

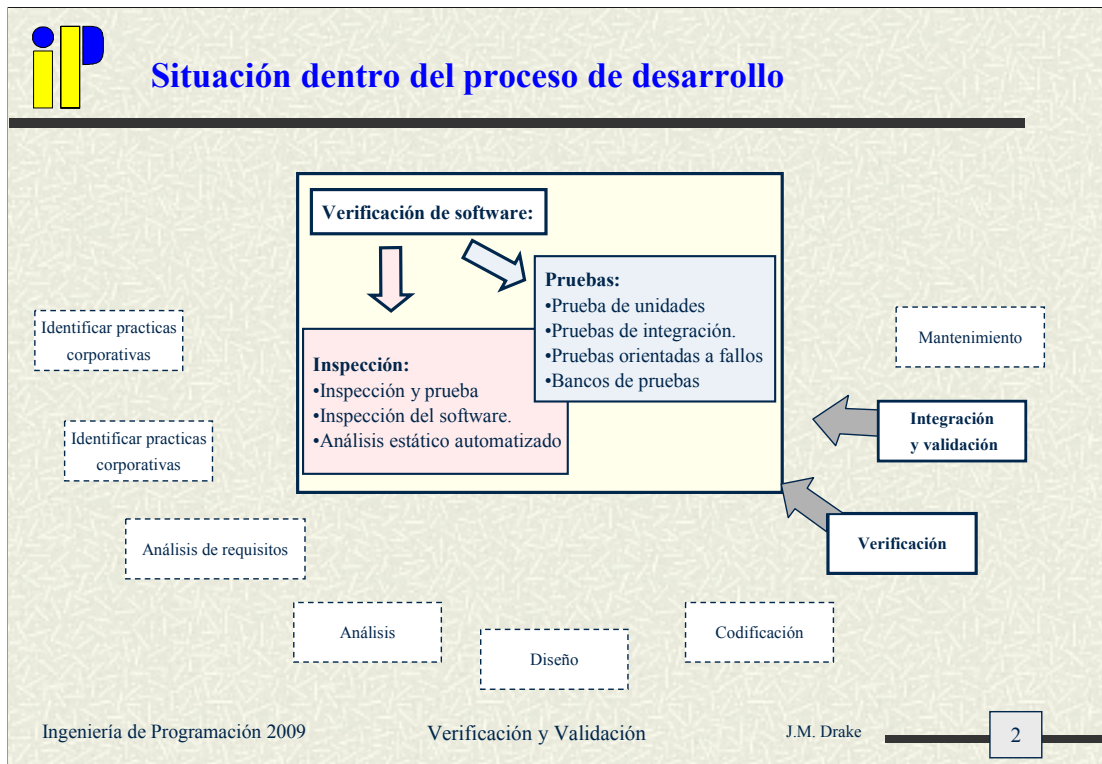


Ingeniería software
4° de Físicas

Verificación y Validación

Ctr José M. Drake y Patricia López
Computadores y Tiempo Real

Ingeniería de Programación 2009 1



El objetivo de este tema es introducir la verificación y validación del software con énfasis en las técnicas de verificación estática y en la prueba dinámica de código. Objetivo de este tema son:

- Comprender la diferencia entre verificación y validación del software.
- Valorar la inspección del software y el análisis estático como métodos de descubrir fallos y mejorar la calidad del software.
- Conocer las técnicas de pruebas para descubrir fallos en el código.
- Analizar las técnicas específicas para las pruebas de componentes y pruebas de sistemas orientados a objetos.
- Importancia de las herramientas CASE para la verificación de software y apoyar el desarrollo de las pruebas.



Verificación y Validación

- **Verificación y Validación (V&V):** Conjunto de procesos de comprobación y análisis que aseguran que el software que se desarrolla está acorde a su especificación y cumple las necesidades de los clientes.
- Existen actividades de V&V en cada etapa del proceso de desarrollo del software
- **Verificación:**
 - ¿Estamos construyendo el producto correctamente?
 - Se comprueba que el software cumple los requisitos funcionales y no funcionales de su especificación.
- **Validación:**
 - ¿Estamos construyendo el producto correcto?
 - Comprueba que el software cumple las expectativas que el cliente espera
- **Importante:** Nunca se va a poder demostrar que el software está completamente libre de defectos

La verificación y validación es el nombre que se da a los procesos de comprobación y análisis que aseguran que el software que se desarrolla está acorde a su especificación y cumple las necesidades de los clientes. La V&V es un proceso de ciclo de vida completo. Inicia con las revisiones de los requerimientos y continúa con las revisiones del diseño y las inspecciones del código hasta la prueba del producto. Existen actividades de V&V en cada etapa del proceso de desarrollo del software. La verificación y la validación no son la misma cosa, aunque es muy fácil confundirlas, Boehm (1979) expresó la diferencia entre ellas de forma sucinta:

- **Verificación:** ¿Estamos construyendo el producto correctamente?
El papel de la verificación comprende comprobar que el software está de acuerdo con su especificación. Se comprueba que el sistema cumple los requerimientos funcionales y no funcionales que se le han especificado.
- **Validación:** ¿Estamos construyendo el producto concreto?
La validación es un proceso más general. Se debe asegurar que el software cumple las expectativas del cliente. Va más allá de comprobar si el sistema está acorde con su especificación, para probar que el software hace lo que el usuario espera a diferencia de lo que se ha especificado.

Es importante llevar a cabo la validación de los requerimientos del sistema de forma inicial. Es fácil cometer errores y omisiones durante la fase de análisis de requerimientos del sistema y, en tales casos, el software final no cumplirá la expectativas de los clientes. Sin embargo, en la realidad, la validación de los requerimientos no puede descubrir todos los problemas que presenta la aplicación. Algunos defectos en los requerimientos solo pueden descubrirse cuando la implementación del sistema es completa.



Técnicas de Verificación y Validación

Inspecciones del Software:

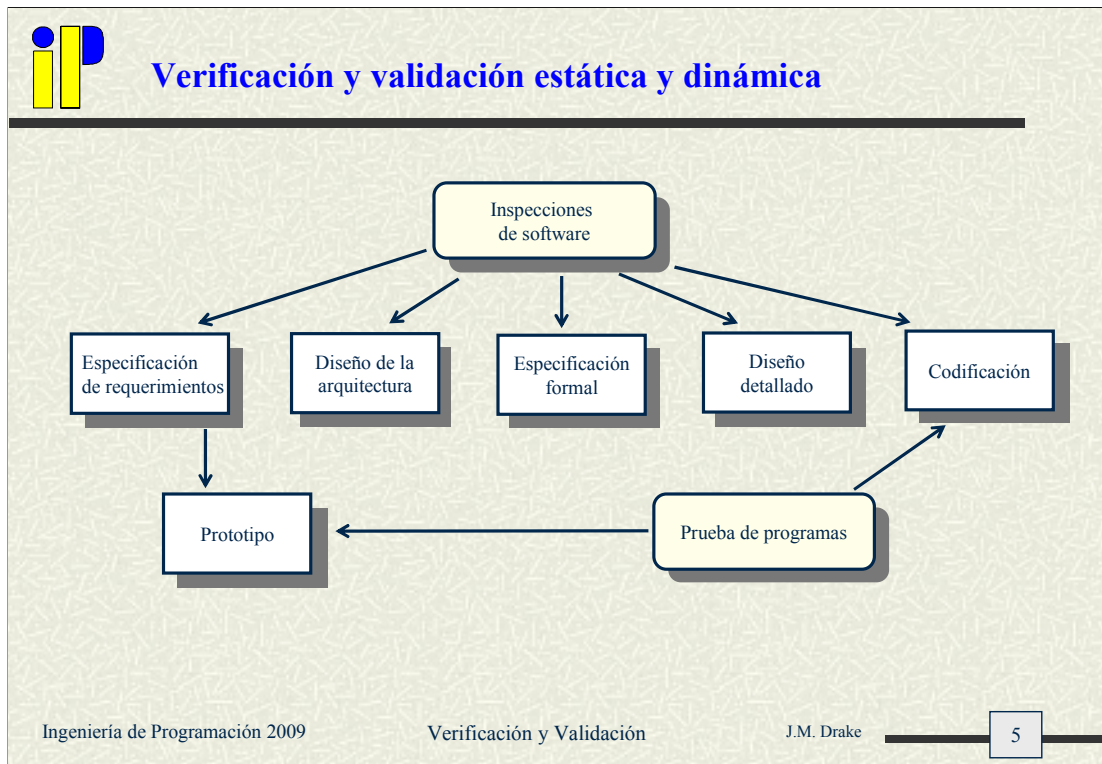
- Se analizan las diferentes representaciones del sistema (diagramas de requerimientos, diagramas de diseño y código fuente) en búsqueda de defectos.
- Son técnicas de validación estáticas => No requieren que el código se ejecute
- Debe realizarse durante todo el ciclo de desarrollo.

Pruebas del Software:

- Se contrasta dinámicamente la respuesta de prototipos ejecutables del sistema con el comportamiento operacional esperado.
- Técnicas de validación dinámicas => El sistema se ejecuta
- Requiere disponer de prototipo ejecutables, por lo que sólo pueden utilizarse en ciertas fases del proceso

Dentro del proceso de Verificación y validación se utilizan dos técnicas de comprobación y análisis de sistemas:


1. **Las inspecciones del software** analizan y comprueban las representaciones del sistema como el documento de requerimientos, los diagramas de diseño y el código fuente del programa. Se aplica a todas las etapas del proceso de desarrollo. Las inspecciones se complementan con algún tipo de análisis automático del texto fuente o de los documentos asociados. Las inspecciones del software y los análisis automatizados son técnicas de verificación y validación estáticas puesto que no requieren que el sistema se ejecute.
2. **Las pruebas del software** consiste en contrastar las respuestas de una implementación del software a series de datos de prueba y examinar las respuestas del software y su comportamiento operacional, para comprobar que se desempeñe conforme a lo requerido. Llevar a cabo las pruebas es una técnica dinámica de la verificación y validación ya que requiere disponer de un prototipo ejecutable del sistema.



En el esquema se muestra el lugar que ocupan las inspecciones y las pruebas dentro del proceso de desarrollo de software. Las flechas indican las fases del proceso en las que se utilizan las técnicas. Las inspecciones de software se pueden utilizar en todas las etapas del proceso, mientras que las técnicas de prueba sólo se pueden cuando está disponible un prototipo o código ejecutable.

Las técnicas de inspección incluyen inspección de programas, análisis automatizado de código fuente y verificación formal. Sin embargo las técnicas estáticas sólo pueden comprobar la correspondencia entre un programa y su especificación (verificación) y no puede probar que el software es de utilidad operacional, y mucho menos que las características no funcionales del software son las correctas. Por lo tanto, para validar un sistema de software, siempre se requieren llevar a cabo ciertas pruebas.

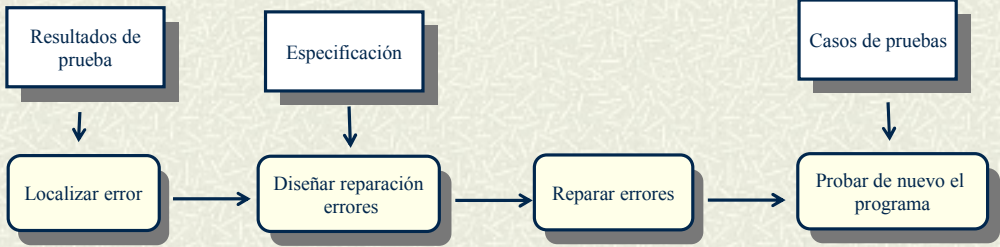
Aunque en la actualidad las inspecciones se utilizan ampliamente, las pruebas de los programas es aún la técnica de verificación y validación predominante.



Proceso de depuración

- Proceso de depuración: Proceso que localiza y corrige los errores descubiertos durante la verificación y validación.
 - Es un proceso complicado pues no siempre los errores se detectan cerca del punto en que se generaron.
 - Se utilizan herramientas de depuración, que facilitan el proceso

- Después de reparar el error, hay que volver a probar el sistema (pruebas de regresión).
 - La solución del primer fallo puede dar lugar a nuevos fallos.



```

graph TD
    A[Resultados de prueba] --> B[Localizar error]
    C[Especificación] --> D[Diseñar reparación errores]
    E[Casos de pruebas] --> F[Probar de nuevo el programa]
    B --> D
    D --> G[Reparar errores]
    G --> F
      
```

Ingeniería de Programación 2009
Verificación y Validación
J.M. Drake

6

Al proceso de eliminación de los errores que se descubren durante las fases de prueba se denomina depuración. Es un proceso independiente que no tiene por qué estar integrado:

- La verificación y validación establece la existencia de defectos en el programa.
- La depuración es el proceso que localiza el origen y corrige estos defectos.

No existe un proceso sencillo para la depuración de programas. Los mejores depuradores buscan patrones en los resultados de las pruebas donde el defecto se detecta, y para localizar el defecto utilizan el conocimiento que tienen sobre el tipo de defecto, el patrón de salida, así como del lenguaje y proceso de programación. El conocimiento del proceso es importante. Los depuradores conocen los errores de los programadores comunes (como olvidad incrementar un contador, errores de direccionamiento de punteros en lenguaje C, etc.) y los comparan contra los patrones observados.

Localizar los fallos es un proceso complejo porque los fallos no necesariamente se localizan cerca del punto en que se detectan. Para localizar un fallo de un programa el programador responsable de la depuración tiene que diseñar programas de prueba adicionales que repitan el fallo original y que ayudan a descubrir el origen del fallo. En estos casos las herramientas de depuración que permiten rastrear el programa y visualizar los resultados intermedios es de una gran ayuda.

Las herramientas de depuración son habitualmente parte de las herramientas de apoyo al lenguaje y que sirven de base al compilador. Proporcionan un entorno especial de ejecución que permiten acceder a las tablas de símbolos del compilador, a través de ella a los valores de las variables del programa. Habitualmente, permiten controlar la ejecución paso a paso sobre el código del programa.

Después de que se descubre el origen del fallo en el programa, este debe corregirse y entonces reevaluar el sistema. Esto implica repetir de nuevo las pruebas anteriores (pruebas de regresión). Estas pruebas se hacen para comprobar que los cambios introducidos resuelven definitivamente el fallo y no introducen nuevos fallos. La estadística muestra que la reparación de un fallo frecuentemente es incompleta y además introduce nuevos fallos.



Inspecciones del software

- Consisten en revisiones sistemáticas tanto de los documentos generados como de los códigos fuentes con el único objetivo de detectar fallos.
 - Las inspecciones se pueden aplicar a la detección de fallos en cualquiera de los documentos generados
 - Permiten detectar entre un 60 y un 90% de los fallos a unos costes mucho más bajos que las pruebas dinámicas.
 - Permiten detectar múltiples defectos en una simple inspección, mientras que las pruebas solo suelen detectar un fallo por prueba.
 - Permite utilizar el conocimiento del dominio y del lenguaje, que determinan los principales tipos de fallos que se suelen cometer.
 - Las inspecciones son útiles para detectar los fallos de módulos, pero no detectan fallos a nivel de sistema, que ha de hacerse con pruebas.
 - La inspecciones no son útiles para la detección de niveles de fiabilidad y evaluación de fallos no funcionales.

Llevar a cabo pruebas sistemáticas de los programas requiere que se desarrollen, ejecuten y examinen diferentes pruebas. Este proceso es muy largo y caro. Cada ejecución de una prueba suele descubrir, en el mejor de los casos, un único fallo, ya que la caída del sistema o la corrupción de los datos que puede implicar hace que sea difícil encontrar el siguiente.

Por el contrario la inspección del software no requiere que el programa se ejecute como lo que se puede utilizar como técnica de verificación antes de que el sistema esté totalmente implementado. Durante una inspección, se examina el código fuente del sistema y se compara con la especificación del mismo que se dispone.

Se ha comprobado estadísticamente que la inspección es una técnica mucho más eficiente para la detección de errores que la verificación basada en pruebas. Es más barato encontrar errores a través de la inspección que con pruebas, y además se considera que el 60% de los errores se detectan mediante una inspección rutinaria, y hasta un 90% de los errores se detectan mediante una inspección sistemática.

Existen dos razones por las que las inspecciones son más efectivas que las pruebas:

- Varios defectos se detectan en una sola inspección. El problema con las pruebas es que sólo pueden detectar único fallo por prueba, ya que los defectos de la primera que se detecte puede afectar a la detección de las siguientes.
- Usa el conocimiento del dominio y del lenguaje de programación que se utiliza. En esencia, es más probable que los revisores vean los tipos de errores que comúnmente ocurre el lenguaje de programación particulares y en los tipos particulares de la aplicación.



Lista de fallos (1)

☛ Fallos de datos:

- ¿Las variables se inicializan antes de que se utilicen los valores?.
- ¿Todas las constantes tienen nombre?
- ¿El límite superior de los arrays es igual al tamaño de los mismos?
- Las cadenas de caracteres tienen delimitadores explícitos.
- ¿Existe posibilidad que el buffer se desborde?

☛ Fallos de control:

- Para cada instrucción condicional ¿Es correcta la condición?
- ¿Todos los ciclo terminan?
- ¿Los bloques están delimitados correctamente?
- En las instrucciones case ¿Se han tomado en cuenta todos los casos?
- Si se requiere un break ¿Se ha incluido?
- ¿Existe código no alcanzable?

☛ Fallos de entrada/salida:

- ¿Se utilizan todas las variables de entrada?
- Antes de que salgan ¿Se le han asignado valores a las variables de salida?
- ¿Provocan corrupción de los datos las entradas no esperadas?

El proceso de inspección siempre se realiza utilizando una lista de los errores comunes de los programadores. Esta se somete a discusión por el personal con experiencia y se actualiza frecuentemente según se vaya teniendo experiencia.

La lista de errores debe ser función del lenguaje de programación que se utilice. Por ejemplo un compilador Ada comprueba que las funciones tienen el número correcto de argumentos, mientras que C no. Muchos de los fallos en C tienen relación con los punteros, estos no se pueden producir en Java.

La cantidad de código que se puede inspeccionar debe estar limitado, ya que a partir de un tiempo de inspección se baja la atención y se hace ineficaz. Existen estudios que establecen que no deberían examinarse más de 125 líneas de código por sesión, y no más de 2 horas.



Lista de fallos (2)

⌘ Fallos de la interfaz:

- ¿Las llamadas a funciones y métodos tienen el número correcto de parámetros?
- ¿Concuerdan los tipos de los parámetros formales y reales?
- ¿Están los parámetros en el orden adecuado?
- ¿Se utilizan los resultados de las funciones?
- ¿Existen funciones o procedimientos no invocados?

⌘ Fallos de gestión de almacenamiento:

- Si una estructura con punteros se modifica, ¿se reasignan correctamente todos los punteros?
- Si se utiliza almacenamiento dinámico, ¿se asigna correctamente el espacio?
- ¿Se desasigna explícitamente la memoria después de que se libera?

⌘ Fallos de gestión de las excepciones:

- ¿Se toman en cuenta todas las condiciones de errores posibles?



Inspección del código fuente: Análisis estático automatizado

- ✦ Los analizadores estáticos de programas son herramientas de software que rastrean el texto fuente de un programa, en busca de errores no detectados por el compilador.
- ✦ En algunos lenguajes de programación no son muy útiles pues el compilador ofrece una gran información acerca de posibles errores (incluso en ejecución).
- ✦ Aspectos habitualmente analizados son:
 - **Análisis de flujo de control:** Identifica y señala los bucles con múltiples puntos de salida y las secciones de código no alcanzable.
 - **Análisis de utilización de datos:** Señala como se utilizan las variables del programa: Variables sin utilización previa, que se declaran dos veces, declaradas y nunca utilizadas. Condiciones lógicas con valor invariante, etc.
 - **Análisis de interfaces:** Verifica la declaración de las operaciones y su invocación. Esto es inútil en lenguajes con tipado fuerte (Java, Ada) pero si en los restantes (C no comprueba los parámetros de una operación).
 - **Análisis de la trayectoria:** Identifica todas las posibles trayectorias del programa y presenta las sentencias ejecutadas en cada trayectoria.
- ✦ Debe utilizarse siempre junto a inspecciones directas del código, pues existen errores que no pueden detectar, por ejemplo la inicialización de una variable a un valor incorrecto

Los analizadores estáticos de programas son herramientas de software que rastrean el texto fuente de un programa y detecta los fallos y anomalías. No requieren que el programa se ejecute, mas bien, analizan sintácticamente el código fuente e identifica las diferentes sentencias que contiene. Después puede detectar si las sentencias están bien formadas, hacer inferencias sobre los flujos de control, y de los valores que se asignan a las variables.

Constituyen un recurso para ayudar a la detección de los fallos. Su función no es habitualmente detectarlos, sino identificar las anomalías o situaciones heterodoxas que pueden serlo.



Pruebas del software

- ‡ Las pruebas se realizan en cuatro etapas:
 - Prueba de unidades (prueba de métodos y clases)
 - se prueba cada método y clase de forma independiente
 - Prueba de integración o de subsistemas
 - se prueban agrupaciones de clases relacionadas
 - Prueba de sistema
 - se prueba el sistema como un todo
 - Prueba de validación (o de aceptación)
 - prueba del sistema en el entorno real de trabajo con intervención del usuario final
- ‡ El descubrimiento de un defecto en una etapa requerirá la repetición de las etapas de prueba anteriores

A excepción de programas pequeños, los sistemas no se prueban como una unidad monolítica. Los sistemas se construyen a partir de subsistemas, que a su vez están se construyen a partir de módulos que están compuestos de operaciones y funciones. Por tanto, el proceso de prueba se lleva a cabo en etapas en las que las pruebas se aplican de forma incremental, en conjunto con la implementación del sistema.

Prueba de unidades: Se prueba cada componente individual de forma independiente (en OO cada clase, con sus correspondientes métodos). Encontrar fallos en módulos de forma aislada es mucho más fácil que encontrarlos cuando ya se encuentran ensamblados dentro de la aplicación.



Tipos de pruebas del software

- # Existen dos tipos diferentes de pruebas:
 - Las pruebas de defectos:
 - Buscan las inconsistencias entre un programa y su especificación.
 - Las pruebas se diseñan para buscar los errores en el código.
 - Demuestran la presencia, y no la ausencia, de defectos
 - Las pruebas estadísticas:
 - Buscan demostrar que satisface la especificación operacional y su fiabilidad.
 - Se diseñan para reflejar la carga de trabajo habitual.
 - Sus resultados se procesan estadísticamente para estimar su fiabilidad (contando el número de caídas del sistema) y sus tiempos de respuesta.

Llevar a cabo pruebas consiste en ejecutar el sistema con datos de entrada específicamente formulados para la prueba que se realiza. La prueba de insuficiencias o defectos del programa se obtienen analizando las respuesta que proporciona y buscando anomalías respecto de lo esperado. Las pruebas se pueden llevar a cabo durante la fase de implementación para verificar que el software se comporta tal como los pretendió el diseñador, y después de que la implementación está completa.

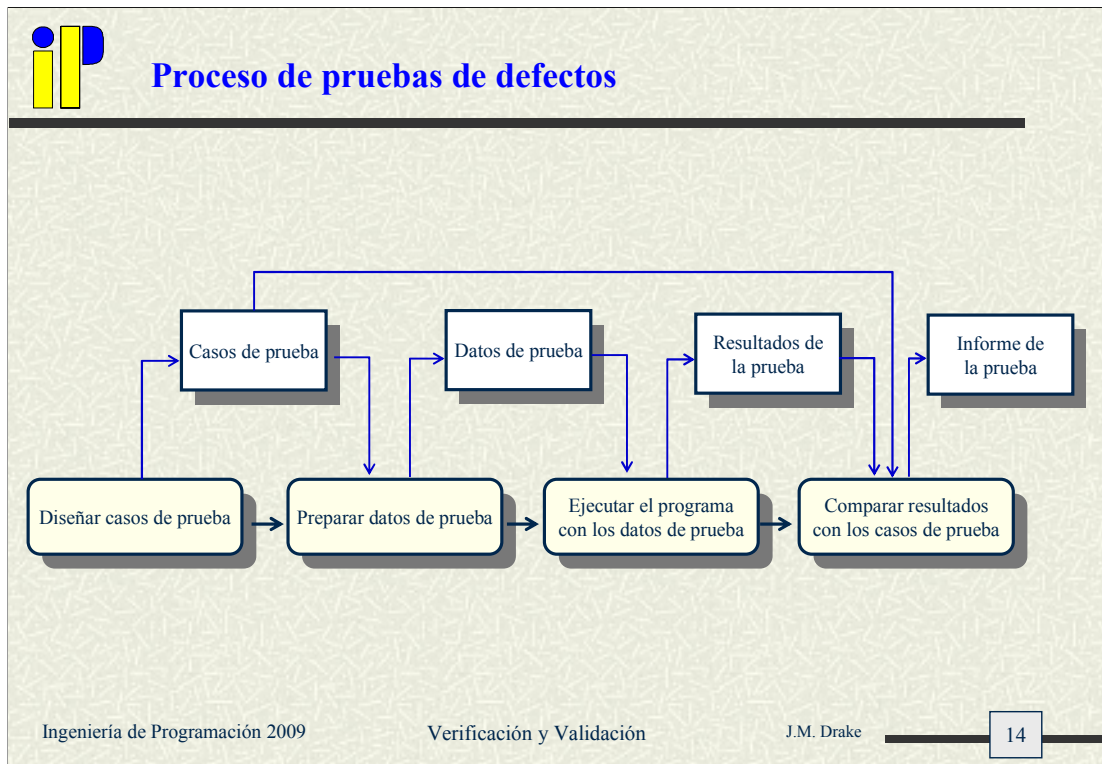
- Existen dos tipos diferentes de prueba, que se utilizan en las diferentes etapas de desarrollo del software:
 1. **Las pruebas de defectos:** Pretenden encontrar las inconsistencias entre un programa y su especificación. Estas inconsistencias se deben habitualmente a los fallos o defectos en el código del programa. Las pruebas se diseñan para revelar la presencia de defectos en el sistema, mas que para evaluar su capacidad operacional.
 2. **Las pruebas estadísticas:** se utilizan para probar el desempeño y la fiabilidad del programa y comprobar como trabaja bajo condiciones operacionales. Las pruebas se diseñan para reflejar las entradas de los usuarios y su frecuencia. Después de llevar a cabo las pruebas, se puede hacer una estimación de la fiabilidad operacional del sistema contando el número de caídas observadas en el sistema. La capacidad del programa se valora midiendo el tiempo de ejecución y el tiempo de respuesta del sistema cuando procesa los datos estadísticos de la prueba.
- No existe una separación clara entre estos dos tipos de pruebas. Durante las pruebas de defectos los desarrolladores obtienen una visión intuitiva de la fiabilidad, y durante las pruebas estadísticas, se descubren obviamente muchos fallos.



Pruebas de defectos

- El objetivo de las pruebas de defecto es detectar los defectos latentes de un sistema software antes de entregar el producto.
 - Una prueba de defectos exitosa es aquella que descubre un fallo, esto es, un comportamiento contrario a la especificación.
 - Las pruebas de defectos demuestran la existencia de un fallo, y no la ausencia de cualquier fallo.

- Las pruebas exhaustivas no son posibles y deben sustituirse por subconjuntos de casos de prueba. Se deben establecer políticas para establecer esos casos de prueba. Por ejemplo:
 - Que todas las instrucciones del programa se ejecuten al menos una vez
 - Se deben probar todas las funciones del sistema que se acceden a través de menú.
 - Si la entrada es introducida por el operador, todas las funciones deben probarse con entradas correctas e incorrectas.

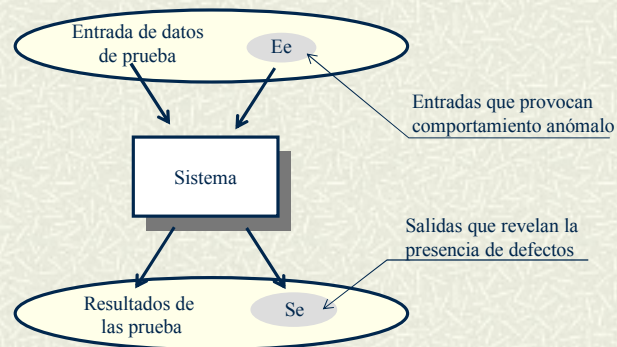


En la figura se muestra un modelo general del proceso de prueba de defectos. Los casos de prueba son especificaciones de las entradas a la prueba y de la salida esperada del sistema, mas una declaración de lo que se prueba. Los datos de prueba son las entradas seleccionadas para probar el sistema. Dichos datos se generan, algunas veces, de forma automática, pero esto no es frecuente, ya que en tal caso no se dispone de las respuesta que hay que esperar del sistema (sin fallo).



Pruebas funcionales (“Caja negra”)

- Las pruebas funcionales o de caja negras es una política de selección de casos de pruebas basado en la especificación del componente o programa.
- Las pruebas se seleccionan en función de la especificación y no de la estructura interna del software.



Las pruebas funcionales o de caja negra son una estrategia para seleccionar las pruebas de fallos basándose en las especificaciones de los componentes y programas, y no del conocimiento de su implementación. El sistema se considera como una caja negra cuyo comportamiento sólo se puede determinar estudiando las entradas y de contrastarlas con las respuestas que proporciona el sistema.

Este enfoque se puede aplicar de igual forma a los sistemas que están organizados como librerías de funciones, o como objetos. El probador introduce las entradas en los componentes del sistema y examina las salidas correspondientes. Si las salidas no son las previstas, entonces la prueba ha detectado exitosamente un fallo en el software.

El problema clave para el probador de defectos es seleccionar la entrada que tienen una alta probabilidad de ser miembro del conjunto Ee. En muchos casos la selección se basa en la experiencia previa de los ingenieros de pruebas. Ellos utilizan el conocimiento del dominio para identificar los casos de prueba que probablemente van a mostrar fallos. También se han propuesto enfoques sistemáticos de la selección de datos de prueba.



Particiones de equivalencia

- ✦ En general es imposible probar un método con todas las combinaciones posibles de entradas
- ✦ Solución: Clasificar los datos de entrada en grupos que tienen un comportamiento similar respecto de la lógica de programa.
- ✦ Una partición de equivalencia es cada grupo de datos de entrada que generan igual respuesta cualitativa.
- ✦ Los casos de pruebas se eligen en función de las particiones:
 - Se elige un caso central por partición: Caso típico.
 - Se elige casos correspondientes a las fronteras con otras particiones: casos atípicos.
- ✦ Las particiones se identifican utilizando la especificación del programa o la documentación del usuario.

Los datos de entrada a un programa se pueden clasificar en diferentes clases, de acuerdo con la funcionalidad del programa. Los datos de un mismo grupo tienen un tratamiento similar por parte del programa. Un enfoque sistemático para las pruebas de defectos se basan en identificar todas las particiones de equivalencia y basar en ellas las pruebas de fallos.

Una vez identificado un conjunto de particiones, se eligen los casos de prueba de cada una de estas particiones. Un buen criterio de selección de los casos de prueba es elegir dichos casos de los límites de las particiones junto con casos cercanos al punto medio de la partición. Con ello, se consideran los casos típicos para los programadores que son los que tienen un tratamiento lejano a las decisiones de control críticas, y valores atípicos que son los que tienen un tratamiento próximo a los casos en que se cambian de tratamiento.



Ejemplo de particiones de equivalencia

- ✦ Método que busca un número en una secuencia de números y devuelve la posición en la que se encuentra (-1 si no lo encuentra)

public int busca (int valor, int[] secuencia)

- ✦ Particiones de equivalencia:

- valorBuscado está en la secuencia / valorBuscado no está en la secuencia
- La secuencia tiene elementos / La secuencia no tiene elementos

- ✦ Casos de prueba

Secuencia	valor
[Vacía]	23
17	17
17	0
17,29,21,23	17
9,35,30,41,17	17
17,18,21,23,29,33,38	23
12,18,21,23,32	17



Pruebas estructurales (“Caja blanca”)

- # En las pruebas estructurales las pruebas se seleccionan en función del conocimiento que se tiene de la implementación del módulo.
- # Se suelen aplicar a módulos pequeños.
- # El probador analiza el código y deduce cuántos y qué conjuntos de valores de entrada han de probarse para que al menos se ejecute una vez cada sentencia del código.
- # Se pueden refinar los casos de prueba que se identifican con pruebas de caja negra.



Ejemplo de prueba estructural

```

Public static int search (int valor,
                        int[] secuencia)
{ int bottom= 0;
  int top =elemArray.length-1;
  int mid;
  int index = -1
  while (bottom <=top)
  { mid=(top+bottom/2;
    if (elemArray[mid] ==key)
      return index;
    else
      {if elemArray[mid]<key)
        bottom =mid+1;
        else
          top=mid-1; }
    return index;
  }
}

```

elemArray	key	r
17	17	1
17	0	-1
17,21,23,29	17	1
9,16,18,30,31,41,45	45	7
17,18,21,23,29,38,41	23	4
17,18,21,23,29,33,38	21	3
12,18,21,23,32	23	4
21,23,29,33,38	25	-1

El ejemplo Java, consiste en una función de búsqueda binaria que busca el entero key dentro del elemArray, retornando en el objeto r un campo booleano r.found que indica si ha sido encontrado, y un campo r.index que retorna la posición del elemento encontrado.

Las 8 pruebas que se proponen, hacen que todas las sentencias del código se ejecuten al menos una vez, y que en todas las sentencias condicionales se elijan cada una de las dos opciones que tienen.



Pruebas de integración

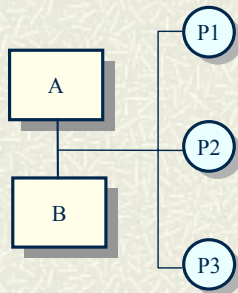
- # Se prueban la respuesta de grupos de módulos interconectados a fin de detectar fallos resultantes de la interacción entre los componentes.
- # Las pruebas de integración se realizan con referencia a las especificaciones del programa.
- # La principal dificultad de las pruebas de integración es la localización de los fallos.
- # Para facilitar la detección de los errores se utilizan técnicas incrementales.

- Una vez que se han probado los componentes individuales del programa, deben integrarse para crear un sistema parcial o completo. En el proceso de integración hay que probar el sistema resultante con respecto a los problemas que surjan de las interacciones de los componentes.
- Las pruebas de integración se desarrollan a partir de la especificación del sistema y se inician tan pronto como estén disponible versiones utilizables de alguno componentes del sistema.
- La principal dificultad que surge en las pruebas de integración es localizar los errores que se descubren durante el proceso. Existen interacciones complejas entre los componentes del sistema y cuando se descubre una salida anómala, es difícil encontrar la fuente de error.
- Para hacer más fácil la localización de errores, siempre se utiliza un enfoque incremental para la integración y prueba del sistema. De forma inicial se deben integrar un conjunto operativo mínimo, y probarlo. Luego se agregan nuevos componentes a esta configuración mínima y se prueba después de que se agrega cada incremento.

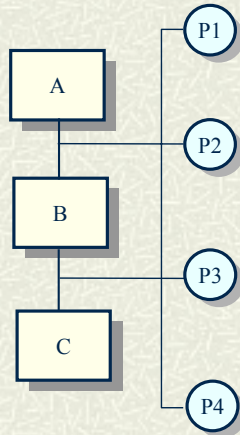


Pruebas de integración incremental

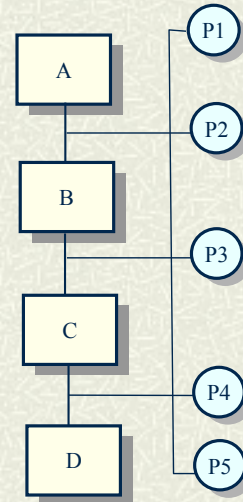
Secuencia de prueba 1



Secuencia de prueba 2



Secuencia de prueba 3





Niveles de pruebas en sistemas orientados a objetos

- En orientación a objetos, la separación entre pruebas de unidades y pruebas de integración no está muy clara, debido a las dependencias entre clases.
- En un sistema orientado a objetos se pueden identificar cuatro niveles de prueba:
 - **Probar las operaciones individuales asociadas a los objetos.**
 - **Probar clases de objetos individualmente (dependencias entre operaciones de la clase)**
 - **Probar grupos de objetos.**
 - **Probar el sistema.**
- Para probar una clase se requiere:
 - Pruebas aisladas de cada operación de su interfaz.
 - La verificación de los valores que se asignan a los atributos de la clases
 - La ejecución de los objetos en todos los estados posibles en la clase.

Las técnicas de pruebas para sistemas orientados a objetos han madurado rápidamente y hoy en día existe una gran cantidad de información disponibles sobre ellas.

En un sistema orientado a objetos se pueden identificar cuatro niveles de prueba:

Probar las operaciones individuales asociadas a los objetos: Es pruebas de funciones y se pueden utilizar las técnicas de caja negra y blanca.

Probar clases de objetos individualmente: Dado que las diferentes funciones de una clase están interrelacionadas requiere un método especial.

Probar grupos de objetos: En este caso nos son apropiadas las integraciones descendente o ascendente. Se requieren nuevos enfoques.

Probar el sistema: La verificación y validación es la comparación con la especificación de requisitos disponibles, y se realiza igual que en el caso funcional estructurado.



Herramienta JUnit

- # JUnit es un conjunto de clases que permite desarrollar y ejecutar conjuntos de pruebas de forma sencilla y sistemática.
 - Está integrada en el entorno Eclipse
 - Orientada a pruebas unitarias (prueba de clases en Java)

- # Los casos de prueba que se generan son clases Java, que pueden ser almacenadas y reejecutadas cuando sea necesario
=> Se generan bancos de pruebas asociados al proyecto

- # Configuración del proyecto para hacer uso de JUnit:
 - Botón derecho en el proyecto => *Configure Build Path* => *Add Library* => seleccionar *JUnit* => *next* => *Elegir JUnit 3.8.1*



Elementos del framework JUnit

- # **Casos de prueba** (Test Case): Clases que incluyen una serie de métodos para probar los métodos de una clase concreta. Por cada clase que queramos crear podemos crear una clase de prueba
- # **Métodos de prueba** : Son los métodos que la herramienta va a ejecutar automáticamente al ejecutar la clase de prueba.
- # **Aserciones** (Asserts): Las pruebas en JUnit se basan en programación con aserciones
- # **Grupos de pruebas** (Test Suite): Sirven para agrupar varias clases de prueba, y poder ejecutarlas todas seguidas.

En los últimos años se han desarrollado un conjunto de herramientas que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. Dicho conjunto se denomina *XUnit*. De entre dicho conjunto, **JUnit** es la herramienta utilizada para realizar pruebas unitarias en Java.

El concepto fundamental en estas herramientas es el **caso de prueba** (*test case*), y la suite de prueba (*test suite*). Los casos de prueba son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de caso de prueba. Mediante las suites podemos organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.

Las pruebas que se van construyendo se estructuran así en forma de árbol, de modo que las hojas son los casos de prueba, y podemos ejecutar cualquier subárbol (suite).

De esta forma, construimos programas que sirven para probar nuestros módulos, y que podremos ejecutar de forma automática. A medida que la aplicación vaya avanzando, se dispondrá de un conjunto importante de casos de prueba, que servirá para hacer pruebas de regresión. Eso es importante, puesto que cuando cambiamos un módulo que ya ha sido probado, el cambio puede haber afectado a otros módulos, y sería necesario volver a ejecutar las pruebas para verificar que todo sigue funcionando.

Aplicando lo anterior a Java, JUnit es un conjunto de clases *opensource* que nos permiten probar nuestras aplicaciones Java. Podemos encontrar información actualizada de JUnit en:



Creación de una clase de prueba (Test Case)

- # Pulsar con el botón derecho sobre la clase a probar y elegir New => Other => Java => JUnit => JUnit Test Case
- # En la ventana que aparece a continuación marcar que deseamos que se cree el método setUp() y pulsar next
- # Seleccionar los métodos que queremos probar y pulsar Finish
- # Se crea la clase de prueba, de nombre *<NombreClase>Test*, que extiende a *junit.framework.TestCase*
- # La clase incluye un método de prueba por cada método que se quiere probar, de nombre *test<nombreMetodo>*
- # Podemos añadir más métodos de prueba (también deberán comenzar por la palabra “test”)



Ejemplo: Clase de prueba para la clase Moda

```
import junit.framework.TestCase;

public class ModaTest extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
    }

    public void testNuevoDato() {
        fail("Not yet implemented");
    }

    public void testModa() {
        fail("Not yet implemented");
    }

    public void testNumVecesValor() {
        fail("Not yet implemented");
    }

    public void testNumDatos() {
        fail("Not yet implemented");
    }
}
```



Métodos de prueba

- ✦ Cuando se ejecute la clase de prueba, todos los métodos de prueba son ejecutados automáticamente.
- ✦ Debemos escribir en ellos los casos de prueba que queramos verificar.
- ✦ En general, la construcción de pruebas sigue siempre el mismo patrón:
 - Construcción de los objetos de prueba: El método *setUp()* de la clase de prueba se ejecuta antes de cada método de prueba, por lo que se emplea para inicializar variables u objetos en el estado que nos interese para cada prueba.
 - Verificación de propiedades de la clase a través de aserciones => Permiten verificar el correcto funcionamiento de los métodos invocados.
- ✦ La ejecución de un método de prueba puede tener tres resultados posibles:
 - Correcto: Finaliza sin producirse ningún fallo
 - Failure: Se ha producido un fallo esperado, de los que estamos comprobando a través del caso de prueba. Se lanza la excepción *AssertionFailedError*
 - Error: Se ha producido un error no esperado

En general la construcción de pruebas sigue siempre estos mismos patrones: construir los objetos de prueba, y llamar al método **assertTrue(...)** (o cualquier otro método *assertXXX(...)* de *TestCase*) para que verifique si cierta propiedad se ha cumplido o no. En el caso de que se cumpla, se supone que la prueba ha sido satisfactoria (en nuestro caso, comparar la matriz calculada con la que debería haber salido).



Aserciones

- Una aserción es una sentencia que nos permite probar una proposición que debería ser siempre cierta de acuerdo con la lógica del programa.
 - Ejemplo: Si invoco el método `clear()` en una lista (elimina todos los elementos)
 - Aserción de comprobación: `list.size() == 0`

- Cada aserción evalúa una expresión booleana que se supone que será cierta si el programa ejecuta correctamente
 - Si no es cierta, el sistema lanza una excepción, poniendo de manifiesto un error en el programa

- Uso de aserciones en JUnit, a través de estos métodos:
 - `void assertTrue (String msj, Boolean cond)`
 - Si `cond` es evaluada a `true` no pasa nada
 - Si `cond` es evaluada a `false`, se lanza la excepción `AssertionFailedError` y se muestra el mensaje `msj`.
 - Ej de la lista
 - `assertTrue("Error en clear", list.size()==0);`
 - `void fail(String msj):` Lanza siempre la excepción `AssertionFailedError` y muestra el mensaje `msj`. (Para mostrar fallos en excepciones)
 - Equivalente a `assertTrue (msj, false);`



Ejemplo

```
import junit.framework.TestCase;

public class ModaTest extends TestCase {

    Moda m;

    // Procedimiento de inicialización (ejecutado antes de cada método)
    protected void setUp() throws Exception {
        super.setUp();
        m = new Moda(3);
    }

    // Procedimiento de prueba para el constructor
    public void testModa() {
        //Numero de datos debe ser 0
        assertTrue("Error en constructor",m.numDatos()==0);
        for (int i=0; i<3; i++){
            assertTrue("Error en constructor para el valor+i, m.numVecesValor(i)==0);
        }
    }
}
```



Ejemplo (cont.)

```
// Procedimiento de prueba de Moda y NuevoDato
public void testModayNuevoDato() {
    // Cuando está vacío debe lanzar la excepción NoHayValores
    try {
        int valor = m.moda();
        fail("Excepcion NoHayValores no lanzada");
    } catch (NoHayValores e) {
    }
    //Añado un elemento
    m.nuevoDato(1);
    // El número de datos debe haber aumentado en 1
    assertTrue("Fallo actualización numDatos",m.numDatos()==1);
    // La moda debe ser 1
    try {
        assertTrue("Fallo Moda", m.moda()==1);
    } catch (NoHayValores e) {
        fail("No se debería haber lanzado la excepcion");
    }
    // Despues de llamar a moda todo se inicializa
    assertTrue("Fallo inicialización", m.numDatos()==0);
}
```



Ejemplo (cont.)

```
//Añado un conjunto de valores
m.nuevoDato(0);
m.nuevoDato(1);
m.nuevoDato(2);
m.nuevoDato(2);
m.nuevoDato(0);
m.nuevoDato(2);
m.nuevoDato(4);
//Comprobamos el número de datos y el numero de veces de cada valor añadido
assertTrue("El numero de datos = "+m.numDatos()+ "deberia ser 6",m.numDatos()==6);
assertTrue("Fallo en numero de veces valor", m.numVecesValor(0)==2);
assertTrue("Fallo en numero de veces valor", m.numVecesValor(1)==1);
assertTrue("Fallo en numero de veces valor", m.numVecesValor(2)==3);

// Comprobamos el valor de la moda
try {
    assertTrue("Fallo Moda", m.moda()==2);
} catch (NoHayValores e) {
    fail("No se deberia haber lanzado la excepcion");
}
}
```



Grupos de prueba

- Se pueden agrupar varias clases de prueba en un grupo de pruebas (Test Suite), de forma que se ejecuten todas juntas.

- Las pruebas que se van construyendo se estructuran así en forma de árbol, de modo que las hojas son los casos de prueba, y podemos ejecutar cualquier subárbol (suite).
 - P.e.: Agrupar todas las clases de prueba de las clases correspondientes a un paquete o a un grupo de clases relacionadas (Moda, MedMaxMin, Acumulador, etc.)

- Para crear un TestSuite
 - Botón derecho en un paquete => New => Other => Java=> JUnit=> JUnit Test Suite
 - En la ventana que aparece, damos nombre al grupo y seleccionamos las clases de prueba que queremos añadir al grupo



Ejecución y Resultados

Para ejecutar una clase de prueba o un grupo de prueba:

- Botón derecho en la clase de prueba => Run as => JUnit Test

Ejecución sin fallos

Ejecución con fallo esperado

Ejecución con fallo inesperado

Package Explorer Navigator JUnit
Finished after 0,047 sec
Runs: 3/3 Errors: 1 Failures: 1
tests.ModaTest [Runner: JUnit 3]
testModa
testModayNuevoDato
testNumeroVecesValor
Failure Trace
junit.framework.AssertionFailedError: Fallo Moda
at tests.ModaTest.testModayNuevoDato(ModaTest.java:34)