PROCESOS DE LA INGENIERIA SOFTWARE
Tema 1

Métodos de prueba

Universidad de Cantabria – Facultad de Ciencias

Patricia López Martínez

Bibliografía

- Bibliografía Básica
  - Ian Sommerville (2011): Software Engineering
    - Capítulo 8 y 15
  - IEEE Computer Society (2014): SWEBOK - Guide to the Software Engineering Body of Knowledge, v3.
    - Capítulo 4
- Bibliografía complementaria
  - Glen J. Myers (2011): The art of software testing

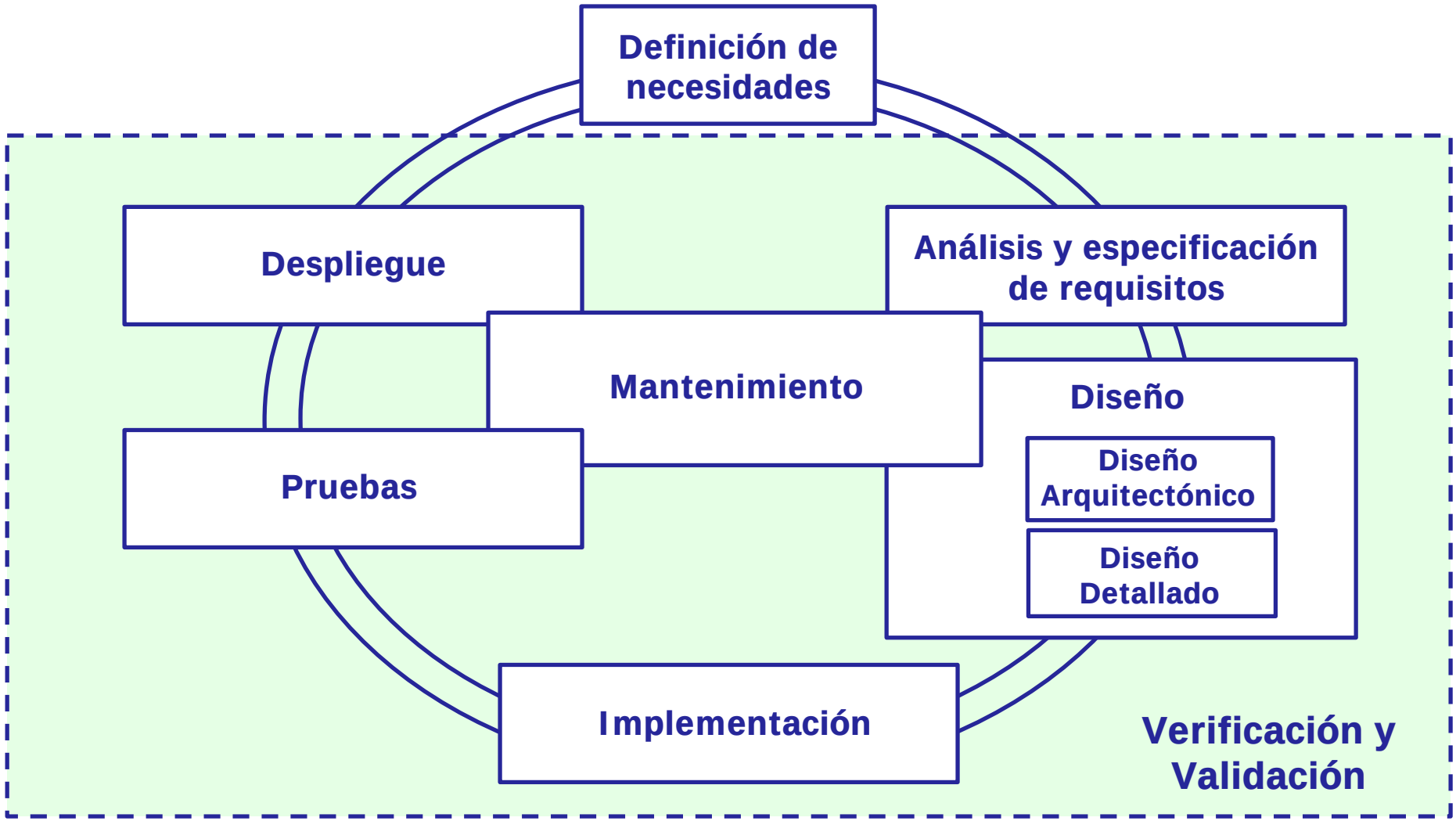
Objetivos

- Repasar los conceptos vistos hasta el momento acerca de:
  - Verificación y validación
  - Pruebas
- Profundizar en los diferentes niveles de pruebas:
  - Pruebas unitarias
  - Pruebas de integración
  - Pruebas de sistema
  - Pruebas de aceptación

Contenido

- Verificación y Validación de Software
- Pruebas Unitarias
- Pruebas de Integración
- Pruebas de Sistema
- Pruebas de Aceptación

Pruebas y V&V de software dentro del ciclo de vida



Verificación y Validación

- **Verificación y Validación (V&V):** Conjunto de procedimientos, actividades, técnicas y herramientas que se utilizan, paralelamente al desarrollo de software, para asegurar que un producto software resuelve el problema inicialmente planteado
  - Se aplica a todos los artefactos generados durante el proceso, no sólo al código

	VERIFICACION	VALIDACION
	¿Estamos construyendo **correctamente** el producto?	¿Estamos construyendo el producto **correcto**?
Garantiza que el software…	Cumple su especificación	Hace lo que el usuario quiere
¿Cómo? Demostrando …	La consistencia y corrección de los artefactos que se generan a lo largo del desarrollo	La corrección del producto final respecto a las necesidades del usuario

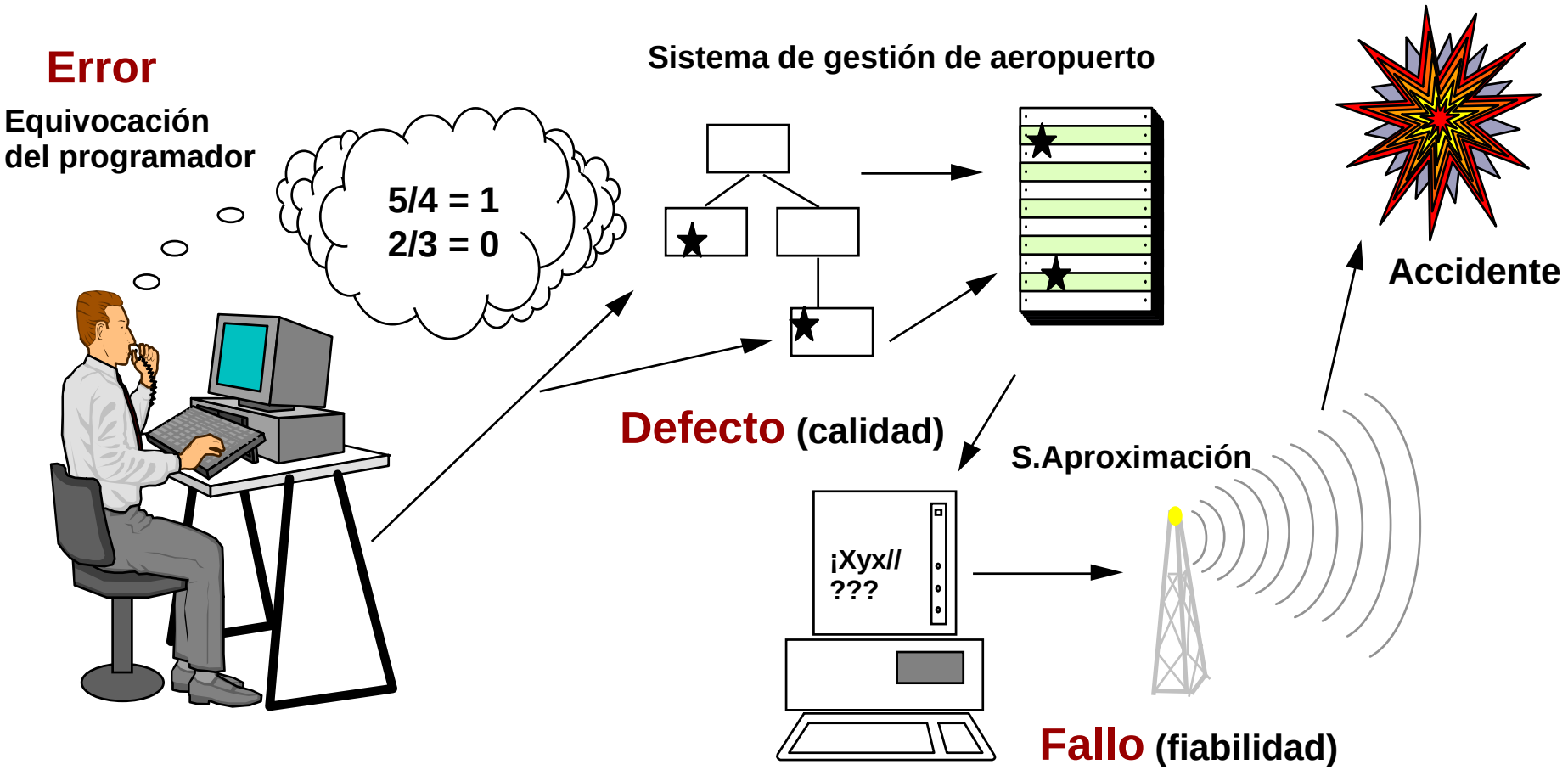
Terminología: Defecto, fallo, error

- **Defecto**: Característica de un sistema con potencial para causar un mal funcionamiento del sistema
  - Es la **causa** de un mal funcionamiento de un sistema
  - *Fault, bug* (en el dominio del software). *Defect* (es más genérico, defecto o fallo)

- **Fallo**: Comportamiento externo del sistema distinto del esperado (en ejecución)
  - Es la **manifestación** de un mal funcionamiento del sistema, aunque no siempre se puede identificar de manera unívoca el defecto asociado, es decir la causa del mismo
  - *Failure*

- **Error:** Tiene diferentes acepciones
  - Acción humana que conduce a la introducción de un defecto
  - Activación de un defecto (en ejecución)
  - Resultado incorrecto
  - Diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto

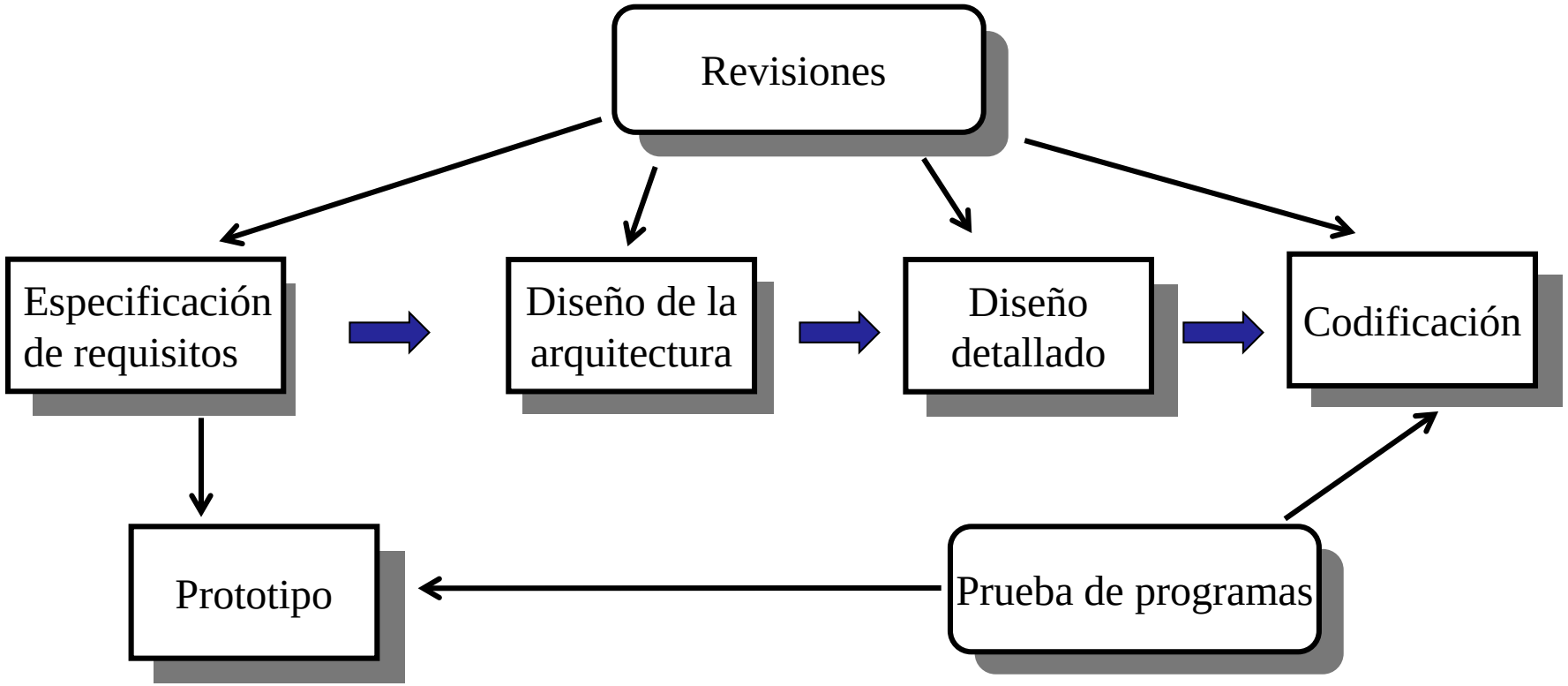
Relación entre Error, Defecto y Fallo



Técnicas de V&V

- Técnicas estáticas => **Revisiones**
  - Principalmente para verificación
  - Pueden llegar a descubrir entre el 60% y el 90% de los fallos

- Técnicas dinámicas => **Pruebas**



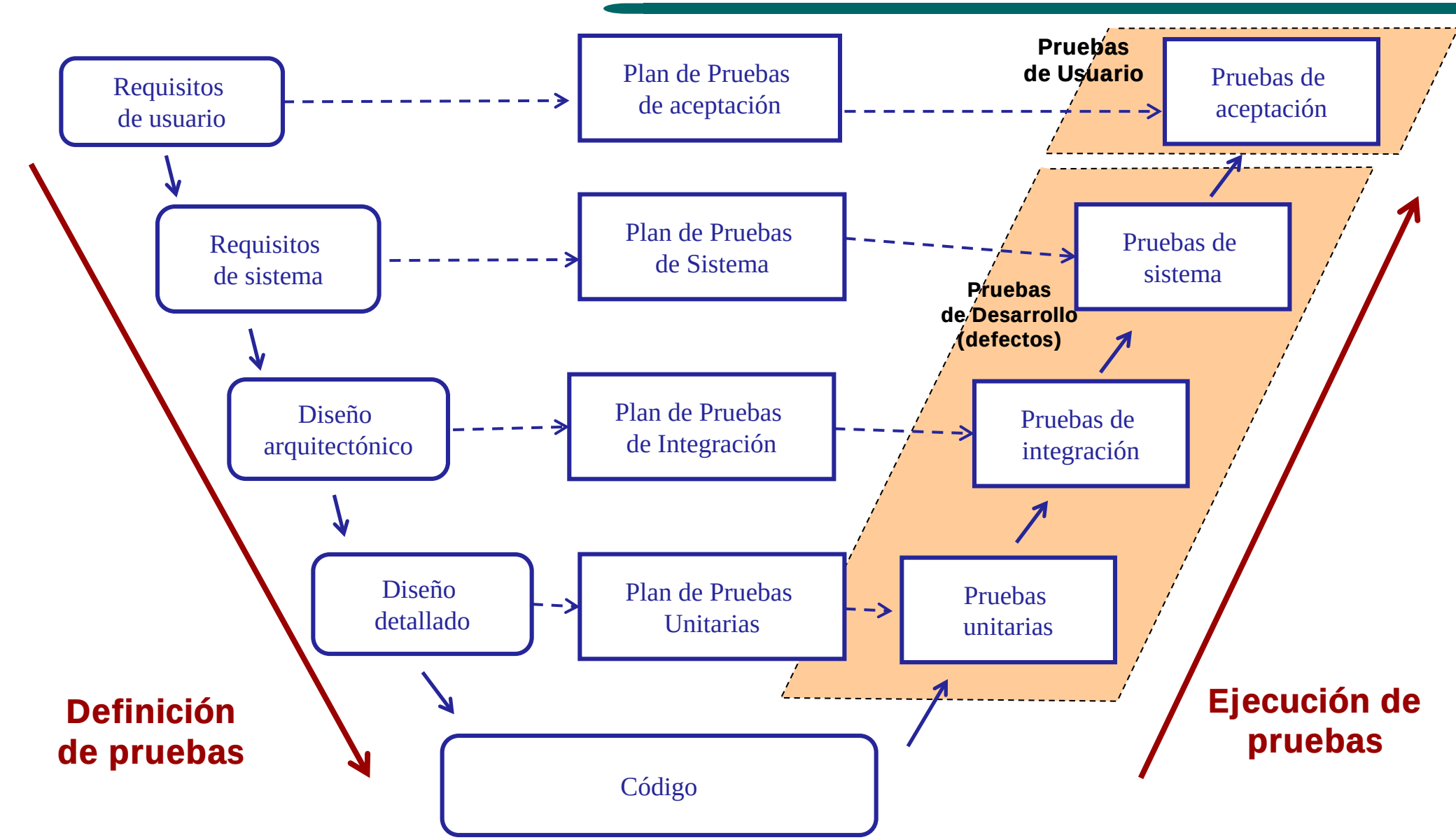
Pruebas de Softare (Testing)

> **"Testing can only show the presence of errors, not their absence"**
> (Dijkstra, 1972)

- Las **Pruebas de Software (Testing)** consisten en:
  - la comprobación **dinámica** del comportamiento de un programa
  - para un **conjunto finito** de casos de prueba, convenientemente seleccionados entre los, habitualmente infinitos, dominios de ejecución,
  - comparándolo con el comportamiento **esperado**
  - para evaluar su **calidad**
  - y mejorarla, identificando posibles **defectos** y problemas.

- El objetivo de las pruebas de software es la **detección de defectos**
  - Descubrir un error es el éxito de una prueba
  - Pruebas de validación vs pruebas de defectos

Niveles de prueba: Estrategia de Diseño y Aplicación



Planificación de V&V: Plan de pruebas

- Es importante planificar un proceso de V&V eficaz y eficiente

- Es necesaria la elaboración de un **plan de pruebas:**
  1. Definición de la fases de pruebas. Por cada fase:
     - Objetivos
     - Mecanismos (tipos de pruebas, criterio de integración, etc.)
     - **Criterios de finalización**
       - Según técnicas, estimación número errores, tasa de errores encontrados
  2. Calendario
  3. Responsables
     - De aplicar los tests, de detectar errores, de resolverlos, etc.
  4. Herramientas
  5. Requisitos de plataforma
  6. Definición de las pruebas de regresión
  7. Monitorización del proceso

Contenido

- Verificación y Validación de Software
- **Pruebas Unitarias**
- Pruebas de Integración
- Pruebas de Sistema
- Pruebas de Aceptación

Pruebas Unitarias

- Verifican el **funcionamiento aislado de piezas de software que** pueden ser probadas de forma separada
  - Subprogramas/Módulos/Clases individuales
  - Componente que incluye varios subprogramas/módulos/clases

- Estas pruebas suelen llevarse a cabo con:
  - Acceso al código fuente probado
  - Ayuda de herramientas de depuración
  - Participación (opcional) de los programadores que escribieron el código

- Técnicas:
  - Caja Negra
    - Partición equivalente, AVL, conjetura de errores, prueba de estados, etc.
  - Caja Blanca
    - Cobertura sentencias, decisiones, condiciones, caminos, etc.

Pruebas Unitarias – Objetos Mock

- Para la prueba de módulos o clases, suelen ser necesarios dos elementos:
  - **Drivers** o controladores, que ejecuten la clase a probar con los casos de prueba definidos
    - Clases de prueba en JUnit
  - **Stubs** u objetos **Mock**, que simulen/controlen el comportamiento de los objetos de los que depende la clase bajo prueba

- Stubs vs Mock
  - Ambos son objetos que ofrecen la misma interfaz que un objeto requerido desde el objeto bajo prueba pero con comportamiento simulado y conocido
    - Stubs => Verificación de estado (**State-Driven Verification**)
      - Requieren la implementación de las clases con comportamiento conocido
    - Mock => Verificación de comportamiento (**Behaviour-Driven Verification**)
      - No requieren la implementación de clases, pero sí un framework que proporcione la funcionalidad necesaria
        - Ej: Mockito, EasyMock, Mock, NSubstitute

Prueba con Stubs vs prueba con Mocks

Stubs / State-Driven verification

```
public class OrderTestStubs {

   @Test
   public void testOrderSendsMailIfUnfilled() {
     Order order = new Order("USB16G", 2);
     MailServiceStub mailer = new MailServiceStub();
     order.setMailer(mailer);
     order.fill();
     assertEquals(1, mailer.numberSent());
  }
```

```
public interface MailService {
     public void send (Message msg);
}

public class MailServiceStub implements MailService {

   private List<Message> messages = new ArrayList<Message>();

   public void send (Message msg) {
     messages.add(msg);
   }
   public int numberSent() {
     return messages.size();
   }
}
```

Mocks / Behaviour-Driven Verification

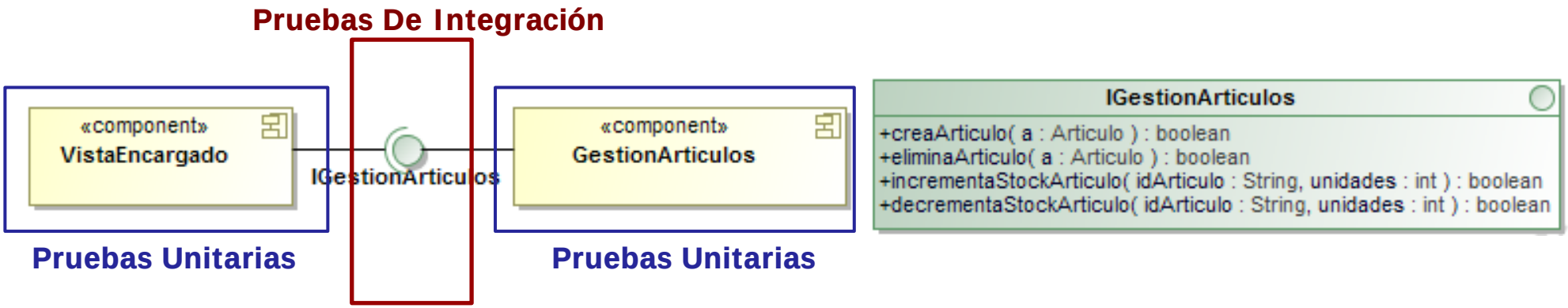
```
public class OrderInteractionTester {

   @Test
   public void testOrderSendsMailIfUnfilled() {
      Order order = new Order("USG16G", 2);

      MailService mailer = mock(MailService.class);
      order.setMailer(mailer);

      order.fill();

      verify(mailer).send((Message) anyObject());
   }
}
```

Contenido

- Verificación y Validación de Software
- Pruebas Unitarias
- **Pruebas de Integración**
- Pruebas de Sistema
- Pruebas de Aceptación

Pruebas de Integración

- Verifican la **interacción** entre los componentes del sistema
  - Compatibilidad de interfaces, de datos intercambiados, etc.



- Errores típicos:
  - Mal uso de interfaces: parámetros en orden inadecuado, con tipos erróneos, etc.
  - Mala interpretación de la semántica de la interfaz
  - Errores de temporización o falta de sincronización
    - Más típicos de sistemas de tiempo real

Pruebas unitarias vs de integración

- En ocasiones, la frontera no está muy bien definida

- Pruebas unitarias:
  - Basado en uso de stubs/Mocks dentro de un framework de pruebas estilo JUnit
  - Se aplican a clases Java aisladas:
    - Clases de dominio
    - Componentes de la capa DAO o de la capa de negocio de manera aislada
      - Si hay dependencias con otros componentes se deben simular con objetos Mock o Stubs
    - Componentes de código de la capa de presentación
  - Cualquier prueba que requiera el uso de elementos externos, como la BBDD, no se considera una prueba unitaria

- Pruebas de integración:
  - En general, se requiere un entorno de pruebas más complejo:
    - Conexión a BBDD real, servidores con inyección de dependencias, etc.
  - Uso de servidores embebidos, o entornos estilo Arquillian

Estrategias/Técnicas de Pruebas de Integración

- **Big Bang**
  - Se combinan todos los componentes a la vez, una vez que se cuenta con el código de todos ellos, probando el sistema como un todo
  - Muy difícil identificar el origen de un error

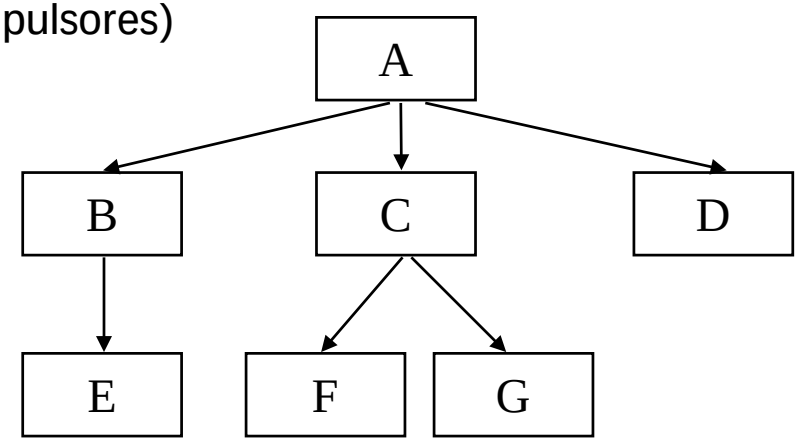
- **Incremental**
  - Consiste en combinar cada nuevo componente con un conjunto de componentes ya probados
  - Estrategias de orden de componentes:
    - Ascendente/Descendente
      - Para sistemas estructurados o jerárquicos
    - Guiadas por la arquitectura o la funcionalidad
      - Para sistemas no estructurados (Orientados a objetos)

- La estrategia está muy ligada a la metodología o ciclo de vida del proyecto

Integración Incremental Ascendente

Incremental Ascendente (Bottom-Up)

- Se combinan los módulos comenzando por los módulos hoja y ascendiendo a través de los niveles de la jerarquía
- Requiere el desarrollo de Controladores (drivers o impulsores)
- Ejemplo:
  1. E, F, G y D estarían probados con pruebas unitarias
  2. B con E
  3. C con F y G
  4. A con B, C y D



Incremental Descendente (Top-Down)

- Se combinan los módulos comenzando por lo más alto de la jerarquía
- Requiere el desarrollo de stubs/mocks
- Dos opciones:
  - Primero en profundidad (ramas)
    - AB, ABE, ABEC, ABECFG, ABECFGD
  - Primero en anchura (niveles)
    - AB, ABC, ABCD, ABCDE, ABCDEFG

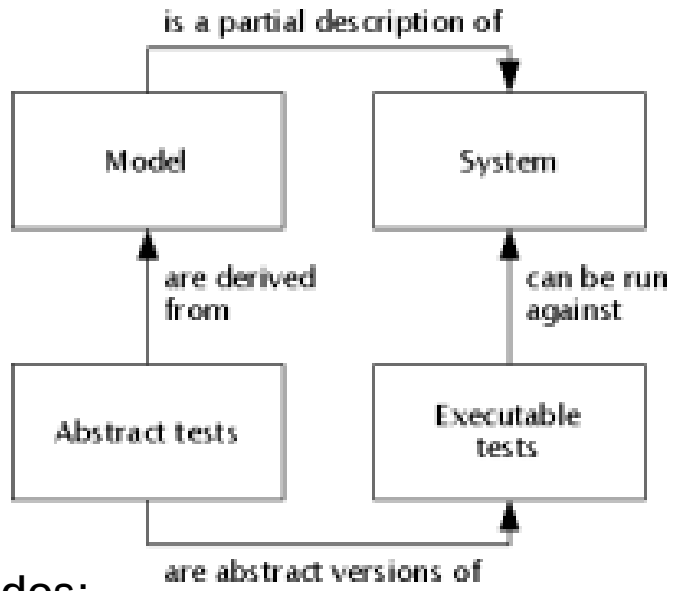
Integración guiada por arquitectura/funcionalidad

- Útil para aplicaciones sin estructura jerárquica, por ejemplo, OO
  - Pruebas **guiadas por subprocesos o hilos funcionales**
    - Clases involucradas en la respuesta a cada entrada o evento del sistema
    - Por ejemplo, guiadas por los escenarios asociados a cada caso de uso
  - Pruebas guiadas por el **uso**
    - Se comienza probando aquellas clases que son más independientes
    - A continuación, las que las usan, y así sucesivamente

¿De dónde extraemos estas pruebas?

Model-based Testing

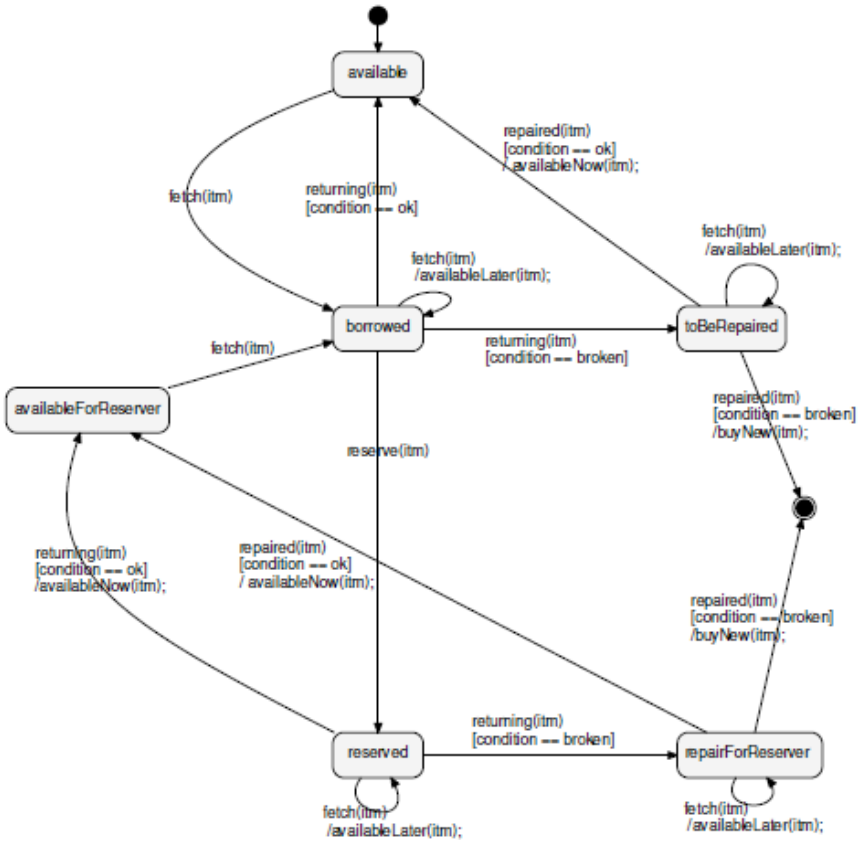
- **Pruebas dirigidas por modelos** (model-based testing): Pruebas de software donde los casos de prueba se derivan total o parcialmente de un modelo que describe algunos (si no todos) los aspectos del sistema bajo prueba (SUT)
  - La generación de los casos de prueba desde los modelos, debería ser, en el caso ideal, automática
  - El sistema bajo prueba (**SUT**, System Under Test) puede ser:
    - Simplemente una clase o un método
    - Un conjunto de componentes
    - Todo el sistema
  - También se suelen usar modelos para especificar los propios tests

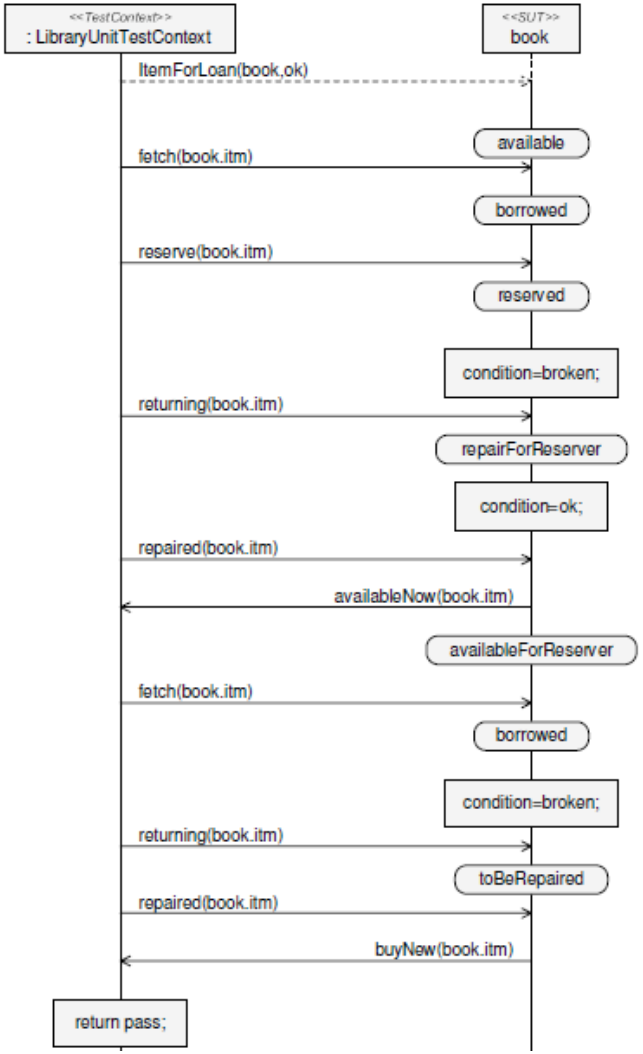


- Para cada nivel de pruebas hay modelos más adecuados:
  - Pruebas unitarias: Diagramas de estado, diagramas de actividad
  - Pruebas de integración: Diagramas de secuencia describiendo escenarios arquitectónicos
  - Pruebas de aceptación: Diagramas de casos de uso

Model-based Testing - Soporte

- Perfiles como el UTP (UML Testing Profile) dan soporte al modelado del propio entorno de pruebas
  - Define elementos como SUT, Test Component, Test Context





Contenido

- Verificación y Validación de Software
- Pruebas Unitarias
- Pruebas de Integración
- **Pruebas de Sistema**
- Pruebas de Aceptación

Pruebas de sistema

- Verifican el **comportamiento del sistema completo**

- Los fallos funcionales se suelen detectar en los otros dos niveles anteriores (unitarias e integración)
  - Desde el punto de vista funcional, la frontera entre pruebas de integración y de sistema no está claro
  - Aunque es en este nivel cuando se prueba la interacción con:
    - Sistemas externos
    - Software legado o reutilizado
    - Dispositivos hardware
    - Entorno operativo

- Este nivel es más adecuado para comprobar propiedades emergentes
  - Generalmente, correspondientes a **requisitos no funcionales**
    - Seguridad, Rendimiento, Fiabilidad, etc.

Pruebas de sistema no funcionales

- Pruebas de **rendimiento**
  - Se comprueban características de rendimiento: tiempo de respuesta, uso de memoria, etc.
- Pruebas de **estrés** o resistencia
  - Se somete al sistema a estados de carga máximos, incluso sobrepasando el máximo establecido
  - Especialmente importante en sistemas distribuidos y en aplicaciones web
- Pruebas de **fiabilidad**
  - Se generan casos de prueba que representen el perfil operacional del software, es decir, que reflejen el modo más habitual de uso y se evalúa estadísticamente la tasa de fallos que se produce
  - También denominadas pruebas estadísticas
- Pruebas de **recuperación**
  - Se fuerza un fallo grave del sistema y se evalúa la capacidad de recuperación del mismo

Pruebas de seguridad

- Pruebas de **seguridad**
  - Se verifica la capacidad del sistema a responder a diferentes ataques de seguridad: accesos no permitidos, virus, etc.
  - Técnicas:
    - Analizar el sistema en busca de vulnerabilidades conocidas (basadas en la experiencia)
      - SQL Poisoning, Buffer Overflow, etc.
    - Atacar el sistema (imposibilidad de prever futuras amenazas)
    - Uso de herramientas de análisis: Password checkers, etc.

Sommerville (2011)

Security checklist

1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.

2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.

3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.

4. If passwords are set, does the system check that passwords are 'strong'? Strong passwords consist of mixed letters, numbers, and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.

5. Are inputs from the system's environment always checked against an input specification? Incorrect processing of badly formed inputs is a common cause of security vulnerabilities.

Pruebas de usabilidad

- La usabilidad de un sistema o herramienta es una medida de su **utilidad**, **facilidad de uso**, **facilidad de aprendizaje** y **apreciación**

- Las pruebas de usabilidad consisten, por tanto, en evaluar de forma sistemática cómo de fácil es usar y aprender a usar un software

- Métodos de medida:
  - **Evaluación heurística**
    - Análisis teórico realizado por un experto en interfaz hombre-computador
    - Se comprueba si se cumplen una serie de reglas o guías
  - **Test de usabilidad**
    - Medida empírica de la usabilidad de una herramienta, sitio web o aplicación, tomada a partir de la observación sistemática de usuarios llevando a cabo tareas reales

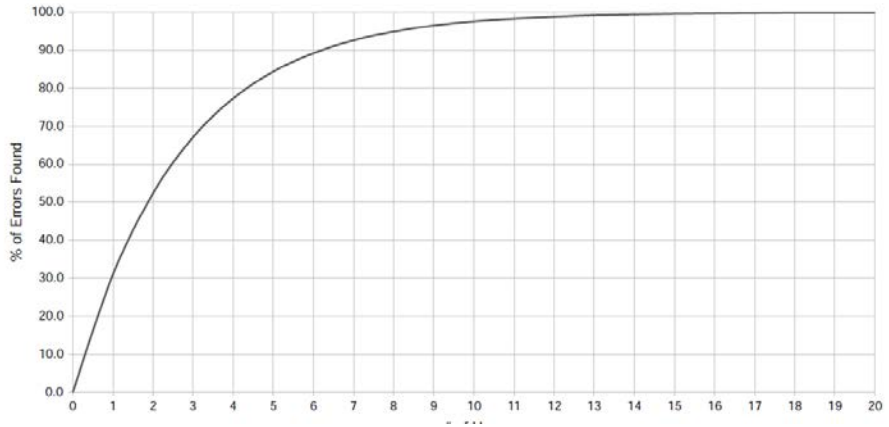
Test de usabilidad – Aspectos a evaluar

- Aspectos típicos que deben ser evaluados:
  1. ¿Es la interfaz adecuada a la preparación o entorno del usuario? ¿Se usa un lenguaje adecuado?
  2. ¿Son legibles los mensajes de usuario y proporcionan la información adecuada?
  3. ¿Existe consistencia en el estilo, formato, convenciones, terminología, etc. de todas las páginas de la aplicación?
  4. Cuando la seguridad es importante, ¿hay suficiente redundancia?
  5. ¿Tiene el sistema un número excesivo de opciones?
  6. ¿El sistema retorna algún tipo de información de recepción de entrada?
  7. ¿Es fácil navegar e interactuar con la interfaz? (Opciones de retorno, necesidad de usar minúsculas o mayúsculas, etc.)
  8. En una segunda interacción con la interfaz ¿El usuario recuerda fácilmente las opciones?

Test de usabilidad – Consideraciones prácticas

- Es necesario aplicar:
  - Mismos tests a varios individuos diferentes (con perfiles diferentes)
  - Mismos tests a los mismos individuos de forma repetida

- ¿Cuántos individuos de prueba (testers) necesitamos?



- ¿Cómo recogemos resultados?
  - Observación , grabación en vídeo, think-aloud
  - Medición de tiempos en realizar tareas, etc.
  - Encuesta final y cuestionarios

Pruebas de sistema funcionales

- Pruebas **Alfa**
  - Validar una versión inicial del software por parte de un grupo reducido de usuarios dentro de la propia empresa desarrolladora
- Pruebas **Beta**
  - Validar una versión inicial del software por parte de un grupo reducido de usuarios en el entorno del cliente
- Pruebas **de Instalación**
  - Validar el funcionamiento del sistema en su entorno final de ejecución

Contenido

- Verificación y Validación de Software
- Pruebas Unitarias
- Pruebas de Integración
- Pruebas de Sistema
- **Pruebas de Aceptación**

Pruebas de aceptación

- El cliente comprueba si el sistema cumple con sus requisitos iniciales para decidir si lo acepta o no
  - Y por tanto, paga
- Idealmente, debería ser llevado a cabo en el entorno del cliente, pero no siempre es posible
- La definición de los casos de prueba suele ser también dirigida por casos de uso, o historias de usuario
- En metodologías ágiles, este nivel de prueba no existe, pues el cliente ha estado plenamente involucrado en el propio proceso de desarrollo y prueba