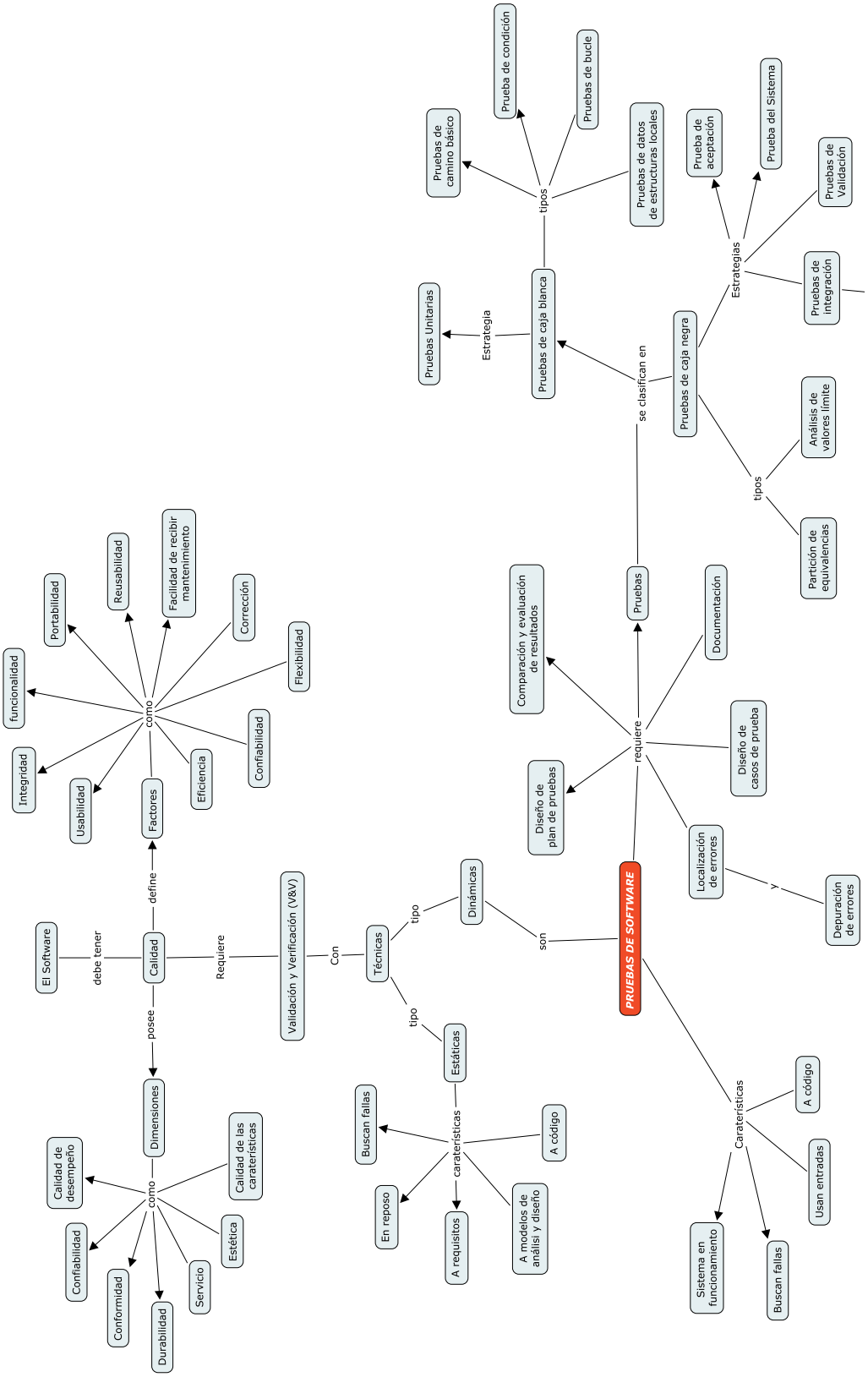
PRUEBAS DE SOFTWARE

Estructura de contenidos

Mapa conceptual
PRUEBAS DE SOFTWARE



INTRODUCCIÓN

En todo sistema de información, después de la etapa de desarrollo se continúa con la implantación del sistema, en donde se requiere de un ejercicio de ingeniería de software no menos importante que el desarrollo. En este punto, antes de hacer la entrega final al cliente, se debe comprobar que el sistema cumple con los requerimientos del usuario y que su funcionamiento es correcto, es decir sin errores o defectos. Para esto se debe implementar pruebas de software.

Algunos constructores de software concentran sus esfuerzos en la fase de desarrollo y descuidan el protocolo de pruebas, olvidando que las pruebas de software son las que ayudan a dar calidad al sistema. En este módulo el estudiante podrá reconocer la importancia de las pruebas de software, las recomendaciones generales, las actividades y las técnicas para el diseño e implementación de pruebas adecuadas que aporten significativamente a la corrección de errores y mejoramiento de la calidad.

Para dar inicio al estudio de este módulo, se recomienda el ver y escuchar con atención el siguiente video: http://youtu.be/bWNRTDAO_7M, que nos recuerda la importancia de las pruebas en el marco de la ingeniería del software.

1. CALIDAD DE SOFTWARE

1.1 Definición

A continuación se presentan algunas definiciones para calidad de software que permiten entender el concepto.

Según el estándar IEEE 6.10-1990, la calidad es "el grado con el que un sistema, componente o proceso cumple con los requisitos especificados y las necesidades o expectativas del cliente o usuario".

(BOLAÑOS, SIERRA, & ALARCÓN, 2008), definen la calidad como un "proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes los producen y a quienes lo utilizan".

Según Boehm (1978) y McCall (1977) en (CATALDI, 2000), la calidad está asociada a tres usos importantes del usuario:

- Características de operación.

- Capacidad para soportar cambios (ser modificado).

- Adaptabilidad a nuevos cambios.

1.2 Dimensión de la calidad

David Garvin (1987) en (PRESSMAN, 2006),plantea ocho dimensiones de la calidad que pueden ser aplicadas al software y que se describen en laTabla 1.

DIMESIÓN	DESCRIPCIÓN
Calidad del desempeño	Presenta el contenido, las funciones y las características especificadas en el modelo de requerimientos.
Calidad de las características	Genera sorpresa y agrado en la primera impresión del usuario.
Confiabilidad	Está disponible cuando se necesita, sin errores y sin fallas.
Conformidad	Es coherente con los estándares locales e internacionales.
Durabilidad	Permite con facilidad el mantenimiento (cambio) y la depuración (corrección).
Servicio	El mantenimiento y la depuración se pueden hacer en un tiempo aceptablemente breve.
Estética	Posee cierta elegancia, flujo único y presencia aceptable por los usuarios en general.
Percepción	Recibe en general buenos comentarios por parte de los usuarios.

Tabla 1: Dimensiones de la calidad del software.

1.3 Factores de la calidad

A partir delos aportes de McCall, Richards y Walters (1977) y de las definiciones del estándar ISO 9126, (PRESSMAN, 2006), presenta los siguientes factores o atributos claves asociados a la calidad del software (Ver Figura 1).

Figura 1: Factores de la calidad de software.

- **Funcionalidad:** satisfacción a las necesidades de adaptabilidad, exactitud, interoperabilidad, cumplimiento y seguridad.

- **Confiabilidad:** cantidad de tiempo que el software se encuentra disponible para su uso, con madurez, tolerancia a fallas y recuperación.

- **Usabilidad:** facilidad para usar, siendo entendible, aprendible y operable.

- **Eficiencia:** uso óptimo de los recursos del sistema.

- **Facilidad de recibir Mantenimiento:** facilidad para realizar reparaciones, es decir, es analizable, cambiable, estable y susceptible a pruebas.

- **Portabilidad:** facilidad para ser llevado a otro ambiente, es decir, es adaptable, instalable y sustituible.

- **Corrección:** cumple con las especificaciones y necesidades del cliente.

- **Integridad:** control de acceso al software o datos de usuarios no autorizados.

- **Flexibilidad:** capacidad para permitir modificaciones cuando el software ya está en operación.

- **Reusabilidad:** Grado en el que puede ser usado por otras aplicaciones.

En particular, para evaluar una interfaz, se debe tener en cuenta los siguientes factores de calidad:

- **Intuitiva:**La interfaz sigue patrones de uso esperados, facilitando la comprensión, localización de operaciones y la entrada de datos.

- **Eficiencia:** Grado en el que es posible localizar o iniciar las operaciones y la información.

- **Robustez:** Capacidad para tratar entradas erróneas de datos o interacción inapropiada del usuario.

- **Riqueza:** interfaz con abundantes características, que permite la personalización según las necesidades del usuario, y la identificación de una secuencia de operaciones comunes por medio de una acción o comando.

1.4 Verificación y Validación (V&V)

En el control de calidad de software se distinguen dos procesos de evaluación propios del proceso de desarrollo de software: la verificación y la validación. IEEE Std 729-1983 da las siguientes definiciones:

Verificación:"Proceso para determinar si los productos de una determinada

fase del desarrollo de software cumplen o no los requisitos establecidos durante la fase anterior".

Pregunta a responder:¿Se ha construido el sistema correctamente?

Validación:"Proceso de evaluación del software al final del proceso de desarrollo para asegurar el cumplimiento de las necesidades del cliente".

Pregunta a responder:¿Se ha construido el sistema correcto?

La validación y verificación se puede realizar usando básicamente dos tipo de técnicas (verFigura 2).

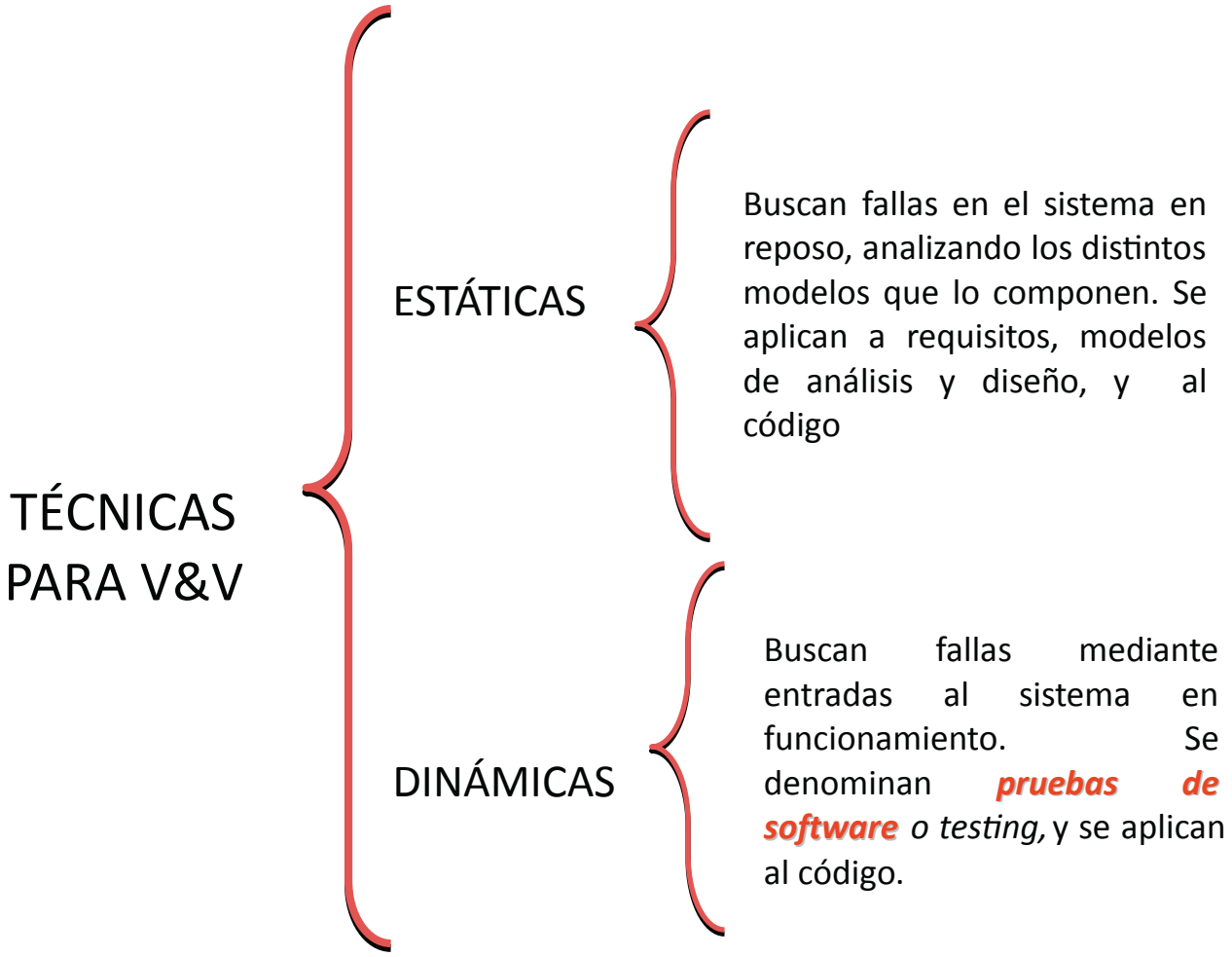


Figura 2: Tipos de técnicas para V&V.

2 PRUEBA DE SOFTWARE

De lo anterior, se puede decir que las pruebas de software son estrictamente necesarias, para determinar de manera dinámica la calidad del software; de esta manera, se garantiza que se ha construido el sistema correcto y de la forma correcta.

2.1 ORIENTACIONES GENERALES

- Realizar revisiones técnicas efectivas, antes de comenzar la prueba.
- La prueba comienza en los componentes del software y fluye hacia la integración de todo el sistema.
- Seleccionar y aplicar la mejor técnica de prueba, de acuerdo al enfoque de desarrollo utilizado.
- Diseñar casos y esbozar el plan de prueba en la fase de diseño del software.
- Un buen caso de prueba permite mostrar un error o un fallo no detectado anteriormente.
- Es fundamental conocer el resultado esperado para determinar si el resultado de la prueba es correcto o no.
- Procurar que en la pruebas, a parte de los desarrolladores, participen programadores que no hayan estado en la fase de codificación y también potenciales usuarios.
- Así como se prueba que el programa funcione correctamente para entradas válidas, también es igualmente importante, comprobar que el programa reaccione correctamente ante entradas no válidas.
- Documentar todo los casos de prueba. Esto permite realizar pruebas de regresión.

2.2 ACTIVIDADES

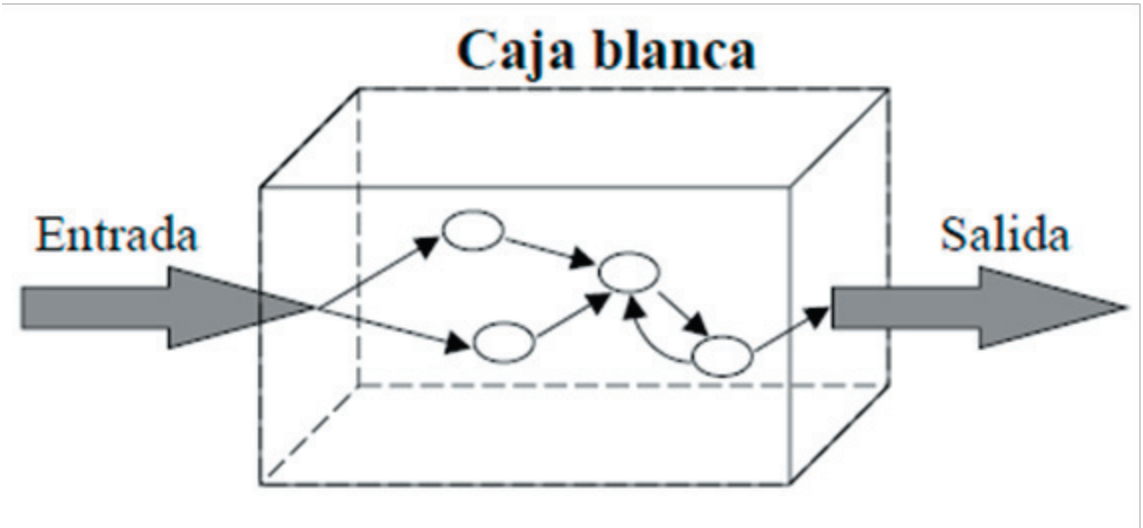
La Figura 3 presenta las principales actividades que se deben desarrollar en la etapa de prueba en el marco del desarrollo software.

Figura 3: Tareas básica en la prueba de software.

2.3 PRUEBAS

Las pruebas de software, se clasifican principalmente en dos categorías: pruebas de caja blanca y pruebas de caja negra.

2.3.1 PRUEBAS DE CAJA BLANCA



Las pruebas de caja blanca, permiten probar la lógica interna del programa y su estructura, realizando las siguientes acciones:

- Ejecución de todas las sentencias (al menos una vez).
- Recorrido de todo los caminos independientes de cada componente.
- Comprobación de todas las decisiones lógicas.
- Comprobación de todos los bucles.
- Implementación de situaciones extremas o límites.

2.3.1.1 PRUEBA DE CAMINO BÁSICO

Esta técnica fue propuesta por Tom McCabe y consiste en definir un conjunto básico de caminos usando la medida de complejidad llamada complejidad ciclomática (VG).

La complejidad ciclomática determina el número de caminos a probar, mediante la siguiente fórmula:

```
V(G) = #Aristas - #Nodos + 2
```

Los pasos en las pruebas de caminos básicos son:

1. Dibujar grafo de flujo.
2. Determinar la complejidad ciclomática del grafo.
3. Determinar los caminos linealmente independientes.
4. Diseñas los casos de prueba.

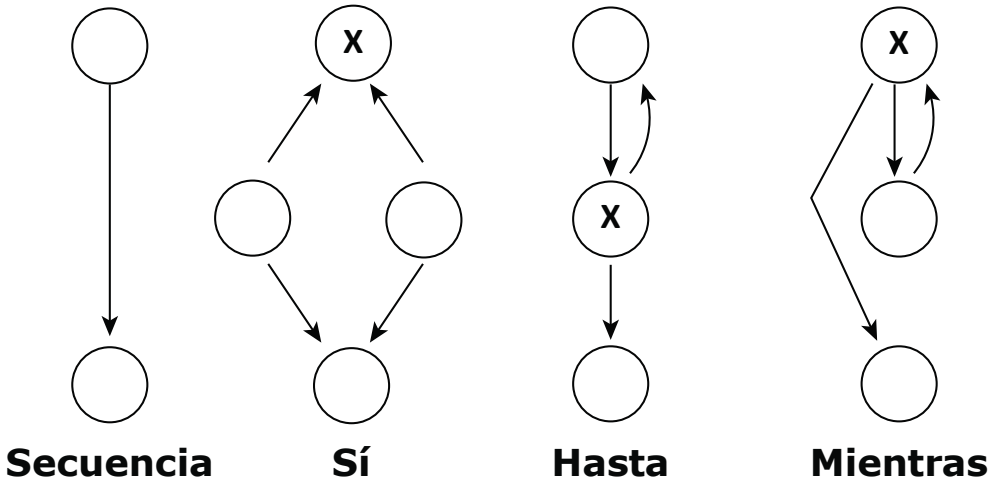


Figura 4: Caminos básicos.

Figura 4: Caminos básicos.

Ejemplo:

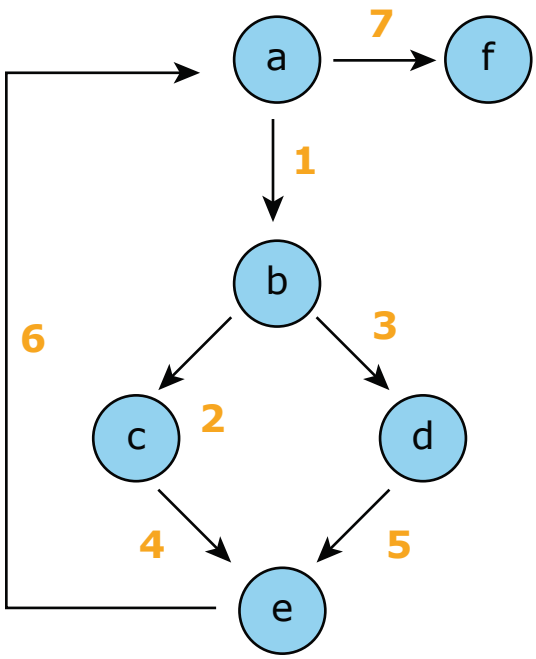
Realizar la prueba de camino básico para el siguiente módulo:

Public intMCD(int x, int y)

{

```
          While ▸ a
          {
               If (x > y) ▸ b
               x = x – y; ▸ c
               else
                    y = y – x; ▸ d
          }                ▸e
return x;              ▸ f
}
```

1. Grafo de flujo:



2. Complejidad Ciclomática:

$$V(G) = \#Aristas - \#Nodos + 2$$

$$V(CDM) = 7-6+2 = 3$$

3. Caminos linealmente independientes:

Como la complejidad ciclomática es 3, entonces existen tres caminos linealmente independientes.

4. Casos de prueba:

	Aristas							
Caminos	1	2	3	4	5	6	7	**Casos de Prueba**
af	0	0	0	0	0	0	1	x=1, y=1, return=1
abdeaf	1	0	1	0	1	1	1	x=1, y=2, return=1
abceaf	1	1	0	1	0	1	1	x=2, y=1, return=1

Tabla 2: Casos de prueba ejemplo de camino básico.

2.3.1.2 Prueba de bucle

Para dicha prueba se requiere, en primer lugar, representar de forma gráfica los bucles, que pueden ser simples, anidados, concatenados y no estructurados, como lo muestra la Figura 5.

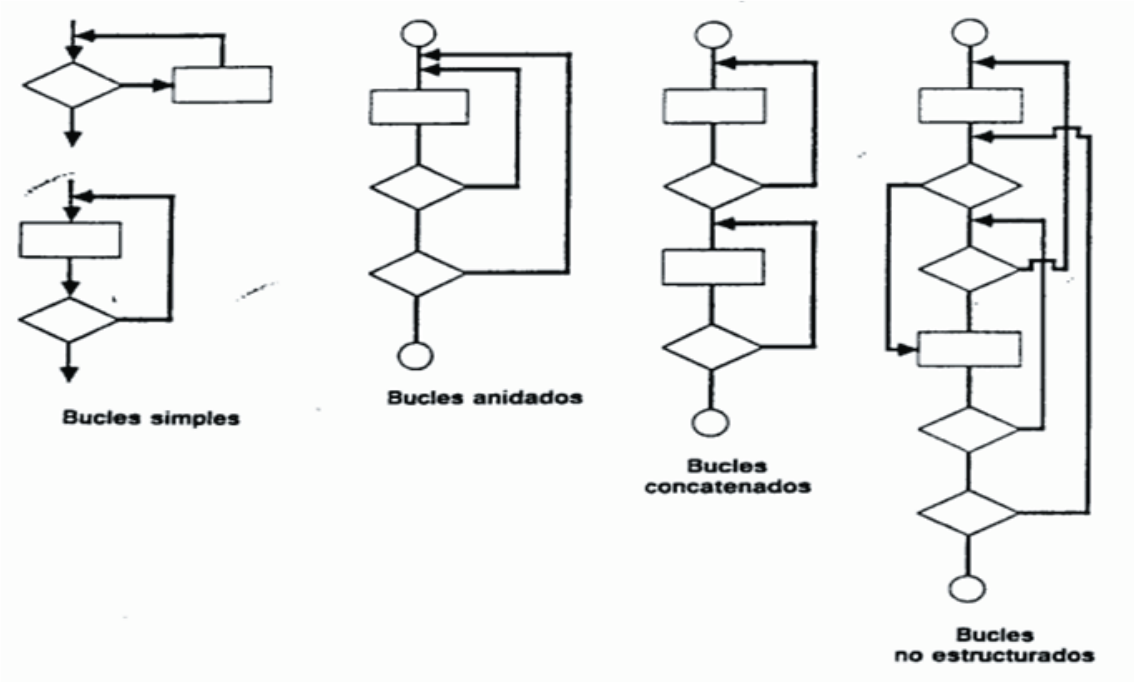


Figura 6: Tipos de bucles.

- Bucles Simples

Con bucles simples, se requieren las siguientes iteraciones, siendo n el número máximo de pasos:

* Saltar por completo el bucle.
* Pasar una sola vez a través del bucle.
* Pasar dos veces a través del bucle.
* Pasar m veces a través del bucle, donde m < n.
* Hacer n-1, n y n+1 pasadas a través del bucle.

- Bucles anidados

En bucles anidados, se requieren los siguientes pasos:

* Comenzar con el bucle más interno.
* Realizar pruebas de bucle simple para el bucle más interno.
* Avanzar hacia afuera y realizar pruebas con el siguiente bucle.
* Continuar hasta que todos los bucles hayan sido probados.

- Bucles concatenados

Si los bucles concatenados son independientes se aplica pruebas de bucles simples, si no independientes se implementa el enfoque de los bucles anidados.

- Bucles no estructurados

Los bucles no estructurados deben ser rediseñados porque comprometen la calidad del diseño. Posteriormente, se realizan pruebas de acuerdo al tipo de bucle resultante.

2.3.1.3 Prueba de condición

Esta prueba evalúa las condiciones lógicas contenidas en un módulo del programa, las cuales pueden ser simples o compuestas.

Condición simple: es una variable lógica (TRUE o FALSE), o una expresión relacional de la forma: E1<operador relacional>E2, donde E1 Y E2 son expresiones aritmética, y el operador relacional es uno de los siguientes: <, >, <=, >=, =, !=.

Condición compuesta: se compone de dos o más condiciones simples y operadores lógicos de tipo NOT, AND, OR y paréntesis.

En general, en la prueba de condición existen los siguientes tipos:

- De cobertura de decisión.
- De cobertura de condición.
- De cobertura de decisión/condición.

Ejemplo:

Definir casos de prueba aplicando cobertura de decisión y de decisión/ condición para el siguiente fragmento de código:

```
public void comprobarhora (int h, int m, int s)
{
    If((h>=0)&&(h<=23))
    {
        If((m>=0)&&(<=59))
        {
            If((s>=0)&&(s<=50))
            {
                System.out.println("La hora digitada es
correcta");
            }
        }
    }
    else
    {
        System.out.println("La hora digitada es incorrecta");
    }
}
```

1. Casos de prueba para cobertura de decisiones:

En el código hay tres decisiones.

D1 ▸ (h>=0) y (h<=23)

D2 ▸ (m>=0) y (m<=59)

D3 ▸ (s>=0) y (s<=59)

2. Datos concretos para los casos de prueba:

Caso	Valor Verdadero	Valor Falso
D1	h=8	h=25
D2	m=25	m=60
D3	s=59	s=75

Figura 7: Datos de prueba de decisión para ejemplo.

Cada decisión debe tomar al menos una vez el valor verdadero y otra el valor falso.

3. Casos de prueba para cubrir todas las decisiones:

Caso de prueba 1: D1 = Verdadero; D2= Verdadero; D3=Verdadero

(h=8; m=25; s=59)

Caso de prueba 2: D1 = Verdadero; D2= Verdadero; D3=Falso

(h=8; m=25; s=75)

Caso de prueba 3: D1 = Verdadero; D2= Falso

(h=8; m=60)

Caso de prueba 4: D1 = Falso

(h=25)

4. Casos de prueba para obtener una cobertura total de decisión/ condición:

En el código hay tres decisiones y cada una contiene dos condiciones.

```
D1 ▸ (h>=0) y (h<=23)
C1.1 • h>=0
C1.2 • h<=23

D2 ▸ (m>=0) y (m<=59)
C2.1 • m>=0
C2.2 • m<=59

D3 ▸ (s>=0) y (s<=59)
C3.1 • s>=0
C3.2 • s<=59
```

5. Datos concretos para los casos de prueba:

Caso	Valor Verdadero	Valor Falso
C1.1	h=10	h=-1
C1.2	h=10	h=24
C2.1	m=30	m=-1
C2.2	m=30	m=60
C3.1	s=50	s=-1
C3.2	s=50	s=70

Figura 8: Datos de prueba de decisión/condición para ejemplo.

6. Casos de prueba para cubrir todas las decisiones/condiciones:

Caso de prueba 1:

C1.1=Verdadero; C1.2=Verdadero;

C2.1=Verdadero; C2.2=Verdadero;

C3.1=Verdadero; C3.2=Verdadero;

(h=10; m=30; s=50)

Caso de prueba 2:

C1.1=Verdadero; C1.2=Verdadero;

C2.1=Verdadero; C2.2=Verdadero;

C3.1=Verdadero; C3.2=Falso;

(h=10; m=30; s=70)

Caso de prueba 3:

C1.1=Verdadero; C1.2=Verdadero;

C2.1=Verdadero; C2.2=Verdadero;

C3.1=Falso; C3.2=Verdadero;

(h=10; m=30; s=-1)

Caso de prueba 4:

C1.1=Verdadero; C1.2=Verdadero;

C2.1=Verdadero; C2.2=Falso;

(h=10; m=30)

Caso de prueba 5:

C1.1=Verdadero; C1.2=Verdadero;

C2.1=Falso; C2.2=Verdadero;

(h=10; m=-1)

Caso de prueba 6:

C1.1=Verdadero; C1.2=Falso;

(h=10)

Caso de prueba 7:

C1.1=Falso; C1.2=Verdadero;

(h=-1)

2.3.1.4 Prueba de estructura de datos locales

Estas pruebas aseguran la integridad de los datos durante todos los pasos de la ejecución del módulo. Se revisa que se cumpla:

- Integridad de datos.
- Inicialización de todas las variables utilizadas.
- Uso adecuado de los límites en los arreglos.
- Declaración correcta de variable.
- Comparaciones entre variables del mismo tipo.
- Control de errores como overflow, underflow, división por cero, etc.

2.3.2 PRUEBAS DE CAJA NEGRA

En este tipo de pruebas se considera el software como una caja negra, sin preocuparse por los detalles procedimentales de los programas, como lo ilustra laFigura 9. De esta manera, los datos de entrada deben generar una salida coherente con las especificaciones; si no es así, es porque se ha encontrado un error el cual debe ser corregido para poder continuar con las pruebas.

Figura 9: Pruebas de caja negra.

Las dos técnicas más comunes son:

- Partición de equivalencia.
- Análisis de valores límite.

2.3.2.1 Partición de equivalencia

En este tipo de prueba, la entrada de un programa se divide en clases de datos de los cuales se puedan derivar casos de prueba, teniendo en cuenta las siguientes reglas:

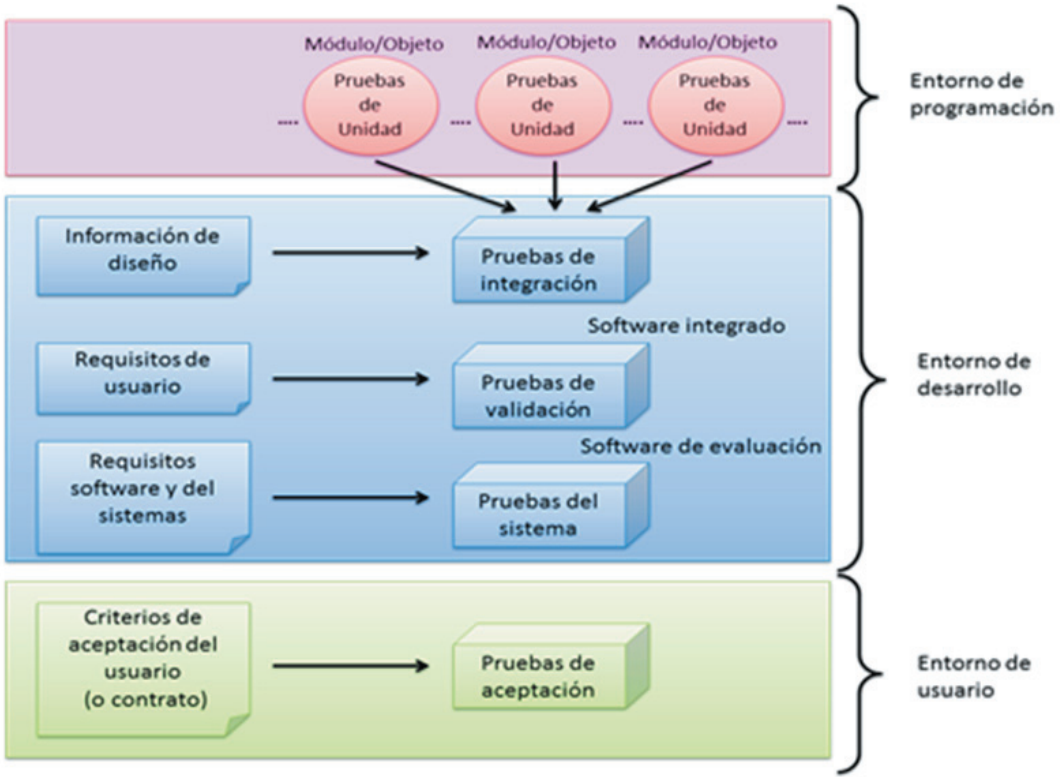
- Si la entrada es un rango o un valor específico, se define una clase de equivalencia válida y dos inválidas.
- Si la entrada es un valor lógico, se define una clase de equivalencia válida y otra inválida.

2.3.2.2 Análisis de valores límite

Después de probar todos los casos en la técnica de participación de equivalencia, se prueban los valores fronterizos de cada clase (valores límite).

2.3.3 ESTRATEGIA DE PRUEBAS

La prueba de software se realiza desde dentro hacia fuera del sistema, como lo ilustra la siguiente figura.



- Pruebas unitarias o de unidad: se comprueba la lógica, funcionalidad y especificación de cada unidad (componente, clase u objeto) del sistema de manera aislada con respecto al resto de unidades.

- Pruebas de integración: se centra en el diseño y la construcción de arquitectura del software, analizando el flujo de información entre unidades a través de las interfaces.

- Pruebas de validación: se comprueba el cumplimiento de los requisitos.

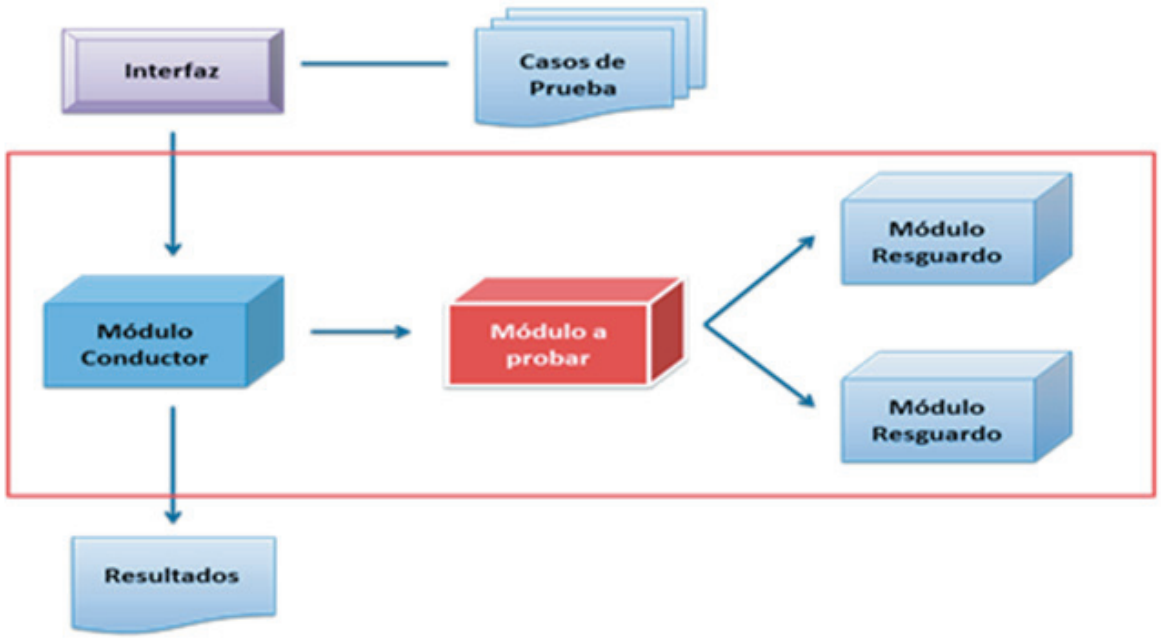
- Pruebas del sistema: se prueba el software y otros elementos del sistema como un todo.

- Pruebas de aceptación: el usuario valida que el producto se ajusta a sus requerimientos.

2.3.4 PRUEBAS UNITARIAS

Corresponden a la evaluación de cada uno de los bloque más pequeños con identidad propia en el sistema, y es realizada por el programador en su entorno de desarrollo.

Las pruebas unitarias usan técnicas de caja blanca, para lo cual se crean módulos conductores y módulos resguardo (siguiente figura).



Un módulo conductor o controlador es un "programa principal" que recibe a través de la interfaz datos de caso de prueba, y los pasa al componente que se desea probar. Los módulos resguardo (en inglés stubs) o "subprogramas tontos", se utilizan para sustituir componentes que son subordinados o llamados desde el componente que se desea probar. Finalmente se generan reportes con los resultados obtenidos.

2.3.5 PRUEBAS DE INTEGRACIÓN

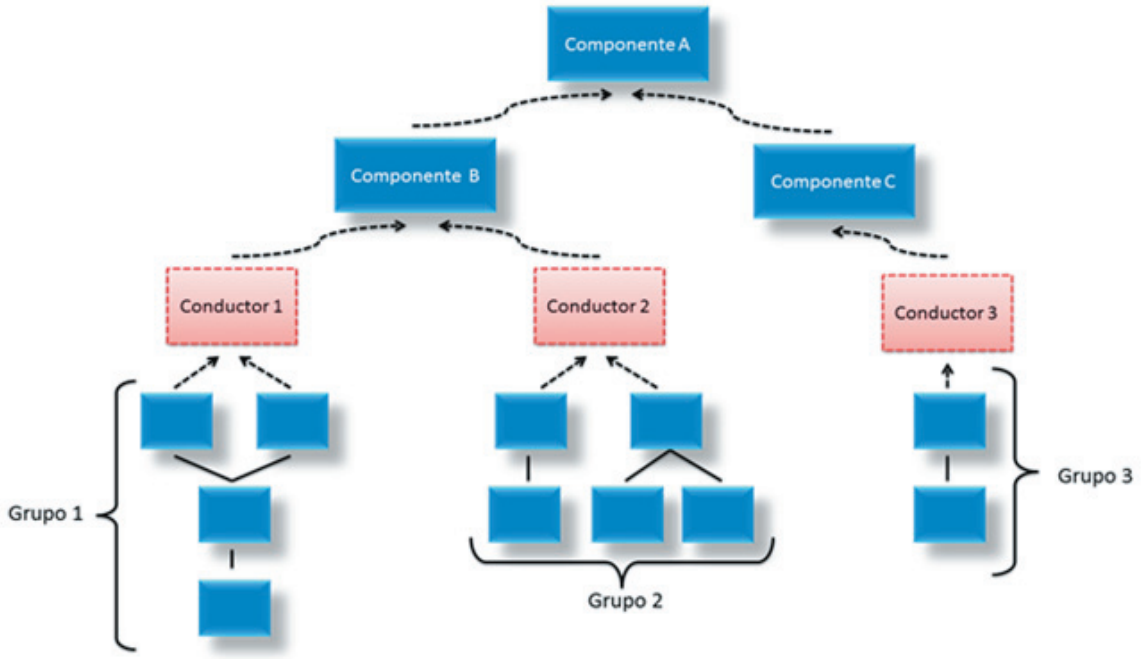
No es suficiente con probar el funcionamiento correcto de cada módulo, se requiere además del funcionamiento en conjunto. Por tal razón, se hacen necesarias las pruebas de integración, las cuales generalmente implementan técnicas de caja negra.

Existen dos estrategias básicas para realizar la combinación de módulos en las pruebas de integración: integración ascendente e integración descendente.

2.3.5.1.1 Pruebas de integración ascendente

En estas pruebas se implementan las siguientes fases:

- Combinación de los componentes del nivel más bajo en grupos.
- Construcción de un módulo conductor para coordinar la E y S de los casos de prueba.
- Probar el grupo.
- Se realizan pruebas de regresión.
- Eliminación de conductores y combinación de grupos en movimiento hacia arriba según la estructura del programa.



Integración ascendente.

En la anterior figura se ilustra la integración ascendente, en donde componentes de nivel inferior se combinan para formar los grupos 1, 2 y 3. El grupo 1 se prueba usando el conductor 1, el grupo 2 usando el conductor 2 y el grupo 3 con el conductor 3. Los conductores 1 y 2 se remueven para que los grupos se pongan en interfaz directa con el componente B. De manera similar, se hace para el grupo 3.

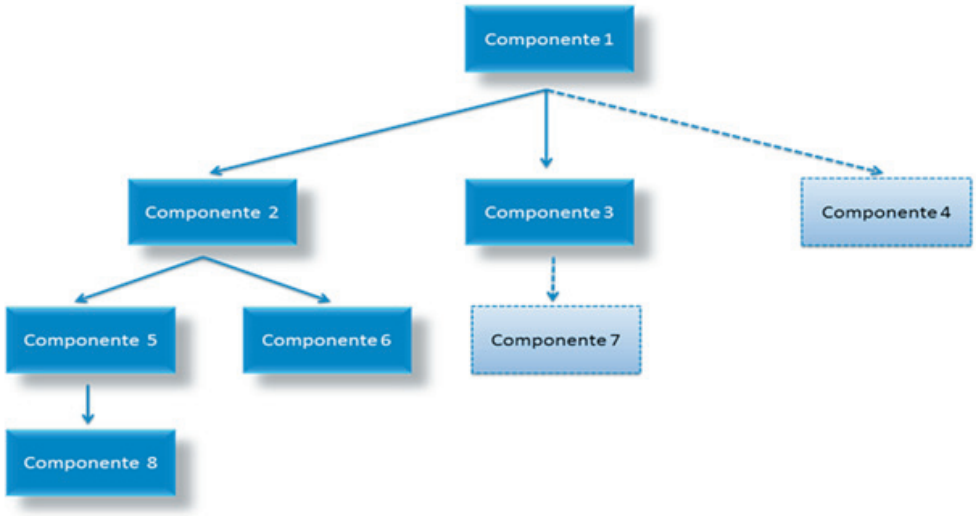
Finalmente los componentes B y C se integran con el componente A, y así sucesivamente según la arquitectura del programa.

2.3.5.1.2 Pruebas de integración descendente

En dichas pruebas se comienza con el módulo superior y se avanza hacía los módulos inferiores, siguiendo las siguientes fases:

- El módulo de control principal se usa como conductor de pruebas, y los módulos inmediatamente subordinados son reemplazados por resguardos.
- Los resguardos se sustituyen por los módulos reales.

- Cada vez que se integra un módulo nuevo, se realiza una prueba.
- Se realizan pruebas de regresión.



Integración descendente.

En la anterior figura se muestra un ejemplo de cómo se integran los módulos o componentes, definiendo una ruta de control mayor según la estructura del programa. En primer lugar, se selecciona la ruta izquierda en la cual se integran los componentes 1, 2 y 5; posteriormente se puede integrar el componente 8 o el componente 6, según se decida. Luego se construyen las rutas de control central y derecha, y se realiza un proceso similar al primero.

2.3.5.1.3 Pruebas de integración sandwich

En este tipo de pruebas se aplica la integración ascendente en los módulos inferiores, y de manera paralela se realiza integración descendente en los componentes superiores. La integración termina cuando ambas técnicas se encuentran en un punto intermedio de la estructura de componentes.

2.3.5.1.4 Pruebas de regresión

Cada vez que se agrega un nuevo componente o módulo en las pruebas de integración, el software cambia y se generan nuevas rutas en el flujo

de datos. De esta manera, se requieren prueba de regresión en las cuales se ejecute algún tipo de pruebas ya realizadas, sean de ascendentes, descendentes o sandwich.

2.3.6 PRUEBAS DE VALIDACIÓN

En la pruebas de validación se verifica el cumplimiento de los requisitos de usuario, con participación del desarrollador y el usuario. Se utiliza la técnica de caja negra, dividida en dos partes:

- Validación por parte del usuario de los resultados.
- Validación de utilidad, facilidad de uso y ergonomía de la interfaz de usuario.
- En las pruebas de validación se encuentran las pruebas alfa y las pruebas beta.
- Pruebas alfa: son realizadas por el cliente en el lugar de desarrollo, y se prueba el sistema aunque no estén terminadas todas las funcionalidades.
- Pruebas beta: Son realizadas por los potenciales consumidores en su entorno, sin presencia del desarrollador.

2.3.7 PRUEBAS DEL SISTEMA

En este tipo de pruebas se verifica el cumplimiento de los requisitos especificados, probando el sistema integrado en su entorno de hardware y software.

2.3.8 PRUEBAS DE ACEPTACIÓN

Estas pruebas son realizadas en el entorno del usuario, para validar la aceptación por parte del cliente, comprobando que el sistema está listo para ser implantado. El usuario puede definir los casos de prueba.

2.4 DISEÑO DE CASOS DE PRUEBA

Se busca seleccionar un conjunto de casos de prueba que tengan la mayor probabilidad de detectar el mayor número posible de errores y fallos, minimizando los recursos empleados para ello.

Se siguen los siguientes pasos:

- Aplicar análisis de valor límite.
- Diseñar casos con la técnica de participación de equivalencia.
- Utilizar la técnica de conjetura de error, combinado entradas y usando casos típicos de error.
- Utilizar técnicas de caja blanca

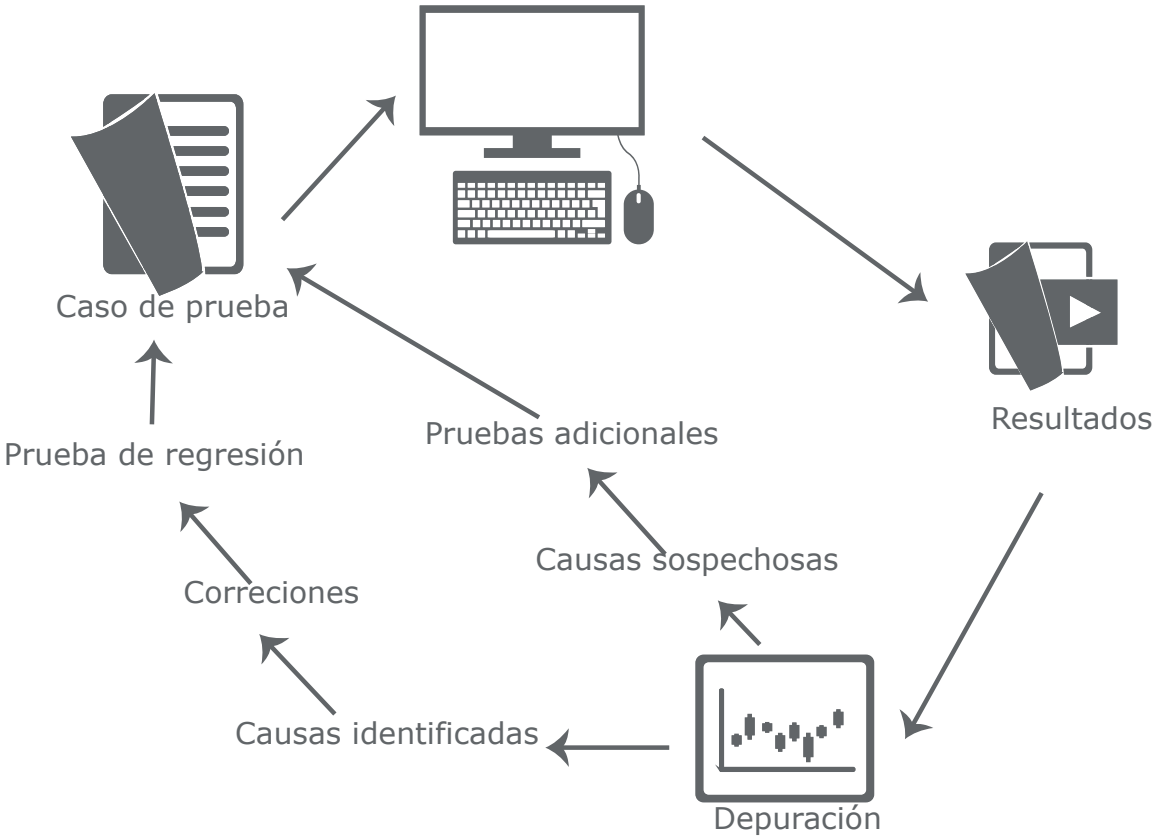
Se pueden utilizar diversos formatos para el diseño de los casos de prueba, los cuales deben incluir entre otros:

- Identificación del caso de prueba
- Caso de uso a probar
- Datos de entrada
- Salidas esperadas
- Secuencia de pasos a seguir
- Estado
- Fecha y hora de realización
- Responsable

Se proporciona como ejemplo un formato para la especificación de caso de prueba en los archivos: Plantilla_Caso_de_prueba.xls y Plantilla_caso_prueba.doc

2.5 DEPURACIÓN DE ERRORES

La depuración de errores se realiza en una dinámica cíclica como lo muestra la siguiente figura.



Proceso de depuración de errores.}

El proceso de depuración inicia con la ejecución de un caso de prueba, valorando los resultados para encontrar la falta de correspondencia entre el valor real y el esperado. Se busca encontrar la causa de la falta y así corregir el error, pero si no se encuentra, se puede tener una causa sospechosa, caso en el cual se realizan pruebas auxiliares para la corrección del error en forma iterativa.

2.6 DOCUMENTACIÓN DE PRUEBAS

Según el estándar IEEE 829-1983, las pruebas deben generar los siguientes documentos mínimos:

- Plan de pruebas.
- Especificación de requerimientos para el diseño de casos de prueba.

- Caso de prueba con descripción del procedimiento y del ítem a probar.
- Reporte de incidentes de prueba.
- Resumen de pruebas.

2.6.1 DISEÑO DEL PLAN DE PRUEBAS

Aunque el plan de pruebas se utiliza en la fase de pruebas, se debe definir en la etapa de diseño, que es cuando se especifica la arquitectura del sistema.

Los principales elementos que se deben especificar en un plan de pruebas de acuerdo con el estándar IEEE 829-2008 son:

- El alcance y los riegos asociados
- Los objetivos y el criterio de finalización de las pruebas (condiciones de suspensión y reanudación)
- El tipo de técnicas a aplicar.
- Método o estrategia de pruebas y tiempo disponible
- Los recursos requeridos para realizar las pruebas: personal, entorno de pruebas, presupuesto
- Priorización de los casos de prueba
- Responsabilidades
- Programación de los casos de prueba

Se proporciona como ejemplo el archivo ***Plantilla_Plan_de_pruebas.doc***, basado en este estándar.

GLOSARIO

Prueba: Proceso mediante el cual se ejecuta de manera sistemática un conjunto de actividades (métodos y técnicas) para encontrar errores.

Caso de prueba: Conjunto de condiciones, datos o variables que servirán para determinar si los requisitos del sistema se cumplen de manera parcial, completa, o no se cumplen.

Error: Discrepancia entre el valor calculado y el valor teórico o esperado, con responsabilidad del desarrollador.

Defecto software: Desviación en el valor esperado por una cierta característica. Defecto de calidad.

Fallo: Consecuencia de un error o un defecto software.

Overflow: Significa desbordamiento en el buffer, cuando la cantidad de datos supera la capacidad pre-asignada. Es un fallo de programación.

Underflow: Significa subdesbordamiento del buffer, cuando se carga datos a una velocidad inferior a la de procesamiento, provocando bloqueos.

BIBLIOGRAFÍA

BOLAÑOS, D., SIERRA, A., & ALARCÓN, M. (2008). *Pruebas de Software y JUnit.* Madrid: Pearson Prentice Hall.

CATALDI, Z. (2000). *Metodología de diseño, desarrollo y evaluación de software educativo. Tesis de Magíster en Informática.* Argentina: Facultad de Informática. Universidad Nacional de la Plata (UNLP).

PRESSMAN, R. (2006). *Ingeniería del Software: Un enfoque práctico. Sexta edición.* McGrawHill.

IEEE Standard Glossary of Software Engineering Terminolgy [IEEE, 1990].

Control de documento Construcción Objeto de Aprendizaje Pruebas de Software	
Desarrollador de contenido Experto temático	José Ricardo Arismendi Santos
Asesor pedagógico	Rafael Neftalí Lizcano Reyes Claudia Milena Hernández
Producción Multimedia	Luis Fernando Botero Mendoza Victor Hugo Tabares
Programadores	Daniel Eduardo Martínez
Líder expertos temáticos	Ana Yaqueline Chavarro Parra
Líder línea de producción	Santiago Lozada Garcés