INGENIERÍA DE SOFTWARE
Programador Universitario

INGENIERÍA DE SOFTWARE II
Licenciatura en Informática

AÑO 2014

Lic. María Isabel Mentz

Facultad de Ciencias Exactas y Tecnología
Universidad Nacional de Tucumán

1. Ingeniería de Software

1.1. Introducción

Ya que actualmente las economías personales, empresariales, nacionales e internacionales dependen cada vez más de las computadoras y sus sistemas de software, es que la práctica de la Ingeniería de Software tiene por objeto la construcción de grandes y complejos sistemas en forma rentable.

Los problemas que se presentan en la construcción de grandes sistemas de software no son simples versiones a gran escala de los problemas de escribir pequeños programas en computadoras.

La complejidad de los programas pequeños es tal que una persona puede comprender con facilidad y retener en su mente todos los detalles de diseño y construcción. Las especificaciones pueden ser informales y el efecto de las modificaciones puede ser evidente de inmediato. Por otro lado, los grandes sistemas son tan complejos que resulta imposible para cualquier individuo recordar los detalles de cada aspecto del proyecto. Se necesitan técnicas más formales de especificación y diseño: debe documentarse adecuadamente cada etapa del proyecto y es esencial una cuidadosa administración.

El término Ingeniería de Software se introduce a fines de la década del 60, al producirse la llamada “Crisis del Software” a causa de la aparición de Hardware de 3ª generación. Estas nuevas máquinas eran de una capacidad muy superior a las máquinas más potentes de 2ª generación y su potencia hizo posible aplicaciones hasta entonces irrealizables.

Las primeras experiencias en la construcción de grandes sistemas de software mostraron que las técnicas de desarrollo hasta entonces conocidas eran inadecuadas. Entonces se presentó la urgente necesidad de nuevas técnicas y metodologías que permitieran controlar la complejidad inherente a los grandes sistemas de software.

Ingeniería de Software = Programming in the Large

Entonces, convenimos en definir:

Ingeniería de Software: Disciplina que trata de la construcción de grandes sistemas que no puede manejar un único individuo. Usa principios metodológicos de la Ingeniería para el desarrollo de sistemas, es decir, sistematiza el desarrollo de sistemas. Adquiere el nivel de una disciplina. Consta de aspectos técnicos y aspectos no técnicos.

El Ingeniero de Software debe tener profundos conocimientos de las técnicas de computación y debe poder comunicarse en forma oral y escrita. Debe comprender los problemas de administración de proyectos relacionados con la

producción de software y debe poder apreciar los problemas de los usuarios del software cuando no lo entienden.

La Ingeniería de Software abarca:

Métodos Incluyen el planeamiento del sistema, análisis de requisitos, diseño de estructura de datos y procedimientos algorítmicos, codificación y testeo.

Herramientas Son soportes automatizados y semiautomatizados para los métodos. Cuando este soporte es un sistema integrado para análisis, diseño, codificación y testeo recibe el nombre de herramienta CASE (COMPUTER AIDED SOFTWARE ENGINEERING).

Procedimientos Definen la secuencia en que se aplican los métodos, los derivables y los controles.

2. Técnicas de Prueba del Software

En oposición a la tarea constructiva que viene desarrollando el Ingeniero de Software hasta este punto, esa nueva etapa de prueba del software tiene un **sentido destructivo**, en cuanto se crean un conjunto de casos de prueba que intentan “demoler” el software que ha sido construido.

Glenn Myers, en su libro sobre prueba de software establece los siguientes **objetivos para la prueba:**

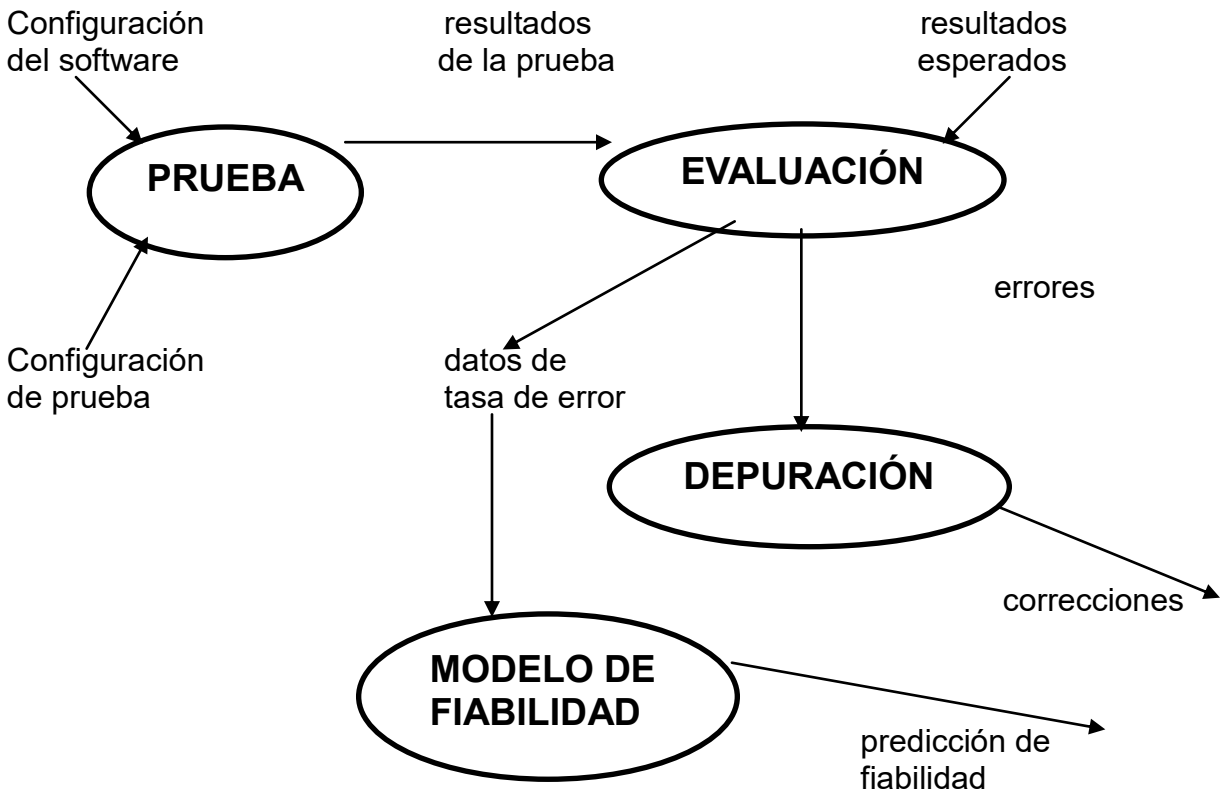
- La prueba es un proceso de ejecución de un programa que **intenta descubrir un error.**
- Un buen caso de prueba es aquél que tiene una **alta probabilidad de encontrar un error** que no había sido descubierto.
- **Una prueba tiene éxito si descubre un error** que no se había encontrado anteriormente.

Los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan un buen indicador de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo.

> **La prueba no puede asegurar la ausencia de errores, sólo puede demostrar que existen defectos en el software.**

2.1 **Flujo de Información de la Prueba**

El flujo de información para la prueba sigue el siguiente esquema:



Obsérvese que las entradas al proceso de prueba son: **1) La configuración del software**, que incluye la especificación de requisitos, las especificaciones de diseño y el código fuente y **2) la configuración de prueba** que incluye un plan y procedimiento de prueba, algunas herramientas de prueba, casos de prueba y resultados esperados.

Si en el proceso de prueba se encuentran muchos errores serios, la calidad y fiabilidad del sistema queda en duda y se requieren más pruebas.

Si se está seguro de que las pruebas son adecuadas para descubrir errores y el proceso de prueba no los descubre, sin duda el usuario sí los descubrirá y el ingeniero deberá corregirlos durante la fase de mantenimiento.

2.2 **Diseño de los casos de prueba**

El diseño de casos de prueba puede requerir tanto o más esfuerzo como la etapa del diseño inicial del producto.

Los métodos de diseño de casos de prueba proporcionan **un enfoque sistemático a la prueba y un mecanismo de ayuda para asegurar la completitud de las pruebas** y conseguir una mayor probabilidad de descubrimiento de errores en el software.

Existen **dos tipos** de pruebas:

- **Pruebas de caja negra**

Conociendo la función específica para la que se ha diseñado el producto de software, se pueden llevar a cabo pruebas que demuestren que **cada función es completamente operativa.**

En el caso del software para computadoras las pruebas se llevan a cabo sobre la interfaz del software. Los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada, que se produce una salida correcta y se mantiene la integridad de la información externa (archivos de datos). Este tipo de prueba **no tiene mucho en cuenta la estructura lógica interna del software.**

- **Pruebas de caja blanca**

Conociendo el funcionamiento del producto se pueden desarrollar pruebas que **aseguren que la operación interna se ajusta a las especificaciones** y que todos los componentes internos se han probado en forma adecuada.

Este tipo de prueba se basa en un minucioso examen de los detalles procedimentales. Se comprueban los mecanismos lógicos del software ya que conjuntos específicos de casos de casos de prueba ejercitan lazos y condiciones. Se puede examinar el estado de cada programa en varios puntos para analizar si coincide con el estado esperado.

El único problema que se presenta es que **para grandes sistemas, la prueba exhaustiva es impracticable.**

2.3 **Prueba de Caja Blanca**

Este método de prueba usa la estructura de control del diseño procedimental para derivar casos de prueba que satisfagan las siguientes **condiciones:**

- Garanticen que se prueba, por lo menos una vez, todos los **caminos independientes** de control de cada módulo.

- Ejerciten por lo menos una vez todas las **decisiones lógicas** en sus versiones verdadera y falsa.

- Ejecuten todos los **lazos** en sus límites y con sus límites operacionales.

- Ejerciten las **estructuras internas de datos** para asegurar su validez.

Este tipo de prueba se usa para descubrir errores que muchas veces pasan inadvertidos a la prueba de caja negra.

2.4 **Prueba de Caja Negra**

Este método de prueba se centra en los requisitos funcionales del software ya que permite al ingeniero de software obtener un conjunto de condiciones de entrada que ejerciten todos los requisitos funcionales de un programa. Es un enfoque complementario a las prueba de caja blanca que permite descubrir otros **tipos de errores** tales como:

1. Funciones incorrectas o ausentes.

2. Errores de interfaz.

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

4. Errores de rendimiento.

5. Errores de inicialización o de finalización.

La prueba de caja negra, generalmente, se aplica en etapas posteriores a las pruebas de caja blanca.

Las pruebas de este tipo se diseñan para responder a las siguientes **preguntas**:

- ¿Cómo se prueba la validez funcional?.
- ¿Qué clase de entradas serán buenos casos de pruebas?.
- ¿Es el sistema particularmente sensible a ciertos valores de entrada?.
- ¿De qué forma están aislados los límites de una clase de datos?.

- ¿Qué volumen y nivel de datos tolerará el sistema?.
- ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?.

2.5 **Conclusiones**

El principal objetivo del diseño de casos de prueba es derivar un conjunto de pruebas que ténganla mayor probabilidad de descubrir los defectos del software. Para ello se usan dos técnicas de diseño de casos de prueba: pruebas de caja blanca y pruebas de caja negra.

Las **pruebas de caja blanca** se centran en las estructuras de control del programa y los casos de prueba deben asegurar que se han probado por lo menos una vez todas las sentencias del programa y se han controlado todas las condiciones.

Este tipo de prueba se considera como una prueba a “pequeña escala” ya que típicamente es aplicada a pequeños componentes de programa. Por el contrario, la prueba de caja negra amplía en enfoque y se podría considerar como una “***prueba a gran escala”.***

Las **prueba de caja negra** son diseñadas para validar los requisitos funcionales sin fijarse en el funcionamiento interno de los programas. Se centran en el ámbito de información de un programa de forma que se proporcione una cobertura completa de prueba.

De todos modos, cuando el sistema se entrega al cliente que lo use, él se hará cargo de seguir llevando a cabo casos de prueba.

3. **Estrategias de prueba de software**

Una estrategia para probar el software **consiste en integrar las técnicas de diseño de casos de prueba** en una serie de pasos bien planificados que dan como resultado la correcta construcción del software. Además, proporciona una guía para el desarrollador del software, para el control de la calidad y para el usuario.

Obviamente, **cualquier estrategia de prueba debe incluir** la planificación de la prueba, el diseño de los casos de prueba, la ejecución de la prueba y la evaluación de los datos resultantes.

En otras palabras, **la prueba es un conjunto de actividades que se deben planificar y llevar a cabo sistemáticamente.**

En general, todas las **plantillas** para la prueba, que son un conjunto de pasos en los que se pueden situar las técnicas de diseño de casos de prueba, tienen las siguientes **características**:

- La prueba **comienza a nivel de módulo y trabaja hacia fuera,** hacia la integración de todo el sistema.
- **Diferentes técnicas de prueba son adecuadas en diferentes momentos**.
- La **prueba debe ser llevada a cabo** por el desarrollador y por un grupo de prueba independiente.
- La **prueba y la depuración** son actividades diferentes, pero la depuración debe incluir cualquier estrategia de prueba.

Una estrategia de prueba combina las **pruebas de bajo nivel**, que testean cada pequeño segmento de código para ver si se ha implementado correctamente, con **pruebas de alto nivel**, que demuestren la validez de las principales funciones del sistema frente a los requisitos del cliente.

3.1 **Verificación y Validación**

La prueba de software es parte de un concepto más amplio llamado verificación y validación.

- **Verificación** Conjunto de actividades que aseguran que el software implementa correctamente una función específica. **¿Estamos construyendo un producto correctamente?.**
- **Validación** Conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. **¿Estamos construyendo el producto correcto?.**

Los métodos de análisis, diseño y de implementación de software actúan para mejorar la calidad al proporcionar **técnicas uniformes y resultados predecibles.**

La prueba consiste en el último bastión desde el que se puede evaluar la calidad y corregir errores. Pero **la prueba no genera calidad**, si ésta no está presente en el software, tampoco lo estará al construir la prueba.

> **La calidad se incorpora al software durante el proceso de ingeniería, cuando se aplican adecuadamente los métodos y herramientas, y durante las revisiones formales efectivas.**

3.2 **Organización para la prueba del software**

En este momento se propone:

- El **desarrollador del software** es responsable de probar las unidades funcionales (módulos del programa), asegurándose que cada una lleva a cabo la función para la cual fue diseñada. Además se puede encargar de la prueba de Integración (paso de prueba que lleva a la construcción y prueba de la estructura total del sistema).

- Una vez que la arquitectura del software está completa entra a jugar un **grupo independiente de prueba**. Este grupo elimina el conflicto de interés presente cuando el constructor de software lo prueba. Trabajando estrechamente vinculados, el constructor y el grupo independiente de prueba, a lo largo del proyecto de software, aseguran que se realicen pruebas exhaustivas. Los errores deberán siempre ser corregidos por el desarrollador.

3.3 **Una Estrategia para la Prueba del Software**

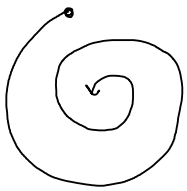
El proceso de ingeniería de software puede verse gráficamente en el siguiente **espiral:**

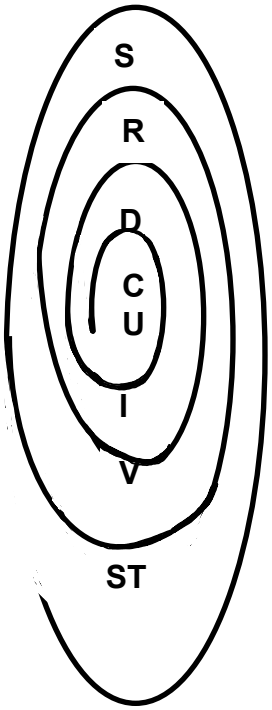
Ingeniería del Sistema

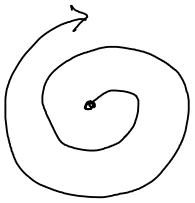
Requisitos

Diseño

Codificación







Prueba de Unidad

Prueba de Integración

Prueba de Validación

Prueba del Sistema

Inicialmente, la **Ingeniería de Software** define el papel del software y conduce el **análisis de requisitos** del software. En esta etapa se establece el campo de información, la funcionalidad, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Moviéndonos hacia el centro del espiral llegamos al **diseño** y por último a la **codificación.**

Para desarrollar el software para computadoras damos vueltas en espiral a través de una serie de **líneas que disminuyen el nivel de abstracción** en cada vuelta.

La estrategia de prueba de software que se propone es **movernos hacia afuera del espiral.**

La **prueba de unidad** comienza en el vértice del espiral y se centra en cada unidad de software tal como está implementada en código fuente.

La prueba avanza, moviéndose hacia fuera del espiral, hasta llegar a la **prueba de integración**, donde el foco de atención es el diseño y la construcción de la arquitectura del software.

Dando otra vuelta hacia fuera del espiral, encontramos la **prueba de validación**, donde se validan los requisitos establecidos en el análisis de requisitos, comparándolos con el sistema que se ha construido.

Finalmente, se llega a la **prueba del sistema** donde se prueba todo el software y otros elementos del sistema como un todo.

Cada vuelta del espiral aumenta el alcance de la prueba.

Dentro del contexto de la Ingeniería de software, el procedimiento de prueba es en realidad una serie de **tres pasos que se llevan a cabo secuencialmente:**

1. Inicialmente la prueba se centra en cada módulo asegurándose de que funciona adecuadamente como unidad. Esta prueba de unidad hace un uso intensivo de la pruebas de caja blanca, ejercitando caminos específicos de la estructura de control del módulo para asegurar un alcance completo y una detección máxima de errores.

2. A continuación se integran los módulos para formar el paquete de software completo. La prueba de integración se dirige a todos los aspectos relacionados con la verificación y la construcción del sistema. Las técnicas que prevalecen son las de diseño de casos de prueba de caja negra, aunque se pueden llevar a cabo algunos casos de prueba de caja blanca para asegurar que se cubren los principales caminos de control.

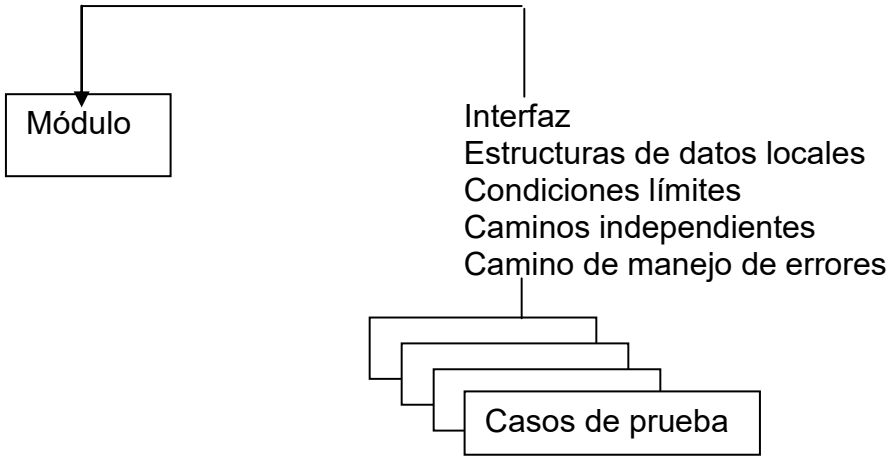
Una vez construido el sistema se llevan a cabo un conjunto de pruebas de alto nivel. Se debe comprobar los criterios de validación establecidos durante el análisis de requisitos. La prueba de validación controla que el software satisfaga los requisitos funcionales, de comportamiento y de rendimiento establecidos. Se usan exclusivamente pruebas de caja negra.

3. El software una vez validado se combina con otros elementos tales como software, usuarios, bases de datos, etc. La prueba de Sistema verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y rendimientos requeridos para el sistema en su totalidad.

3.4 **Pruebas de Unidad**

La prueba de unidad centra el proceso de verificación en la menor unidad de diseño**, el módulo**. Se usa como guía el modelo de **diseño detallado** y se prueban los caminos de control importantes con el fin de descubrir errores dentro del ámbito del módulo. Esta prueba está orientada a las pruebas de caja blanca y el paso de prueba se puede llevar a cabo en paralelo para múltiples módulos.

Las pruebas que se realizan como parte de la prueba de unidad se muestran en la siguiente figura:



Se prueba la **interfaz** del módulo para asegurar que la información fluye, desde y hasta el módulo, adecuadamente.

Se controlan las **estructuras de datos locales** para asegurar que los datos mantienen temporalmente su integridad durante todos los pasos del algoritmo.

Se prueban las **condiciones límites** para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento.

Se ejercitan los **caminos independientes** o básicos de la estructura de control para asegurar que todas las sentencias del módulo se ejecutan al menos una vez.

Finalmente, se prueban todos los **caminos de manejo de errores**.

Obviamente, al iniciar cualquier prueba es preciso **probar el flujo de datos de la interfaz del módulo** ya que, si los datos no ingresan correctamente, las pruebas restantes no tienen sentido. Las siguientes preguntas deben ser satisfechas:

¿El número de parámetros de entrada es igual al número de argumentos?.
¿Coinciden los atributos de los parámetros y los argumentos?.
¿Coinciden los sistemas de unidades de parámetros y los argumentos?.
¿El número de argumentos transmitidos a los módulos es igual al número de parámetros?.
¿Son iguales los atributos de los argumentos transmitidos que los atributos de los parámetros?.
¿Son iguales sus sistemas de unidades?.
¿Son correctos el número, los atributos y el orden de los argumentos de las funciones incorporadas?.
¿Existen referencias a parámetros que no estén asociados con el punto de entrada local?.
¿Entran solamente parámetros alterados?.
¿Las definiciones de variables globales son consistentes entre módulos?.
¿Se pasan las restricciones como argumentos?.

Si además el módulo lleva a cabo **Entrada/Salida externa**, se deben satisfacer, adicionalmente las siguientes preguntas:

¿Son correctos los atributos de los archivos?.
¿Son correctas las sentencias de apertura?.
¿Cuadran las especificaciones de formato con las sentencias de E/S?.
¿Cuadra el tamaño del buffer con el tamaño del archivo?.
¿Se abren los archivos antes de usarlos?.
¿Se tienen en cuenta las condiciones de EOF?.
¿Se manejan los errores de E/S?.
¿Hay algún error textual en la información de salida?.

Ya que las **estructuras de datos locales** son una fuente potencial de errores se deben diseñar pruebas para descubrir errores de las siguientes **categorías:**

1. **Tipificación impropia o inconsistente.**
2. **Inicialización o valores implícitos erróneos.**
3. **Nombres de variables incorrectos (mal escritos o truncados).**
4. **Tipos de datos inconsistentes.**
5. **Excepciones de desborde, por arriba o por abajo, o de direccionamiento.**

Además de las estructuras de datos locales, durante la prueba de unidad, se debe comprobar el **impacto de los datos globales sobre el módulo**.

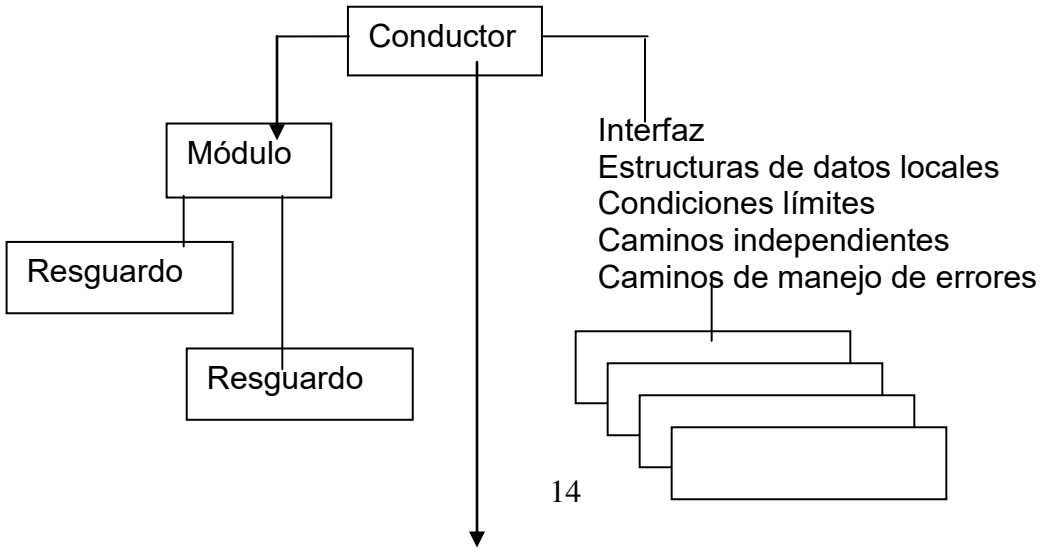
Se debe hacer una comprobación selectiva de los **caminos de ejecución**, diseñar casos de prueba para detectar errores debidos a **cálculos incorrectos,** comparaciones incorrectas o flujos de control inapropiados.

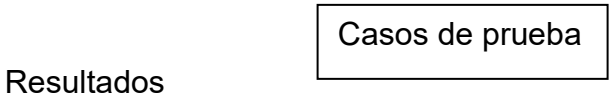
Las comparaciones y el flujo de control están fuertemente emparejados, por lo tanto, los casos de prueba deben descubrir errores tales como:

Comparaciones entre tipos de datos diferentes,
operadores lógicos o de precedencia incorrectos,
igualdad esperada cuando los errores de precisión la hacen poco probable,
variables o comparadores incorrectos,
finales de lazos inadecuados,
fallo de salida cuando se encuentra una iteración divergente,
variables de control de lazos modificadas de forma inadvertida o inadecuada,
etc.

La **prueba de límites** es la última tarea en el paso de prueba de unidad. Es muy importante ya que el software falla en condiciones límites, es decir, cuando se procesa el n-ésimo elemento de un arreglo n-dimensional, cuando se hace la i-ésima iteración en un lazo de i pasos o cuando se encuentran los valores máximo o mínimo permitidos. Los casos de prueba que ejerciten las estructuras de datos, el flujo de control y los valores de datos por encima de los máximos o por debajo de los mínimos son muy apropiados para descubrir errores.

El diseño de casos de prueba de unidad comienza una vez que se ha desarrollado, revisado y verificado en sus sintaxis el código fuente. **Se repasa la información de diseño para tener una guía para definir los casos de prueba** que tendrán una mayor probabilidad de descubrir errores de las categorías antes mencionadas. **Cada caso de prueba debe ir acompañado de un conjunto de resultados esperados.**





Ya que el módulo no es un programa independiente, para cada prueba de unidad se debe escribir un cierto software que conduzca **(DRIVER)** o resguarde **(RESGUARDO)** la prueba.

En la mayoría de los casos, un **DRIVER** no es más que un programa principal que acepte los casos de prueba, pase esos datos al módulo a probar, e imprima los resultados relevantes.

Por el contrario, los **RESGUARDOS** reemplazan módulos que son llamados por el módulo que se está probando. Un resguardo es un subprograma que usa la interfaz del módulo que lo contiene, lleva a cabo una mínima manipulación de datos, imprime una verificación de entrada y vuelve el control al módulo.

Ambos tipos de programas son **software adicional que no forma parte del producto del software final.** De acuerdo al grado de simplicidad que tenga el sistema pueden ser fáciles de escribir. Hay casos en que su uso se pospone hasta que se haga la prueba de integración.

La prueba de unidad se simplifica cuando se diseña un módulo con un **alto grado de cohesión**, ya que cuando el módulo se dirige a una función, se reduce el número de casos de prueba y los errores se pueden predecir y descubrir fácilmente.

3.5 **Prueba de Integración**

La prueba de integración es una **técnica para construir sistemáticamente la estructura del sistema de software** a la vez que se llevan a cabo pruebas para detectar errores asociados con la iteración. Obviamente, se espera que la estructura esté construida de acuerdo con el diseño.

Las razones por las que se producen errores al poner todos los módulos previamente probados, interactuando entre sí son:

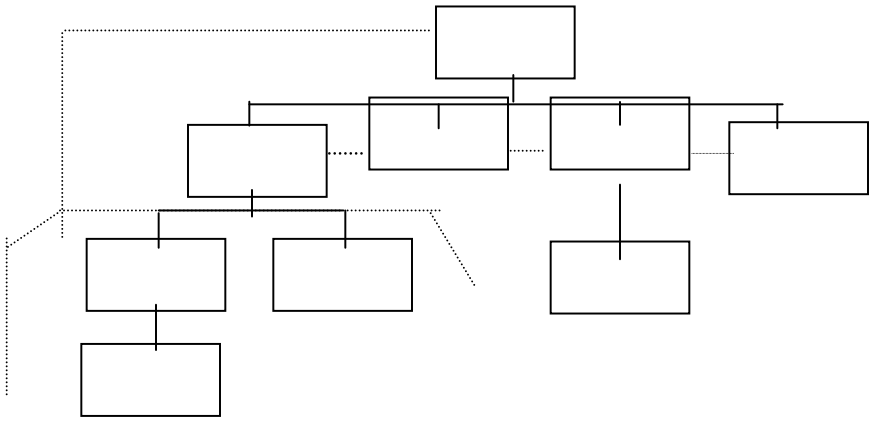
Se pueden perder datos en una interfaz,
Un módulo puede inadvertidamente afectar de una manera no deseada a otro,
Las subfunciones combinadas pueden no producir la función principal deseada,
Las estructuras de datos globales pueden presentar problemas,
El nivel de imprecisión aceptable para un módulo puede crecer hasta niveles inaceptables en el conjunto,
Etc.

Existe una tendencia a intentar una **integración no incremental** (enfoque Big-Bang) donde se combinan todos los módulos por anticipado y se prueba el programa en conjunto. En general esta metodología es caótica ya que se encuentran un gran número de errores, la corrección se hace muy difícil porque es muy complicado aislar la causa de los errores.

Por oposición, la **integración incremental** permite que el programa se construya y pruebe en pequeños segmentos fáciles de aislar y corregir y se puede aplicar un enfoque de prueba sistemático.

Integración Descendente

Se integran los módulos **moviéndose hacia abajo en la jerarquía de control**, comenzando por el programa principal. Los módulos subordinados se van incorporando en la estructura, ya sea por la forma primero-en –profundidad o bien la forma primero-en-ancho.



El proceso de integración se lleva a cabo en **5 pasos**:

1. Se usa el **módulo principal como conductor** de pruebas, disponiendo resguardos para todos los módulos directamente subordinados a él.

2. Se van **sustituyendo los resguardos subordinados** uno a uno por los módulos reales, de acuerdo a la forma de integración elegida.

3. Se llevan a cabo **pruebas** cada vez que se integra un módulo nuevo.

4. Tras terminar cada conjunto de pruebas, **se reemplaza otro resguardo** por el módulo real.

5. Se hace la **prueba de regresión**, es decir, todas o algunas de las pruebas anteriores, para asegurar que no se han producido nuevos errores.

El proceso se repite y continúa hasta que se haya construido la estructura del programa completo.

Integración Ascendente

Esta prueba comienza con la construcción y prueba de los **módulos atómicos** (módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el procesamiento requerido de los módulos subordinados está siempre disponible y no se hacen necesarios los resguardos.

La estrategia de integración se implementa mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en grupos llamados **construcciones** que realizan una subfunción específica.
2. Se escribe un **conductor** (programa de control de prueba) para coordinar la entrada y salida de casos de prueba.
3. **Se prueba el grupo.**
4. **Se eliminan los conductores y se combinan los grupos** moviéndose hacia arriba en la estructura del programa.

3.5.1 Ventajas y Desventajas

La principal **desventaja del enfoque descendente** es la necesidad de resguardos y la dificultad de prueba asociada a ellos. La principal ventaja es que se pueden probar de antemano las principales funciones de control.

La mayor **desventaja de la integración ascendente** es que el programa como entidad no existe hasta que se ha añadido el último módulo. Pero permite una mayor facilidad de diseño de casos de prueba y no hay que escribir resguardos.

En general, el mejor compromiso puede ser un planeamiento combinado, llamado **prueba sándwich**, que use el enfoque descendente para los niveles superiores de la estructura del programa y el enfoque ascendente para los niveles subordinados.

A medida que progrese la prueba, su encargado debe identificar los **módulos críticos**, que son los que tienen una o más de las siguientes características:

1) Está dirigido a varios requisitos del software.
2) Tiene un mayor nivel de control (está alto en la estructura).
3) Es complejo o propenso a errores.
4) Tiene requisitos de rendimiento muy definidos.

Los módulos críticos deben probarse lo antes posible, y la prueba de regresión debe centrarse en ellos.

3.6 **Prueba de Validación**

Una vez que el software está completamente ensamblado como un paquete, que se han encontrado y corregido los errores de interfaz, puede comenzar la **prueba de validación.**

> **La validación se logra cuando el software funciona de acuerdo con las expectativas del cliente, definidas en las especificaciones de requisitos.**

La validación de software se consigue mediante **una prueba de caja negra** que demuestran la conformidad con los requisitos. Se hace un plan de pruebas y se usa un procedimiento de prueba para definir los casos de prueba específicos que se usarán para demostrar la conformidad con los requisitos. Ambos estarán diseñados para asegurar que se satisfacen con los requisitos funcionales, que se alcanzan los requisitos de rendimiento, que la documentación es correcta e inteligible y que alcanzan los requisitos de portabilidad, compatibilidad, recuperaciones de errores, facilidad de mantenimiento, etc.

Una vez que se procede con cada caso de prueba de validación puede darse **una de dos condiciones:**

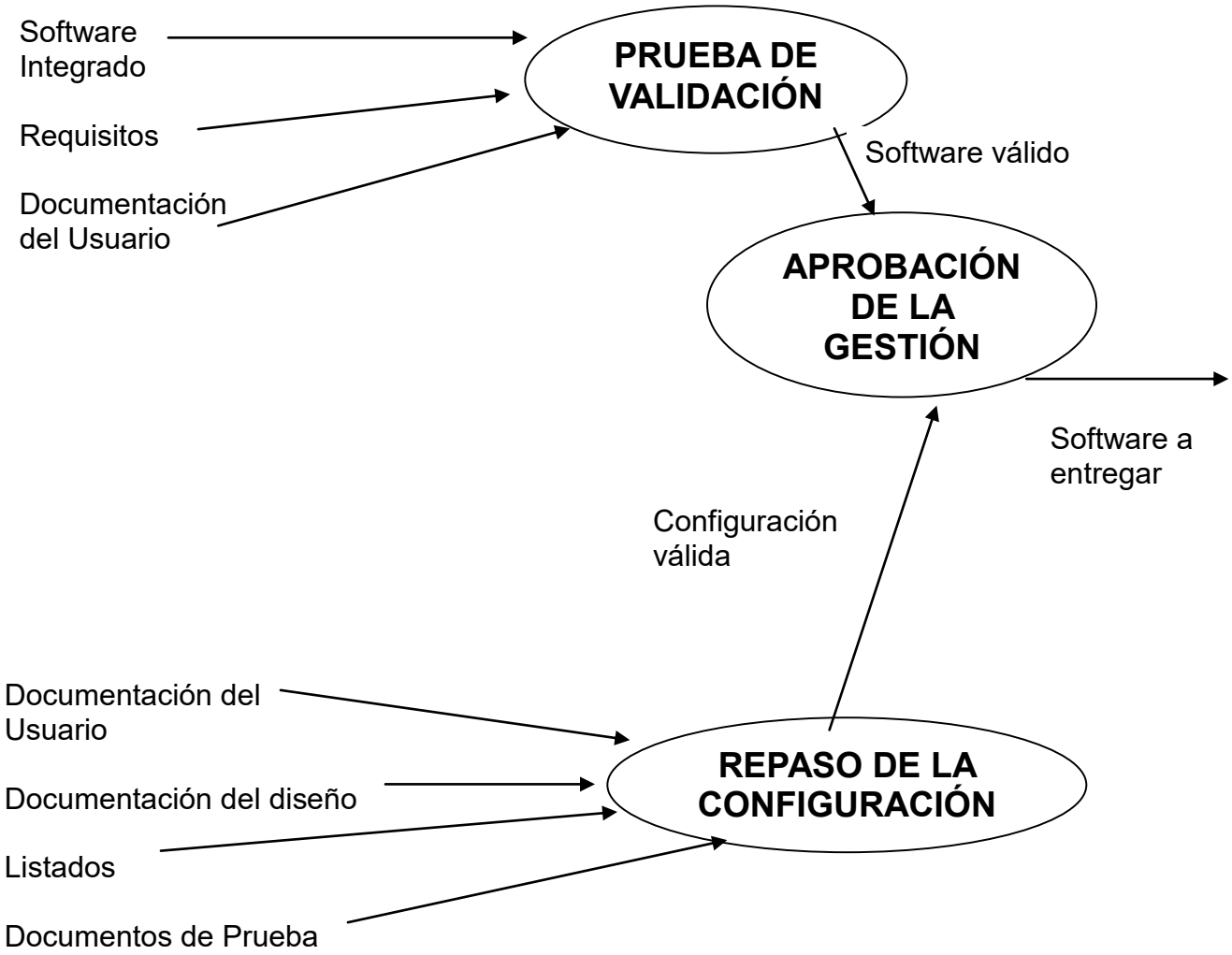
1) Las características de funcionamiento o rendimiento están de acuerdo a las especificaciones y **son aceptables**, o

2) Se descubre una desviación de las especificaciones y se crea una **lista de deficiencias**. Estos errores generalmente se corrigen al terminar el plan negociando con el cliente un método para resolverlos.

Un elemento importante de la prueba de validación es el **repaso de la configuración** (a veces llamado **auditoría**) que intenta asegurar que todos los

elementos de la configuración de software se han desarrollado adecuadamente, están catalogados y tienen suficiente detalle para facilitar el mantenimiento.

Esta etapa o proceso se ilustra en la siguiente figura:



Pruebas Alfa y Beta

Cuando se construye software a medida para un cliente, se llevan a cabo una serie de **pruebas de aceptación** para que él mismo valide todos los requisitos. Esto se debe a que es imposible prever cómo el cliente usará realmente el programa debido a errores de interpretación delas instrucciones de uso, en el sentido, por ejemplo, de que una salida puede ser muy clara para quien realiza la prueba pero ininteligible para un usuario normal.

Una prueba de aceptación puede ir desde el informal paso de prueba hasta la ejecución sistemática de una serie de prueba bien planificadas.

La **prueba Alfa** es conducida por el cliente en el lugar de desarrollo, con el desarrollador presente para registrar errores y problemas de uso. Es decir, la prueba Alfa se lleva a cabo en un **ambiente controlado.**

La **prueba Beta** se lleva a cabo en ambientes del cliente y es conducida por el usuario final del sistema. El desarrollador normalmente no está presente, ya que se trata de una aplicación en vivo en un **entorno no controlado** por el equipo de desarrollo. Los resultados, anotados, de esta prueba llevan a modificaciones para producir un producto para toda la base de clientes.

3.7 Prueba del Sistema

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

Prueba de Recuperación

El tiempo de recuperación ante los fallos y la tolerancia del sistema a los mismos es muy importante en los sistemas de software. En general, esto de alguna manera nos dice que los fallos de procesamiento no deben provocar el cese en el funcionamiento del software.

Una **prueba de recuperación** fuerza el fallo del software de alguna manera y verifica que la recuperación se lleve a cabo adecuadamente.

Si se ha previsto una **recuperación automática**, es decir que el propio sistema la lleva a cabo, se debe controlar la correctitud de los datos y el re-arranque.

Si, por el contrario, la recuperación requiere de la participación humana, se evalúan los tiempos medios de reparación para ver si están dentro de los límites aceptables.

Prueba de Seguridad

Cualquier sistema computacional que maneje información sensible es objeto de **actividades piratas**, que intentan entrar en los sistemas para obtener ganancias personales ilícitas.

La **prueba de seguridad** intenta verificar que los mecanismos de protección incorporados al sistema lo protegerán de las entradas o accesos impropios.

Durante la prueba, el encargado de la misma debe intentar hacerse con las claves de acceso por cualquier medio externo al oficio, puede atacar al sistema con software diseñado para romper las claves, debe bloquear el sistema negando el servicio a otras personas, debe producir a propósito errores del sistema, intentando entrar durante la recuperación, etc.

Con suficiente tiempo y recursos, toda buena prueba de seguridad finalmente accede al sistema. El rol del diseñador del sistema es hacer que el costo de entrar sea mayor que el valor de la información obtenida al hacerlo.

Prueba de Resistencia

Las pruebas de resistencia están diseñadas para enfrentar al sistema con **situaciones anormales**. El responsable de la prueba debe preguntarse, ¿a qué potencia se puede ponerlo a trabajar antes de que falle?.

Durante la prueba de resistencia se hace que el **sistema demande recursos** en cantidad, frecuencia o volúmenes anormales, y se estudia como reacciona el sistema ante estas anormalidades.

Prueba de Rendimiento

La prueba de rendimiento se diseña para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado.

Este tipo de prueba se ha venido haciendo durante todos los pasos del proceso de prueba, pero solamente se puede asegurar el rendimiento del sistema cuando están integrados todos los elementos del mismo.

4. El arte de la Depuración

La depuración aparece como consecuencia de una prueba efectiva ya que, si un caso de prueba descubre un error, **la depuración es el proceso que resulta en la eliminación del error.**

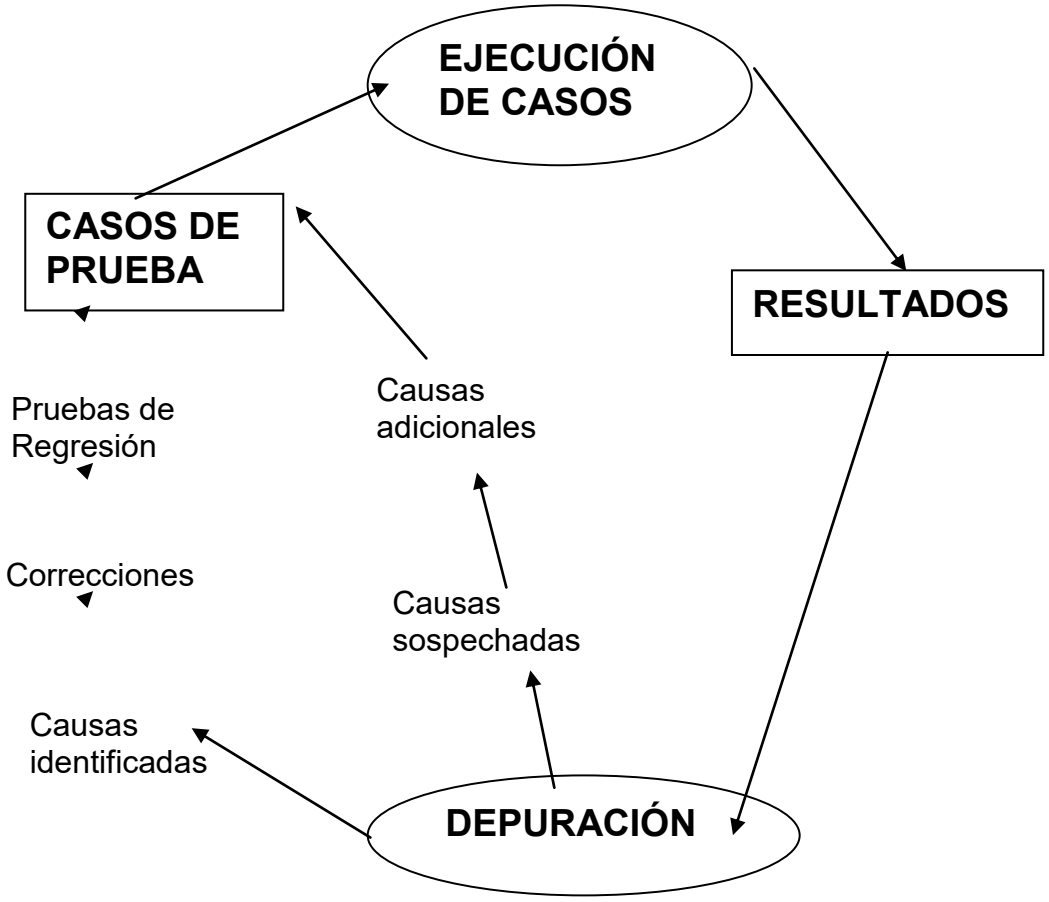
Aunque la depuración puede y debe ser un proceso ordenado, es el único proceso que aún conserva características que lo definen como artístico ya que es un **proceso mental que conecta un síntoma con una causa.**

4.1 El Proceso de Depuración

La depuración no es una tarea de prueba, pero sí **es una consecuencia de la prueba.**

En la siguiente figura se muestra como el proceso de depuración comienza con la ejecución de un caso de prueba, cuando se evalúan los resultados y no se corresponden con los esperados.

Estos datos que no corresponden, en muchos casos, son un síntoma de una causa que permanece oculta. La depuración se encarga de buscar esa causa, llevando así a la corrección del error.



Si no se puede encontrar la causa, se diseñan casos de prueba que ayuden a validar las sospechas de donde está hasta que se la encuentra, se corrige y se eliminan los errores.

La razón del porqué de la depuración se dice que está más asociada a atributos personales que a una metodología, está relacionada con las características de los errores que se producen:

- El síntoma y la causa pueden ser remotos entre sí.

- El síntoma puede desaparecer temporalmente al corregir otro error.
- El síntoma puede producirse por una causa que no sea un error (redondeos).
- El error puede ser humano y de difícil detección.
- El síntoma puede ser resultado de problemas de tiempo y no de procesamiento.
- Puede ser difícil reproducir en forma precisa las condiciones de entrada (en un sistema de tiempo real el orden de la entrada no está determinado).
- El síntoma puede aparecer en forma intermitente.
- El síntoma puede producirse debido a causas que se distribuyen por un serie de tareas ejecutándose en diferentes procesadores.

En general, existen **tres enfoques** para proponer la depuración:

- **Fuerza Bruta:** Se ponen controles (escrituras) en los programas hasta que alguien logra detectar el error. Es el enfoque más común y menos eficiente.

- **Vuelta Atrás:** Se usa con éxito en pequeños programas, recorriendo manualmente el código hacia atrás, partiendo desde donde se detectó el síntoma, hasta que se llega a ubicar el error.

- **Eliminación de causas:** Se usa la inducción o deducción particionando en forma binaria el dominio de los datos relacionados con la ocurrencia del error, con el fin de aislar las posibles causas. Se llega a una hipótesis de causa y se usan los datos anteriores para probar o revocar esa hipótesis. Si la hipótesis es prometedora, se refinan los datos a fin de intentar aislar el error.