

# Ingeniería del Software

## Tema 4. Pruebas



Universidad de Alcalá

# Contenidos

---

- ▶ **Parte I. Conceptos generales**
  - ▶ Definiciones y conceptos
  - ▶ Limitaciones de las pruebas
  - ▶ Técnicas de prueba
- ▶ **Parte II. Pruebas unitarias con JUnit**
- ▶ **Parte III. Pruebas de integración con mock objects**
- ▶ **Parte IV. Pruebas de sistema**

---

# Parte I. Conceptos generales

# Pruebas de Programas

---

- ▶ El objetivo de las pruebas es mostrar que un programa hace lo que está destinado a hacer y descubrir los defectos del programa antes de que se ponga en uso.
- ▶ Cuando prueba el software, ejecuta un programa utilizando datos artificiales.
- ▶ Verifica los resultados de la ejecución de prueba para detectar errores, anomalías o información sobre los atributos no funcionales del programa.
- ▶ ¿Puede revelarse la presencia de errores? NO de su Ausencia.
- ▶ Las pruebas forman parte de un proceso de verificación y validación más general, que también incluye técnicas de validación estática

# Fines de las Pruebas de Programas

- ▶ Demostrar al desarrollador y al cliente que el software cumple con sus requisitos.
- ▶ Para el software personalizado, esto significa que debe haber al menos una prueba para cada requisito en el documento de requisitos. Para los productos de software genéricos, significa que debe haber pruebas para todas las características del sistema, más combinaciones de estas características, que se incorporarán en la versión del producto.
- ▶ Para descubrir situaciones en las que el comportamiento del software es incorrecto, indeseable o no se ajusta a su especificación.
- ▶ Las pruebas de defectos están relacionadas con eliminar el comportamiento no deseado del sistema, como los bloqueos del sistema, las interacciones no deseadas con otros sistemas, los cálculos incorrectos y la corrupción de datos.

# Fines de las Pruebas de Programas

---

- ▶ El primer objetivo lleva a las pruebas de validación.
- ▶ Espera que el sistema funcione correctamente utilizando un conjunto determinado de casos de prueba que reflejen el uso esperado del sistema.
- ▶ El segundo objetivo lleva a la prueba de defectos.
- ▶ Los casos de prueba están diseñados para exponer defectos. Los casos de prueba en prueba de defectos pueden ser deliberadamente oscuros y no necesitan reflejar cómo se usa normalmente el sistema.

# Fines de las Pruebas de Programas

---

## ► Pruebas de validación:

Demostrar al desarrollador y al cliente del sistema que el software cumple con sus requisitos.

Una prueba exitosa muestra que el sistema funciona según lo previsto.

## ► Prueba de defectos

Para descubrir fallas o defectos en el software donde su comportamiento es incorrecto o no conforme con su especificación

Una prueba exitosa es una prueba que hace que el sistema funcione incorrectamente y, por lo tanto, expone un defecto en el sistema.

# Verificación vs. Validación

---

## ▶ **Validación**

- ▶ Evaluación de un sistema o componente, durante o al final del proceso de desarrollo, para comprobar si se satisfacen los requisitos especificados
- ▶ *¿Estamos construyendo el sistema correcto?*

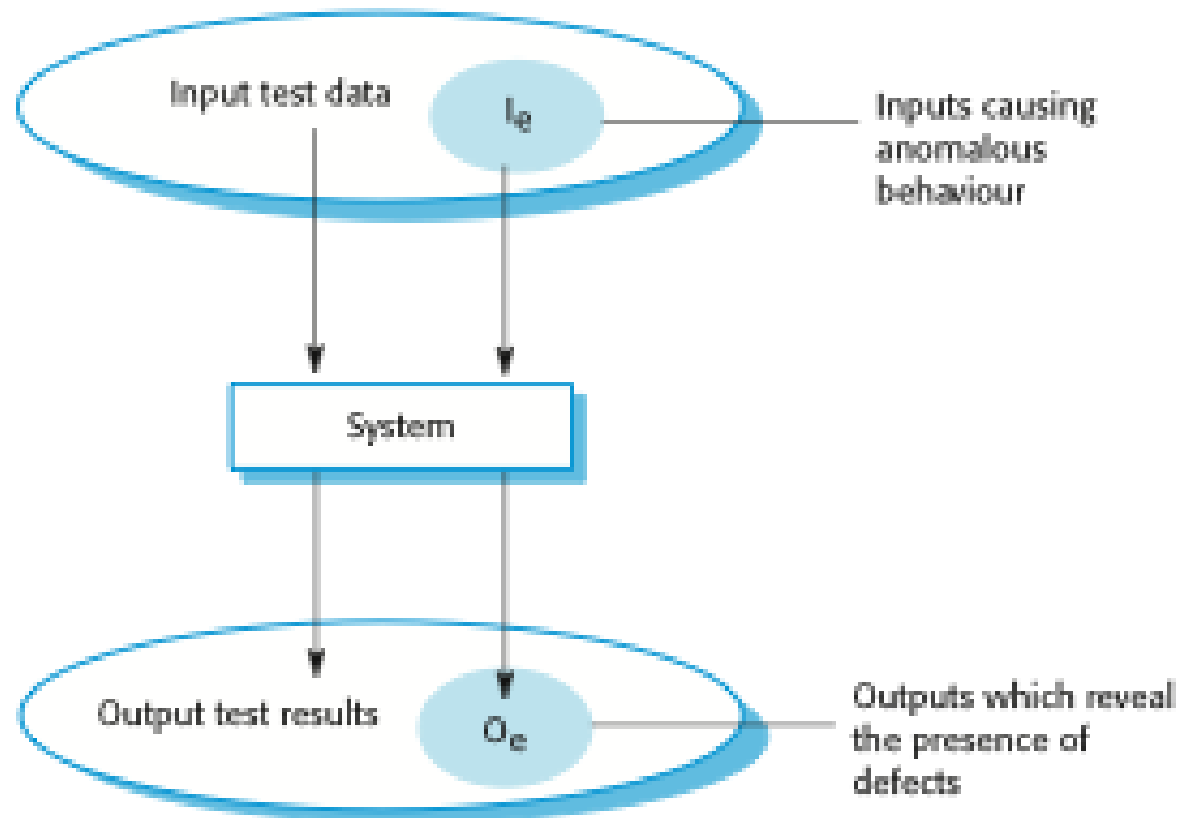
## ▶ **Verificación**

- ▶ Evaluación de un sistema o componente para determinar si los productos obtenidos satisfacen las condiciones impuestas
- ▶ Evaluación de la calidad de la implementación
- ▶ *¿Estamos construyendo correctamente el sistema?*



# Modelo entrada-salida de pruebas

---



# Verificación y validación

---

- ▶ Verificación: ¿"Estamos construyendo el producto correctamente"?  
El software debe ajustarse a su especificación.
- ▶ Validación: ¿"Estamos construyendo el producto correcto"?  
El software debe hacer lo que el usuario realmente requiere.

# Necesidad de la V & V

---

- ▶ El objetivo de V & V es establecer la confianza de que el sistema es "apto para su propósito". Depende del propósito del sistema, las expectativas del usuario y el entorno de marketing.
- ▶ Propósito del software: El nivel de confianza depende de cuán crítico sea el software para una organización.
- ▶ Expectativas del usuario: Los usuarios pueden tener bajas expectativas de ciertos tipos de software.
- ▶ Entorno de marketing: Obtener un producto en el mercado temprano puede ser más importante que encontrar defectos en el programa.

# Inspección y Prueba

---

- ▶ Inspecciones de software: análisis de la representación estática del sistema para descubrir problemas (verificación estática)

Puede ser complementado por documento basado en herramientas y análisis de código.

- ▶ Pruebas de software enfocadas a la observación del comportamiento del producto (verificación dinámica)

El sistema se ejecuta con datos de prueba y se observa su comportamiento operacional.

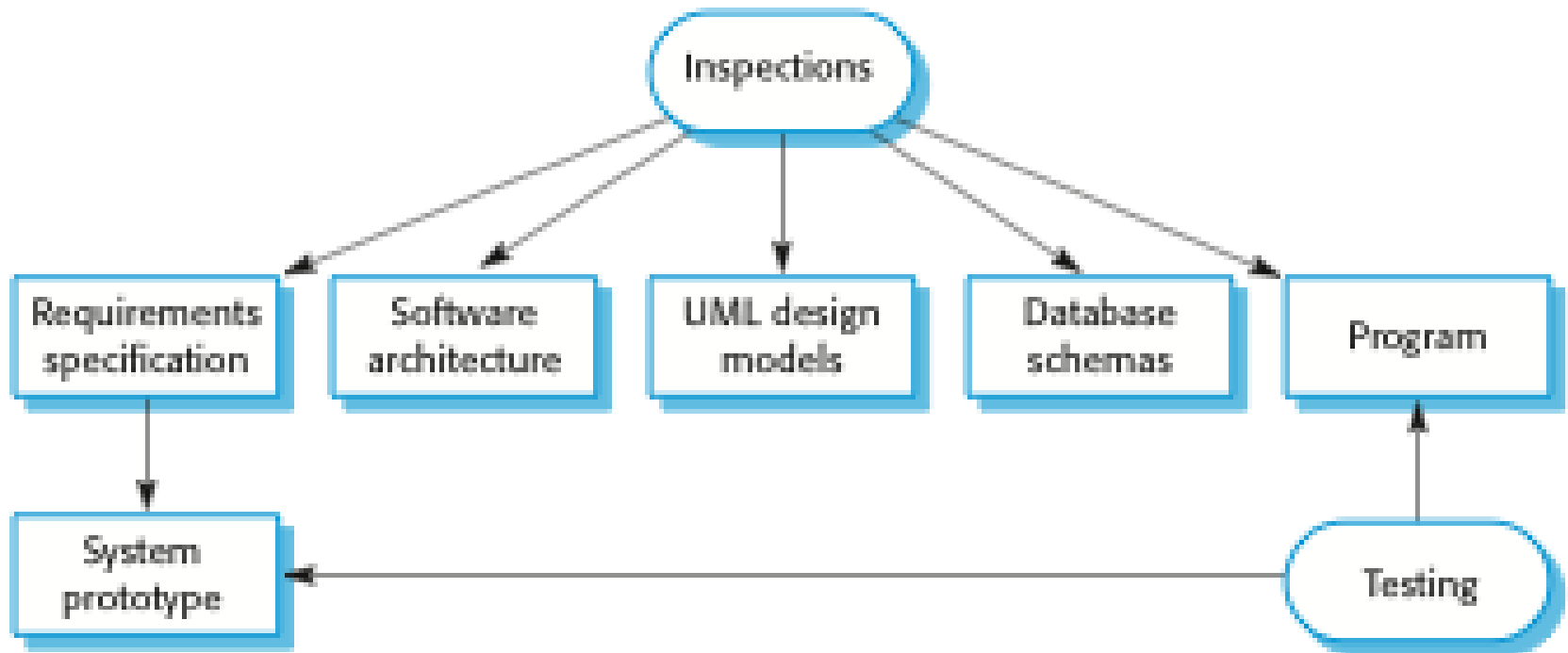
# Técnicas de prueba basadas en el código

---

- ▶ Partimos de que no puede hacerse una prueba exhaustiva (excepto en módulos triviales).
- ▶ **Pruebas de caja negra**
  - ▶ Basadas en el flujo de datos
  - ▶ Se usan cuando se quiere probar lo que hace el software pero no como lo hace
  - ▶ En Java cuando “miramos el javadoc”
- ▶ **Pruebas de caja blanca**
  - ▶ Basadas en el flujo de control
  - ▶ Pruebas unitarias que se usan cuando se conoce la estructura interna y el funcionamiento del código a probar
  - ▶ En Java, cuando “miramos el código”
- ▶ Ambas son dinámicas (es necesario ejecutar el software para detectar errores)

# Inspección y Prueba

---



# Inspección del Software

---

- ▶ Se trata de personas que examinan las representaciones del software con el objetivo de descubrir anomalías y defectos.
- ▶ Las inspecciones no requieren la ejecución de un sistema, por lo que se pueden usar antes de la implementación completa del software.
- ▶ Pueden aplicarse a cualquier representación del sistema (requisitos, diseño, datos de configuración, datos de prueba, etc.).
- ▶ Se ha demostrado que son una técnica eficaz para descubrir errores de programa.

# Ventajas de la Inspección

---

- ▶ Durante las pruebas, los errores pueden enmascarar (ocultar) otros errores. Debido a que la inspección es un proceso estático, no tiene que preocuparse por las interacciones entre los errores.
- ▶ Las versiones incompletas de un sistema se pueden inspeccionar sin costos adicionales. Si un programa está incompleto, no se necesita desarrollar artefactos de prueba especializados para probar las partes que están disponibles.
- ▶ Además de buscar defectos en los programas, una inspección también puede considerar atributos de calidad más amplios de un programa, como el cumplimiento de los estándares, la portabilidad y la capacidad de mantenimiento.



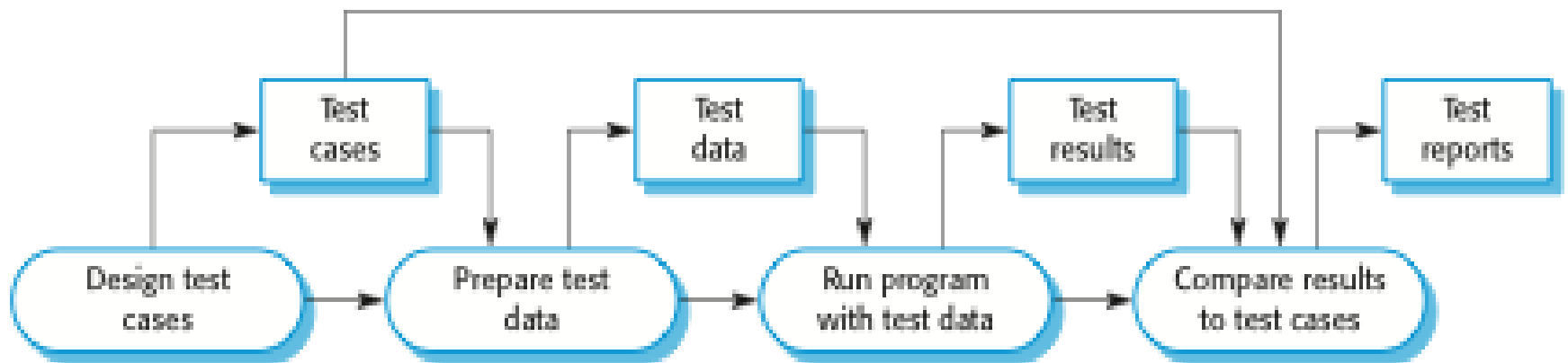
# Inspección y Prueba

---

- ▶ Las inspecciones y pruebas son complementarias y no se oponen a otras técnicas de verificación.
- ▶ Ambos deben ser utilizados durante el proceso V & V.
- ▶ Las inspecciones pueden verificar la conformidad con una especificación pero no la conformidad con los requisitos reales del cliente.
- ▶ Las inspecciones no pueden verificar características no funcionales, como el rendimiento, la facilidad de uso, etc.

# Modelo de proceso de pruebas de software

---



# Etapas de las Pruebas

---

- ▶ Pruebas de desarrollo, donde el sistema se prueba durante el desarrollo para descubrir errores y defectos.
- ▶ Pruebas de lanzamiento, donde un equipo de pruebas independiente prueba una versión completa del sistema antes de que se lance a los usuarios.
- ▶ Pruebas de usuario, donde los usuarios o usuarios potenciales de un sistema prueban el sistema en su propio entorno.

# Pruebas de Desarrollo

---

- ▶ Las pruebas de desarrollo incluyen todas las actividades de prueba que lleva a cabo el equipo que desarrolla el sistema.
- ▶ Prueba de unidad, donde se prueban unidades de programa individuales o clases de objetos. Las pruebas unitarias deben centrarse en probar la funcionalidad de los objetos o métodos.
- ▶ Pruebas de componentes, donde se integran varias unidades individuales para crear componentes compuestos. Las pruebas de componentes deben centrarse en probar las interfaces de componentes.
- ▶ Pruebas del sistema, donde algunos o todos los componentes de un sistema están integrados y el sistema se prueba como un todo. Las pruebas del sistema deben centrarse en probar las interacciones de los componentes.

# Prueba Unitaria

---

- ▶ La prueba unitaria es el proceso de probar componentes individuales de forma aislada.
- ▶ Es un proceso de prueba de defectos.
- ▶ Las unidades pueden ser:
  - Funciones o métodos individuales dentro de un objeto.
  - Clases de objetos con varios atributos y métodos.
  - Componentes compuestos con interfaces definidas utilizadas para acceder a su funcionalidad.

# Prueba Unitaria de Objetos de Clases

---

- ▶ La cobertura completa del examen de una clase implica probar todas las operaciones asociadas con un objeto.
- ▶ Interrogación de todos los atributos del objeto.
- ▶ Probar el objeto en todos los estados posibles.
- ▶ La herencia hace que sea más difícil diseñar pruebas de clase de objeto, ya que la información que se va a probar no está localizada.

# Pruebas Automáticas

---

- ▶ Siempre que sea posible, las pruebas unitarias deben automatizarse para que las pruebas se ejecuten y verifiquen sin intervención manual.
- ▶ En las pruebas unitarias automatizadas, utiliza un marco de automatización de pruebas (como JUnit) para escribir y ejecutar las pruebas del programa.
- ▶ Los marcos de prueba de unidad proporcionan clases de prueba genéricas que se extienden para crear casos de prueba específicos. Luego, pueden ejecutar todas las pruebas que haya implementado e informar, a menudo a través de alguna GUI, sobre el éxito de las pruebas.

# Componentes de pruebas automáticas

---

- ▶ Una parte de configuración, donde inicializa el sistema con el caso de prueba, a saber, las entradas y las salidas esperadas.
- ▶ Una parte de llamada, donde se llama al objeto o método a probar.
- ▶ Una parte de aserción donde se compara el resultado de la llamada con el resultado esperado. Si la afirmación se evalúa como verdadera, la prueba ha sido exitosa si es falsa, entonces ha fallado.



# Efectividad de las pruebas unitarias

Los casos de prueba deben mostrar que, cuando se usan como se espera, el componente que se está probando hace lo que se supone que debe hacer.

Si hay defectos en el componente, estos deben ser revelados por los casos de prueba.

Esto lleva a 2 tipos de caso de prueba unitaria:

El primero de ellos debe reflejar el funcionamiento normal de un programa y debe mostrar que el componente funciona como se espera.

El otro tipo de caso de prueba debe basarse en la experiencia de donde surgen problemas comunes. Debería usar entradas anormales para verificar que se procesen correctamente y no bloqueen la componente.

# Tipos de pruebas

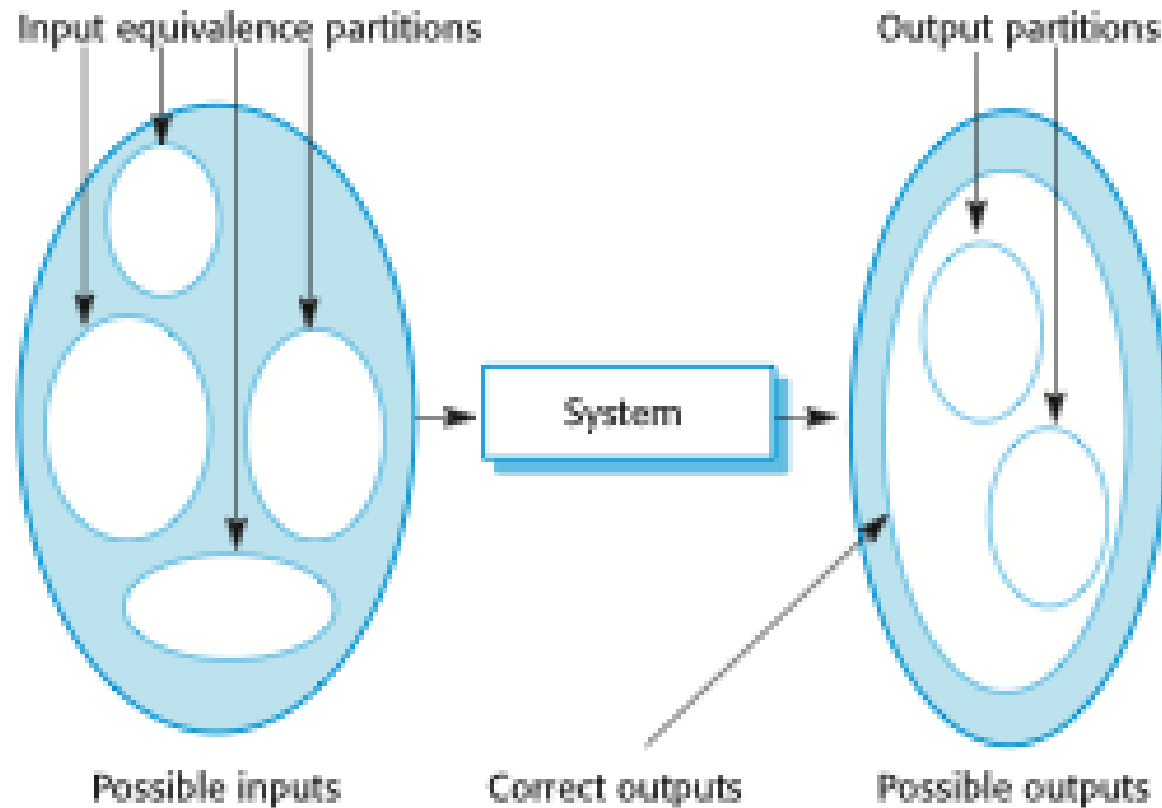
---

- ▶ Pruebas de partición, donde se identifican grupos de entradas que tienen características comunes y se deben procesar de la misma manera. Debe elegir las pruebas dentro de cada uno de estos grupos.
- ▶ Pruebas basadas en pautas, donde se usan las pautas para elegir los casos de prueba. Estas pautas reflejan la experiencia previa de los tipos de errores que los programadores suelen cometer al desarrollar componentes.

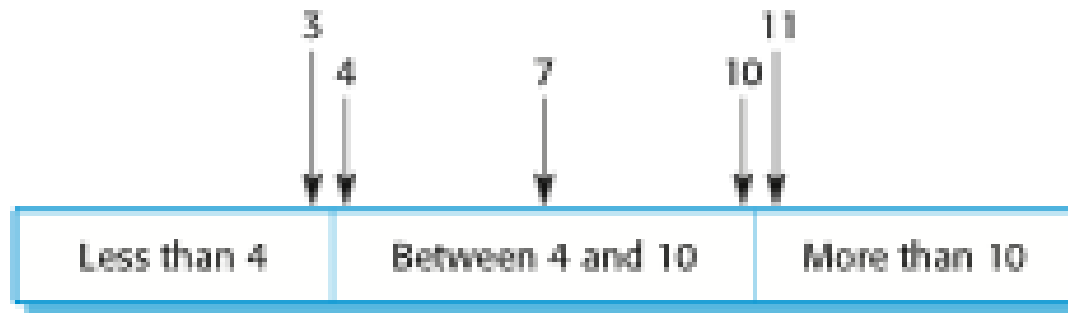
# Pruebas de partición

- ▶ Los datos de entrada y los resultados de salida a menudo caen en clases diferentes donde todos los miembros de una clase están relacionados.
- ▶ Cada una de estas clases es una partición o dominio de equivalencia donde el programa se comporta de una manera equivalente para cada miembro de la clase. Los casos de prueba deben ser elegidos de cada partición.

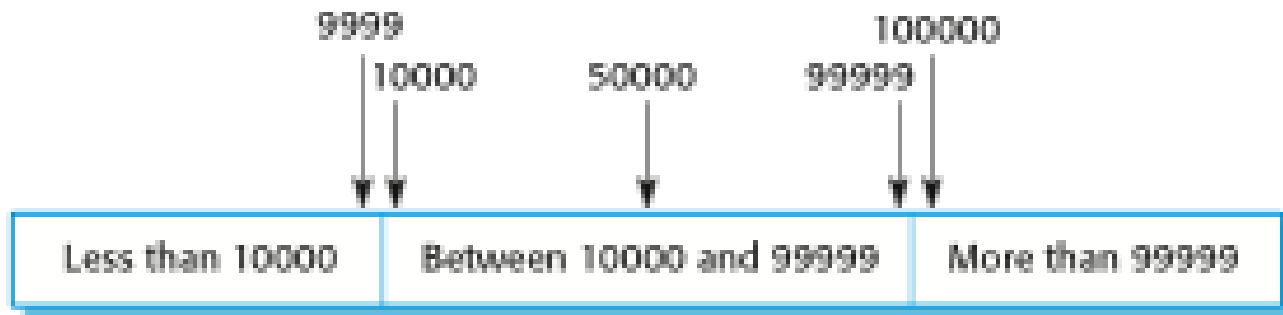
# Equivalencia en la partición



# Particiones Equivalentes



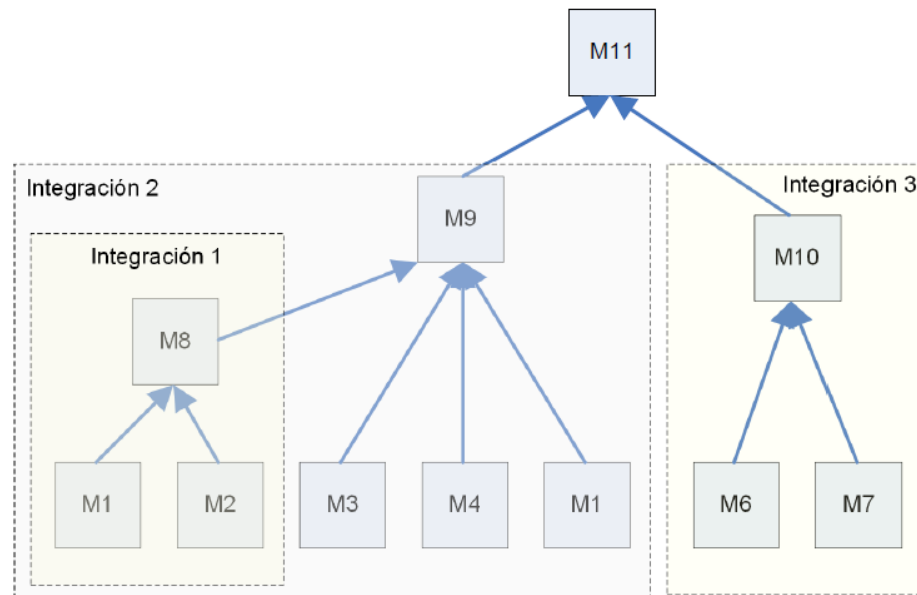
Number of input values



Input values

# Pruebas de integración

- ▶ Unión de varios componentes unitarios, módulos y subsistemas
- ▶ Se comprueba que módulos ya probados en las pruebas unitarias de forma aislada, pueden interactuar correctamente.



# Aproximaciones de integración

---

- ▶ Aproximaciones:

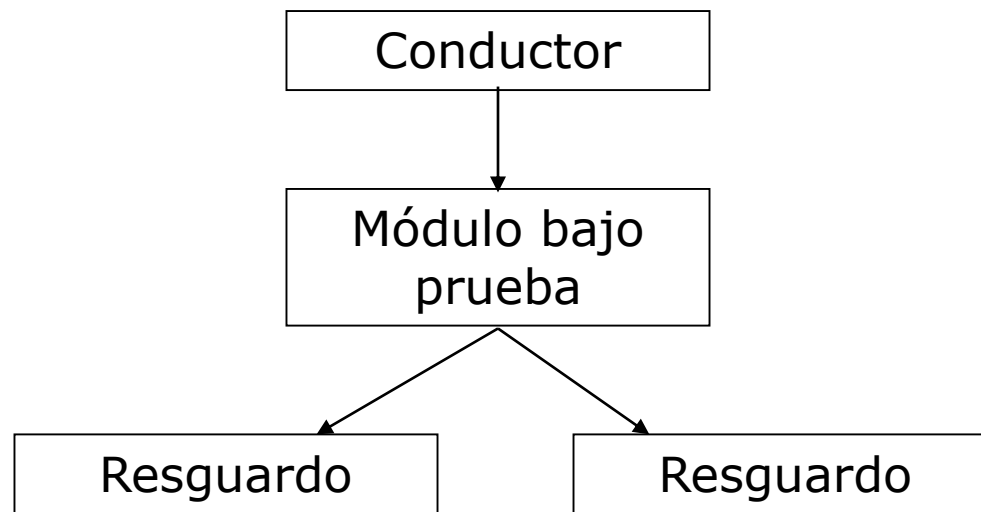
- ▶ Big-bang
- ▶ Descendente (Top-down)
- ▶ Ascendente (Bottom-up)
- ▶ Sandwich: Combinación de las 2 anteriores.

- ▶ Pueden ser necesario escribir objetos simulados:

- ▶ **Conductor**: simulan la llamada al módulo pasándole los parámetros necesarios para realizar la prueba
- ▶ **Resguardos**: módulos simulados llamados por el módulo bajo prueba

# Conductores y Resguardos

---



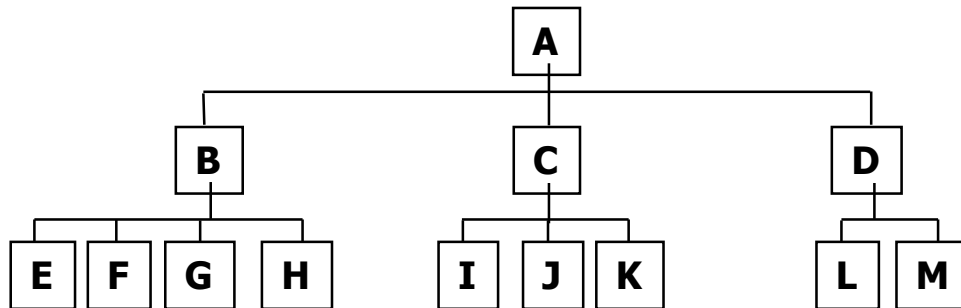


# Integración descendente

---

## ► **Descendente (*Top-down*)**

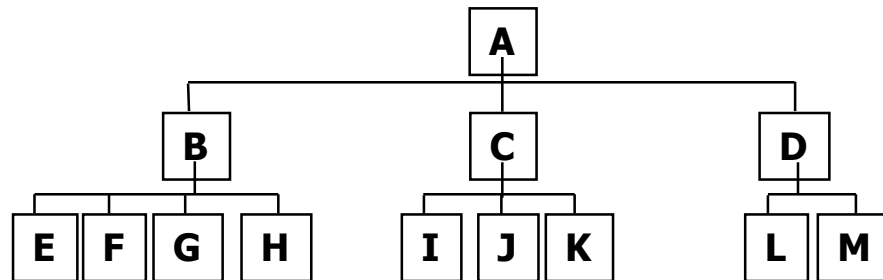
- Probar el módulo *A* primero
- Escribir *módulos simulados, resguardos (stubs)* para B, C, D
- Una vez eliminados los problemas con *A*, *probar el módulo B*
- Escribir resguardos para E, F, G, H etc.



# Integración ascendente

## ► Ascendente (*Bottom-up*)

- Escribir *conductores* para proporcionar a E los datos que necesita de B
- Probar E independientemente
- Probar F independientemente, etc.
- Una vez E, F, G, H han sido probados, el subsistema B puede ser probado, escribiendo un *conductor* para A

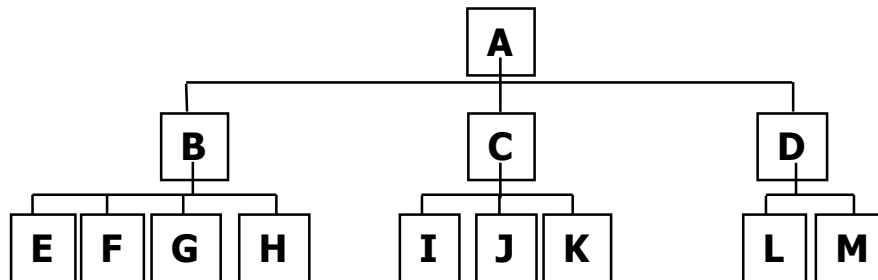


# Integración Big-Bang! y Sandwich

---

## ► **Big-Bang**

- Escribir y probar A, B, C, D, E, F, G, H, I, J, K, M *a la vez*.

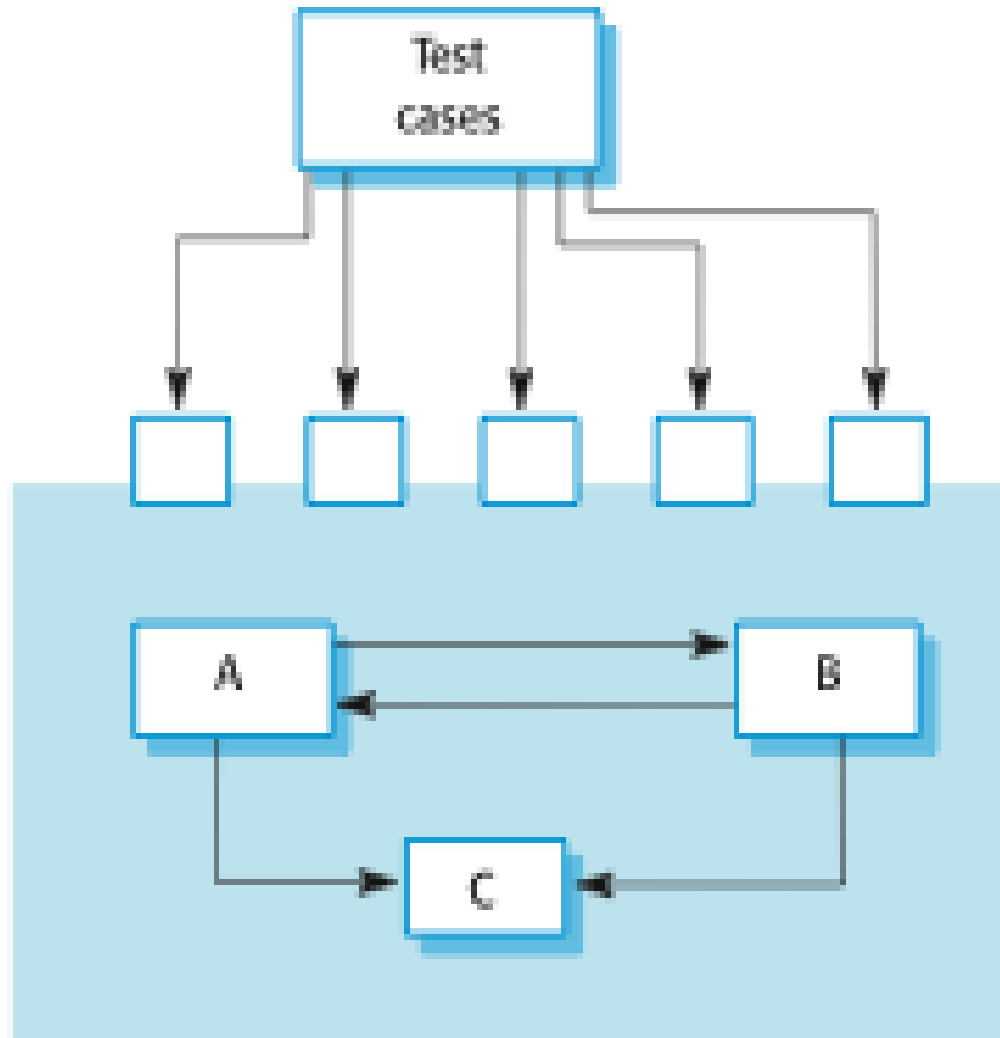


## ► **Sandwich**

- Combinación de ascendente y descendente

# Interface testing

---



# Pruebas de Interfaz

Los objetivos son detectar fallos debidos a errores de interfaz o suposiciones no válidas sobre las interfaces.

## Tipos de interfaz

- ▶ Interfaz de parámetros: Datos pasados de un método o procedimiento a otro.
- ▶ Interfaces de memoria compartida: La memoria se comparte entre procedimientos o funciones.
- ▶ Interfaces de procedimientos: El subsistema encapsula un conjunto de procedimientos a los que pueden llamar otros subsistemas.
- ▶ Interfaces de paso de mensajes: Los subsistemas solicitan servicios de otros subsistemas

# Errores de Interfaz

- ▶ ~~Mal uso de la interfaz:~~ Un componente que llama a otro componente y comete un error en el uso de su interfaz, por ejemplo entregar parámetros en un orden incorrecto.
- ▶ Malentendido interfaz: Un componente que llama incorpora suposiciones sobre el comportamiento del componente llamado que son incorrectos.
- ▶ Errores de tiempo: El componente llamado y el componente de llamada operan a diferentes velocidades y se accede a la información desactualizada.

# Líneas generales en las pruebas de interfaz

- ▶ Diseñar pruebas para que los parámetros de un procedimiento llamado estén en los extremos de sus rangos.
- ▶ Diseñar pruebas que hacen que el componente falle. Utilizar las pruebas de estrés en los sistemas de paso de mensajes.
- ▶ En los sistemas de memoria compartida, variar el orden en que se activan los componentes.

# Pruebas de sistema

- ▶ Las pruebas del sistema durante el desarrollo implican la integración de componentes para crear una versión del sistema y luego probar el sistema integrado.
- ▶ El enfoque en las pruebas del sistema es probar las interacciones entre los componentes.
- ▶ Las pruebas del sistema comprueban que los componentes son compatibles, interactúan correctamente y transfieren los datos correctos en el momento correcto a través de sus interfaces.
- ▶ Las pruebas del sistema prueban el comportamiento emergente de un sistema.



# Pruebas de sistema y de componente

---

- ▶ Durante la prueba del sistema, los componentes reutilizables que se han desarrollado por separado y los sistemas estándar pueden integrarse con los componentes desarrollados recientemente. Luego se prueba el sistema completo.
- ▶ Los componentes desarrollados por diferentes miembros del equipo o sub-equipos pueden integrarse en esta etapa. La prueba del sistema es un proceso colectivo más que individual.
- ▶ En algunas empresas, las pruebas del sistema pueden involucrar a un equipo de pruebas independiente sin la participación de diseñadores y programadores.

# Pruebas con casos de uso

---

- ▶ Los casos de uso desarrollados para identificar las interacciones del sistema se pueden utilizar como base para las pruebas del sistema.
- ▶ Cada caso de uso usualmente involucra varios componentes del sistema, por lo que probar el caso de uso obliga a que estas interacciones ocurran.
- ▶ Los diagramas de secuencia asociados con el caso de uso documentan los componentes e interacciones que se están probando.

# Estrategia de Pruebas

---

- ▶ La prueba exhaustiva del sistema es imposible, por lo que se debe desarrollar políticas de prueba que definan la cobertura de prueba requerida del sistema.

Ejemplos de políticas de prueba:

- Se deben probar todas las funciones del sistema a las que se accede a través de los menús.
- Se deben probar las combinaciones de funciones a las que se accede a través del mismo menú.
- Todas las funciones deben probarse con la entrada correcta e incorrecta.

# Pruebas de regresión

---

Las pruebas de regresión prueban el sistema para verificar que los cambios no hayan "roto" el código que funcionaba anteriormente.

En un proceso de prueba manual, la prueba de regresión es costosa pero, con la prueba automatizada, es simple y directa. Todas las pruebas se vuelven a ejecutar cada vez que se realiza un cambio en el programa.

Las pruebas deben ejecutarse con éxito antes de que se confirme el cambio.

# Pruebas de lanzamiento (release)

- ▶ La prueba de lanzamiento es el proceso de probar un lanzamiento particular de un sistema que está diseñado para ser usado fuera del equipo de desarrollo.
- ▶ El objetivo principal del proceso de prueba de lanzamiento es convencer al proveedor del sistema de que es lo suficientemente bueno para su uso.
- ▶ Por lo tanto, las pruebas de lanzamiento tienen que mostrar que el sistema entrega su funcionalidad, rendimiento y confiabilidad específicos, y que no falla durante el uso normal.
- ▶ La prueba de lanzamiento generalmente es un proceso de prueba de caja negra donde las pruebas solo se derivan de las especificaciones del sistema.

# Pruebas de lanzamiento y de sistema

- ▶ La prueba de lanzamiento es una clase de prueba del sistema.
- ▶ Diferencias importantes:
- ▶ Un equipo separado que no haya estado involucrado en el desarrollo del sistema, debe ser responsable de las pruebas de lanzamiento.
- ▶ Las pruebas del sistema realizadas por el equipo de desarrollo deben centrarse en descubrir errores en el sistema (pruebas de defectos). El objetivo de las pruebas de lanzamiento es verificar que el sistema cumple con sus requisitos y es lo suficientemente bueno para uso externo (pruebas de validación).

# Pruebas de prestaciones

- ▶ Parte de las pruebas de lanzamiento puede implicar probar las propiedades emergentes de un sistema, como el rendimiento y la confiabilidad.
- ▶ Las pruebas deben reflejar el perfil de uso del sistema. Las pruebas de rendimiento generalmente implican la planificación de una serie de pruebas en las que la carga aumenta constantemente hasta que el rendimiento del sistema se vuelve inaceptable.
- ▶ Las pruebas de estrés son una forma de pruebas de rendimiento en las que el sistema se sobrecarga deliberadamente para probar su comportamiento hasta que falla.

# Prueba de usuario

---

- ▶ La prueba del usuario o cliente es una etapa en el proceso de prueba en la que los usuarios o clientes proporcionan información y consejos sobre la prueba del sistema.
- ▶ La prueba del usuario es esencial, incluso cuando se han llevado a cabo pruebas exhaustivas del sistema y de la versión.
- ▶ La razón de esto es que las influencias del entorno de trabajo del usuario tienen un efecto importante en la confiabilidad, el rendimiento, la facilidad de uso y la solidez de un sistema. Estos no pueden ser replicados en un entorno de prueba.

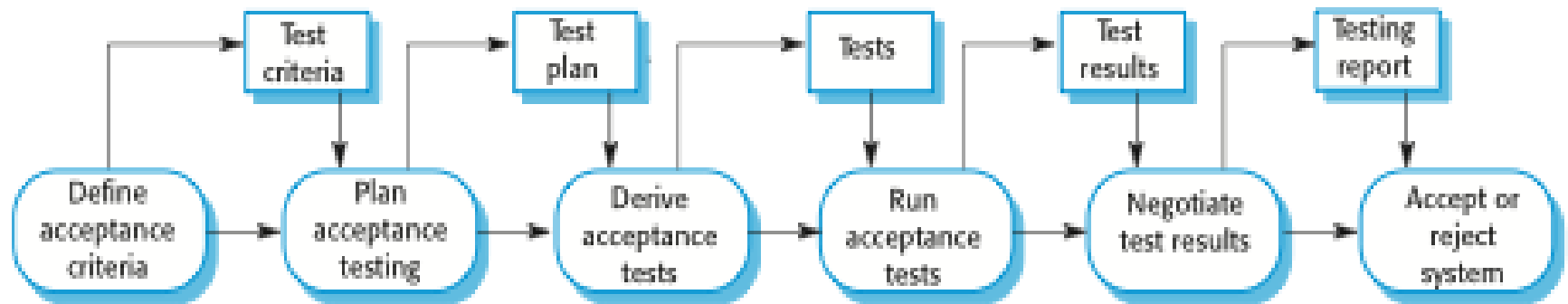


# Tipos de prueba de usuario

- ▶ Prueba alfa: Los usuarios del software trabajan con el equipo de desarrollo para probar el software en el sitio del desarrollador.
- ▶ Prueba beta: Un lanzamiento del software se pone a disposición de los usuarios para permitirles experimentar y plantear los problemas que descubren con los desarrolladores del sistema.
- ▶ Test de aceptación: Los clientes prueban un sistema para decidir si está listo o no para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente. Principalmente para sistemas personalizados.

# The acceptance testing process

---



# Etapas en las pruebas de aceptación

---

- ▶ Definir criterios de aceptación.
- ▶ Plan de pruebas de aceptación.
- ▶ Derivar pruebas de aceptación.
- ▶ Ejecutar pruebas de aceptación.
- ▶ Negociar los resultados de la prueba
- ▶ Rechazar / aceptar sistema

# Puntos Clave: Prueba de Software

---

## ► Definición

Proceso para verificar que el software cumple criterios específicos de calidad, corrección, completitud, exactitud, eficacia y eficiencia, para asegurar su buen funcionamiento y la satisfacción del cliente final.

**Software:** conjunto de elementos de un sistema informático: programas, datos, documentación, procedimientos y reglas.

## ► Objetivo:

- Se asume que todo software contiene defectos.
- Los defectos se manifestarán en fallos.
- Los fallos serán detectados o bien por las pruebas o bien por los usuarios.
- Las pruebas deben poder **trazarse** hasta los requisitos.

# Puntos Clave: :Limitaciones de la prueba

---

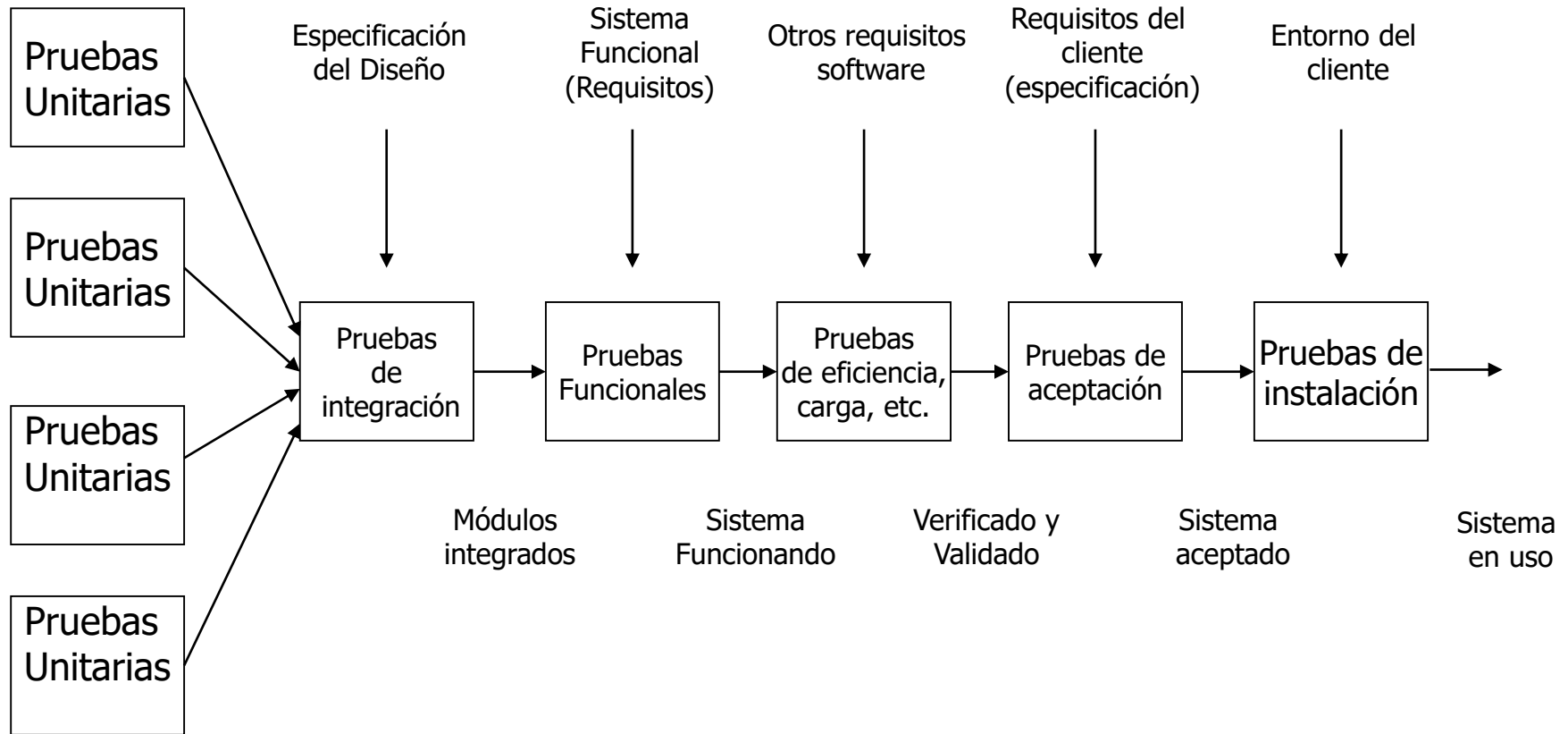
- ▶ Es imposible en la práctica hacer una prueba exhaustiva.
  - ▶ Por lo tanto, la clave está en saber seleccionar las pruebas que más probabilidad tienen de encontrar errores.
- ▶ Selección de los equipos de prueba.
  - ▶ Idealmente, la prueba debería hacerla un equipo independiente.
- ▶ ¿Cuándo terminar de probar?
  - ▶ No hay una respuesta definitiva a esta pregunta. Lo más que puede hacerse es intentar desarrollar modelos estadísticos de detección de fallos.
    - ▶ Véase cuadro “Técnicas de crecimiento de la fiabilidad del software” en el libro.

# Puntos Clave: Proceso de fase de pruebas

---

- ▶ Para asegurar el correcto funcionamiento de un sistema de SW completo y garantizar la satisfacción del cliente se debe implementar una estrategia de pruebas que incluya:
  - ▶ Pruebas unitarias: individual
  - ▶ Pruebas de integración: en conjunto
  - ▶ Pruebas del sistema: compatibilidad con requisitos
    - ▶ Funcionalidad: cumple requisitos
    - ▶ Rendimiento
    - ▶ Instalación: en la plataforma HW/SW de operación
    - ▶ Aceptación: con el cliente

# Puntos Clave: Proceso de fase de pruebas



# Puntos Clave: Pruebas unitarias

---

- ▶ Proceso de prueba donde se estudia de manera aislada un componente de SW
  - ▶ Por ejemplo un clase de Java
- ▶ Se centran en las pruebas de los programas / módulos generados por los desarrolladores
  - ▶ Generalmente, son los propios desarrolladores los que prueban su código.
  - ▶ Puede hacerse mediante técnicas de caja blanca y caja negra
  - ▶ Con caja blanca: medición de la cobertura, ramas, etc.



# Puntos Clave: beneficios y limitaciones de las pruebas unitarias

---

## ► Beneficios

- Permite arreglar los errores detectados sin cambiar la funcionalidad del componente probado
- Contribuye al proceso de integración de componentes del Sistema de SW al permitir asegurar el funcionamiento correcto de los componentes de manera individual

## ► Limitaciones

- No permite identificar errores de integración ni errores de desempeño del Sistema de SW
- No son prácticas para probar todos los casos posibles para los componentes que se prueban
- Su compatibilidad con los diferentes modelos de desarrollo de SW varía, siendo más compatible con el *Desarrollo dirigido por pruebas* (p.e., en metodologías ágiles como XP)

# Puntos Clave: Pruebas de sistema

---

- ▶ Pruebas para analizar que todo el sistema es conforme con los requisitos y las especificaciones.
  - ▶ No está solamente relacionado con el software
  - ▶ Estos tipos de pruebas incluyen:
    - ▶ Pruebas funcionales
    - ▶ Pruebas de eficiencia (rendimiento)
    - ▶ Pruebas de aceptación
    - ▶ Pruebas de instalación
    - ▶ Otras:
      - Pruebas de recuperación (Recovery testing)
      - *Pruebas de seguridad (Security) – Datos protegidos contra ataques*
      - Pruebas de que el sistema es seguro (Safety testing) – Operación segura
      - *Pruebas de estrés (Stress testing)*
      - Pruebas de interfaces (Interface testing)

# Líneas generales en las pruebas

---

Elegir entradas que obliguen al sistema a generar todos los mensajes de error.

Diseñar entradas que causan desbordamientos de buffers de entrada.

Repetir la misma entrada o serie de entradas varias veces

Forzar salidas inválidas.

Forzar resultados de los cálculos demasiado grandes o demasiado pequeños.

# Puntos Clave

- ▶ Las pruebas solo pueden mostrar la presencia de errores en un programa. No puede demostrar que no hay fallos.
- ▶ Las pruebas de desarrollo son responsabilidad del equipo de desarrollo de software. Un equipo independiente debe ser responsable de probar un sistema antes de que sea entregado a los clientes.
- ▶ Las pruebas de desarrollo incluyen pruebas unitarias, en las que prueba los objetos individuales y los métodos de prueba de componentes en los que prueba grupos relacionados de objetos y pruebas de sistemas, en los que prueba sistemas parciales o completos.

---

# Parte II. Pruebas unitarias con JUnit

# Pruebas unitarias con xUnit

---

- ▶ Para hacer pruebas unitarias se usan *frameworks* en entornos de pruebas automatizados
  - ▶ Un *framework* es un conjunto de clases relacionadas que proveen un conjunto de funcionalidades para realizar una tarea específica
- ▶ En el caso de la realización de pruebas unitarias xUnit es el *framework* más usado para hacer pruebas unitarias automatizadas

# Pasos generales para realizar pruebas unitarias en xUnit

---

1. Cada prueba a realizar se programa como un método OO
  - ▶ Determinar la clase del sistema de SW que se va a probar, esté o no programada
  - ▶ Determinar los métodos de la clase que se van a probar
  - ▶ Definir los métodos de prueba usando aserciones (assert) para verificar que los métodos a probar producen las salidas esperadas al introducir ciertos valores de entrada
2. Se reúnen todos los métodos en una colección (*suite*) de pruebas
3. Se ejecuta la colección de pruebas
4. El framework xUnit invoca cada prueba de la colección de pruebas
  - Útil en las pruebas de regresión
5. El framework xUnit reporta los resultados

# Los diferentes xUnit

---

- ▶ Para poder usar el Framework xUnit se necesita su implementación en el lenguaje de programación usado en las clases a probar. Las implementaciones de xUnit son:
  - ▶ JUnit: xUnit para Java
  - ▶ VbUnit: xUnit para Objetos Visual Basic y Objetos COM (Component Object Model)
  - ▶ Cunit: xUnit para lenguaje C
  - ▶ CPPUnit: xUnit para lenguaje C++
  - ▶ csUnit, MbUnit, NUnit: xUnit para lenguajes Microsoft .NET como C#, Vb.NET, J# y C++
  - ▶ DBUnit: xUnit para proyectos con Bases de Datos
  - ▶ Dunit: xUnit para Borland Delphi 4 y posteriores
  - ▶ PHPUnit: xUnit para PHP
  - ▶ PyUnit: xUnit para Python



# Pruebas unitarias con JUnit

---

## ► ¿Qué es **JUnit**?

- JUnit es la implementación del Framework xUnit para el lenguaje Java
- Creado en 1997 por Erich Gamma y Kent Beck
- Se usa para realizar pruebas unitarias de clases escritas en Java en un entorno de pruebas
- Es un Framework muy sencillo, con muy pocas clases, que puede aprenderse a usar muy rápidamente por lo que se ha hecho muy popular
- Es *open source* y su código está disponible en:

`http://www.junit.org/`

# Concepto de Caso de Prueba (TestCase)

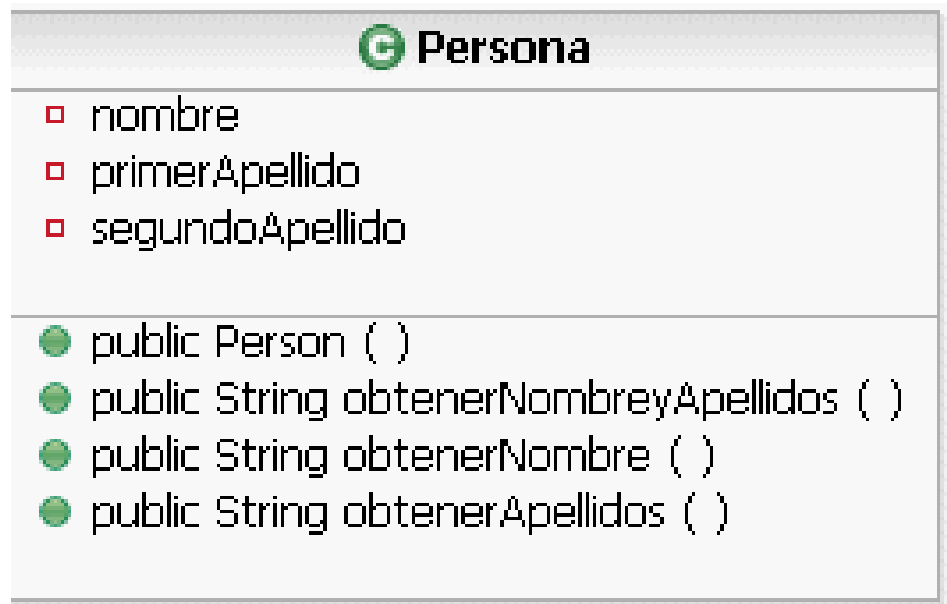
---

- ▶ Un caso de prueba (TestCase) es un conjunto de métodos que prueban las funcionalidades de una clase Java.
- ▶ La clase `TestCase` del framework JUnit contiene los métodos básicos para escribir un caso de prueba particular.

# Creación de un caso de prueba sencillo

---

- ▶ Para ver como crear un caso de prueba sencillo adaptando JUnit vamos a probar los métodos de la clase **Persona**.



# Pasos para crear un caso de prueba en JUnit

---

- ▶ Para crear un caso de prueba particular hay que crear una subclase de la clase `TestCase` de JUnit
- ▶ El nombre de la clase de prueba particular a crear debe contener con la palabra `Test` (no necesario en JUnit4)
  - ▶ Para el ejemplo de la clase `persona` tenemos:

```
import junit.framework.TestCase;
```

```
    public class PersonaTest extends TestCase {  
  
    }
```

# Pasos para crear los métodos de prueba

---

- ▶ Para escribir métodos de prueba se debe:
  1. Definir las precondiciones de la prueba y la funcionalidad a probar
  2. Especificar el resultado esperado de la prueba o postcondición para chequear si se cumple. Esto se hace usando los métodos `asserts...()` del framework JUnit.
  3. Implementar el código para que los métodos de prueba verifiquen las aserciones definidas.
  4. Ejecutar el método de prueba

# Pasos para crear los métodos de prueba

---

- ▶ La signatura de los métodos de prueba en la versión 3.8 de Junit es:

```
public void testSuCasoDePrueba ( ) { }
```

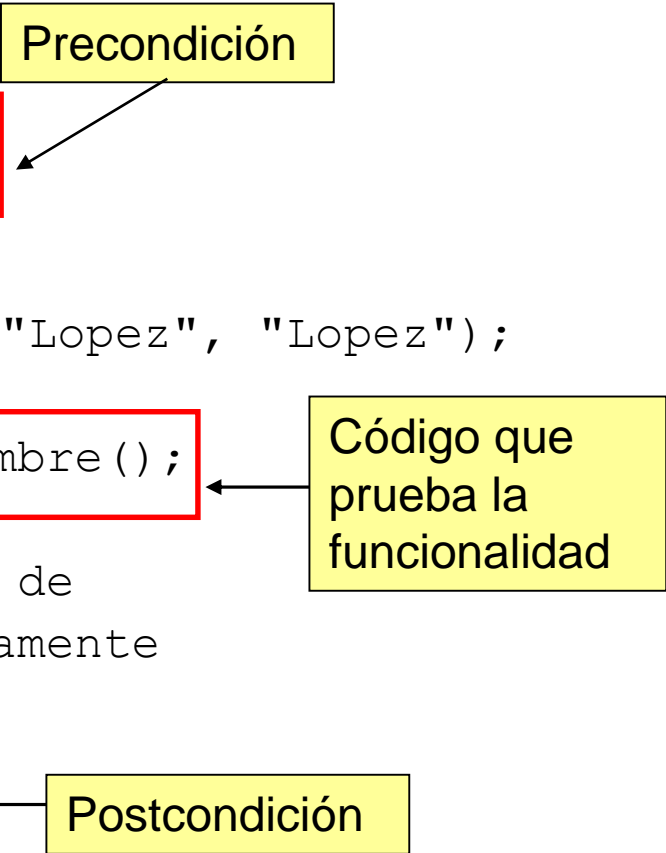
- ▶ El nombre del método de prueba debe comenzar con la palabra `test` y a continuación el nombre del método de prueba particular.
  - ▶ La primera letra de cada palabra del nombre del método debe ser mayúscula

# Prueba del método **ObtenerNombre** de la clase **Persona**

```
public void testObtenerNombre() {
```

```
String expResult = "Vicentin";
```

Precondición



```
Persona instancia =  
    new Persona ("Vicentin", "Lopez", "Lopez");
```

```
String result = instancia.obtenerNombre();
```

Código que  
prueba la  
funcionalidad

```
//Prueba si el método obtenerNombre de  
//la clase Persona funciona correctamente
```

```
assertEquals(expResult, result);
```

Postcondición

```
}
```

# Prueba del método **ObtenerApellidos** de la clase **Persona**

---

```
public void testObtenerApellidos () {
```

```
    String expectedResult = "Lopez Lopez";
```

```
    Persona instancia =  
        new Persona ("Vicentin", "Lopez", "Lopez");
```

```
    String result = instancia.obtenerApellidos();
```

```
    //Prueba si el método obtenerNombre de  
    //la clase Persona funciona correctamente
```

```
    assertEquals(expectedResult, result);
```

```
}
```



# Prueba del método **ObtenerNombreyApellidos** de la clase **Persona**

---

```
public void testObtenerNombreyApellidos () {
```

```
    String expectedResult = "Vicentin Lopez Lopez";
```

```
    Persona instancia =  
        new Persona ("Vicentin", "Lopez", "Lopez");
```

```
    String result = instancia.obtenerNombreyApellidos();
```

```
    //Prueba si el método obtenerNombre de  
    //la clase Persona funciona correctamente
```

```
    assertEquals(expectedResult, result);
```

```
}
```

# Código del caso de prueba, clase **Persona**

---

```
import junit.framework.TestCase;

public class PersonaTest extends TestCase {

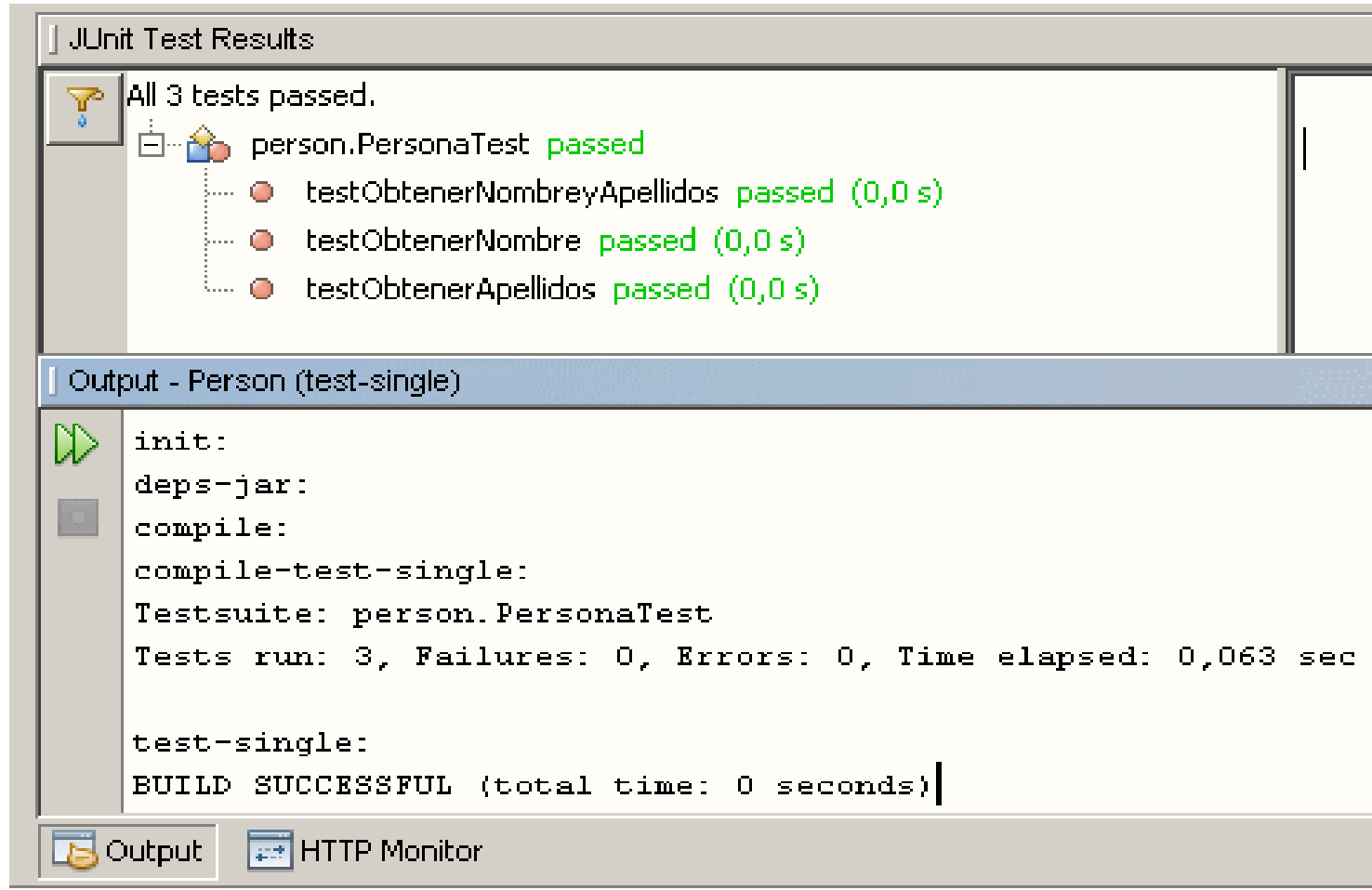
    public void testObtenerNombreyApellidos() {
        String expectedResult = "Vicentin Lopez Lopez";
        Persona instancia = new Persona ("Vicentin", "Lopez", "Lopez");
        String result = instancia.obtenerNombreyApellidos();
        assertEquals(expResult, result);
    }
    public void testObtenerNombre() {
        Persona instancia = new Persona ("Vicentin", "Lopez", "Lopez");
        String expectedResult = " Vicentin ";
        String result = instancia.obtenerNombre();
        assertEquals(expResult, result);
    }
    public void testObtenerApellidos() {
        Persona instancia = new Persona (" Vicentin ", "Lopez", "Lopez");
        String expectedResult = "Lopez Lopez";
        String result = instancia.obtenerApellidos();
        assertEquals(expResult, result);
    }
}
```

# Ejecución de casos de prueba

---

- ▶ Para ejecutar un caso de prueba se usa el método `run()` de la clase `TestCase()`.
- ▶ En *Netbeans* un caso de prueba particular se ejecuta seleccionando en el menú `Run | Runfile` y en la ventana desplegable se selecciona el nombre del caso de prueba que se desea ejecutar
- ▶ Cuando se ejecuta un caso de prueba JUnit proporciona dos tipos de información
  - ▶ Información que indica si las pruebas pasaron o fallaron
  - ▶ Si la prueba falló indica la causa de la falla

# Ejemplo de resultados en *Netbeans*



# Eliminación del código duplicado

---

- ▶ En caso de prueba particular puede haber código duplicado en cada método de prueba al tener que crear objetos y estructuras de datos comunes
- ▶ Ej. clase persona: en cada método de prueba se crea un instancia de Persona
- ▶ Esto podría hacerse una sola vez y definir esta instancia como el contexto para ejecutar cada uno de los métodos del caso de prueba **testPersona()**

# Código del caso de prueba de la clase **Persona**

```
import junit.framework.TestCase;

public class PersonaTest extends TestCase {

    public void testObtenerNombreyApellidos() {
        String expectedResult = "Vicentin Lopez Lopez";
        Persona instancia = new Persona ("Vicentin", "Lopez", "Lopez");
        String result = instancia.obtenerNombreyApellidos();
        //Prueba del método obtenerNombreyApellidos de la clase Persona.
        assertEquals(expectedResult, result);
    }

    public void testObtenerNombre() {
        Persona instancia = new Persona ("Vicentin", "Lopez", "Lopez");
        String expectedResult = "Vicentin";
        String result = instancia.obtenerNombre();
        assertEquals(expectedResult, result);
    }

    public void testObtenerApellidos() {
        Persona instancia = new Persona ("Vicentin", "Lopez", "Lopez");
        String expectedResult = "Lopez Lopez";
        String result = instancia.obtenerApellidos();
        assertEquals(expectedResult, result);
    }

}
```

Código duplicado

# Ejemplo completo con `setUp()` y `tearDown()`

---

```
import junit.framework.TestCase;
public class PersonaTest extends TestCase {
    private Persona instancia;
    @Override
    protected void setUp() {
        instancia = new Persona ("Vicentin", "Lopez", "Lopez");
    }
    public void testObtenerNombreyApellidos() {
        String expResult = "Vicentin Lopez Lopez";
        String result = instancia.obtenerNombreyApellidos();
        assertEquals(expResult, result);
    }
    public void testObtenerNombre() {
        String expResult = "Vicentin";
        String result = instancia.obtenerNombre();
        assertEquals(expResult, result);
    }
    public void testObtenerApellidos() {
        String expResult = "Lopez Lopez";
        String result = instancia.obtenerApellidos();
        assertEquals(expResult, result);
    }
    @Override
    protected void tearDown() {
        instancia = null;
    }
}
```

---

# Parte III. Pruebas de integración con MockObjects



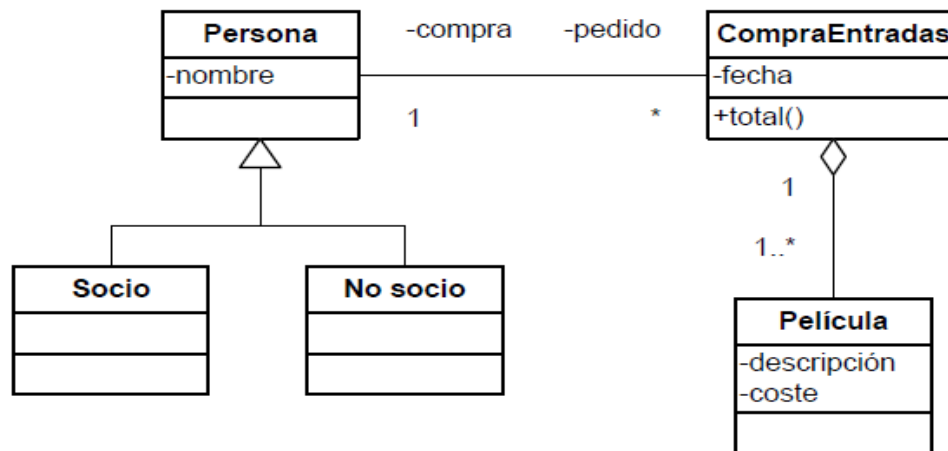
# Introducción

---

- ▶ A veces es necesario hacer pruebas de una clase aislando sus objetos colaboradores.
  - ▶ Por ejemplo, en las pruebas de integración
- ▶ Esta necesidad se presenta cuando la instanciación de los objetos colaboradores es muy costosa, p.e., si se requiere activar una base de datos
- ▶ Una de las técnicas para poder probar clases relacionadas con otras es a través de objetos simulados (*Mock Objects*)
  - ▶ Simulan el comportamiento de los objetos colaboradores para la realización de las pruebas

# Objetos simulados (*Mock Objects*)

- ▶ Objetos que permiten realizar pruebas de manera aislada de clases que tienen asociaciones con otras clases, llamadas clases colaboradoras
  - ▶ Permiten hacer la prueba de una clase aislándola de las clases con las que interactúa
  - ▶ Simulan a los objetos colaboradores de la clase que esté probando, adoptando la interfaz pública de los objetos colaboradores



## ¿Cuándo usar *Mock Objects*?

---

- ▶ Cuando en una aplicación no se puede probar los objetos que no tienen relaciones de asociación antes de probar los objetos que sí las tienen
- ▶ Cuando no se cuenta con la implementación de los objetos colaboradores
- ▶ Cuando usar los objetos colaboradores para las pruebas es muy costoso, como por ejemplo, usar una base de datos

# Ventajas

---

- ▶ Son de fácil instanciación, ya que su objetivo es comportarse como los objetos que simulan
- ▶ Son infalibles, ya que si se detecta un fallo haciendo la prueba de la clase con la que colabora se sabe que el fallo está en dicha clase
- ▶ Varias herramientas disponibles: *JMock*, *Nmock*, *Easymock*...
- ▶ Permiten reducir el tiempo de las pruebas al sustituir las clases colaboradoras por un código mucho más simple

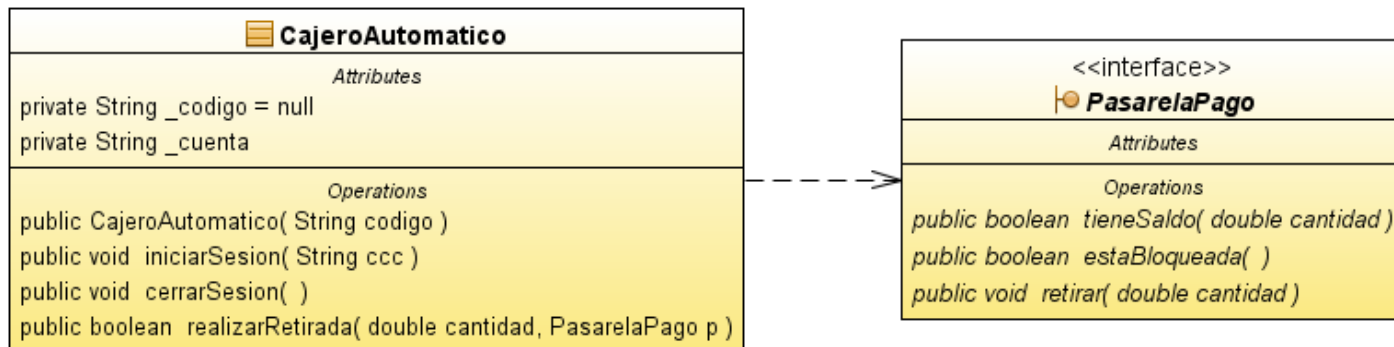
# Uso de los objetos simulados con *EasyMock*

---

1. **Crear el *mock***, a partir de una interfaz.
  - ▶ A través del método `createMock()`
2. **Preparar el *mock* estableciendo las expectativas (modo grabación)**
  - ▶ Definir todas las invocaciones que se espera que ocurran
  - ▶ Método `expect()` y otros métodos
3. **Iniciar el *mock* (modo reproducción)**
  - ▶ Informar que ya se establecieron todas las expectativas y que comienza a funcionar el objeto *mock*.
  - ▶ Método `replay()`
4. **Ejecutar el método a probar** realizando la prueba correspondiente
5. **Verificar el *mock***
  - ▶ Consiste en verificar que se hayan realizado todas las invocaciones que se esperaban, en el orden y cantidad correctas.
  - ▶ Esto se hace a través del método `verify()`

# Ejemplo de uso de *Mock Objects*

- ▶ Probamos una clase denominada **CajeroAutomatico**.
- ▶ Esta clase en el diseño final utiliza otra clase denominada **PasarelaPago** con la que mantiene una comunicación durante las transacciones de los clientes con el cajero



# Ejemplo de uso: método a probar

---

```
public class CajeroAutomatico {  
    ///...  
  
    public boolean realizarRetirada(double cantidad, PasarelaPago p)  
    {  
        assert(_cuenta != null );  
        if (p.estaBloqueada())  
            return false;  
        if (p.tieneSaldo(cantidad)){  
            p.retirar(cantidad);  
            return true;  
        }  
        return false;  
    }  
}
```

# Proceso para usar *Mock Objects*

---

## I. Crear los *Mock Objects* necesarios

- Se debe crear un *Mock Object* por cada objeto colaborador que utilice la clase a probar
- Las clases de los *Mock Objects* deben ser subclasses de las clases de los objetos colaboradores para evitar errores de compilación
- Para crear el mock se invoca a `createMock()`, método que proviene de un import estático en la clase de prueba

```
public class CajeroAutomaticoTest {  
  
    private CajeroAutomatico classUnderTest;  
    private PasarelaPago mock ;  
  
    @Before  
    public void setUp() {  
        classUnderTest = new CajeroAutomatico("1111111111");  
        classUnderTest.iniciarSesion("1234");  
        /* (1) Crea un mock para la clase colaboradora*/  
        mock = createMock(PasarelaPago.class);  
    }  
    ...  
}
```



# Proceso para usar *Mock Objects*

2. Definir el comportamiento esperado de los *Mock Objects*, estableciendo las expectativas (grabación):
  - Qué métodos van a ser llamados desde la clase a probar
  - Con qué parámetros y (opcional) sus valores de retorno
  - La secuencia en las que se van a invocar los métodos
  - El número de invocaciones a cada método

```
@Test
public void testRealizarRetiradaBasico() throws Exception {
    System.out.println("realizarRetirada");

    /* (2) En estado de "grabación", se le dice al objeto simulado
       las llamadas que debe esperar y cómo responder a ellas.
    */
    expect(mock.estaBloqueada()).andReturn(false);
    expect(mock.tieneSaldo(500)).andReturn(true);
    mock.retirar(500);
    /* (3) Ahora, el objeto simulado comienza a esperar las llamadas*/
    replay(mock);

    ...
}
```

# Proceso para usar *Mock Objects*

---

3. Iniciar el mock (reproducción)
4. Ejecutar el método que se vaya a probar realizando la prueba correspondiente

```
@Test
public void testRealizarRetiradaBasico() throws Exception {
    ...
    replay(mock);

    boolean result = classUnderTest.realizarRetirada(500, mock);
    assertTrue(result);

    /* Forzamos a que la ausencia de llamadas sea un error también*/
    verify(mock);
}
```

# Proceso para usar *Mock Objects*

---

5. Decir a cada *Mock Object* involucrado en la prueba que verifique si se cumplieron las expectativas
  - Comprobar al finalizar la prueba que los *Mock Objects* cumplieron las expectativas. Esto se puede hacer de manera automática usando las herramientas que automatizan la técnica de los *Mock Objects*

```
realizarRetirada
  Unexpected method call tieneSaldo(3.0):
    tieneSaldo(500.0): expected: 1, actual: 0
    retirar(500.0): expected: 1, actual: 0)
realizarRetiradaMasComplejo
```

```
realizarRetirada
  Expectation failure on verify:
    estaBloqueada(): expected: 1, actual: 0)
```

# Referencias

---

- ▶ Pressman, R.S. Ingeniería del software. Ed. McGraw-Hill. 2012.
- ▶ Sommerville, I. Ingeniería del software. Ed. Addison Wesley. 2006.