Caso de prueba del camino 8:
Entrada válida: value = 1
Entrada válida: min = 0
Entrada válida: max = 0
Resultado esperado: falso
Resultado obtenido: falso

En este caso de prueba *value* es mayor que *min*, pero mayor *max*, por lo cual no se sigue ejecutando la sentencia de la condición (tercer if) y el método devuelve falso.

Caso de prueba del camino 9:
Entrada válida: value = 1
Entrada válida: min = 0
Entrada válida: max = 10
Resultado esperado: verdadero
Resultado obtenido: verdadero

En este caso de prueba, *value* es mayor que *min* y menor *max*. El método devuelve verdadero, esto quiere decir que *value* se encuentra en el intervalo de *min* y *max*. La herramienta JUnit se utiliza para ejecutar los casos de prueba automáticamente una vez que se ejecute la aplicación en el Netbeans. Los errores detectados de manera general en la aplicación durante el proceso de ejecución de las pruebas unitarias fueron:

- algunas de las expresiones regulares establecidas no están implementadas correctamente. Ejemplo: la expresión implementada en el método *isAIpAddress*.
- existen condiciones lógicas incorrectas en los métodos y mal posicionadas.
- el cálculo para convertir de KB a GB es incorrecto. Este error fue introducido en la implementación de la fórmula.
- en algunos casos los mensajes no están en correspondencia con los errores detectados.
- no se muestra el mensaje de error cuando que se compara un valor entero con una cadena. Este error fue encontrado en uno de los métodos de la implementación de la herramienta.

B. *Prueba de Integración*

Las pruebas de integración descendiente se realizaron con la técnica de caja negra, y los métodos partición equivalente y análisis de valores límites. Los módulos subordinados al sistema base se fueron incorporando a la estructura de la forma primero en anchura. La integración primero en anchura, incorpora todos los módulos directamente subordinados a cada nivel, moviéndose por la estructura de forma horizontal. El sistema base y los módulos de la herramienta HMAST, como se puede apreciar en la Fig. 3 fueron sustituidos por M1 hasta M6, donde M1 es el sistema base y de M2 hasta M6 son los módulos DHCP, MySQL, SSH, Apache y Bacula respectivamente. Cuando se realizó la integración se tuvo en cuenta probar todas las funcionalidades, pero el requisito de prioridad alta *adicionar módulo al servidor* no puede pasar por alto, debido a la importancia que tiene en las pruebas de integración. Este requisito permite al usuario ver la relación claramente del módulo correspondiente con el sistema base. La interfaz de adicionar módulo en el servidor contiene un solo campo de entrada que requiere un miembro de un conjunto. El tipo de campo es una lista de los módulos adicionar donde el administrador debe seleccionar el módulo que desea añadir.

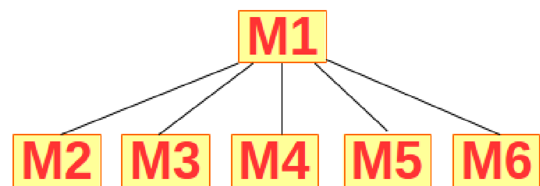

Fig. 3 Integración descendiente de forma primero en anchura

Al realizar cada prueba de integración descendente para cada módulo se realizaron pruebas de regresión las cuales permitieron ejecutar un subconjunto de pruebas que se habían llevado a cabo con anterioridad. Para la realización del caso de prueba con el método de partición equivalente se tuvieron en cuenta los siguientes aspectos:

- la interfaz de adicionar módulo al servidor está constituida por el atributo *Lista de módulos* y el campo de dicho atributo puede tomar los siguientes valores: *DHCP, MySQL, Bacula, Apache o SSH,*
- la condición de entrada asociada al atributo es el valor que está definido en la lista de selección,

- la clase de equivalencia válida es seleccionar un módulo, ejemplo *DHCP*,
- la clase de equivalencia no válida es no seleccionar un módulo y seleccionar la opción *Aceptar*. En este caso se le muestra un mensaje de error al administrador indicándole que debe seleccionar un módulo.

Las no conformidades detectadas de manera general en la aplicación durante el proceso de ejecución de las pruebas de integración fueron:

- cuando se adicionó el módulo *Bacula* no se redireccionó para la página de la funcionalidad *Instalar* ya que el servicio no estaba instalado y no redireccionó para la página principal si ya se encuentra instalado el servicio,
- no se actualiza la tabla de la lista de los módulos que han sido adicionados al servidor,
- cuando se instala, no se puede ver las configuraciones generales del servicio,
- luego de corregir la anterior no conformidad se detectó que los vínculos del menú lateral que tienen relación con las configuraciones generales no funcionan ya que las páginas todavía no tenían la nueva estructura del sistema base,
- al cerrar la pestaña de trabajo del servicio y no salir de la aplicación, si el administrador desea acceder al área del servicio se le indica que el módulo no está activo en el sistema.

C. Prueba de Validación

Las pruebas de aceptación fueron ejecutadas en un entorno controlado con la presencia de los desarrolladores. Estas pruebas fueron realizadas al sistema base y a los módulos DHCP y MySQL en la primera versión liberada de la herramienta HMAST. Además se realizaron con la técnica de caja negra, y los métodos partición equivalente y análisis de valores límites. Un ejemplo de caso de prueba (ver TABLA I) aplicado en la herramienta HMAST fue la funcionalidad *adicionar una asignación estática*. El uso del método partición equivalente permitió trabajar con cada atributo como si fuera una clase equivalente y para completar la aplicación de la prueba anterior se utilizó el método de análisis de valores límites, empleando los atributos *Tiempo de aceptación* y *Máximo tiempo* donde se probó valores justo por encima y justo por debajo del valor máximo y mínimo respectivamente que pueden tomar dichos atributos.

TABLA I
CASO DE PRUEBA: ADICIONAR UNA ASIGNACIÓN ESTÁTICA

Caso de Prueba de Aceptación	
Código Caso de Prueba: HMAST_DHCP_15_57	**Nombre Historia de Usuario:** Gestionar asignaciones estáticas del servicio DHCP
Nombre de la persona que realiza la prueba: Amaury Viera Hernández (cliente)	
Descripción de la Prueba: Probar que no se adicione la asignación estática del servicio DHCP introduciendo valores inválidos.	
Condiciones de Ejecución: El usuario debe estar autenticado en el sistema. Debe existir al menos un servidor adicionado en la aplicación con el servicio DHCP instalado.	
Entrada / Pasos de ejecución: 1. Seleccionar la opción *Asignaciones estáticas* del menú lateral izquierdo de la aplicación. 2. Luego seleccionar la opción *Adicionar* en la parte superior derecha del sistema. 3. Introducir algunos o todos los valores incorrectos en la interfaz. Los campos son: Nombre: I(//////////) Nombre de la máquina: I(·····45$$) Tipo hardware: I(Seleccione) Dirección física: I(00:1c0:41) Dirección IP: I(10.53.3.22m) Tiempo de aceptación: I(10000000000) Máximo tiempo: I(-1) 4. Seleccionar la opción *Aceptar*.	
Resultado Esperado: El sistema muestra los campos en rojo donde se introdujeron los valores incorrectos y los mensajes correspondientes a los errores.	
Evaluación de la Prueba: Satisfactoria	

Las pruebas de aceptación se realizaron en tres iteraciones (ver TABLA II), donde se detectaron un total de diez no conformidades, de las cuales, nueve eran de impacto alto, medio y bajo respectivamente, y una de recomendación. Las no conformidades detectadas de manera general fueron:

- no se actualiza la lista de módulos en el instante que se adicionó el módulo al servidor,
- la aplicación permite enviar correo con campos vacíos que son obligatorios,
- el sistema permite adicionar un servidor que ya existe, con el mismo nombre y dirección *IP*,
- la aplicación no permite editar ni eliminar a los usuarios a partir de diez en la lista de usuarios,
- se muestra un error al tratar de iniciar el servicio *MySQL* en el servidor,
- se instala el servicio *MySQL* pero lanza un error al concluir la instalación, y dicho mensaje debe ser escrito en el idioma de Español.

TABLA II
NO CONFORMIDADES

Iteración	No. C	Imp. Alto	Imp. Medio	Imp. Bajo	R
1	9	3	2	3	1
2	1		1		
3	0				

Total	10	3	3	3	1

D. Prueba del Sistema

Luego de realizar las pruebas unitarias y de integración hay que asegurarse que estén correctamente implementados los mecanismos de la seguridad y conocer la cantidad de trabajo que puede soportar el sistema.

1.1) Prueba de Seguridad

Esta prueba se ejecutó con la ayuda de la herramienta Nikto en Linux. En el ejemplo que se muestra en la Fig. 4, se realizaron las siguientes actividades:

- se escanearon los directorios CGI mediante el parámetro *-Cgidirs*,
- se revisó la base de datos en busca de errores de sintaxis a través del parámetro *-dbcheck*,
- se habilitó la detección de intrusos mediante técnicas de evasión con el uso del parámetro *-evasion 15*. Donde *1* es la codificación aleatoria URI (no-UTF8) y *5* es el parámetro falso,
- se definió el formato de salida *htm* a través del parámetro *-Format*, utilizando *-o* para nombrar el reporte a generar,
- se especificó la técnica de mutación mediante el parámetro *-mutate 235*. Donde *2* adivina los nombres de archivo de contraseña, *3* enumera los nombres de usuario a través de Apache y *5* intenta de forzar los nombres de subdominios,

localhost / 127.0.0.1 port 8080/hmast	
Target IP	127.0.0.1
Target hostname	localhost
Target Port	8080/hmast
HTTP Server	Jetty(9.2.2.v20140723)
Start Time	2015-03-02 11:15:43
Site Link (Name)	http://localhost:8080/hmast/
Site Link (IP)	http://127.0.0.1:8080/hmast/

URI	/
HTTP Method	HEAD
Description	Number of sections in the version string differ from those in the database, the server reports: jetty(9.2.2.v20140723) while the database has: 40.7.2.2.41. This may cause false positives.
Test Links	http://localhost:8080/hmast/ http://127.0.0.1:8080/hmast/
OSVDB Entries	OSVDB-0

Nikto Scan Summary	
Software Details	Nikto 2.1.4
CLI Options	-h localhost:8080/hmast -Cgidirs -dbcheck -Display 1234DEPV -evasion 15 -Format htm -o resultado1.htm -mutate 235 -Tuning 0123456789abc
Hosts Tested	1
Start Time	Sun Mar 1 11:15:43 2015
End Time	Sun Mar 1 11:16:54 2015
Elapsed Time	71 seconds

Fig. 4 Pruebas de seguridad

- se utilizó el parámetro *-Tuning 0123456789abc* para reducir el número de pruebas realizadas contra un objetivo. Donde *0* permite que un archivo sea cargado en el servidor de destino, *1* un archivo o un ataque desconocido que se ha visto en los registros del servidor web, *2* por defecto los archivos mal configurados o protegidos con una contraseña errónea, *3* divulgación de información, *4* inyección (*XSS / Guion / HTML*), *5* recursos que permiten a los usuarios remotos recuperar los archivos no autorizados en el directorio raíz del servidor web, *6* recursos que permiten la denegación de servicio contra la aplicación de destino, el servidor web o el host, *7* permite a los usuarios remotos recuperar archivos no autorizados desde cualquier punto, *8* recursos que permiten a los usuarios ejecutar un comando del sistema o iniciar una *shell*, *9* cualquier tipo de ataque que permite inyección SQL, *a* permite al cliente acceder a un recurso que no tiene acceso, *b software* o programas instalados podrían ser identificados positivamente y *c* el *software* permite la inclusión remota de código fuente. La vulnerabilidad detectada es que existe incompatibilidad entre la versión del controlador de aplicaciones del servidor *jetty* (9.2.2.v20140723) con la versión del gestor base de datos (40.7.2.2.41).

1.2) Las pruebas de Resistencia y Rendimiento

Las pruebas se ejecutaron en un ambiente con las siguientes características: para poder acceder al sistema el servidor debe estar corriendo, la computadora contiene 1TB de disco duro y 1GB de RAM, la red está conectada a 100 Mbps y se utilizó la herramienta JMeter v2.12 para GNU/Linux. En la TABLA III muestran los resultados obtenidos en 5 pruebas realizas para *5*, *50* y *100* usuarios que acceden a la página de inicio del sistema tras realizar la autenticación. En cada una de las pruebas se registra el rendimiento que representa la cantidad de peticiones por segundo que es capaz de atender el sistema.

Los resultados muestran que con la configuración descrita anteriormente el sistema tiene un mejor comportamiento para una carga de *50 usuarios*, donde la media de peticiones atendidas por segundo es de 38.98.

TABLA III
PRUEBAS DE RESISTENCIA Y RENDIMIENTO

Usuarios	R1	R2	R3	R4	R5	Media
5	30.09	38.59	40.08	39.20	38.15	**37.38**
50	42.45	40.21	37.61	39.09	35.55	**38.98**
100	38.67	36.00	38.29	39.90	38.36	**38.24**

III. CONCLUSIONES

Las experiencias adquiridas en el proceso de ejecución de pruebas a la herramienta HMAST permitieron realizar una descripción explícita y elocuente de los métodos, técnicas, estrategias de pruebas y las herramientas utilizadas. Se diseñó para cada nivel de prueba los casos de pruebas correspondientes. El uso de las herramientas JMeter, JUnit y Nikto facilitó la ejecución de las pruebas de rendimiento y resistencia, unitarias y de seguridad respectivamente. Durante las pruebas se detectaron no conformidades significativas para su posterior corrección mejorando la calidad del sistema.

REFERENCIAS

[1] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 306, La Habana 2005.
[2] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 286, La Habana 2005.
[3] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 294, La Habana 2005.
[4] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 296-297, La Habana 2005.
[5] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 310-311, La Habana 2005.
[6] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 312-315, La Habana 2005.
[7] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 316, La Habana 2005.
[8] Roger S. Pressman, "Ingeniería de Software" Un enfoque práctico, 5ª ed., parte 1, pp. 317-318, La Habana 2005.
[9] JMeter. The Apache Software Foundation. http://jmeter.apache.org
[10] JUnit. Writing JUnit Tests in NetBeans IDE. https://netbeans.org/kb/docs/java/junit-intro.html
[11] Nikto. Intro al escáner de web Nikto. http://www.securityartwork.es/2014/02/26/escaner-de-webs-nikto/