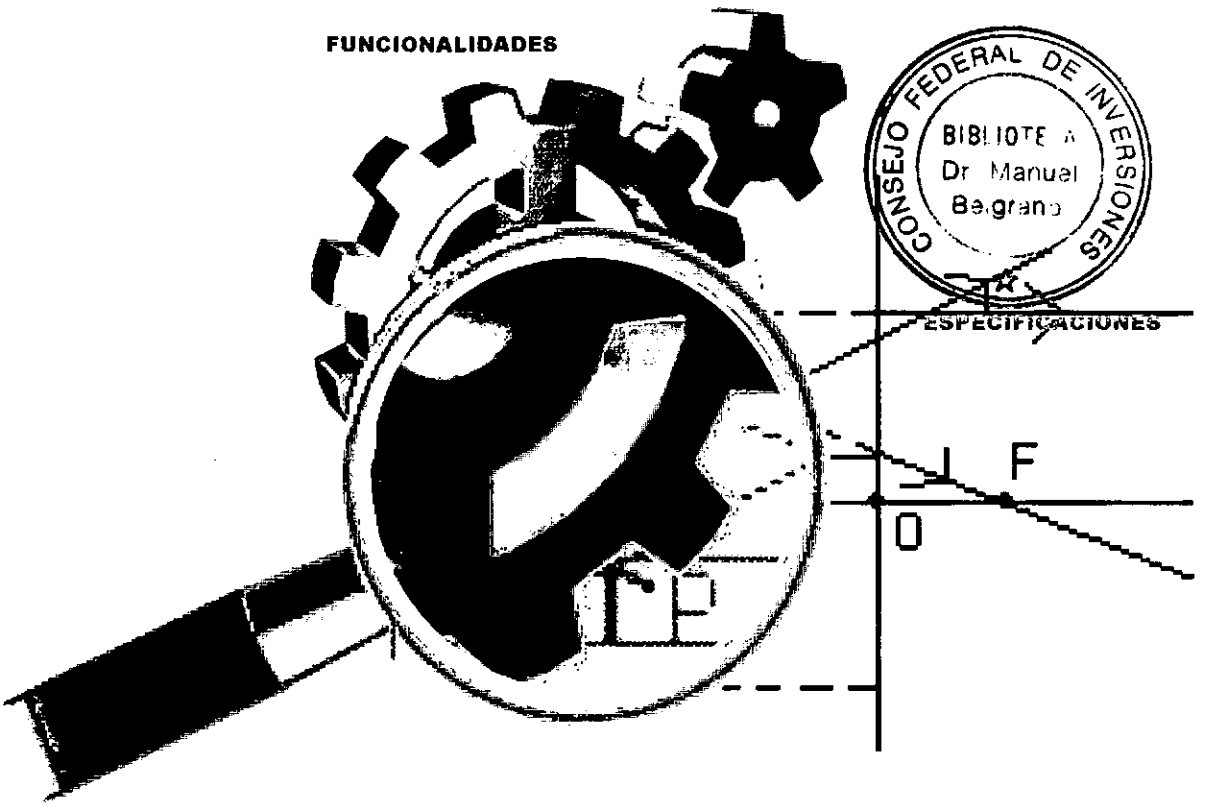
" LA CALIDAD EN EL PROCESO DE TESTING DE SISTEMAS "



CFI - 2002

PROGRAMA:

" SAN LUIS – GESTIÓN PARA LA IMPLEMENTACIÓN

DE UNA PROVINCIA DIGITAL "

PROYECTO:

" LA CALIDAD EN EL PROCESO

DE TESTING DE SISTEMAS "

ÍNDICE

RESUMEN GERENCIAL DE PROYECTO

El objetivo de este proyecto consiste en proveer al Gobierno de la Provincia de San Luis requisitos mínimos, lineamientos y documentación necesaria para generar una estrategia de testing de aplicativos metódicamente, permitiendo llevar a cabo los procedimientos de testing en forma adecuada y coherente, maximizando la cantidad de errores encontrados y obteniéndose así un producto final más confiable y de mayor calidad.

En la actividad 1 se observa el producto de una intensa investigación de la cual surge el desarrollo conceptual de los procedimientos teóricos necesarios para llevar a cabo el testing de aplicativos. Entre ellos podemos mencionar: definiciones, principios y atributos de la prueba de sistemas, métodos de prueba de software (pruebas de caja negra, pruebas de caja blanca, pruebas no funcionales), niveles de prueba, diseño de casos de prueba, prueba de sistemas web, etc.

En la actividad 2 se detalla un conjunto elaborado de requisitos mínimos, estrategias y procedimientos adecuados a las necesidades de la S.E.T.I., contemplando, entre otras cosas: la planificación de los procesos de prueba, plan de pruebas, estrategia, alcance, niveles de prueba, diseño, elaboración y ejecución de los casos de prueba, mecánica de la prueba, análisis de riesgo, criterios de finalización de las pruebas, ejecución del plan de pruebas, seguimiento de las mismas, etc.

En la actividad 3 se describe un conjunto de plantillas que sirven de base para la generación de la documentación de los resultados obtenidos de la ejecución

de los procedimientos e implementación de las estrategias de testing generadas en este proyecto. Las mismas sirven para: el registro, revisión y aprobación de casos de prueba, captura de pantallas y seguimiento de errores.

En la actividad 4 se muestra la aplicación en un caso práctico de testing sistema basado en las técnicas y estrategias generadas en las actividades anteriores. Para llevar a cabo el mismo la S.E.T.I. propuso que se realizara un testing funcional sobre tres módulos del Sistema de Capacitación en su versión alfa.

INTRODUCCIÓN

La implementación del proyecto "Autopista de la Información" en la Provincia de San Luis tiene como objetivo mejorar la calidad de vida del pueblo sanluiseño mediante el uso de las tecnologías de la información. En este marco, la Secretaría de Estado de Tecnologías de la Información ha contratado el desarrollo de aplicativos para diversas áreas.

El chequeo de presencia de errores, o testing, es una etapa indispensable en todo proceso de desarrollo de software. Hasta el momento, el Gobierno de la Provincia de San Luis realizaba el proceso de testing de los desarrollos y aplicativos contratados en forma prácticamente artesanal. Es inconcebible que esta situación perdure teniendo en cuenta que la puesta en producción de los aplicativos de la Autopista de la Información afecta a múltiples áreas del Gobierno en una forma sin precedentes.

La necesidad de un marco eficaz para ser utilizado en el testing de cada uno de los aplicativos hace evidente que debe sistematizarse el proceso.

El resultado del proceso de testing debe reflejar, como mínimo, si el software cumple con los requerimientos del Gobierno de la Provincia de San Luis, comportándose según lo esperado e interactuando correctamente con los otros sistemas y con tiempos de respuesta aceptables.

Considerando la magnitud y el número de sistemas contratados, la complejidad natural de las interacciones demandadas y que los procesos de prueba de software culminan con la aceptación de los mismos, surge la necesidad de

replantear las estrategias de testing. Las estrategias resultantes deben basarse en un sustento metodológico preciso y ordenado, ya que sin el mismo, la calidad del proceso de testing se ve deteriorada, generando consecuencias no deseadas en los procesos de auditoría del proyecto.

La ejecución de este proyecto brindará un trabajo de intensa investigación que permitirá generar estrategias y lineamientos necesarios para lograr procesos de testing confiables y coherentes donde se maximice la cantidad de errores encontrados.

OBJETIVO ESPECÍFICO

Proveer los requisitos mínimos y documentación necesaria para generar una estrategia de testing de aplicativos contemplando, entre otras cosas: la planificación de los procesos de prueba, la elaboración y ejecución de los casos de prueba y, finalmente, la evaluación de los resultados.

INFORME DEL DESARROLLO DE LA ACTIVIDAD N° 1

1 ENUNCIADO DE LA ACTIVIDAD

Investigación y desarrollo conceptual de los procedimientos y pasos a llevar a cabo en el testing de aplicativos.

En esta actividad se elaborarán, entre otros:

- Introducción a la verificación y validación de software.
- Definiciones y conceptos.
- Descripción de las fases del proceso de pruebas.
- Explicación de la importancia de la planificación de las pruebas.

2 DESARROLLO

La prueba de software es una actividad que de una u otra manera es llevada a cabo en algún momento al menos por su desarrollador original. Para ello, el equipo de desarrollo puede planear y realizar un proceso de prueba enfocado a un ambiente en el que espera operará su producto. Aunque esto es relativamente aceptable, puede no ser suficiente. Muchas veces el ambiente de operación pensado por el equipo de desarrollo puede diferir del ambiente que se presenta para la persona que utiliza finalmente dicho de software, por lo que en la prueba desarrollada se pueden haber ignorado algunas situaciones.

Lo ideal para llevar a cabo el testing de software es contar con un equipo de prueba de software (ver punto 2.8.1) que realice la misma a lo largo de todas las etapas del desarrollo (análisis, especificación, diseño, implementación y mantenimiento), planteando la estrategia adecuada que se debe aplicar en cada una de ellas.

La prueba de software es una actividad necesaria que ayuda a establecer la calidad de éste. La importancia de la misma se ve reforzada con el paso de los años convirtiéndose en un factor fundamental, llegándose a plantear que la prueba de software se aplique en cada una de las etapas del desarrollo como algo necesario.

En general, se puede decir que la prueba de software permite determinar si el producto generado satisface las especificaciones establecidas. Así mismo, una prueba de software permite detectar la presencia de errores que pudieran generar salidas o comportamientos inapropiados durante la ejecución del mismo.

Debemos tener en claro y diferenciar testing o prueba de software de depuración de software (debugging); el primero tiene como objetivo encontrar o detectar errores y el segundo busca el problema que los generó. La confusión surge porque cuando se encuentra un error se busca el por qué del mismo para resolver el problema. El límite entre testing y debugging es que en el testing por cada vez que se encuentra un error se debe registrar formalmente en un documento (en la Actividad 3 se generarán la plantillas para tal fin) y una vez finalizado el mismo, el equipo de desarrollo utiliza los resultados del testing,

registrados en el documento llevado a cabo para tal fin, para realizar la depuración de los errores.

En esta actividad se presentan técnicamente los conceptos teóricos sobre la prueba de software en general, orientado a objeto, basado en componentes y web. La misma no pretende desarrollar toda la teoría de testing de sistemas, sino más bien, dar los conceptos que forman la base para plantear lineamientos confiables de una estrategia de testing que sean coherentes y que maximicen la cantidad de errores encontrados.

2.1 PRUEBA DE SISTEMAS

Una de las actividades asociadas al desarrollo de software es el proceso prueba o "testing" del mismo. Debido a la importancia que ha adquirido el testing de software se ha establecido que el mismo es una actividad fundamental dentro de cada una de las etapas del proceso de desarrollo.

La prueba es indispensable, puesto que a partir de ella se puede determinar la calidad de los productos implementados; a pesar de esto, no es difícil percibir como su importancia se ha subestimado y en ocasiones hasta ignorado.

La **prueba o testing de software** se define como: "*Una actividad en la cual un sistema o componente es ejecutado bajo condiciones especificas, se observan o almacenan los resultados y se realiza una evaluación de algún aspecto del sistema o componente*". Aunque una definición más usada para la prueba de software es: "*El proceso de ejecutar un programa con el fin de encontrar errores*".

Cuando se habla de condiciones específicas, en la definición anterior, se puede suponer la presencia de una especie de ambiente de operación de la prueba, para el cual deben existir determinados valores para las entradas y las salidas, así como también ciertas condiciones que delimitan a dicho ambiente de operación. Todo lo mencionado se plasma sobre un plan de pruebas.

El **Plan de prueba** establece las estrategias, recursos y cronograma de ejecución de las pruebas. La **estrategia** se refiere al tipo de pruebas, sus objetivos, nivel de análisis y porcentajes de los resultados aceptables sobre las pruebas que se llevarán a cabo en cada etapa del desarrollo e implementación de los sistemas y componentes.

Las pruebas de software crecen a medida que se progresa en la creación del software hasta que se adquiera una estabilidad en las evoluciones del ciclo de vida, lo que significa, un gran esfuerzo en las últimas fases. Debido a esto, es indispensable mantener el modelo de pruebas a lo largo del ciclo de vida del software, aunque sea necesario cambiar dicho modelo debido a modificaciones o refinamiento natural que sufre el software en la creación.

El **modelo de prueba** describe como se van a probar los componentes ejecutables con distintos métodos de prueba, como de integración y sistema, y como han de ser probados aspectos del sistema. El modelo de pruebas está compuesto de *Casos de prueba*, *Procedimientos de prueba* y *Componentes de prueba*, los cuales serán explicados a continuación.

Se define un **caso de prueba** como: "*Un conjunto de entradas, condiciones de ejecución y resultados esperados diseñados para un objetivo particular*".

En un proceso de prueba de software, se pueden identificar las siguientes acciones:

I. *preparar una serie de casos de prueba,*

II. *llevar a cabo dichos casos de prueba,*

III. *decidir cuándo suspender la prueba,*

IV. *evaluar los resultados generados por la prueba,*

V. *emitir un criterio de evaluación.*

De las mismas surgen cuestionamientos tales como: ¿Cómo seleccionar casos de prueba representativos?, ¿Cuántas pruebas realizar? o bien ¿Cómo decidir si es o no de calidad el producto evaluado?, los cuales permiten entender que cada una de estas acciones requiere especial atención.

El caso de prueba especifica una forma de probar el sistema, incluyendo que probar, los datos de entrada y salida, y las condiciones de la prueba. Surge a raíz de un requerimiento o un conjunto de requerimientos cuya implementación justifica la realización de una prueba.

El **Procedimiento de prueba** especifica como realizar uno o más casos de prueba o parte de ellos. Tiene instrucciones precisas de como interactuar con un sistema o componente, o la forma de configurar un componente de prueba.

Es posible que un procedimiento de prueba sea combinado con uno o varios casos de prueba, y viceversa.

Un **componente de prueba** automatiza uno o más procedimientos de prueba.

Puede ser desarrollado utilizando cualquier lenguaje de programación o herramienta de automatización. Se utiliza para probar componentes mediante la provisión de los datos de entrada y evaluación de los datos de salida. Puede incluir el monitoreo de actividad entre los componentes que se prueban y la generación de reportes con los resultados de la prueba.

Un **defecto** representa una anomalía en el funcionamiento de los sistemas. Es utilizado para describir aquellos aspectos que los desarrolladores utilizarán para rastrear y corregir la falla.

Los términos tales como falla, equivocación y error, pueden considerarse como sinónimos, sin embargo, dentro del contexto de prueba de software no es prudente realizar esta suposición. Con el propósito de evitar confusiones y presentar conceptos básicos en materia de pruebas, se presentan estas definiciones:

a) Equivocación (mistake):

Acción del ser humano que produce un resultado incorrecto.

b) *Defecto o falta (fault):*

Proceso o definición de dato incorrecto en un programa de computadora. El resultado de una equivocación.

c) *Falla (failure):*

Resultado incorrecto. El resultado de una falla.

d) *Error (error):*

Magnitud por la que el resultado es incorrecto.

A continuación se definen dos conceptos que se deben tener bien en claro en la prueba de software ya que la misma forma parte de un concepto más amplio, referenciado generalmente como verificación y validación.

La **Verificación** se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. Ésta responde a la pregunta: ¿Se está construyendo el producto correctamente?

La **Validación** se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. Ésta responde a la pregunta: ¿Se está construyendo el producto correcto?, es decir, ¿Se ajusta a los requisitos del cliente?

La prueba confirma los niveles de calidad alcanzados en todas las etapas del ciclo de vida mediante la aplicación adecuada de métodos y herramientas.

Las estrategias actuales de verificación y validación se aplican a todas las etapas del ciclo de vida. Por ejemplo, los cuatro criterios básicos para la verificación y validación de las especificaciones de requisitos y de diseño son:

- **Completitud:** una especificación es completa si todas sus partes están presentes y cada parte está completamente desarrollada.
- **Consistencia:** interna (las especificaciones no deben entrar en conflicto unas con otras) y externa (las especificaciones no deben entrar en conflicto con especificaciones o entidades externas). Los detalles de las especificaciones deben tener claros antecedentes en especificaciones anteriores o en los objetivos del sistema.

- **Factibilidad:** una especificación es factible si los beneficios del sistema exceden sus costos.
- **Verificabilidad:** una especificación es verificable si se puede identificar una técnica económicamente factible que permita determinar si un software satisfará o no una especificación.

2.1.1 PRINCIPIOS PARA PROBAR SOFTWARE

Los siguientes son principios que han surgido de las mejores prácticas de pruebas de software:

- La definición del resultado esperado a la salida del programa es una parte integrante y necesaria del caso de prueba. Si el resultado esperado de la prueba no ha sido predefinido cabe la posibilidad que un resultado plausible pero erróneo se interprete como correcto.
- El grupo encargado de realizar la prueba de software debe ser, en lo posible, distinto del que realiza el desarrollo del mismo.
- Examinar a conciencia el resultado de cada prueba. Ocurre que errores que suelen aparecer casualmente son errores expuestos por los casos de prueba pero que se escaparon a la detección por falta de inspección.
- Evitar los casos de prueba desechables a menos que el programa sea verdaderamente desechable. Es una práctica sentarse frente al monitor, improvisar un caso de prueba y pasarlo al programa. El problema está en la pérdida de tiempo, ya que cuando se tenga que probar nuevamente (por

haberse corregido algún error) se tendrá que reinventar la prueba. El planeado y documentación de los casos de prueba juegan un papel importantísimo en el testing de software.

- Determinar cuándo finaliza la prueba, ya que no definir esto desde un comienzo trae muchos problemas.
- La probabilidad de encontrar errores adicionales en un módulo del programa es proporcional al número de errores ya encontrados en dicho módulo. Contrariamente a lo que la intuición indicaría, esto ha sido verificado en muchos programas.
- No alterar nunca el programa para que la prueba resulte más fácil.
- La prueba como cualquier otra actividad debe comenzar con la definición de sus objetivos.

2.2 ESTRATEGIAS DE PRUEBAS

Una estrategia de prueba de software integra las técnicas de diseño de casos de prueba en una serie de pasos planificados que dan como resultado una correcta construcción del software. La estrategia proporciona un plano o guía que describe los pasos a llevar a cabo en el testing de software, cuándo se deben planificar y realizar esos pasos, y cuánto esfuerzo, tiempo y recurso se van a requerir.

El Plan de trabajo para realizar las pruebas responde a las preguntas: qué, cuándo y cómo llevar a cabo la misma.

Una **estrategia de prueba** es un procedimiento que busca reducir los riesgos de cometer errores. Esta debe:

- Prever los riegos más peligrosos
- Señalar la forma de reducir los riesgos (tácticas de prueba)
- Distinguir los posibles riesgos dentro del proceso total

Entre las tácticas de prueba podemos mencionar: los planes de pruebas específicos, técnicas particulares, uso de herramientas, etc.

En resumen, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de los casos de prueba, la ejecución de las pruebas la agrupación y evaluación de los datos resultantes.

En la Actividad 2 del presente proyecto se desarrollará la estrategia de prueba de software que se adecue a las necesidades de la S.E.T.I.

Nota:

Los **riesgos** son condiciones que pueden conducir a pérdidas. Un riesgo es peligroso en la medida en que sea posible (probable).

Las **pruebas** tienen por objeto reducir la posibilidad de riesgos mientras que una táctica busca reducir la posibilidad de que un riesgo específico tenga lugar.

La **estrategia** es una planeación de tácticas que busca reducir los riesgos en el proceso de desarrollo.

2.2.1 FUNDAMENTOS DE LA PRUEBA DEL SOFTWARE

Los desarrolladores de software son por naturaleza constructivos mientras que lo que se busca con la prueba es buscar la forma de mostrar que tiene errores. En el testing se requiere que se descarten las ideas preconcebidas sobre la *corrección* del software que se acaba de desarrollar y se supere cualquier conflicto de intereses que aparezcan cuando se descubren errores. La prueba de software es uno de los pasos de la Ingeniería de Software que se puede ver como destructivo en lugar de constructivo, pero esto sólo es un punto de vista. Desde otra perspectiva más optimista, la prueba de software intenta de descubrir la máxima cantidad errores.

2.2.1.1 Objetivos de la Prueba de Software

La prueba del software es una fase clave del ciclo de vida de un producto. En ella se determina la disposición del producto para ser entregado basándose en criterios preestablecidos de correctitud y calidad. La actividad de verificación del software debe ser, por lo tanto, cuidadosamente planificada con el objeto de poder asegurar que la fase de prueba garantiza esos niveles de correctitud y calidad.

Existen dos enfoques a la hora de probar un software que no son auto-excluyentes y de hecho son complementarios:

- El hecho de realizar una prueba no garantiza la ausencia de defectos, sino solamente se demuestra la existencia de éstos.

- La prueba de software se realiza con el propósito de encontrar algo que difiera a las especificaciones planteadas para el producto o bien, para detectar la presencia de situaciones que pudieran generar resultados inapropiados.

Aunque ambos enfoques pueden orientar el sentido de una prueba, entre los objetivos principales de la prueba de software podemos mencionar:

I. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.

II. Un buen caso de prueba es aquél que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.

III. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Otra característica deseable del proceso de prueba es que sea capaz de encontrar diferentes clases de errores en la mínima cantidad de tiempo posible y con el mínimo esfuerzo.

2.2.1.2 Facilidad de Prueba

La facilidad de prueba del software tiene que ver con que si el software es probable o no y de serlo, cuan fácil de probar es.

Los siguientes puntos de comprobación permiten verificar si un software es fácil de probar:

Operatividad: Cuando mejor funcione, más eficientemente se puede probar.

Observabilidad: Lo que se ve es lo que se prueba.

Controlabilidad: Cuanto mejor se pueda controlar el software, más se puede automatizar y optimizar.

Capacidad de Descomposición: Controlando el ámbito de las pruebas, se puede aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión.

Si el sistema de software está construido con módulos independientes entonces los módulos del software se pueden probar independientemente.

Simplicidad: Cuanto menos haya que probar, más rápidamente podremos probarlo.

Estabilidad: Cuanto menos cambios, menos interrupciones a las pruebas.

- Los cambios del software son poco frecuentes.
- Los cambios del software están controlados.
- Los cambios del software no invalidan las pruebas existentes.
- El software se recupera bien de los fallos.

Facilidad de Comprensión: Cuanta más información se tenga, más inteligentes serán las pruebas.

- El diseño se ha entendido perfectamente.
- Las dependencias entre los componentes internos, externos y compartidos se han entendido perfectamente.
- Se han comunicado los cambios del diseño.
- La documentación técnica es instantáneamente accesible.
- La documentación técnica está bien organizada.
- La documentación técnica es específica y detallada.
- La documentación técnica es exacta.

2.2.1.3 Atributos de una Buena Prueba

Una buena prueba está caracterizada por cumplir los siguientes atributos:

I. Tiene una alta probabilidad de encontrar un error.

II. No debe ser redundante.

III. Debería ser la mejor de las pruebas realizables.

IV. No debería ser ni demasiado sencilla ni demasiado compleja.

2.2.2 FACTORES DE PRUEBA EN EL DESARROLLO DE SOFTWARE

Los factores de prueba son aquellos aspectos que deben ser aprobados como parte de una estrategia de prueba de software o corresponden a los riesgos que la estrategia considera:

- Riesgo: son aquellos factores que pueden producir resultados incorrectos, y
- Factor de prueba: correctitud.

Debemos tener en cuenta que no todo factor de prueba tiene que ser considerado sino que esto depende de la clase de software que se desarrolle.

Entre los factores de prueba más comunes podemos mencionar:

Correctitud: adecuación del software a las necesidades del usuario, es decir, si las especificaciones y los objetivos satisfacen al mismo. Para probar este factor debemos responder a la pregunta: ¿El sistema hace lo que se espera que haga de acuerdo a lo requerido?

Fiabilidad o confiabilidad: cantidad de fallas por período de tiempo. Definir qué son fallas y posibles tipos. Para probar este factor debemos responder a la pregunta: ¿El sistema es confiable todo el tiempo?

Eficacia: consumo de recursos que hará el sistema. Tiempos de respuesta a consultas específicas. Para probar este factor debemos responder a la pregunta: ¿Se ejecutará el sistema en mi hardware lo mejor posible?

Integridad: mantener datos coherentes pase lo que pase. Definir hasta qué punto el sistema detecta errores en los datos (inconsistencia). Para probar este factor debemos responder a las preguntas: ¿Es seguro el sistema?, ¿Es consistente?

Completitud: grado en que se han implementado las funciones requeridas. Para probar este factor debemos responder a la pregunta: ¿Se han implementado todos los casos de uso, módulos, funciones, etc.?

Testeabilidad: facilidad de probarlo. Para probar este factor debemos responder a la pregunta: ¿Se puede probar el sistema?

Flexibilidad: facilidad para cambiar componentes. Posibilidad de agregar o quitar módulos ante nuevas necesidades del usuario. Para probar este factor debemos responder a la pregunta: ¿Se puede cambiar el software?

Mantenibilidad: facilidad para mantenerlo. Calidad de la documentación, comentarios en fuentes, etc. Para probar este factor debemos responder a la pregunta: ¿El software se puede corregir?

Reutilización: reuso de los componentes para otros sistemas u otros eventos del sistema. Administrar correctamente los repositorios de componentes. Para probar este factor debemos responder a la pregunta: ¿Puedo reusar alguna parte del sistema?

Interoperabilidad: posibilidad de acoplarlo con otros sistemas. Para probar este factor debemos responder a la pregunta: ¿Puede interactuar el sistema con otro?

Acoplabilidad: conectividad entre módulos. Para probar este factor debemos responder a las preguntas: ¿Cuál es el grado de conectividad del sistema?, ¿Los módulos son independientes?

Auditabilidad: facilidad para comprobar las acciones que los usuarios llevan acabo en el sistema. Para probar este factor debemos responder a las

preguntas: ¿Permite auditar?, ¿Es auditable?, ¿Registra las acciones llevadas a cabo por los usuarios?

Facilidad de uso: esfuerzo requerido para aprender, manejar e interpretar lo que realiza el software. Para probar este factor debemos responder a la pregunta: ¿Se puede usar con facilidad el sistema?

Portabilidad: esfuerzo de trasferencia de una plataforma de hardware-software a otra. Para probar este factor debemos responder a la pregunta: ¿Se puede usar en otra máquina?

2.3 MÉTODOS DE PRUEBA DEL SOFTWARE

"*La prueba del software es un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación*".

Los métodos de pruebas de software pueden ser clasificados en tres grandes grupos: Prueba de Caja Blanca, Prueba de Caja Negra y Prueba de Requerimientos no Funcionales.

2.3.1 PRUEBA DE CAJA BLANCA

La prueba de caja blanca, denominada también prueba de caja de cristal es un método de diseño de casos de prueba que usa la estructura de control del diseño del programa para los casos de prueba, es decir, se aprovecha del conocimiento de la estructura interna del mismo.

La prueba de caja blanca permite obtener casos de prueba que:

I. Garanticen que se recorran por lo menos una vez todos los caminos independientes de cada módulo.

II. Recorran todas las decisiones lógicas en sus vértices verdadero y falso.

III. Ejecuten todos los ciclos en sus límites y con sus límites operacionales

IV. Recorran las estructuras internas de datos para asegurar su validez.

La pruebas de caja blanca enfocan su atención a los detalles procedimentales del software, por ello la implementación de estas pruebas depende fuertemente de la disponibilidad de código fuente. Este tipo de pruebas, permiten generar casos para probar y validar los caminos de cada módulo, las condiciones lógicas, los bucles y sus límites, así como también para las estructuras de datos. Las pruebas de caja blanca también son conocidas como pruebas estructurales.

Una justificación para la realización de este tipo de prueba (en vez de invertir el tiempo asegurando que se han alcanzado los requisitos funcionales del programa -prueba de la caja negra-, y que a su vez surge como ventaja de emplear tiempo y energía probando las minuciosidades lógicas de un Software) la encontramos en :

I. Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa. Generalmente se cometen más errores al diseñar o implementar funciones que no son habituales realizar.

II. A menudo creemos que un camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma normal.

III. Los errores de digitación (escritura) son aleatorios.

Algunas de las pruebas más significativas dentro de este enfoque se detallan en los puntos subsiguientes:

2.3.1.1 Notación de Grafo de Flujo

A continuación se definen los conceptos sobre grafos en que se basan las pruebas estructurales.

Grafo de flujo: representan el flujo de control de un programa y ayudan en la obtención de conjuntos de prueba. Está formado por un conjunto de nodos y arcos.

Nodos: cada nodo del grafo representa una o más sentencias del programa.

Arcos: son líneas dirigidas que unen dos nodos. Los arcos entre nodos representan flujo de control. Un arco debe terminar en un nodo.

Región: es un área rodeada por nodos y arcos.

Nodo predicado: es un nodo que contiene una condición.

Caminos independientes: es un camino que introduce por lo menos un nuevo conjunto de sentencias de proceso (debe moverse por lo menos por un arco nuevo en el camino).

Medida de la complejidad ciclomática *V(G)*: da un valor límite para el número de pruebas que se deben diseñar y ejecutar para garantizar que se cubren

todas las sentencias del programa. Este límite es el *número ciclomático*, que mide el número de caminos linealmente independientes para un grafo de flujo dado, y que viene dado por la fórmula:

V(G) = 1 + **d** ; donde **d** es el número de nodos predicado del grafo G.

Conjunto básico: colección de caminos que garantizan la ejecución por lo menos una vez de todas las sentencias del programa.

La Figura 1 muestra un ejemplo de grafo de flujo con nodos predicado, con V(G) = 1 + 3 = 4.

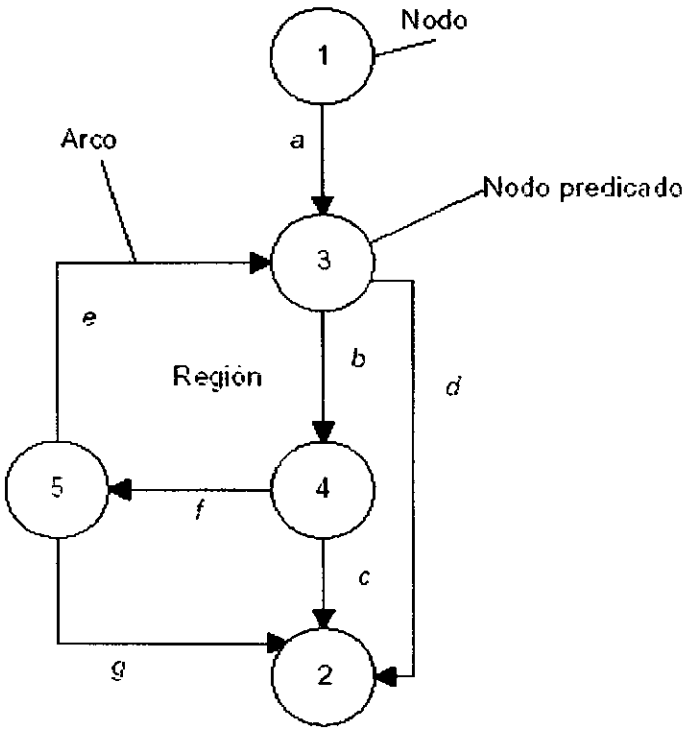


Figura 1 – Grafo de Flujo

2.3.1.2 Matrices de Grafos

Son útiles para mecanizar la obtención del grafo de flujo y la determinación de un conjunto básico de caminos:

- En cada celda se indica la arista que conecta dos nodos dados.
- La representación en forma de matriz de conexiones reemplaza el identificador de la arista por un 1.
- Peso de enlace: da información adicional sobre la matriz de conexión. De forma sencilla un peso 1 indica que existe una conexión y un peso 0 indica que no existe conexión.

La Figura 2 muestra la matriz de grafo y la matriz de conexiones correspondiente al grafo de flujo de la figura 1.

NODO

NODO	1	2	3	4	5
1			*a*		
2					
3		*d*		*b*	
4		*c*			*f*
5		*g*	*e*		

NODO

	1	2	3	4	5
1			1		
2					
3		1		1	
4		1			1
5		1	1		

Figura 2 – Matriz de Grafo y Matriz de Conexiones

2.3.1.3 Estrategias de Pruebas Estructurales

A continuación se describen brevemente las diferentes estrategias de pruebas estructurales o de Caja Blanca.

2.3.1.3.1 Prueba por Cobertura de Sentencias

Consiste en seleccionar casos de prueba que garanticen que cada sentencia o secuencia de sentencias sin puntos de decisión ha/ n sido probada/ s al menos una vez.

2.3.1.3.2 Prueba por Cobertura de Ramas

Consiste en seleccionar un conjunto de caminos de tal forma que cada rama del programa cubre en al menos un camino. Un 100% de cobertura de ramas garantiza un 100% de cobertura de sentencias, pues toda sentencia está en alguna rama.

2.3.1.3.3 Prueba de Caminos

Es la estrategia de prueba de Caja Blanca más exhaustiva y consiste en seleccionar casos de prueba de tal forma que cada camino posible del programa es ejecutado al menos una vez.

En este tipo de prueba se realiza un análisis sobre una representación gráfica de un programa denominada grafo de control. En este grafo, los nodos representan bloques de instrucciones de un programa y los flujos de ejecución

para dichas instrucciones se representan por medio de aristas. A partir de este grafo, se puede identificar un conjunto básico de caminos de ejecución, sobre el cual se pueden realizar pruebas con el propósito de probar el flujo de ejecución de los caminos en una unidad.

Este método permite al diseñador derivar una medida de la complejidad lógica de un programa y usarla como una guía para definir un conjunto básico de caminos de ejecución. Se garantiza que los casos de prueba que verifican el conjunto básico ejecutan todas las sentencias del programa al menos una vez. La derivación de casos de prueba se realiza según los siguientes pasos:

I. Derivar un grafo de flujo partiendo del diseño o del código fuente.

II. Determinar la complejidad ciclomática de este grafo de flujo.

III. Determinar un conjunto básico de caminos linealmente independientes.

IV. Preparar casos de prueba que fuercen la ejecución de cada camino del conjunto básico.

La existencia de un simple bucle puede implicar la existencia de un número infinito de caminos y también pueden existir caminos inalcanzables para cualquier valor de la entrada. Debido a estos problemas se han sugerido otras estrategias que intentan eliminarlos:

2.3.1.3.3.1 Prueba del camino simple

Consiste en la ejecución de todos los caminos simples (un camino simple es aquél que no contiene la misma rama más de una vez).

2.3.1.3.3.2 Prueba de caminos linealmente independientes

Consiste en la ejecución de todos los caminos linealmente independientes a través del grafo de flujo del programa. Un concepto que debemos recordar para esto es el de número ciclomático, el cuál es igual al número de casos de prueba requeridos para satisfacer la estrategia.

La obtención del *índice de efectividad de las pruebas* permite conocer el grado de extensión en que los casos de prueba satisfacen una estrategia de prueba particular para un programa dado y un conjunto de casos de prueba. Siendo T una estrategia de prueba para cubrir una clase de objetos (como caminos, caminos simples, caminos linealmente independientes, ramas o sentencias), el *índice de efectividad de las pruebas* se define como la proporción entre el número de objetos probados al menos una vez y el número total de objetos.

Los apartados siguientes muestran, con un mayor grado de profundidad, las estrategias de prueba del camino y la prueba de ciclos.

2.3.1.3.4 Prueba de Condiciones

Basándose de igual forma en un grafo de control, pueden generarse casos de prueba para elementos individuales de expresiones lógicas. De esta forma, se pretende probar cada condición con todas sus posibles alternativas.

2.3.1.3.5 Prueba de Ciclos

A partir del grafo de control, pueden generarse casos de prueba para las iteraciones definidas en los programas con el propósito de verificar si se realizan de forma correcta.

Los posibles errores en las construcciones de bucles o ciclos son:

- De iniciación.
- De indexación o incremento.
- En los límites.

Esta técnica se centra exclusivamente en la validez de las construcciones de bucles. Se definen cuatro clases de bucles diferentes y un conjunto de pruebas para cada tipo de bucle:

2.3.1.3.5.1 Bucles simples

Si *n* es el máximo número de pasos permitidos por el bucle, entonces:

I. Saltar el bucle completamente.

II. Pasar una sola vez por el bucle.

III. Hacer *m* pasos por el bucle con $m < n$.

IV. Hacer $n-1$, n y $n+1$ pasos por el bucle.

2.3.1.3.5.2 Bucles anidados

I. Comenzar en el bucle más interior. Disponer todos los demás bucles en sus valores mínimos.

II. Realizar pruebas de bucle simple con el bucle más interior mientras se mantienen los bucles exteriores con valores mínimos. Añadir pruebas de fuera de rango o de valores excluidos.

III. Progresar hacia fuera realizando pruebas para el siguiente bucle, manteniendo los demás bucles exteriores en valores mínimos y los demás bucles anidados con valores "típicos".

IV. Continuar hasta que todos los bucles hayan sido probados.

2.3.1.3.5.3 Bucles concatenados

Pueden ser probados como bucles simples si cada bucle es independiente de los otros. Si no es así, se puede utilizar el método de bucles anidados.

2.3.1.3.5.4 Bucles no estructurados

Bajo ningún concepto son aceptables. Los bucles no estructurados se deben rediseñar, no se deben probar.

2.3.1.3.6 Prueba de Definición de Datos

Estas pruebas son realizadas con el objetivo de encontrar posibles contradicciones o redundancias en la definición de los datos utilizados en el software. Para ello, se realiza un análisis del comportamiento de cada uno de los datos o cada una de los flujos de ejecución.

2.3.2 PRUEBA DE CAJA NEGRA

Las pruebas de Caja Negra, también llamada pruebas funcionales, de comportamiento o a gran escala, se centran en los requisitos funcionales del software ignorando la estructura de control del mismo. Esta permite al Ingeniero de Software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa.

Este tipo de pruebas concentra la atención en generar casos de prueba que permitan verificar los requisitos funcionales de un programa. A diferencia de las pruebas de caja blanca, que se basan en la lógica interna del software, este tipo de pruebas se concentra en su funcionalidad, por lo que mucho del trabajo se realiza interactuando con la interfaz del software. Los casos de prueba generados bajo este enfoque, se diseñan a partir de valores entrada y salida. De esta forma, se puede determinar la validez de una salida para un conjunto de entradas proporcionadas.

La aplicación de pruebas de caja negra permiten detectar errores como:

I. Funciones incorrectas o ausentes.

II. Errores de interfaz.

III. Errores en estructuras de datos o en accesos a bases de datos externas.

IV. Errores de rendimiento.

V. Errores de inicialización y de terminación.

Las pruebas se diseñan para responder, entre otras, las siguientes preguntas:

I. ¿Cómo se prueba la validez funcional?

II. ¿Qué clases de entrada compondrán buenos casos de prueba?

III. ¿Es el sistema particularmente sensible a ciertos valores de entrada?

IV. ¿De qué forma están aislados los límites de una clase de datos?

V. ¿Qué volúmenes y niveles de datos tolerará el sistema?

VI. ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?

En el ciclo de vida del software, la prueba de la Caja Blanca debería realizarse al principio del proceso de prueba, mientras que la prueba de la Caja Negra debería realizarse en etapas posteriores.

Estas son algunas de las pruebas (métodos) más conocidas en este contexto:

2.3.2.1 Partición Equivalente

La partición equivalente intenta definir un caso de prueba que descubra clases de errores y por tanto reducir el número de casos de prueba necesarios. Ésta

se basa en una evaluación de las clases de equivalencia para una condición de entrada.

La idea de esta técnica, consiste en dividir los valores válidos y no válidos para entradas y salidas en un número reducido de particiones, de forma que, el comportamiento del software sea el mismo para cualquier valor contenido en una partición particular. El propósito principal de una partición es reducir la cantidad de casos de prueba generados en el proceso. Para llevar a cabo los mismos, se procede dividiendo el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba.

Las clases de equivalencia se pueden definir de acuerdo con los siguientes criterios o directrices:

I. Si una condición de entrada especifica un rango, se definen una clase de equivalencia válida (dentro del rango) y dos inválidas (por debajo del rango y por encima del rango).

II. Si una condición de entrada requiere un valor específico, se definen una clase de equivalencia válida (el valor específico) y dos inválidas (por debajo del valor y por encima del valor).

III. Si una condición de entrada especifica un miembro de un conjunto, se definen una clase de equivalencia válida (dentro del conjunto) y una inválida (fuera del conjunto).

IV. Si una condición de entrada es lógica, se definen una clase válida (verdadero) y una inválida (falso).

De igual forma se construyen las particiones de equivalencia para los valores del dominio de salida.

2.3.2.2 Análisis de los Valores Límite

La generación de casos de prueba en esta técnica, se enfoca en los valores límites bajo la consideración de que existe una tendencia a fallar precisamente cuando el software trabaja con valores extremos de la /s variable /s de entrada /s. Generalmente, los valores establecidos para generar los casos de prueba son el mínimo, valores un poco arriba del mínimo, valor máximo y valores un poco arriba del máximo.

Este método complementa la partición equivalente dado que selecciona casos de prueba en los "bordes" de una clase. Además de centrarse en las condiciones de entrada, también deriva casos de prueba para el dominio de salida. Las directrices para derivar casos de prueba son:

I. Para una condición de entrada que especifica un rango limitado por los valores ***a*** y ***b***, los casos de prueba deben incluir los valores ***a*** y ***b*** y los valores justo por debajo de ***a*** y justo por encima de ***b***.

II. Para una condición de entrada que especifica un número de valores, los casos de prueba deben incluir el valor mínimo, el máximo y los valores justo por debajo del mínimo y justo por encima del máximo.

III. Aplicar las directrices I y II a las condiciones de salida.

IV. Si las estructuras de datos internas tienen límites preestablecidos, debe diseñarse un caso de prueba que ejercite la estructura en sus límites.

2.3.2.3 Pruebas Según la Experiencia

En este tipo de testing la generación de casos de prueba se realiza a partir de la intuición y la experiencia. La idea básica es redactar una lista de las posibles fallas o de las posibles situaciones en las cuales suele ocurrir algún problema y así desarrollar casos de prueba basados en la información contenida en estas listas.

2.3.2.4 Tablas de Decisión

Este tipo de prueba permite describir el comportamiento de un programa a partir de un conjunto de acciones que este realiza cuando se opera bajo determinadas condiciones. En este enfoque, las condiciones pueden ser interpretadas como entradas de un programa y las acciones como las salidas producidas. Para ello se pueden utilizar conectores lógicos **y** (and), **o** (or) y **no** (not). Al involucrar aspectos de lógica este tipo de prueba se hace más rigurosa y permite además transformar una especificación en lenguaje natural en una especificación más formal.

2.3.2.5 Técnicas de Grafos de Causa-Efecto