Pruebas de componentes

Terminología

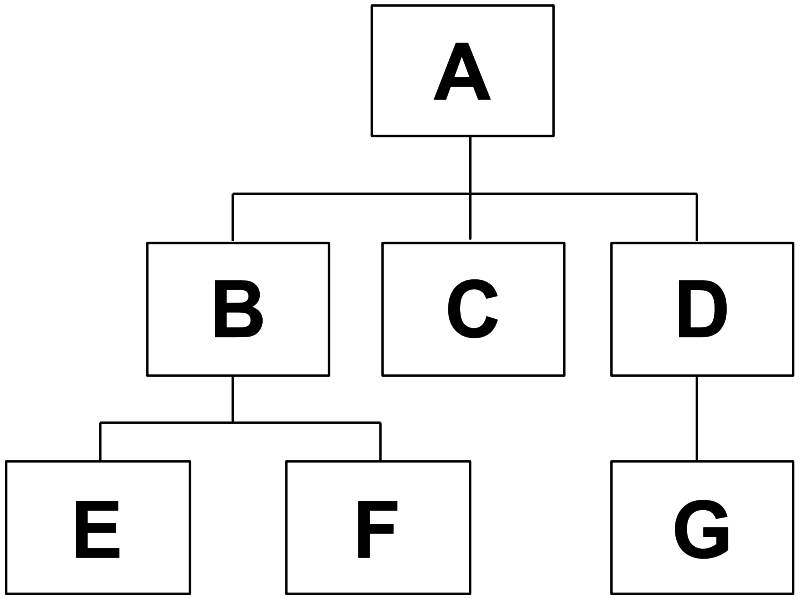
- Primer nivel de prueba: verificación unitaria o de componentes
  - **Objetos de prueba**: clases, módulos, scripts, etc.
  - **Entorno de prueba**: los objetos de prueba vienen directamente del "escritorio del programador", por lo tanto requiere un alto nivel de cooperación con desarrollo.
  - **Objetivos de prueba**: comprobar que toda la funcionalidad del objeto de pruebas es completa, se desempeña y funciona adecuadamente de acuerdo a su especificación
    - Es importante probar:
      - La robustez de los componentes (tests negativos)
      - La eficiencia (uso de recursos)
      - Mantenibilidad (estructura del código, modularidad, cumplimiento de estándares, etc)

Estrategia de pruebas (componentes)

- Técnicas estáticas (analíticas): analizar el producto (p.e. programa) para deducir su correcta operación
- Técnicas dinámicas: experimentar con el comportamiento de un producto para ver si el producto actúa como es esperado
  - Testing de caja blanca
    - Basado en el código fuente del programa, el cual se utiliza para diseñar los casos de prueba
  - Testing de caja negra
    - Basado en la especificación del programa
- “Test first” development ¿qué tipo de técnica utilizará para generar los casos de prueba?

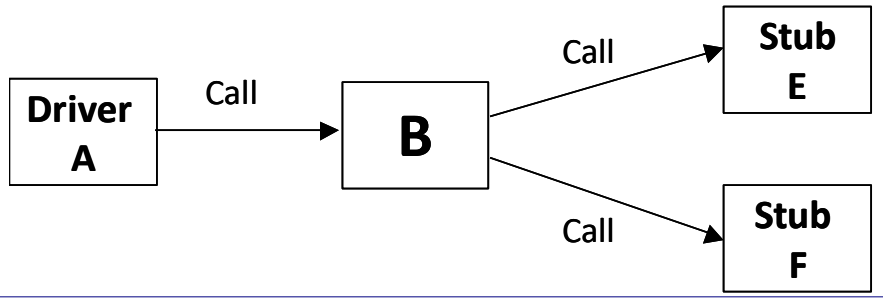
Módulos y componentes

- El módulo A “usa” a los módulos B, C, D.
- El módulo B “usa” a los módulos E y F
- El módulo D “usa” al módulo G



Pruebas de componentes

- Ejemplo: quiero probar al módulo B de forma aislada – No uso los módulos A, E y F
  - El módulo B es usado por el módulo A
    - Debo simular la llamada del modulo A al B – *Driver*
    - Normalmente el Driver es el que suministra los datos de los casos de prueba
  - El módulo B usa a los módulos E y F
    - Cuando llamo desde B a E o F debo simular la ejecución de estos módulos – *Stub*
  - Se prueba al módulo B con métodos que veremos más adelante

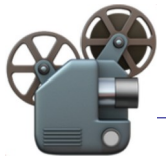


Drivers y Stubs

- Stub (simula la actividad del componente omitido)
  - Es una pieza de código con los mismos parámetros de entrada y salida que el módulo faltante pero con una alta simplificación del comportamiento.
    - De todas formas, tiene un **costo** de realización
  - Por ejemplo puede producir los resultados esperados leyendo de un archivo, o pidiéndole de forma interactiva a un testeador humano, o no hacer nada (siempre y cuando esto sea aceptable para el módulo bajo test)
  - Si el stub devuelve siempre los mismos valores al módulo que lo llama es probable que esté módulo no sea testeado adecuadamente.

Drivers y Stubs (cont.)

- *Driver*
  - Pieza de código que simula el uso (por otro módulo) del módulo que está siendo testeado. Es menos costoso de realizar que un stub
  - Puede leer los datos necesarios para llamar al módulo bajo test desde un archivo, GUI, etc
  - Normalmente es el que suministra los casos de prueba al módulo que está siendo testeado



Pruebas de integración

Terminología

- Segundo nivel de prueba: testing de integración de unidades o componentes
  - **Objetos de prueba**: conjuntos relacionados de: clases, módulos, scripts, etc.
  - **Entorno de prueba**: se utiliza también el entorno de desarrollo y los drivers y stubs ya disponibles (si fuera necesario)
- **Objetivos de prueba**: revelar problemas de interfaz, así como también conflictos entre los componentes integrados
  - Es importante probar:
    - Intercambio de datos y comunicaciones
  - ¿Pueden omitirse las pruebas de componentes? ⇒ Dificultad en el diagnóstico

Estrategia de pruebas (integración)

- ¿En qué orden y de qué forma deberíamos probar la integración de los componentes?
- En la práctica, los diversos componentes van a estar disponibles en momentos diferentes.
- Una práctica común es realizar el test de integración a medida que los componentes están disponibles
  - ¿Qué desventajas puede tener esto?
- El responsable de las pruebas debe elegir una estrategia de integración que optimice:
  - Tiempo que se insume (ahorro del tiempo)
  - Costo del entorno de pruebas

Estrategias de integración (cont.)

- No incremental
  - Big-Bang
- Incrementales
  - Bottom-Up (Ascendente)
  - Top-Down (Descendente)

  } Para sistemas estructurados de forma estrictamente jerárquica

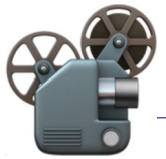
  - Por disponibilidad (Ad hoc)
  - Integración *Backbone* (esqueleto)
- El objetivo es lograr combinar módulos o componentes individuales para que trabajen correctamente de forma conjunta

Consideraciones

- Restricciones
  - La arquitectura del sistema
  - El plan de proyecto
  - El plan de pruebas
- La estrategia de pruebas puede ser un mix de varias alternativas distintas según aspectos de:
  - Riesgo
  - Disponibilidad
  - Costo
  - Otros...

Derecho a las pruebas de sistema...

- ¿Qué pasa si no se realizan pruebas adecuadas a nivel de componente/integración y se pasa derecho a las pruebas a nivel de sistema?
  - Video ilustrativo: Fixing bugs in your code (fuente: YouTube)



Pruebas de sistema

Terminología

- Tercer nivel de prueba: testing de sistema, en donde se comprueba que el sistema cumple con los requisitos especificados.
  - **Objetos de prueba**: el sistema integrado como un todo
  - **Entorno de prueba**: lo más cercano posible al ambiente de producción (error común → utilizar el ambiente de desarrollo)
    - **Objetivos de prueba**: validar que el sistema completo cumple con la especificación de sus requisitos funcionales y no funcionales
    - Es importante verificar:
      - Documentación y omisión de requisitos
  - ¿Qué pasa si no tenemos los requisitos documentados o la documentación existente no es clara al respecto?

Pruebas de aceptación

Terminología

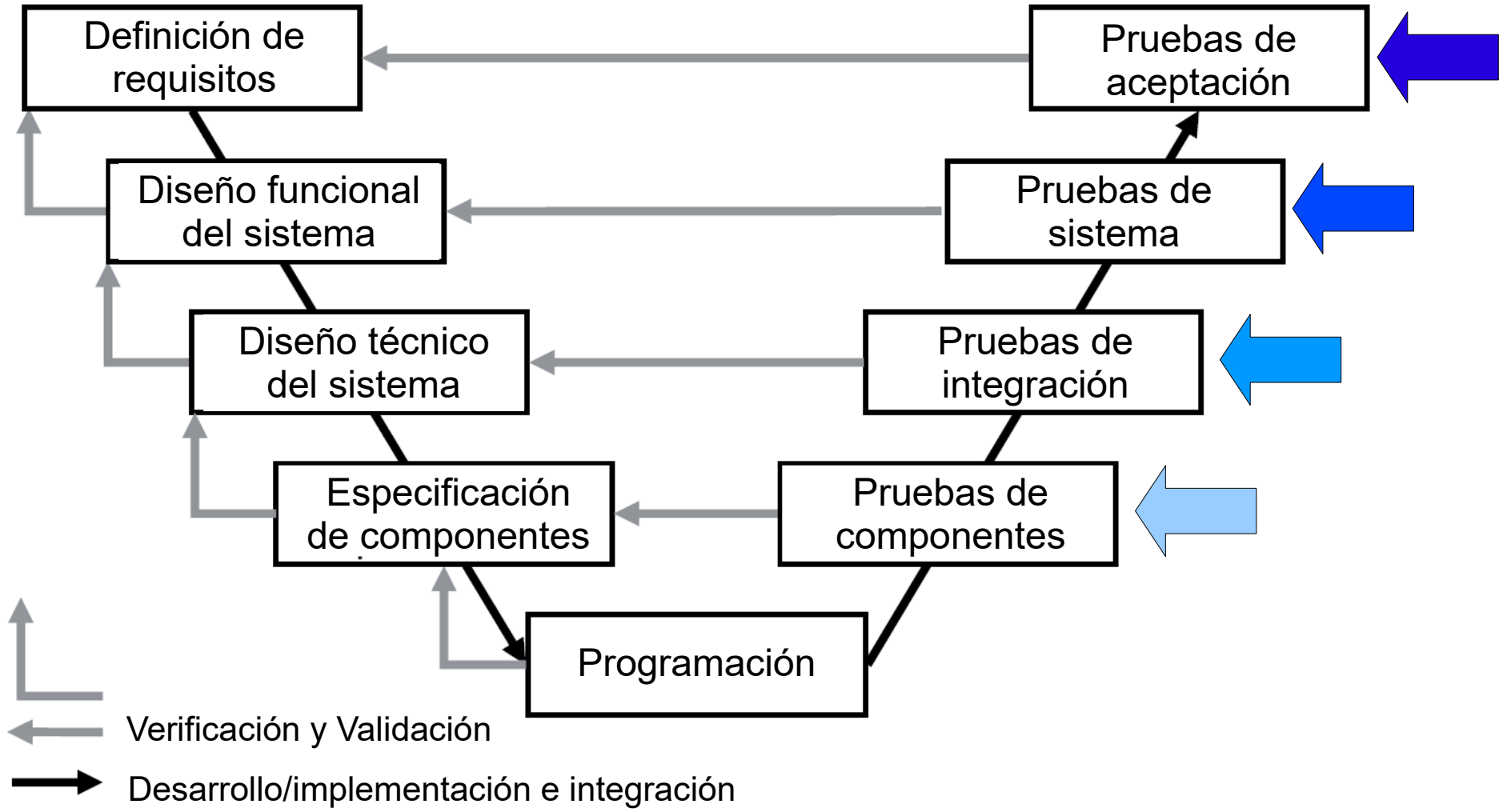
- Cuarto nivel de prueba: testing de aceptación, en donde se comprueba que el sistema es adecuado al uso y necesidades del cliente (puede realizarse como parte de las pruebas de niveles inferiores o distribuido en varios niveles de pruebas)
  - **Objetos de prueba**: el sistema (o parte de este) bajo la perspectiva del usuario/cliente
  - **Entorno de prueba**: entorno de producción
    - **Objetivos de prueba**: validar que construimos el producto "correcto"
    - **Bases de prueba**: cualquier documento que describa el sistema desde el punto de vista del usuario/cliente: casos de uso, procesos de negocio, user storys, etc.

Tipos de pruebas de aceptación

- Pruebas de aceptación contractuales
- Pruebas de aceptación de usuario
- Pruebas de aceptación operacionales
- Pruebas de campo (*alpha/beta* testing y *dogfood* tests)

Ref: Libro “Software Testing Foundation” Sección 3.5, capítulo 3.

Proceso en V


Definición de requisitos
Diseño funcional del sistema
Diseño técnico del sistema
Especificación de componentes
Programación
Pruebas de componentes
Pruebas de integración
Pruebas de sistema
Pruebas de aceptación
Verificación y Validación
Desarrollo/implementación e integración

El software se pone en producción y después… ¿sigo probando?

¡Estamos en vivo!

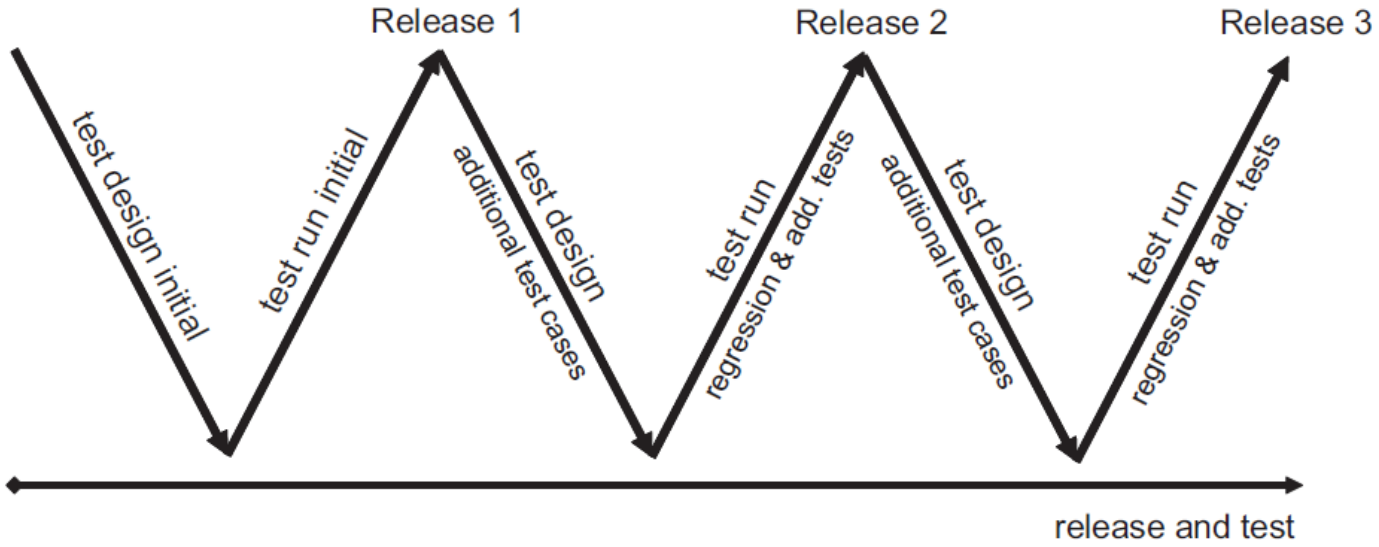
- Luego de que el sistema pasa todos los niveles de prueba y se da el OK para poner en producción… los problemas no terminan… comienzan!
  - Video ilustrativo: Trying to fix a bug in production (fuente: YouTube)

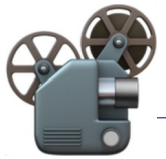
Pruebas de nuevas versiones del sistema

- Lo que vimos anteriormente es solo el **principio** del ciclo de vida del software.
- Luego de la instalación de la primera versión del software, el mismo puede ser usado por años o décadas. Siendo cambiado, actualizado y extendido muchas veces.
- El software no se “gasta” con el uso, ni los defectos se originan por dicha causa.
- El mantenimiento puede ser correctivo, adaptativo o evolutivo (lo veremos más adelante en “evolución del software”).

Pruebas de regresión

- Tanto en el mantenimiento como en procesos iterativos incrementales, es necesario realizar pruebas de regresión: verificar que los cambios o agregados realizados a la nueva versión no hayan degradado o “roto” el funcionamiento del sistema.





Tipos de pruebas y técnicas de generación de casos de prueba

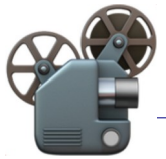
Spillner capítulos 4 y 5

Tipos genéricos de pruebas

- **Pruebas funcionales**: verifican las entradas/salidas del sistema.
  - Las técnicas de caja negra (basadas en la especificación) son las más usadas.
    - Testing basado en los requisitos y en los procesos de negocio.
- **Pruebas no funcionales**: verifican en qué grado se cumplen los requisitos no funcionales (confiabilidad, usabilidad y eficiencia).
  - Se pueden usar las pruebas funcionales para recrear el escenario donde la característica no-funcional debe ser medida.
- **Pruebas de la estructura del software**: se basan en la estructura de los artefactos (código, requisitos, arquitectura).
  - Las técnicas de caja blanca son las más usadas.
- **Pruebas relacionadas a los cambios**
  - Pruebas de regresión: verificar que nuevas faltas no se introdujeron o se enmascararon con los cambios realizados.

Técnicas de verificación

- **Técnicas estáticas** (análisis): analizar el producto para deducir su correcta operación
  - Análisis de código
  - Análisis estático
- **Técnicas dinámicas** (pruebas): experimentar con el comportamiento de un producto para ver si el producto actúa como es esperado
  - Caja negra
  - Caja blanca
  - Basadas en la experiencia



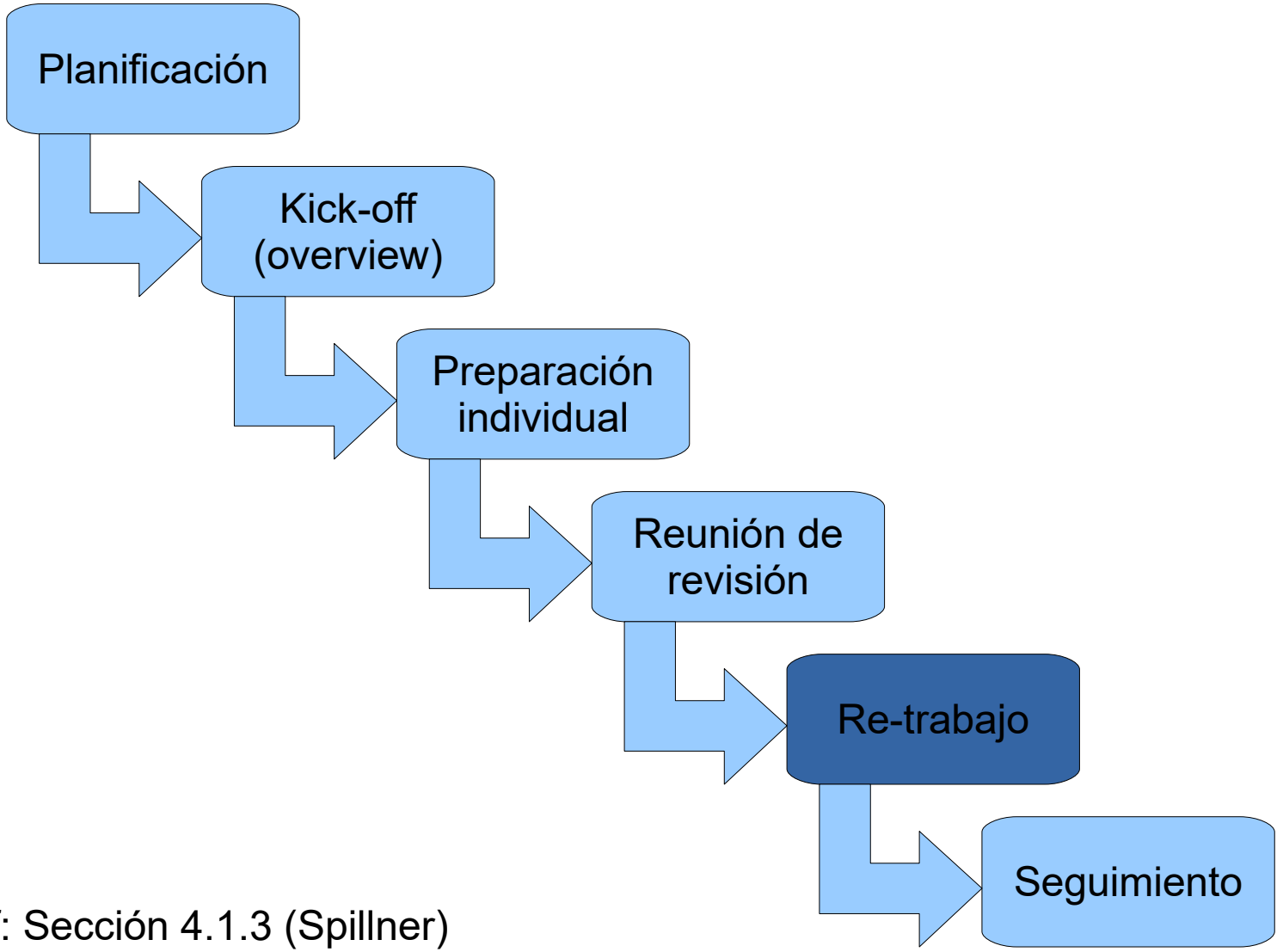
Pruebas estáticas

- Análisis de código (*Structured group evaluations*)
  - Se revisa el código buscando defectos.
  - Se puede llevar a cabo en grupos.
  - Recorridas e Inspecciones (técnicas conocidas con resultados conocidos).
- Análisis estático (*Static analysis*)
  - La entrada es el código fuente del programa y la salida es una serie de defectos detectados.
  - El análisis es realizado por herramientas (compiladores y otros).

Análisis de código

- **Revisiones**: término común utilizado para las diferentes técnicas de análisis estático realizadas por personas.
  - Se revisa el código buscando problemas en algoritmos y otras faltas.
- Algunas técnicas:
  - Recorridas e Inspecciones
    - **Objetivo**: Criticar al producto y no a la persona que lo construyó
    - Permiten:
      - Remoción de defectos **barata** de forma temprana. Menor tiempo de desarrollo.
        - El costo y tiempo necesario para las pruebas dinámicas decrece ya que el objeto de pruebas tiene menos defectos.
      - Como son realizadas en grupo, permiten un **aprendizaje mutuo**. Se unifica y mejora el estilo y método de programación, lo que produce una mejora en la calidad de los siguientes productos.
      - **No** deben usarse para **evaluar** a los programadores.

Proceso general de revisión



Ref: Sección 4.1.3 (Spillner)

Revisiones: roles y responsabilidades

- Líder (*manager*): selecciona los artefactos a ser revisados, asigna los recursos necesarios y selecciona el equipo de revisión (generalmente no participa de la reunión de revisión).
- Moderador: es el responsable de ejecutar la revisión (cada una de sus etapas). Organiza la discusión.
- Autor: presenta y explica el código (o documento).
- Revisores: expertos técnicos que participan de la revisión.
- Secretario: escribe el reporte de la reunión (problemas, acciones a tomar, decisiones y recomendaciones).

Tipos de revisiones

- En relación al **objeto** examinado:
  - Revisión de productos generados a partir del proceso de desarrollo de software.
  - Revisión del proyecto o del proceso de desarrollo en sí (*management reviews*).
- Tipos de revisiones de productos:
  - Recorridas (Walkthroughs).
  - Inspecciones (*Inspections*).
  - Revisiones técnicas (*Technical reviews*) → estudiar del libro.
  - Revisiones informales (Informal reviews) → estudiar del libro.

Recorridas

- Es un método informal y manual.
  - **Objetivo principal**: descubrir defectos, problemas y ambigüedades.
  - Otros objetivos: educar a la audiencia sobre el producto, aprendizaje mutuo, mejorar el producto, discutir formas alternativas de implementación, etc.
- Se “recorre” el producto (o se simula su comportamiento en caso de ser código) en busca de defectos.
- Es bastante más informal que otros tipos de revisiones (inspecciones por ejemplo).

Inspecciones

- Es uno de los métodos más formales de revisión. Sigue un proceso formal y prescriptivo.
- Objetivo: encontrar defectos y elementos poco claros, midiendo la calidad del objeto bajo revisión y mejorar la calidad del proceso de inspección y de desarrollo de software.
- Se examinan arterfactos (no solo aplicables al código) buscando defectos comunes.
- Los revisores (inspectores) utilizan listas de comprobación (*checklists*). Estas listas dependen del lenguaje de programación y de la organización. Por ejemplo revisan:
  - Uso de variables no inicializadas.
  - Asignaciones de tipos no compatibles.

Análisis de código: datos interesantes

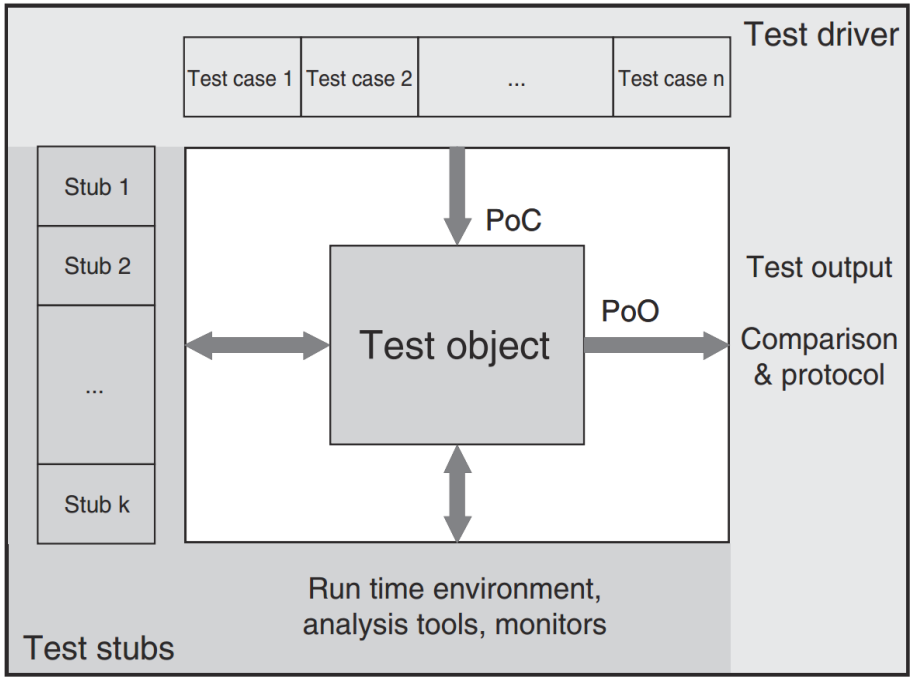
- Fagan:
  - Con Inspección se detectó el 67% de las faltas detectadas
  - Al usar Inspección de Código se tuvieron 38% menos fallas (durante los primeros 7 meses de operación) que usando recorridas
- Ackerman et. al.:
  - Reportaron que 93% de todas las faltas en aplicación de negocios fueron detectadas a partir de inspecciones
- Jones:
  - Reportó que inspecciones de código permitieron detectar 85% del total de faltas detectadas

Análisis estático

- Objetivo: (al igual que las revisiones) encontrar defectos o partes propensas a defectos en los documentos.
  - Diferencia: la revisión la realiza una herramienta.
  - El documento a ser analizado debe seguir una cierta estructura formal para poder ser chequeado por una herramienta .
- ¿Por qué estático? No requieren ejecución del código a analizar
- Es mayoritariamente un análisis sintáctico:
  - Instrucciones bien formadas (violación de sintaxis)
  - Desviación de estándares
  - Anomalías en el flujo de control o de datos
- Complementan al compilador

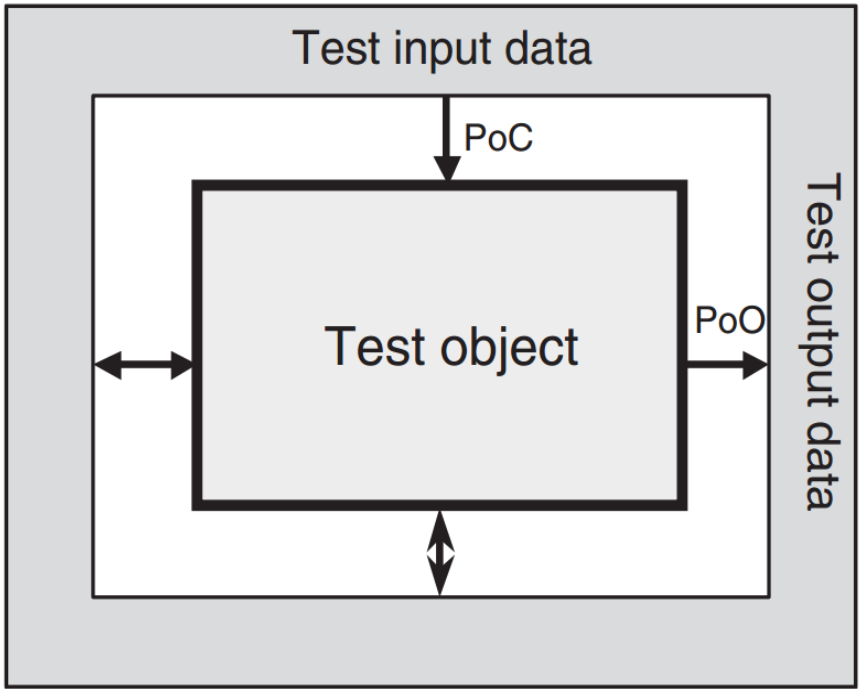
Pruebas dinámicas

- Probar el sistema ejecutando el objeto de pruebas.
- Tres categorías generales (refieren al diseño de los CP):
  - Pruebas de caja negra (también llamadas pruebas basadas en la especificación)
  - Pruebas de caja blanca (también llamadas pruebas estructurales)
  - Pruebas basadas en la experiencia



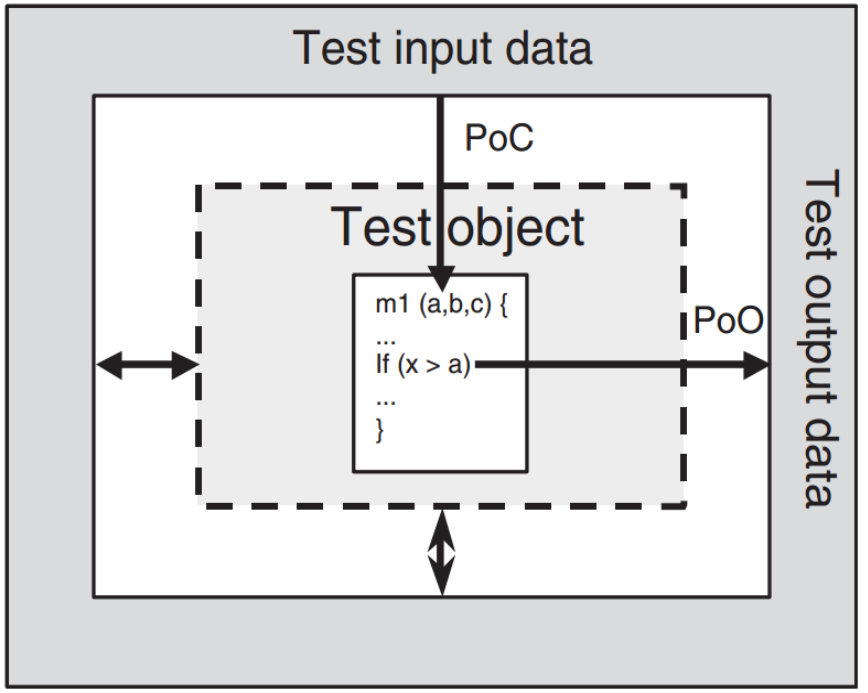
Enfoque caja negra vs. caja blanca

Black box approach

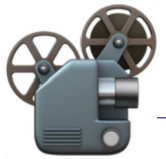


PoC and PoO “outside” the test object

White box approach



PoC and/or PoO “inside” the test object



Técnicas de caja negra

Técnicas de caja negra

- La estructura inherente o diseño del objeto de prueba no es considerado para el diseño de los casos de prueba
- Algunas técnicas:
  - Particiones en clases de equivalencia
  - Análisis de valores límites
  - Pruebas de transición de estados
  - Pruebas basadas en la lógica (grafos causa efecto, tablas de decisión y *pairwise testing)*
  - Pruebas basadas en casos de uso
- Vamos a ver las marcadas en azul, las otras las estudian del libro

Particiones en clases de equivalencia

- Clase de equivalencia: conjunto de entradas para las cuales suponemos que el software se comporta igual
- Proceso de partición de equivalencia:

  1)Identificar el dominio de entrada y de salida

  2)Identificar las clases de equivalencia del dominio de entrada

  3)Definir los casos de prueba
- Propiedades de un buen caso de prueba
  - Reduce significativamente el número de los otros casos de prueba
  - Cubre un conjunto extenso de otros casos de prueba posibles

Particiones en Cl. de Eq. (cont.)

- Identificación de las clases de equivalencia:
  - Cada condición de entrada separarla en 2 o más grupos.
  - Identificar las clases válidas así como también las clases inválidas.
  - Ejemplos:
    - “la numeración es de 1 a 999” → clase válida 1 <= num <= 999, 2 clases inválidas num < 1 y num > 999.
    - “el primer carácter debe ser una letra” → clase válida el primer carácter es una letra, clase inválida el primer carácter no es una letra.
  - Si se cree que ciertos elementos de una clase de eq. no son tratados de forma idéntica por el programa, dividir la clase de eq. en clases de eq. diferentes.

Particiones en Cl. de Eq. (cont.)

- Proceso de definición de los casos de prueba
  1. Asignar un número único a cada clase de equivalencia.
  2. Hasta cubrir todas las clases de eq. con casos de prueba, escribir un nuevo caso de prueba que cubra tantas clases de eq. válidas, no cubiertas, como sea posible.
  3. Escribir un caso de prueba para cubrir una y solo una clase de equivalencia para cada clase de equivalencia inválida (evita cubrimiento de errores por otro error).

Valores límite

- La experiencia muestra que los casos de prueba que exploran las condiciones límite producen mejor resultado que aquellas que no lo hacen.
- Las condiciones límite son aquellas que se hallan “arriba” y “debajo” de los márgenes de las clases de equivalencia de entrada y de salida.
- Diferencias con partición de equivalencia
  - Elegir casos tal que los márgenes de las clases de eq. sean probados (el límite y los adyacentes a ambos lados).
  - Se debe tener muy en cuenta las clases de eq. de la salida (esto también se puede considerar en particiones de equivalencia).

Valores límites

- Ejemplos
  - La entrada son valores entre -1 y 1 → Escribir casos de prueba con entrada 1, -1, 1.001, -1.001
  - Un archivo de entrada puede contener de 1 a 255 registros → Escribir casos de prueba con 1, 255, 0 y 256 registros
  - Se registran hasta 4 mensajes en la cuenta a pagar (UTE, ANTEL, etc) →Escribir casos de prueba que generen 0 y 4 mensajes. Escribir un caso de prueba que pueda causar el registro de 5 mensajes. LÍMITES DE LA SALIDA
- USAR EL INGENIO PARA ENCONTRAR CONDICIONES LÍMITE

Ejercicio

- En el ejemplo del triángulo detectar:
  - Clases de equivalencia de la entrada
  - Valores límite de la entrada
  - Clases de equivalencia de la salida
  - Valores límite de la salida

Ejercicio (cont.)

- Algunas clases de equivalencia de la entrada:
  - La suma de dos lados es siempre mayor que la del tercero.
  - La suma de dos lados no siempre es mayor que la del tercero.
    - Combinación para todos los lados.
  - No ingreso todos los lados.
- Algunos valores límite de la entrada:
  - La suma de dos lados es igual a la del tercero.
    - Combinación para todos los lados.
  - No ingreso ningún lado.

Ejercicio (cont.)

- Algunas clases de equivalencia de la salida
  - Triángulo escaleno
  - Triángulo isósceles
  - Triangulo equilátero
  - No es un triángulo válido
- Algunos valores límite de la salida
  - Quizás un “triángulo casi válido”
    - Ejemplo: lado1 = 1, lado2 =2, lado3= 3