Técnicas de verificación y validación de software. Pruebas

M.I. Capel

ETS Ingenierías Informática
y Telecommunicación
Universidad de Granada
Email: manuelcapel@ugr.es

Desarrollo de Software

Índice

Índice

Índice

Índice

Índice

Prueba de Software

Concepto

La actividad denominada *prueba de software* (testing) consiste realmente en un conjunto de actividades sistemáticas que se planifican antes de comenzar el desarrollo de software (codificación) y realizadas por *personal especializado* y, para proyectos muy grandes, por un equipo de expertos en pruebas (“testers").

Motivación

Las consecuencia de los fallos en el software cuestan a la economía norteamericana una cantidad estimada de 59.5 KM USD por año (G. Tassey, *The economic impacts of inadequate infrastructure for software testing, 2002*).

Se estima que se podrían suprimir 22.2 KM USD de las pérdidas anuales si se probara el software de manera adecuada durante todas las fases del ciclo de desarrollo del mismo.

Motivación

Las consecuencia de los fallos en el software cuestan a la economía norteamericana una cantidad estimada de 59.5 KM USD por año (G. Tassey, *The economic impacts of inadequate infrastructure for software testing, 2002*).

Se estima que se podrían suprimir 22.2 KM USD de las pérdidas anuales si se probara el software de manera adecuada durante todas las fases del ciclo de desarrollo del mismo.

Costes por software defectuoso (Basili–Boehm)

Corrección de defectos en la industria del software

Fase del desarrollo	Coste / defecto (USD)
Diseño y compilación	139
Compilación o encuadernación	455
Integración y pre–producción	977
En el mercado	7,136

El coste promedio a la industria por corregir cada defecto de un software que haya salido de la responsabilidad del equipo de desarrollo y que sea detectado por el cliente que recibe el producto es de 14,102 USD.

Costes por software defectuoso (Basili–Boehm)

Corrección de defectos en la industria del software

Fase del desarrollo	Coste / defecto (USD)
Diseño y compilación	139
Compilación o encuadernación	455
Integración y pre–producción	977
En el mercado	7,136

El coste promedio a la industria por corregir cada defecto de un software que haya salido de la responsabilidad del equipo de desarrollo y que sea detectado por el cliente que recibe el producto es de 14,102 USD.

Característica fundamental de las pruebas del software

- Las pruebas sólo pueden verificar el sistema y su operación respecto de criterios predeterminados o *requerimientos* del software
- La calidad del software ha de encontrarse dentro de éste, no es algo que se decida o se pueda incluir en la fase de pruebas

Origen de los defectos del software

Porcentaje de defectos respecto de la fase del desarrollo

%	Fase (introducidos en)
85	Diseño y codificación
< 2	Compilación y encuadernación
< 2	Integración
> 5	En el mercado

Los objetivos fundamentales de la prueba de software

- Identificar el origen y la magnitud de los riesgos en el desarrollo que se puedan reducir mediante la aplicación de un plan de pruebas
- Realizar comprobaciones con sistematicidad para reducir los riesgos identificados
- Saber cuándo hemos de terminar las pruebas de un software
- Gestionar las pruebas como otro proyecto dentro del desarrollo completo de un sistema informático

Concepto Clave 1

Todos los proyectos en Ingeniería suelen adolecer de defectos en el nuevo sistema o producto creado Por consiguiente, tomar la decisión de no tratar de encontrar dichos defectos, es decir, no realizar pruebas del software, no hará que los defectos desaparezcan

Verificación y Validación

Verificación

¿Estamos construyendo el producto correctamente?

Verificación se refiere al conjunto de tareas que aseguran que el software producido implementa correctamente una funcion especifica.

Validación

¿Estamos construyendo el producto correcto?

Validación se refiere a un conjunto diferente de tareas para poder seguir(“traceable”) al software hasta los requisitos.

Verificación y Validación

Verificación

¿Estamos construyendo el producto correctamente?

Verificación se refiere al conjunto de tareas que aseguran que el software producido implementa correctamente una funcion especifica.

Validación

¿Estamos construyendo el producto correcto?

Validación se refiere a un conjunto diferente de tareas para poder seguir(“traceable”) al software hasta los requisitos.

Verificación y Validación

Verificación

¿Estamos construyendo el producto correctamente?

Verificación se refiere al conjunto de tareas que aseguran que el software producido implementa correctamente una funcion especifica.

Validación

¿Estamos construyendo el producto correcto?

Validación se refiere a un conjunto diferente de tareas para poder seguir(“traceable”) al software hasta los requisitos.

¿Qué es y no es la actividad “prueba de software"?

Ejemplo: cómo probar un coche antes de comprarlo

¿Qué exámenes o comprobaciones debemos hacer antes de comprarnos un coche nuevo?

- Desde el punto de vista de usuarios–conductores del vehículo, no podemos repetir todos los diferentes tipos de pruebas que ha realizado el fabricante antes de sacarlo al mercado.
- Hemos de limitar nuestras pruebas a comprobaciones factibles a través de Internet o en el concesionario de automóviles.

Ejemplo de las pruebas a realizar

Según el conductor del vehículo

- Validar la financiabilidad de la compra
- Que nos guste
- Comprobar el grado de confort
- Capacidad, versatilidad, mantenibilidad
- Prestaciones:
  - Gasto combustible / 100 Km
  - Tipo de combustible
  - Tiempo mínimo para conseguir velocidad
  - Estabilidad en curvas
  - Estabilidad en alta velocidad ($>$ 180 KM/h)
  - Tiempo promedio entre mantenimientos

Tipos de pruebas en el ejemplo

Correspondencia con la terminología de pruebas

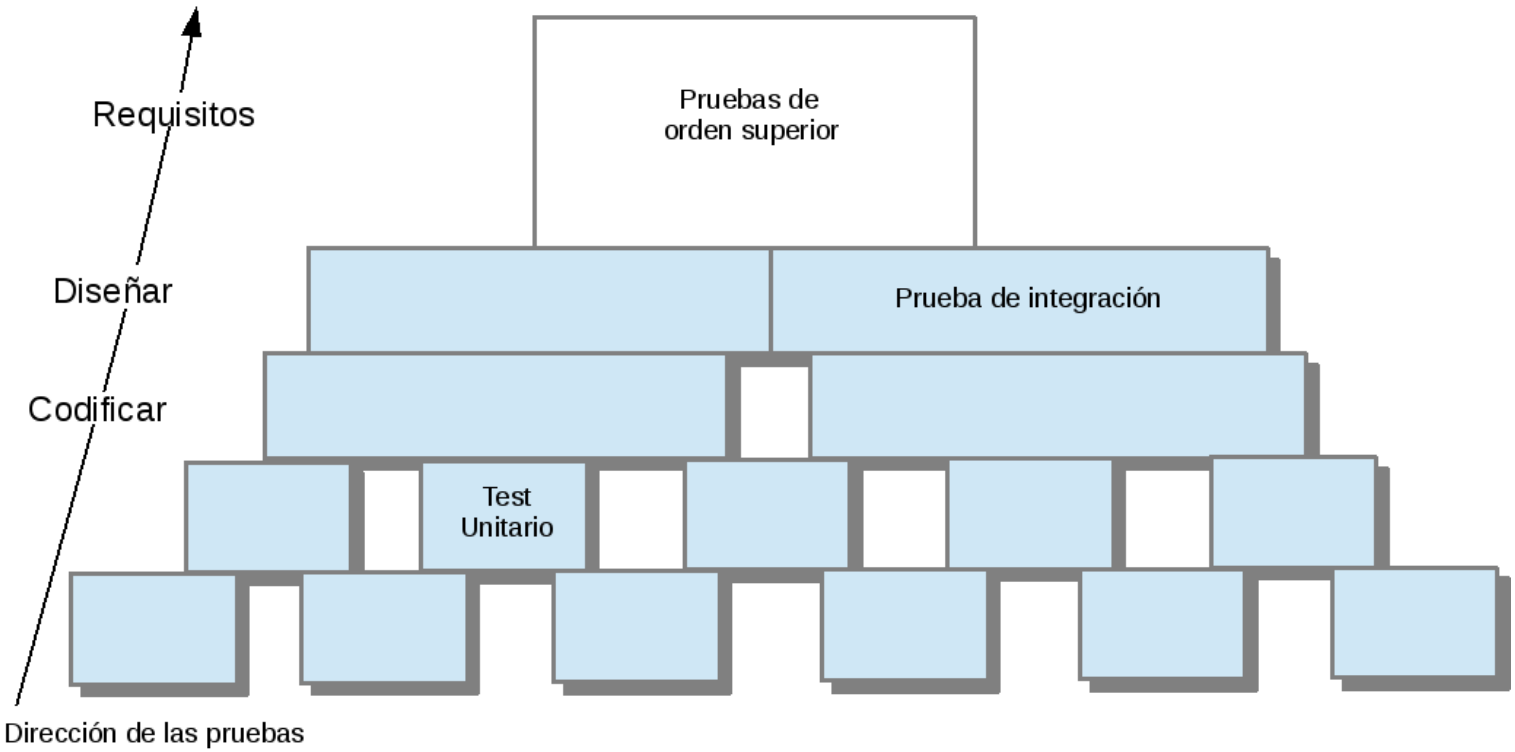
Tipo prueba	Nombre técnico
Examinar el coche sin conducirlo	Pruebas estáticas
Comprobar caracter. sin conducirlo	Pruebas funcionales y estructurales
Probar a conducirlo	Prueba de rendimiento

Técnicas de Prueba de Software

Características Comunes

- Las pruebas comienzan a nivel de componentes y se desarrollan “hacia arriba”,es decir, hacia la integración del sistema informático global.
- Diferentes enfoques de Ingeniería de Software necesitan distintas técnicas de prueba, en diferentes momentos
- La prueba (testing) y la depuración (debugging) de software son actividades completamente diferentes
- Se han de incluir pruebas de *bajo nivel* para verificar que han sido correctamente programados los diferentes segmentos de código y también pruebas de *alto nivel* que validan las funciones principales del sistema

Proceso *procedural* de la prueba del software



Proceso *procedural* de la prueba del software II

Protocolo

1. *Pruebas unitarias*: comprueban caminos específicos en la estructura de control de un componente
2. *Prueba de integración*: los componentes se ensamblan para formar un paquete software completo
3. *Las pruebas de validación*: el software ha de cumplir todos los requisitos: *funcionales*, *comportamiento*, *rendimiento*...
4. *Pruebas del sistema*: comprueba que todos los elementos (software+hardware) se combinan bien y que se consigue el rendimiento/funcionamiento global esperado

Proceso *procedural* de la prueba del software III

¿Cuándo sabemos que hemos probado el software *lo suficiente*?

- Respuesta: probablemente nunca: cada vez que el usuario lo ejecuta, el programa está siendo probado
- Criterios más rigurosos para determinar el fin de las pruebas:
  - *Cleanroom Software Engineering*: utilización de técnicas estadísticas de utilización
  - Utilización de modelos estadísticos y teoría de confiabilidad del software para predecir la completitud de las pruebas
  - Recogiendo muestras (y métricas) durante la realización de pruebas del software

Validación

Concepto

- Se trata de la culminación de la prueba de integración
- Desaparecen las diferencias entre: software convencional, OO y aplicaciones Web
- La validación tiene éxito cuando el software funciona de la manera que espera el usuario
- *Software Requirements Specification*, si se ha desarrollado: contendrá los criterios de validación

Criterios de Validación

Concepto

- Se consigue demostrar la conformidad con los requisitos del software a través de un plan de prueba y de un procedimiento
- Los casos de prueba, una vez realizados, pueden suponer la aceptación o la elaboración de una lista de deficiencias

Antecedentes de “Alpha, Beta Testing"

- Sirve para que el desarrollador conozca la forma en que los usuarios utilizarán el software que ha desarrollado y pueda modificarlo de tal forma que sea aceptado
- Los *tests-alpha* se realizan en un entorno controlado, en presencia del desarrollador

“Beta Testing"

- Los *tests-beta* son una aplicación *viva* del software en un entorno que no puede ser controlado por el desarrollador
- El cliente registra todos los problemas (reales o imaginados) y los envía al desarrollador periódicamente
- Los tests *alpha* y *beta* se utilizan para validar un producto por muchos usuarios
- *Test de Aceptación de Cliente*: es una variante del *beta*-test

Sistemas y sus pruebas

Antecedentes

- El software se incorpora junto con otros elementos en los sistemas
- Son muy importantes los pasos seguidos en el diseño y prueba del software para conseguir un alta probabilidad de integración en sistemas grandes
- La *Prueba del Sistema* se trata de realizar una serie de tests que sirven para ejercitar de una manera completa un sistema

Recuperación

Concepto

Se trata de una prueba del sistema que fuerza al software a fallar de varias maneras Comprueba se realiza la recuperación del sistema de forma adecuada La automatización implica:

- reinicialización
- *checkpointing*
- recuperación de datos, etc.

Seguridad

Concepto

- Este tipo de prueba intenta verificar que los mecanismos de protección construidos con el sistema lo protegen de intrusiones
- El *tester* juega el papel de un *hacker*
- Una buena prueba de seguridad ha de conseguir finalmente penetrar el sistema
- El diseño del sistema ha de garantizar que conseguir una intrusión es más costoso que el valor de la información que se podría obtener

Stress

Concepto

- Este tipo de prueba se utiliza para confrontar a los sistemas con situaciones de funcionamiento anormales
- “How high can we crank this up before it’s all screwed up!!!"
- Ejecuta el sistema de un modo que demande recursos en cantidad, frecuencia, o volumen anormales
- El *tester* intenta *romper* el programa
- Una variación es el denominado *test de sensibilidad*

Rendimiento

Concepto

- Esta prueba se diseña para comprobar que el rendimiento en tiempo de ejecución del software dentro de un sistema integrado y en su contexto
- A menudo, se combinan con pruebas de *stress* y normalmente necesitan instrumentación software y hardware para realizarlas:
  - Medición de ciclos del procesador
  - *Event Logs*, p.e.: interrupciones/unidad tiempo
  - Muestrear estados de dispositivos periódicamente

Despliegue

Concepto

- Ejercita el software en cada plataforma y sistema operativo en el que vaya a funcionar
- Prueba todos los procedimientos de instalación del software y el software de instalación especializado
- Examina toda la documentación que vaya a ser utilizada para introducir el software producido a los usuarios finales

Aseguramiento de la Calidad y Pruebas de Software

Miller (1977):

"The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large–scale and small–scale systems"

Aseguramiento de la Calidad y Pruebas de Software II

- V%V incluye un conjunto amplio de actividades para asegurar la calidad
- La prueba de software es la última posibilidad para evaluar la calidad de un software y, de manera más práctica, para descubrir errores ocultos
- Pero no introduce *calidad* al sistema: “If Quality isn’t there before you begin testing, it won’t be there when you’re finished testing”

Principios para realizar buenas pruebas del software

1. Los riesgos de un sistema informático de negocio se pueden reducir si se buscan sistemáticamente los defectos del software.
2. Las pruebas *positivas* y *negativas* aplicadas a dicho software contribuyen de manera importante a la reducción de los riesgos.
3. Las pruebas estáticas y de ejecución contribuyen también a su reducción.
4. Actualmente, la actividad de pruebas del software ha de ser también soportada por herramientas para automatización de pruebas.

Principios para realizar buenas pruebas del software II

5. La prioridad más importante a la hora de realizar las pruebas es la de afrontar primero los riesgos más importantes del sistema.
6. Después, la segunda prioridad ha de ser la de comprobar las actividades más frecuentes (*regla 80/20*) del negocio.
7. Análisis estadísticos (distribución de Weibull de patrones de aparición de defectos y otros errores) son muy efectivos para predecir cuándo se aconseja terminar la actividad de pruebas un software.
8. Probar siempre el sistema de la misma manera que lo utilizarán sus posibles usuarios.

Principios para realizar buenas pruebas del software III

9. Considerar que los defectos del software son el resultado de un proceso y no convertirlo en una cuestión personal que vaya a implicar a desarrolladores concretos.
10. Buscar defectos en un software es una inversión, pero también es otro coste del proyecto.

La prueba de software como una profesión

Esta actividad requiere de una mentalidad diferente de la que posee el desarrollador de software, así como de conceptos y habilidades significativamente distintas de dicho personal técnico.

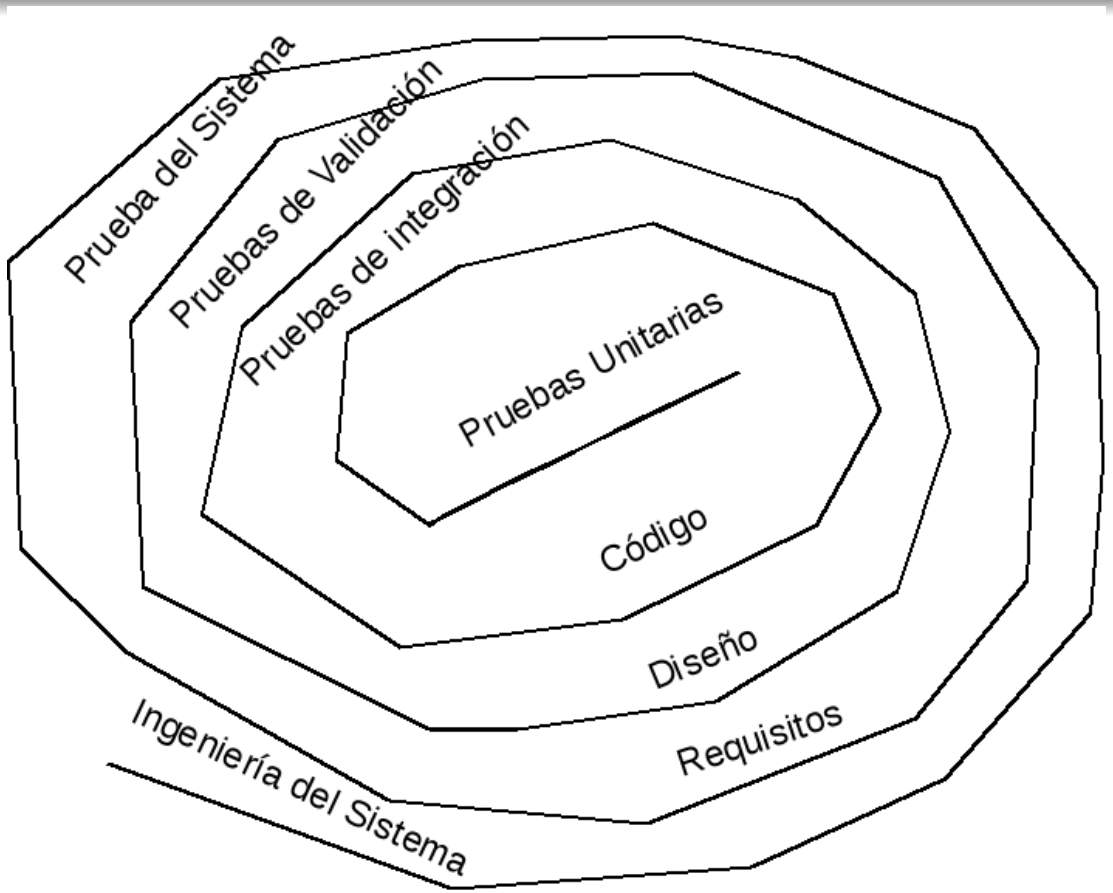
Otros conceptos–clave

- Para conseguir mayor eficacia, los planes y las actividades de pruebas han de enfocarse en riesgos conocidos del negocio para el que se desarrolla el sistema informático.
- Ningún proyecto de pruebas para un sistema de negocio puede llegar a comprobar todos los posibles defectos del software debido a las limitaciones de presupuesto, habilidades del personal técnico y recursos en general.
- El desafío más importante para un *"tester"* con experiencia consiste en identificar qué partes del sistema no se van a probar produciendo el menor impacto posible en futuras pérdidas del negocio.

Relación entre las pruebas de software y el ciclo de desarrollo

- La prueba y el desarrollo de software son actividades relacionadas, ya que el éxito de ambos procesos es muy interdependiente.
- Los procesos de prueba y desarrollo de software dependen de los procesos de soporte del otro, tales como gestión de requerimientos, búsqueda de defectos, cambios y control de versiones.
- Las pruebas deben comenzar al principio de un proyecto de desarrollo porque cualquier producto del proyecto es un excelente candidato para ser probado.

Estrategia de Pruebas de Software



Estrategia de Pruebas de Software II

Comentarios

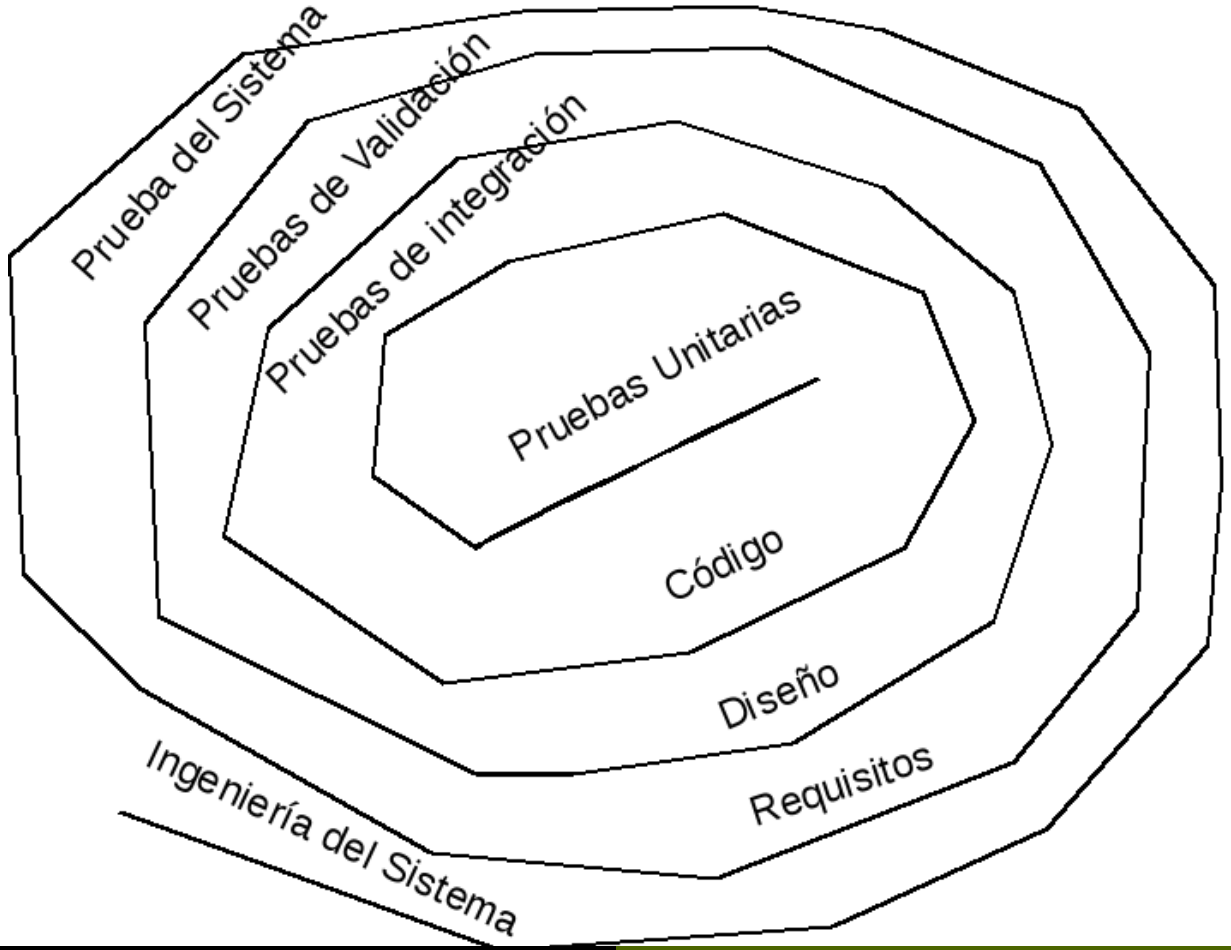
- Para producir software: se disminuye el nivel de abstracción en cada vuelta (sentido contrario al de las agujas del reloj).
- Para probar el software, seguimos la espiral hacia afuera en la dirección de las agujas del reloj ampliando el ámbito de la prueba en cada vuelta.

Estrategia de Pruebas de Software II

Comentarios

- Para producir software: se disminuye el nivel de abstracción en cada vuelta (sentido contrario al de las agujas del reloj).
- Para probar el software, seguimos la espiral hacia afuera en la dirección de las agujas del reloj ampliando el ámbito de la prueba en cada vuelta.

Estrategia de Pruebas de Software III



El Plan de Pruebas

- Documenta la planificación de las pruebas de un software
- “‘Master Test Plan":
  1. Enumeración de los sistemas y/o aplicaciones que van a ser probados
  2. Riesgos, requisitos y objetivos de las pruebas a realizar
  3. Ámbito y limitaciones de este Plan
  4. Fuentes de la “*experticia*" en el negocio, necesario para planificar y ejecutar las pruebas
  5. Fuentes de la “*experticia*" en desarrollo de software, necesario para planificar y ejecutar las pruebas
  6. Fuentes de los datos implicados en las pruebas
  7. Entornos de Pruebas y su Gestión
  8. Estrategia de Pruebas

Plan de Pruebas II

Comentarios

- Los items (1)–(8) del Plan de Pruebas anterior describen las habilidades, proceso y marco tecnológico necesario para que se puedan realizar las pruebas
- La identificación de los elementos señalados por (1)–(8) son un pre–requisito para obtener una prueba con éxito de la aplicación o del sistema

El Plan de Pruebas III

- "Master Test Plan" (2):
  9. Para cada fase del Desarrollo del Sistema, REPETIR
     - Fase concreta del Desarrollo
     - ¿Cómo sabemos que podemos comenzar las pruebas?
     - ¿Cómo sabemos que podemos acabar las pruebas?
     - Borrador de la lista de casos de prueba (ID, título, descripción)
     - Borrador planificación escrita del caso de prueba
     - Borrador planificación escrita del caso de ejecución
     - Borrador análisis resultados casos de ejecución
     - Planificación de informes
  10. Elaborar borrador con la planificación completa de las pruebas

Plan de Pruebas IV

Comentarios

- La lista de elementos enumerados en el item (9) constituyen las actividades de planificación de las pruebas específicas de la fase de desarrollo del software
- El primer borrador del Plan de Pruebas contendrá una planificación de pruebas que sigue muy de cerca la planificación del desarrollo del software encargado
- Se irá adquiriendo más información y entendiendo mejor el tipo de pruebas que quedan por realizar
- En la revisión de la planificación inicial de las pruebas salen a la luz nuevos aspectos del desarrollo detallado del software, que no eran evidentes al comienzo de éste

Casos de Prueba

Concepto

Los casos de prueba son las condiciones generales que hay que probar para una aplicación o sistema–software, que se obtienen de los requerimientos de la aplicación de negocio o de la especificación del software producido

- Uno de los resultados más importantes es el “Master Test Plan": lista de casos de prueba necesarios para verificar cada fase del desarrollo del software encargado
- El equipo de pruebas escribe los casos de prueba para cada subsistema o módulo de la aplicación, cuya planificación ha de coordinarse con el equipo de desarrollo

Casos de Prueba II

Propósito

- El “Master Test Plan" documenta de forma global “qué" hay que probar de la aplicación–software y “por qué"
- Los casos de prueba se refieren más bien al “cómo" hay que realizar las pruebas

ámbito

- Cada caso de prueba se centra en sólo 1 de las actividades incluidas en el plan de pruebas completo
- Se deben identificar algunas de las actividades más elementales del negocio, que el software ha de soportar
- Definir casos de prueba para cada una de las actividades

Completitud de los Casos de Prueba

- Confirmar que hemos tenido en cuenta la realización de todas las actividades requeridas del negocio en los casos de prueba diseñados
- Comprobar que se han cubierto todos los requerimientos de la aplicación o la especificación del software encargado para ese negocio
- Contenidos de un Caso de Prueba:
  1. ID único (indicado en el Plan de Pruebas)
  2. Título independiente (indicado en el Plan de Pruebas)
  3. Breve descripción (indicado en el Plan de Pruebas)
  4. Fase del Desarrollo en la que se aplica (indicado en el Plan de Pruebas)

Completitud de los Casos de Prueba

Contenidos de un Caso de Prueba (2)

5. Objetivos específicos de la prueba y sus medidas de logro
6. Datos sugeridos para realizar la prueba
7. Herramientas de prueba sugeridas
8. Procedimiento para comenzar la prueba
9. Procedimiento para detener la prueba adecuadamente
10. Prueba de un procedimiento de reseteo para repetir la prueba

Completitud de los Casos de Prueba

Contenidos de un Caso de Prueba (3)

11. REPETIR los pasos de ejecución de la prueba:
    - Número de paso
    - Acción asociada
    - Resultados esperados.
    - TRACEAR e INFORMAR resultados actuales
12. TRACEAR e INFORMAR primera vez que se ejecuta (fecha/hora)
13. TRACEAR e INFORMAR número de intentos hasta ejecutar la prueba con éxito
14. TRACEAR e INFORMAR lista de defectos del software
15. TRACEAR e INFORMAR ¿Ejecutada con éxito ?(Sí|No)

Validación de los casos de Prueba

Objetivos

- Si no se validan los casos de prueba, existe la posibilidad de que el equipo de pruebas esté informando de defectos del software y *pasos de ejcución* de la prueba fallidos, en lugar de defectos en la elaboración del caso de prueba
- Se trata de volver a revisar las *acciones* y los resultados asociados a los *pasos* de ejecución de la prueba
- Se comparan con los requerimientos del software, especificaciones (requerimientos y software) y se obtiene asesoramiento del experto del negocio

Validación de los casos de Prueba II

Lista de acciones asociadas a los *pasos* de la prueba

- Documentación apropiada de las interacciones del usuario final con la aplicación
- Emulación de as actividades rutinarias del negocio
- Si los resultados esperados casan con los resultados reales, se dice que el *paso* de ejecución "*pasó*" la prueba
- En caso contrario, se dice que el *paso* ha fallado la prueba y se necesita un análisis y diagnóstico posterior

Validación de los casos de Prueba III

Diferencias entre resultados reales y esperados

- Causa de resultados esperados incorrectos
  1. Falta de comprensión adecuada del "*probador*", que se transmite erróneamente al programador
  2. Si probador, desarrollador y experto en el negocio están de acuerdo en los resultados esperados, el fallo se debe a un defecto en el proceso de desarrollo o en el código

Obtención del Plan de Pruebas

Idea general

- Al comienzo del proceso de desarrollo hay suficiente información para elaborar un borrador del Plan de Pruebas
- Pero no hay nivel de detalle todavía para escribir cada uno de los casos de prueba
- Tales detalles aparecen en orden inverso a como van apareciendo las fases del proceso de desarrollo
- La forma en que se planifican las pruebas parece funcionar al contrario de lo esperado:
  1. Pruebas de la post-implementación
  2. Pruebas de la implementación final
  3. Pruebas de la implementación inicial

Diagrama de Desarrollo en “V" (2)

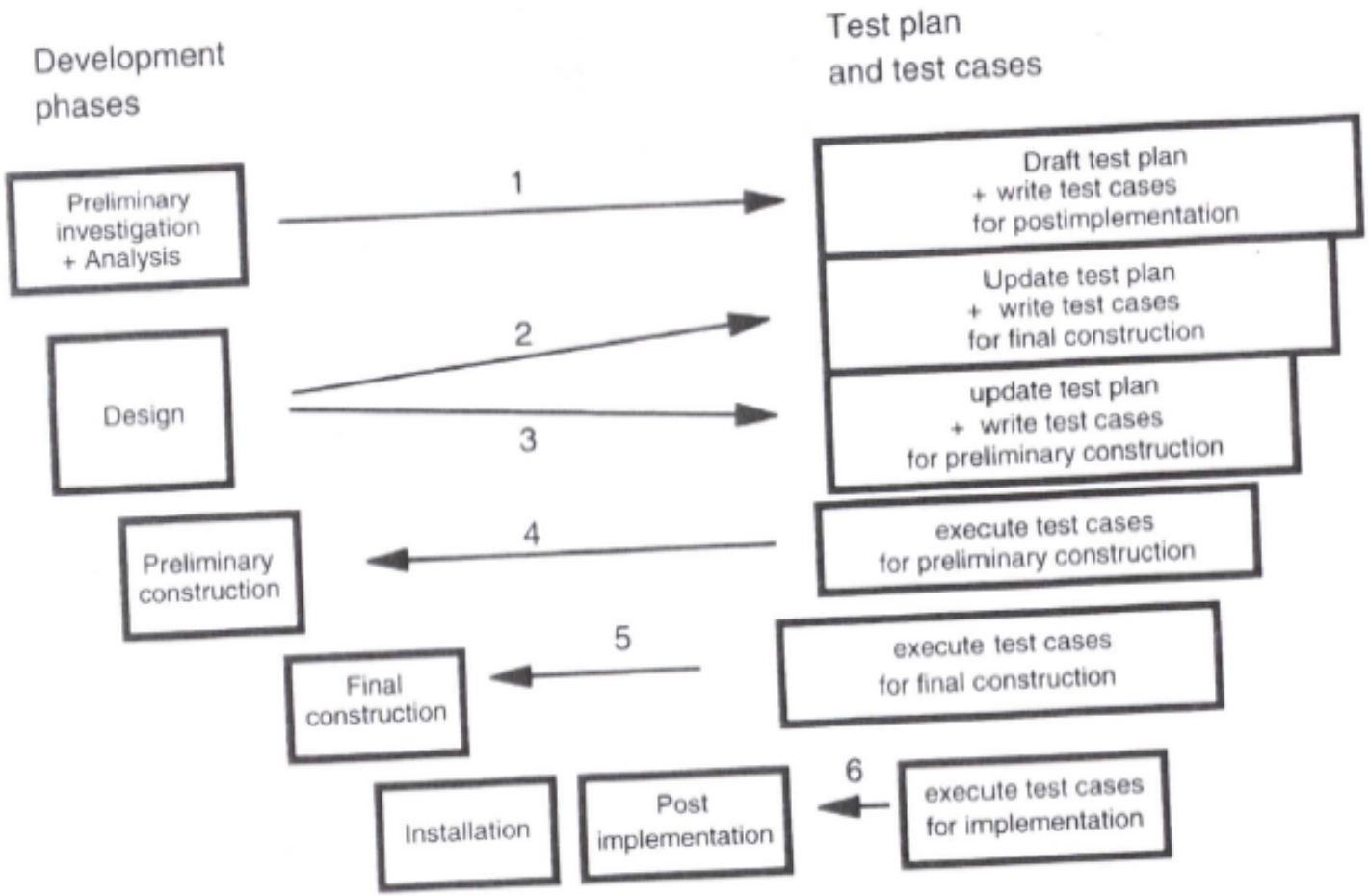


Diagrama de Desarrollo en “V" (3)

Interpretación del diagrama

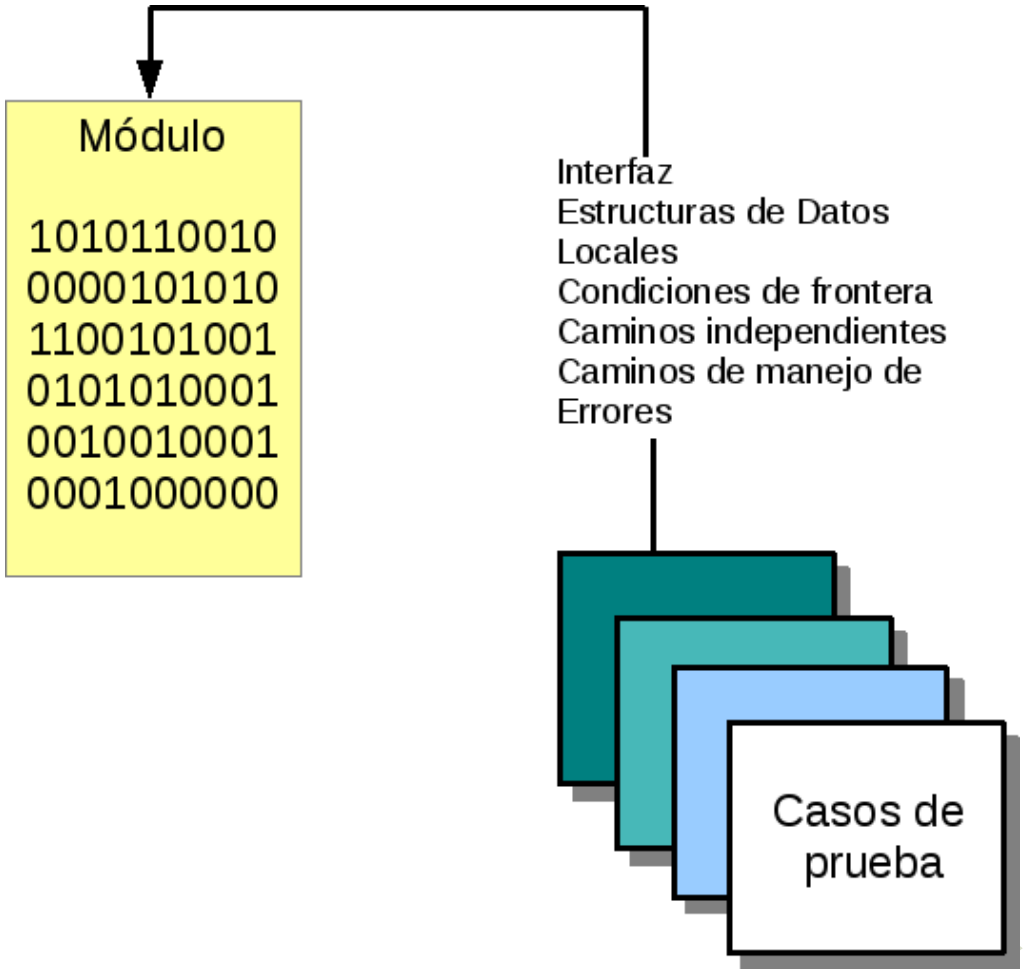
- (1) Elaboración Plan de Pruebas y caso de prueba para *post-implementación*
- (2) Escribir caso de prueba para la implementación final
- (3) Escribir caso de prueba para la construcción inicial del software
- (4) Ejecutar el caso de prueba anterior (3)
- (5) Ejecutar el caso de prueba para la implementación final del software
- (6) Comienza la post–implementación y se ha de ejecutar su caso de prueba, finalmente

Pruebas Unitarias

Antecedentes

- Concentra el esfuerzo de comprobación en la unidad más pequeña de diseño de software
- Trata de encontrar errores en la lógica interna de proceso o en las estructuras de datos de un componente
- Se prueban caminos de control importantes para descubrir errores dentro del ámbito de cada prueba
- Los errores que se pueden producir y la complejidad de las comprobaciones están, por tanto, limitados

Pruebas Unitarias II



Pruebas Unitarias III

Prueba Unitaria

- Lo primero es comprobar el flujo de datos a través de la interfaz del componente
- Las Estructuras de Datos han de ser *ejercitadas*
- Determinar el impacto que producen en los datos globales de la aplicación
- Seleccionar los caminos de ejecución a comprobar
- Diseñar casos para descubrir errores de computación, comparaciones incorrectas o flujos de control impropios
- Un buen diseño de software ha de anticipar errores y establecer su tratamiento cuando estos se produzcan

Pruebas Unitarias IV

Prueba Unitaria

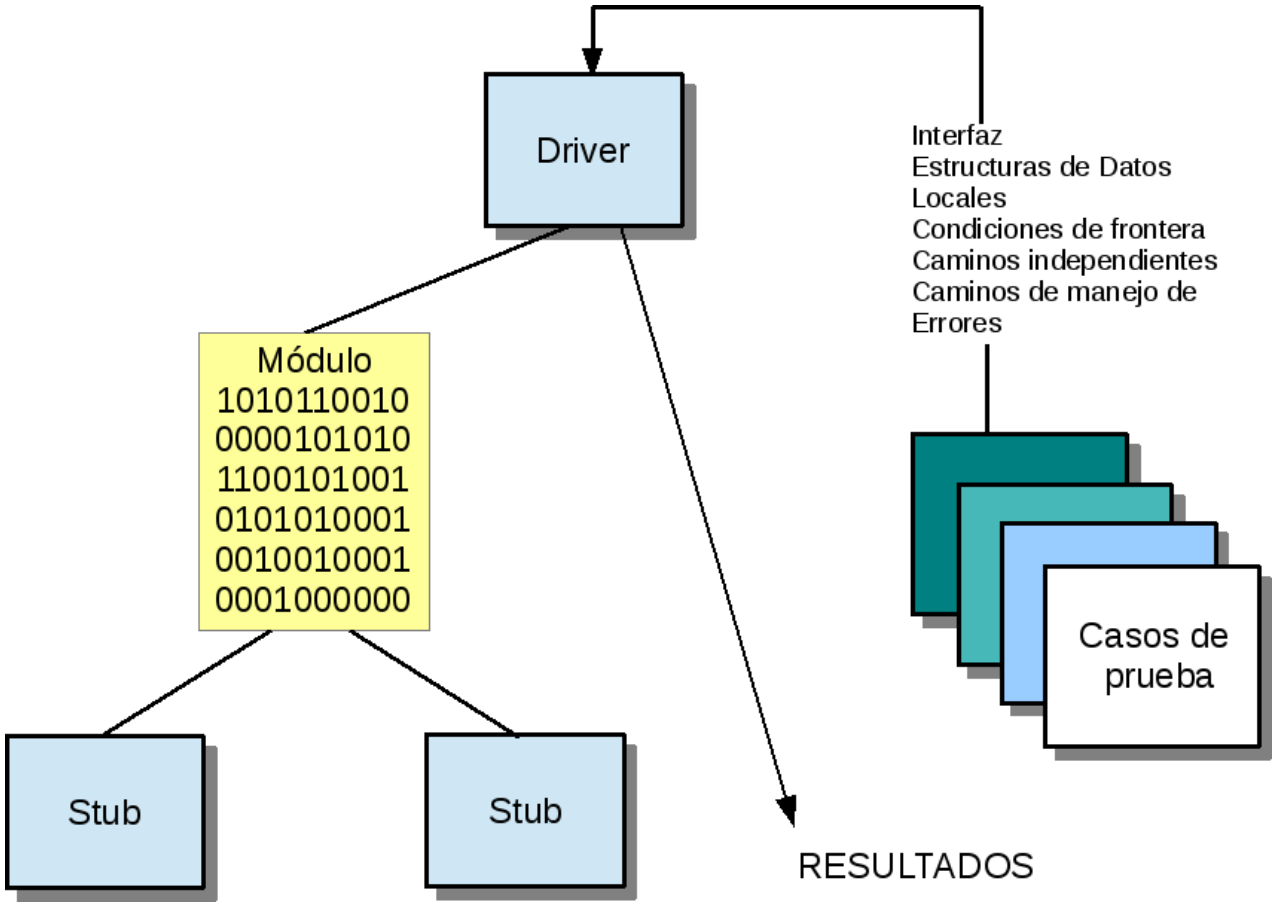
- El procesamiento de grandes estructuras de datos frecuentemente falla en los límites de la estructura
- Hay que programar casos de prueba para detectar errores en la frontera de los datos locales a cada componente

Pruebas unitarias

Procedimientos

- Las pruebas unitarias se consideran normalmente como adjuntas a la actividad de codificación del software
- El diseño de cada prueba unitaria puede ocurrir antes de que comience la codificación de un software, o después
- Para probar componentes hay que desarrollar:
  - *Driver*–software
  - *Stubs*
- Un *stub* utiliza la interfaz de un módulo subordinado (pero sin pasarle la llamada) y devuelve el control al componente
- Drivers y stubs representan la sobrecarga del software debida a su prueba

Entorno para Pruebas Unitarias



Pruebas de Integración

Cuestión Fundamental

Si todos los componentes software, tras efectuar las pruebas unitarias, funcionan individualmente ¿Por qué dudar de que funcionen también cuando se combinan todos en una aplicación?

Pruebas de Integración (II)

Causas de la falta de composicionalidad de las pruebas

- Problemas de interfaz entre componentes
- Cuando se combinan sub–funciones pueden no producir la función principal deseada
- Problemas de composición entre estructuras de datos globales repartidas entre componentes
- Los errores individuales de los componentes se amplifica cuando se combinan varios de ellos

Pruebas de Integración (III)

Concepto (Pressman, 2010)

"Se trata de una técnica sistemática para construir la arquitectura del software mientras que, al mismo tiempo, se realizan pruebas para descubrir errores asociados con la interactuación de los componentes (problema de interfaces) de un software"

Se combinan bien con la construcción de la arquitectura de software del sistema completo

Pruebas de Integración (IV)

Objetivo

Utilizando componentes comprobados con pruebas unitarias, producir un prueba comprensiva de la arquitectura de software, que se considera el *deliverable* más importante de la actividad denominada diseño del sistema

Técnicas de Pruebas de Integración

Integración incremental vs. "Big–Bang"

Todas las pruebas de los componentes combinadas de una sola vez, y el programa se prueba entero: El Caos!!!!

La *integración incremental* es la antítesis del *Big–Bang*

El caso de las interfaces

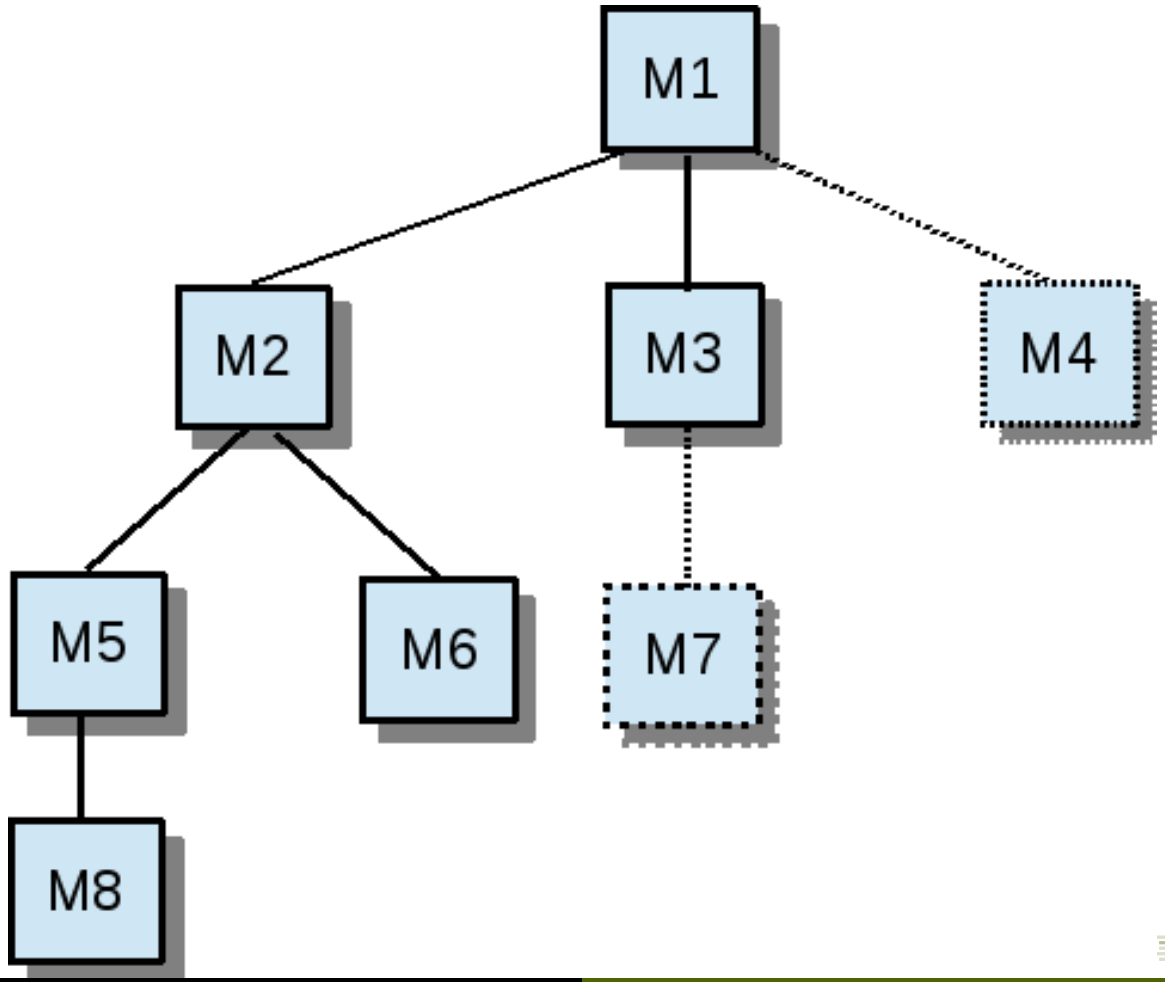
La prueba de las interfaces no se presta a la utilización de esta técnica

Técnicas de Pruebas de Integración (II)

Integración Top-Down

- Se trata de una prueba de integración incremental
- Las pruebas de los componentes se integran recorriendo la jerarquía de control (se supone en forma de árbol), hacia las *hojas*
- Comenzar en *módulo de control principal*(raíz del árbol)
- Se incorporan a la estructura recorriéndola
- Seleccionar el orden de integración de las pruebas (recorriendo el árbol en profundidad o en anchura) dependerá de las características de la aplicación

Integración Top-Down II



Integración Top-Down III

Pasos del Proceso de Integración Top–Down

1. *Driver* de la prueba: módulo de control principal
2. *Stubs*: módulos subordinados al principal
3. *Estrategia*: reemplazo de cada *stub* por el componente real al que representa, después volver a ejecutar la prueba
4. Se realizará una *prueba de regresión* para asegurar que no se han introducido nuevos errores

Integración Top-Down IV

Pros y *Cons*

- Si se corresponde la jerarquización de las pruebas con la estructura del programa, los errores se encuentran antes
- Se puede "implementar" y demostrar una función completa del software si se utiliza un recorrido en profundidad del árbol que representa el control de las pruebas
- Inicialmente, resulta una forma de probar el software poco complicada
- Pueden aparecer problemas *logísticos*
- Ocasiona falta de datos significativos sobre los que ejercitar las pruebas aplicación

Integración Top-Down V

Resolución del problema de flujo de datos

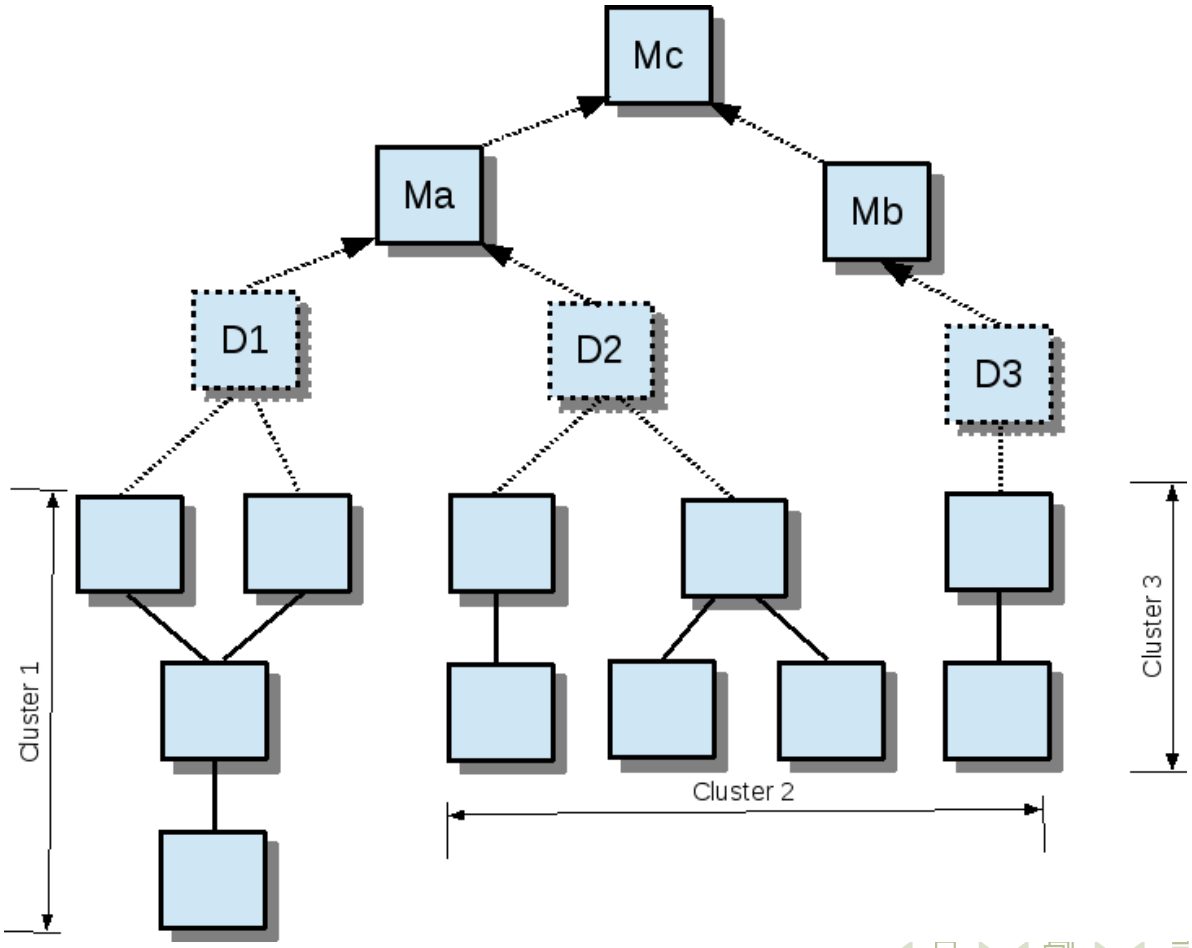
1. Integrar el software *de abajo a arriba* en la jerarquía mientras las pruebas lo hacen al contrario:
   - Retrasar pruebas hasta que los *stubs* sean reemplazados por módulos reales
   - Implica realizar más pruebas de regresión
2. Desarrollar *stubs "inteligentes"*, que realicen funciones limitadas que simulan el módulo actual

Técnicas de Pruebas de Integración (III)

Integración Bottom–Up

Se comienza construyendo y probando primero *módulos atómicos*, es decir, componentes en los niveles más bajos de la estructura del programa, eliminando, de esta forma, la necesidad de crear *stubs* en la estructura de la prueba de integración

Integración Bottom–Up II



Integración Bottom–Up III

Pasos del Proceso de Integración Bottom–Up

1. Los módulos atómicos se combinan en “builds"
2. Se programa un *driver* para cada *build*
3. Se prueba el *build*
4. Se van eliminando *drivers* y los *builds* se combinan en *clusters*
5. Desplazamiento hacia arriba en la estructura jerárquica de la prueba de integración

Integración Bottom–Up IV

Pros y *Cons*

- Facilita el encontrar los fallos en las interfaces de los módulos de bajo nivel
- No es necesario programar *stubs* para llevar a cabo la prueba de integración
- Los componentes de la Interfaz de Usuario del sistema ahora son los últimos en ser comprobados
- Induce más regresión cuando se encuentran fallos en los módulos de nivel superior de la jerarquía de integración

Pruebas estáticas

Concepto

- Para realizar este tipo de pruebas no es necesario ejecutar la aplicación que queremos probar
- Reducción de defectos del software debido a la mejora de la documentación, a partir de la cual se desarrolla
- Ayudan a conseguir la operación correcta por el usuario final, cuando el software está ya desarrollado y entregado
- Es una de las formas más costo–efectivas de identificar los probables defectos en el software de las aplicaciones
- Son recomendables para cualquier fase del desarrollo, excepto en la de instalación del software

Motivación

Observación:

Unas pocas horas bien aprovechadas de *pruebas estáticas* sobre la documentación de un proyecto de desarrollo pueden ahorrar muchas horas de corrección y rediseño de software.

Documentación para revisar

1. Software Development Manager/ Gestor de Desarrollo de Software
2. Software Developers/ Desarrolladores del Software
3. Testers / Equipo de Pruebas
4. Administrator / Administrador del Sistema
5. Documentación de Usuario

Documentos

(1)Del Gestor

- Requisitos (*"Requirements"*) del Software
- Planificación del Proyecto de Desarrollo

(2)Del Desarrollador

- Casos de Uso
- Diseño (diagramas de clases, . . .), Diagramas de Flujo de Datos
- Diseño de Bases de Datos y archivos
- Interfaces
- Especificaciones sobre conectividad (red)
- Especificaciones sobre Seguridad
- Especificaciones de Pantalla/Ventanas/Páginas
- Código

Documentos

(1)Del Gestor

- Requisitos (*"Requirements"*) del Software
- Planificación del Proyecto de Desarrollo

(2)Del Desarrollador

- Casos de Uso
- Diseño (diagramas de clases, . . .), Diagramas de Flujo de Datos
- Diseño de Bases de Datos y archivos
- Interfaces
- Especificaciones sobre conectividad (red)
- Especificaciones sobre Seguridad
- Especificaciones de Pantalla/Ventanas/Páginas
- Código

Documentos II

(1)Del “Tester"

- Planificación de las Pruebas
- Casos de Prueba
- Especificación del Entorno de Pruebas
- Prueba de Fuentes de Datos y Preparación
- Instalación de Herramientas de Prueba de Software

Técnicas de Pruebas Estáticas

Revisión de Contenidos (basadas en)

- Pruebas de Escritorio (“Desk Checking")
- Inspecciones
- Pruebas Estructuradas de Recorrido (“Walkthroughs")

Pruebas funcionales

Concepto

- Validar el comportamiento del software con respecto a lo que tiene que hacer, tal como aparece documentado
- Ejercitar lo más completamente que sea posible el software que soporta la actividades diarias de un negocio, mediante la ejecución de *casos de prueba*:
  - Se obtienen a partir de los *casos de uso* previos
  - Inicialmente para componentes individuales: menús,ítems de datos, páginas web, bases de datos, etc.
  - Finalmente, se prueban los componentes software de forma conjunta y en orden inverso al del código producido

Pruebas funcionales II

Operatoria

- Se ejecutan los casos de prueba para los diferentes caminos de negocio (business paths)
- La ejecución de los casos de prueba se realiza en orden inverso al del código producido para el sistema

Objetivos

- Prueba de navegación del usuario
- Pruebas de ventanas de transacciones
- Prueba del flujo de las transacciones
- Prueba de ventanas de reporte
- Prueba de flujo de ventanas de reporte
- Pruebas de eliminación/ actualización/ recuperación/ creación de registros en una base de datos

Prueba de Regresión

Concepto

- El software cambia, durante una prueba de integración, cada vez que se añade un módulo. Tales cambios pueden producir problemas en el comportamiento de funciones que antes funcionaban perfectamente.
- Se trata de la re-ejecución de algunos subconjuntos de pruebas que ya han sido realizadas para asegurar que los cambios no se han propagado produciendo efectos colaterales indeseados en el resto del software ya comprobado

Prueba de Regresión II

Contenidos de la prueba:

1. Un ejemplar representativo de todas las posibles comprobaciones, que ejerciten todas las funciones del software a probar
2. Comprobaciones que se centran en la prueba de componentes software que han sido cambiados
3. Comprobaciones adicionales que se enfocan a la prueba de funciones de software que podrían verse afectadas por los cambios

Técnicas de Pruebas de Caja Blanca

- Se trata de verificar la corrección de las sentencias, caminos en el código, condiciones, bucles y flujo de datos del software producido
- Para realizar este tipo de pruebas se necesita tener acceso al *fuente*, así como a: requisitos, casos de uso, datos y ejecutables
- Suelen ser realizadas por el propio programador depurando el código que ha producido
- Cuanto mayor sea la cobertura lógica (algoritmos, protocolos, procedimientos, etc.), menos defectos se descubrirán posteriormente con otros tipos de pruebas

Técnicas de Pruebas de Caja Blanca II

Cobertura de Sentencias

- Se trata de verificar qué ocurre si cada sentencia del código se ejecuta al menos una vez
- También qué procentaje en promedio de las líneas de código fuente de un programa consiguen ejecutarse
- Linea de base: *cuanto mayor sea la cobertura de la prueba del código fuente, el número de defectos del software que encontremos después será menor*

Técnicas de Pruebas de Caja Blanca III

Cobertura de Ramas

- Se trata de determinar qué porcentaje de las ramas (verdadero/falso) lógicas de un programa se han ejecutado
- La coexistencia de ramas que nunca se han ejecutado con sentencias no comprobadas suele ser la causa de muchos errores en la lógica de los programas
- El escoger probar las ramas simples en un programa antes de pasar a hacerlo con ramas de condiciones compuestas hace las pruebas de este tipo más cortas
- Para probar todas las posibles combinaciones de valores de verdad en ramas de condiciones compuestas se utilizarán *tablas de verdad*

Técnicas de Pruebas de Caja Blanca IV

Definición

Una secuencia de sentencias de un programa desde la primera sentencia ejecutable a través de una serie de otras sentencias hasta llegar a una sentencia *return/stop/end/exit*

Cobertura de Caminos

- Determinar qué porcentaje de los caminos en el código fuente de un programa han sido ejecutados

Técnicas de Pruebas de Caja Blanca V

Cobertura de Bucles

- Consiste en determinar el porcentaje de sentencias *loop–while–repeat–do* de un programa que hayan sido ejecutadas completamente
- El objetivo es forzar a ejecutar el bucle en el programa: 0, 1, n/2, n y n+1 veces
- Los valores extremos de los límites (0,n+1) comprueban condiciones inesperadas o inapropiadas del bucle
- Bucles no ejecutados completamente y los que se ejecutan sólo dentro de sus límites esperados son una *bomba de tiempo* en los programas

Pruebas de Caja Negra

- Verificar la corrección del comportamiento del software que soporta directamente la actividad diaria en un negocio
- Se le denomina también: *cobertura del comportamiento* de un software
- Tener acceso a los requisitos, casos de uso, datos y sólo al código ejecutable y sus datos
- Por desarrolladores o “testers" siempre que no hayan producido el código que se va a verificar
- Se hacen pruebas *positivas* (comportamiento esperado del software) y *negativas* (comportamiento inesperado)
- Las pruebas negativas suelen sacar a la luz más defectos del software

Identificación de Datos

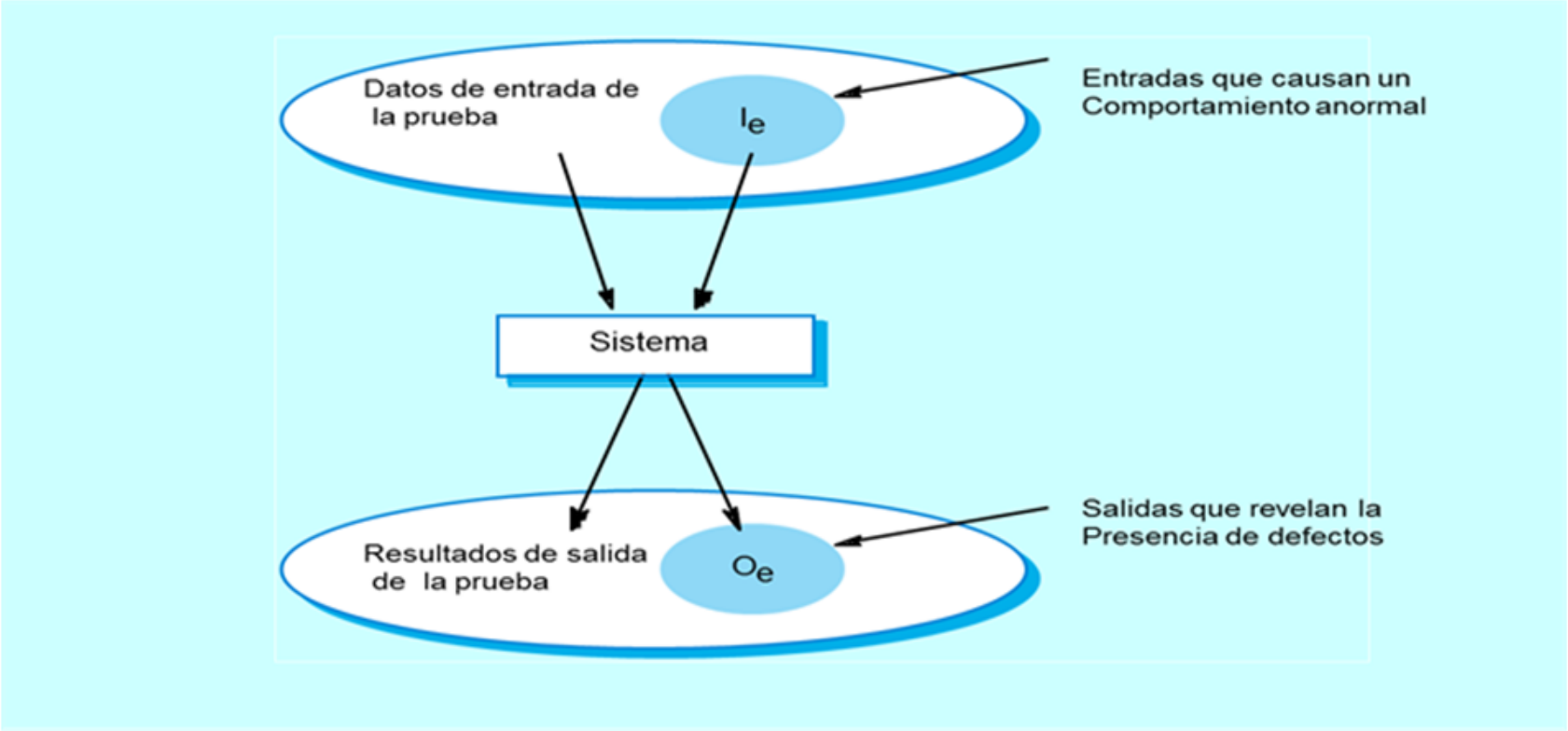


Figura: Descubrimiento de defectos del software desde los datos

Técnicas de Pruebas de Caja Negra I

Clases de Equivalencia

- Identificar grupos de datos de entrada que tienden a hacer que la aplicación se ejecute de la misma manera
- Reducen sustancialmente el número de datos de prueba necesarios para verificar un comportamiento específico
- Se aplica, por ejemplo, agrupando por campos de datos con valores que se pueden clasificar de otra forma (ID/clase de personal)
- Se deben seleccionar clases de equivalencia que incluyan separadamente valores del comienzo, del final y del medio de un campo de valores

Técnicas de Pruebas de Caja Negra II

Cobertura de los Valores Esperados

- Encontrar reglas de negocio en los requisitos de la aplicación que definan los resultados esperados
- Se trata de obtener valores–resultados de prueba a partir de sus valores de entrada asociados
- Elaborar una tabla de combinaciones válidas de entradas a clases de equivalencia con valores frontera para las reglas de negocio que se espera proporcionen los resultados

Técnicas de Pruebas de Caja Negra III

Ejemplo Cobertura de Valores Esperados (1)

Input	Input	Output
Edad	Copago	Habitación
0-6	50 EUR	50 EUR
7–17	75 EUR	100
18–35	100	150
36–49	125	300
50-74	200	350
75-99	250	400

Tabla: Reglas de negocio

Técnicas de Pruebas de Caja Negra

Ejemplo Cobertura de Valores Esperados (2)

Input	Input	Output
Edad	Copago	Habitación
0-6	50 EUR	50 EUR
-1	50 EUR	error-edad no está en el rango
7–17	75 EUR	100
5	75 EUR	error–el copago no es válido
6	75 EUR	100
7	75 EUR	100

Tabla: Tabla de combinaciones

Pruebas Estructurales

Concepto

- Se trata de validar el comportamiento del software que utilizan (dan soporte) las aplicaciones de usuario
- Reduce el riesgo de fallo del denominado *software de plataforma*
- Son pruebas de tipo *no–funcional*
- Las pruebas de *caja blanca* no se pueden aplicar
- La mayoría de las pruebas de *caja negra* tampoco

Técnicas de Pruebas Estructurales I

Pruebas de Interfaz

- Probar el paso de datos entre la aplicación y los distintos componentes de la plataforma se realiza correctamente
- Han de ser probados: archivos de datos, APIs, peticiones a bases de datos, mensajes por la red
- Se puede realizar en 4 pasos:
  1. Producir datos pero mantener la trasferencia inhibida
  2. Eliminar los inhibidores de transferencia de datos
  3. Escribir tests para que la aplicación pida datos desde otros componentes de la plataforma
  4. Permitir que los componentes de la plataforma–software alimenten con datos reales de entrada a la aplicación

Técnicas de Pruebas Estructurales II

Pruebas de Seguridad

- Utilizar clases de equivalencia para probar el comportamiento relativo a la seguridad
- Puede implicar la encriptación de las palabras de paso, lo cual complica la prueba basada en clases de equivalencia
- El equipo de pruebas ha de verificar la degradación del rendimiento de la aplicación por motivo de esta prueba
- Desde el punto de vista de las pruebas de regresión, conviene realizar pruebas de seguridad lo antes posible en la aplicación

Técnicas de Pruebas Estructurales III

Pruebas de Instalación

- Se trata de comprobar si la nueva aplicación se ha situado correctamente en entorno de producción
- Es necesario probar el proceso de instalación para asegurase que los clientes podrán hacer funcionar la aplicación
- Normalmente se ha de contar con una plataforma software+hardware análoga al entorno de utilización para hacer la prueba de la aplicación
- Hay que proporcionar al cliente información acerca de si el proceso de instalación se desarrolló correctamente

Técnicas de Pruebas Estructurales IV

Pruebas de Humo

- Configurar una aplicación instalada consiste en seleccionar entre una lista de opciones cómo ha de operar el software para cumplir con las reglas específicas
- Fijar parámetros de arranque (“start up"): ubicación archivos, número de sesiones, ID/password, zonas...
- Reglas de procesamiento: clases de seguridad, planificación de arranque y backups del sistema,...

Técnicas de Pruebas Estructurales V

Pruebas de Humo II

- Este tipo de prueba se utiliza para verificar de forma no exhaustiva que una aplicación software instalada puede ser posteriormente bien configurada
- La dificultad consiste en identificar las combinaciones de configuración más probables de la aplicación para probarlas
- Cada nueva configuración se prueba diferencialmente con respecto a una que anteriormente funcionó

Técnicas de Pruebas Estructurales VI

Pruebas de Administración

- Se trata de una ampliación de las pruebas funcionales de las actividades de un negocio
- Prueba de las actividades de *soporte* a un negocio
- Los componentes administrativos de una aplicación podrían haberse desarrollado antes que los funcionales, en ese caso los resultados de la prueba serán el punto de comienzo de la prueba de estos ultimos
- Si ocurre los contrario, los archivos de configuración manual utilizados para probar la funcionalidad de la aplicación se pueden utilizar como los resultados esperados de las pruebas de componentes administrativos

Técnicas de Pruebas Estructurales VII

Pruebas de Backup y Recuperación

- Los archivos de *backup* se utilizan para restaurar el software a un estado próximo al que tenía antes de fallar
- Si no se prevén situaciones de fallo en las aplicaciones de negocio durante su desarrollo, entonces probablemente éstas no se recuperarán en la eventualidad de un fallo
- Realización de backups de archivos críticos del negocio: *master files*, *transacciones* y *antes* y *después* de instalación de imágenes del sistema
- Hay que realizar varios backups, interrumpir la aplicación anormalmente y restaurar la aplicación utilizando sólo los backups guardados

Pruebas de Rendimiento

Concepto

- Se trata de validar la *velocidad de ejecución* del software respecto de la *necesidad de velocidad* del negocio, tal como se expresó en el documento de requerimientos
- La denominada *velocidad* del software se consigue como una buena combinación del tiempo de respuesta y la carga de trabajo cuando se producen *cargas–pico* debidas a los usuarios activos
- Pruebas de rendimiento: una serie de tests que introducen de forma incremental más carga de trabajo por una combinación de transacciones de un negocio para varias ventanas temporales de ejecución de la aplicación

Metas Importantes de las Pruebas de Rendimiento

Desafíos a Afrontar

1. Identificar correctamente a qué transacciones y actividades concretas del negocio se les necesita *medir el rendimiento*
2. Por cada grupo de transacciones relevante del negocio, determinar la *utilización–pico* de recursos computacionales y cuándo ocurren (ventanas de tiempo) los picos
3. Determinar cuántos picos de carga de trabajo hay que comprobar para conseguir una buena prueba de rendimiento

Técnicas de Planificación de las Pruebas

Pasos para pruebas de carga de trabajo

- Llevar a cabo una intensificación de la carga de trabajo del software hasta llegar a una situación de *carga–pico*
- Ejecutar medidas de rendimiento en el entorno de dicho pico de carga de trabajo
- Llevar a cabo una disminución de la carga de trabajo del software desde la última situación de *carga–pico*

Técnicas de Planificación de las Pruebas-II

Documentación de Requisitos de Carga de Trabajo

- El análisis de carga de trabajo de una aplicación software debe identificar los grupos de transacciones y sus requisitos de rendimiento

Grupo Transacciones	Tiempo respuesta
Navegar menús	3s. (máx)
Des/Conexión	3s. (máx)
Información productos	4s. (máx)
Operación compra	7s. (máx)
Búsqueda catálogo	10s. (máx)
Pago con Tarjeta	30s. (máx)
Envío	24h. (máx)

Tabla: Ejemplo de aplicación de compras por Internet

Técnicas de Planificación de las Pruebas-II

Documentación de Requisitos de Carga de Trabajo

- El análisis de carga de trabajo de una aplicación software debe identificar los grupos de transacciones y sus requisitos de rendimiento

Grupo Transacciones	Tiempo respuesta
Navegar menús	3s. (máx)
Des/Conexión	3s. (máx)
Información productos	4s. (máx)
Operación compra	7s. (máx)
Búsqueda catálogo	10s. (máx)
Pago con Tarjeta	30s. (máx)
Envío	24h. (máx)

Tabla: Ejemplo de aplicación de compras por Internet

Técnicas de Planificación de las Pruebas II

Documentación de Picos de Carga de Trabajo

- Para caracterizar un pico de carga lo importante es determinar los usuarios activos en ese momento, no los usuarios conectados
- La competencia por los recursos entre usuarios de una aplicación sólo se produce dentro de una misma ventana temporal
- Instrumentación de las pruebas de carga-pico: determinar *cómo de bien* compiten las transacciones respecto de diferentes cargas de trabajo
- Hay que añadir la previsión de uso máximo de cada grupo de transacciones a la planificación de la carga de trabajo

Técnicas de Planificación de las Pruebas III

Documentación de Picos de Carga de Trabajo (2)

Grupo Transacciones	Tiempo respuesta	Activos	Fecha de la actividad
Navegar menús	3s. (máx)	2000	L-V: 12-13h
Des/Conexión	3s. (máx)	2000	L-V: 12-13h
Información productos	4s. (máx)	2000	L-V: 12-13h
Operación compra	7s. (máx)	500	S: 9-11h
Búsqueda catálogo	10s. (máx)	2000	L-V: 12-13h
Pago con Tarjeta	30s. (máx)	500	S: 9-11h

Tabla: Planificación de la carga de trabajo para obtener rendimiento

Técnicas de Planificación de las Pruebas IV

Documentación de Picos de Carga de Trabajo (3)

- Para obtener una estimación del rendimiento de la aplicación en todo momento, hay que averiguar cuántos *picos de carga de trabajo distintos* hay que comprobar

Grupo Transacciones	Tiempo respuesta	Activos	Fecha de la actividad
Navegar menús	3s. (máx)	500	S: 9-11h
Des/Conexión	3s. (máx)	500	S: 9-11h
Información productos	4s. (máx)	500	S: 9-11h
Operación compra	7s. (máx)	500	S: 9-11h
Búsqueda catálogo	10s. (máx)	500	S: 9-11h
Pago con Tarjeta	30s. (máx)	500	S: 9-11h

Tabla: Planificación de la carga (sábados) para obtener rendimiento

Técnicas de Planificación de las Pruebas IV

Documentación de Picos de Carga de Trabajo (3)

- Para obtener una estimación del rendimiento de la aplicación en todo momento, hay que averiguar cuántos *picos de carga de trabajo distintos* hay que comprobar

Grupo Transacciones	Tiempo respuesta	Activos	Fecha de la actividad
Navegar menús	3s. (máx)	500	S: 9-11h
Des/Conexión	3s. (máx)	500	S: 9-11h
Información productos	4s. (máx)	500	S: 9-11h
Operación compra	7s. (máx)	500	S: 9-11h
Búsqueda catálogo	10s. (máx)	500	S: 9-11h
Pago con Tarjeta	30s. (máx)	500	S: 9-11h

Tabla: Planificación de la carga (sábados) para obtener rendimiento

Técnicas de Ejecución de las Pruebas

Idea general

- Para poder predecir el rendimiento de la aplicación hay que crearse los picos de carga de trabajo en un entorno de pruebas
- Esto se realiza en 3 pasos:
  1. Intensificación de la carga de trabajo hasta alcanzar el pico
  2. Medida de rendimiento en el pico
  3. Disminución de la carga de trabajo desde el pico

Prueba de Rendimiento de un Componente

Idea general

- Tener una estimación temprana acerca de si la inclusión de un nuevo componente software nos puede llevar cerca del valor máximo del tiempo de respuesta previsto en la planificación de carga de trabajo

Prueba de Rendimiento de un Componente II

Rendimiento de "*viaje completo*" (roundtrip)

- Se trata de la medida del tiempo de respuesta de la aplicación desde que el usuario envía su petición hasta que los resultados se han mostrado completamente
- Esta medida de rendimiento incluye:
  - Procesamiento implicado y realizado en la parte del usuario
  - Comunicación necesaria a través de la red
  - Procesamiento secundario en otros computadores de soporte para la transacción

Prueba de Rendimiento de un Componente III

Ejemplo de Medida de Rendimiento de "*Viaje Completo*"

0.5s	venta de compra en el cliente
0.2s	transmisión a la red
2.4s	registro en servidor
1.3s	generar orden al almacén (servidor)
0.7s	generar confiración compra (servidor)
0.1s	tramsión confirmación (red)
0.6s	mostrar registro confirmación (cliente)

Total tiempo de respuesta $=$ $5.8s$ $<$ tiempo máximo en pico (previsto en la planificación de carga de trabajo)

Caso de Estudio: "*Compras del Sábado*"

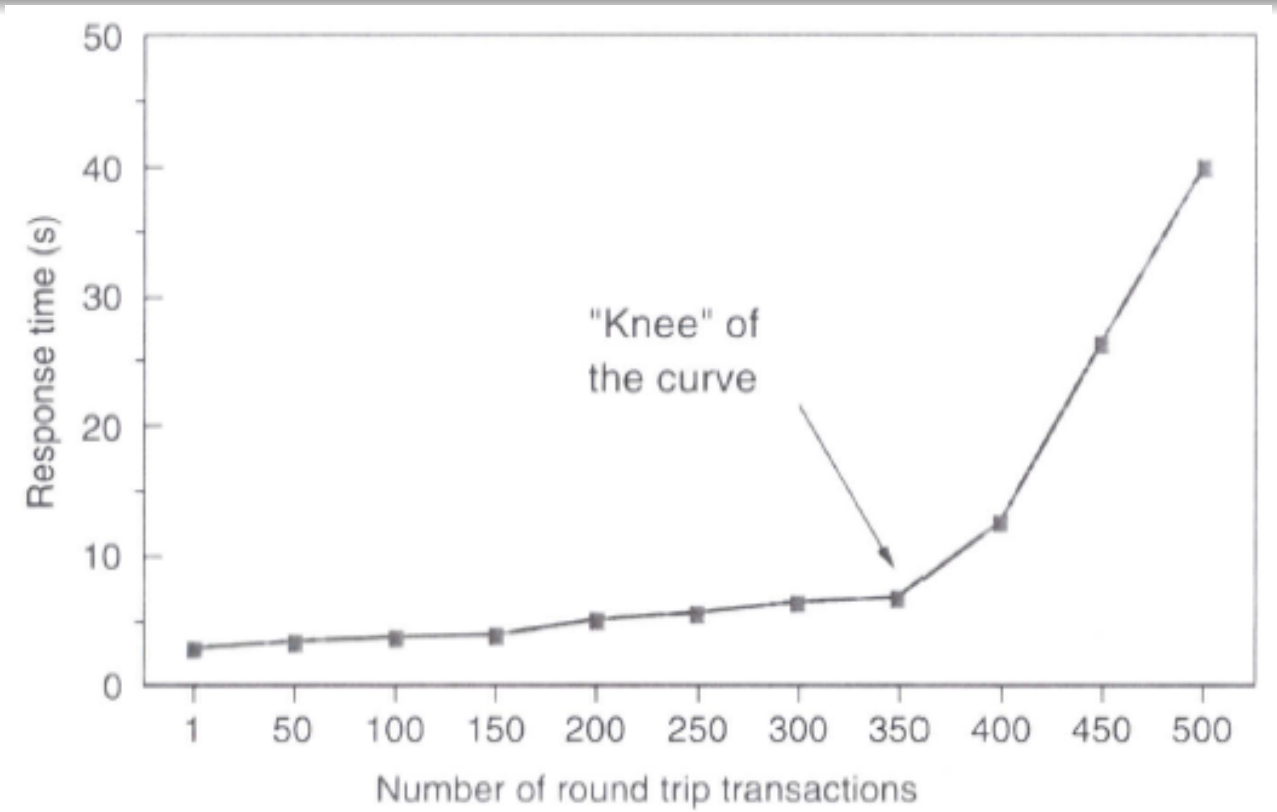


Figura: Rendimiento de viaje completo

Caso de Estudio: "*Compras del Sábado*"

Comentarios a la figura

- En un entorno de pruebas, se lanza la ejecución de la aplicación, inicialmente sin usuarios activos
- Se lanzan cada vez más copias de la misma transacción
- Cada una de ellas representa a un usuario activo, y observar la forma de la gráfica bajo unas condiciones de carga de trabajo creciente
- El tiempo (eje de ordenadas) representa el *peor tiempo de respuesta* de todas las transacciones actualmente activas
- El tiempo de respuesta promedio entre todas las transacciones no se puede utilizar para determinar si se cumple o no el requisito de rendimiento

Caso de Estudio: "*Compras del Sábado*"

Comentarios a la figura (2)

- El codo o *rodilla* ("*knee*") de la gráfica se debe a algún tipo de embotellamiento en el que se interfieren las transacciones, que se produce en algún momento de sus caminos de procesamiento
- La gráfica no informa de las causas del embotellamiento, sólo de las circunstancias en las que se produce
- La única forma de descubrir dónde se produce el *codo* es ejecutar las pruebas incrementando la carga de trabajo hasta que aparezca en la gráfica

Prueba de Rendimiento de un Componente IV

Previo a la Prueba de Carga de Trabajo

1. En un *sistema vacío*, para todas las transacciones, el rendimiento de viaje completo inicial ha de ser menor que el tiempo máximo en pico previsto en los requerimientos del software
2. El rendimiento de viaje completo de las transacciones empeorará conforme éstas compitan por recursos en un sistema más ocupado
3. Todas las transacciones (color verde) que cumplan (1) están preparadas para la prueba de carga de trabajo

Rendimiento de viaje completo de las transacciones

Grupo Transacciones	Tiempo respuesta	Activos	Fecha	Rendimiento
Navegar menús	3s. (máx)	500	S:9-11h	4.5s
Des/Conexión	3s. (máx)	500	S:9-11h	2.0s
Información productos	4s. (máx)	500	S:9-11h	15.3s
Operación compra	7s. (máx)	500	S: 9-11h	3.0s
Búsqueda catálogo	10s. (máx)	500	S: 9-11h	1.6s
Pago con Tarjeta	30s. (máx)	500	S: 9-11h	103s

Tabla: Planificación de la carga (sábados) para obtener rendimiento

Caso de Estudio: "*Compras del Sábado*"

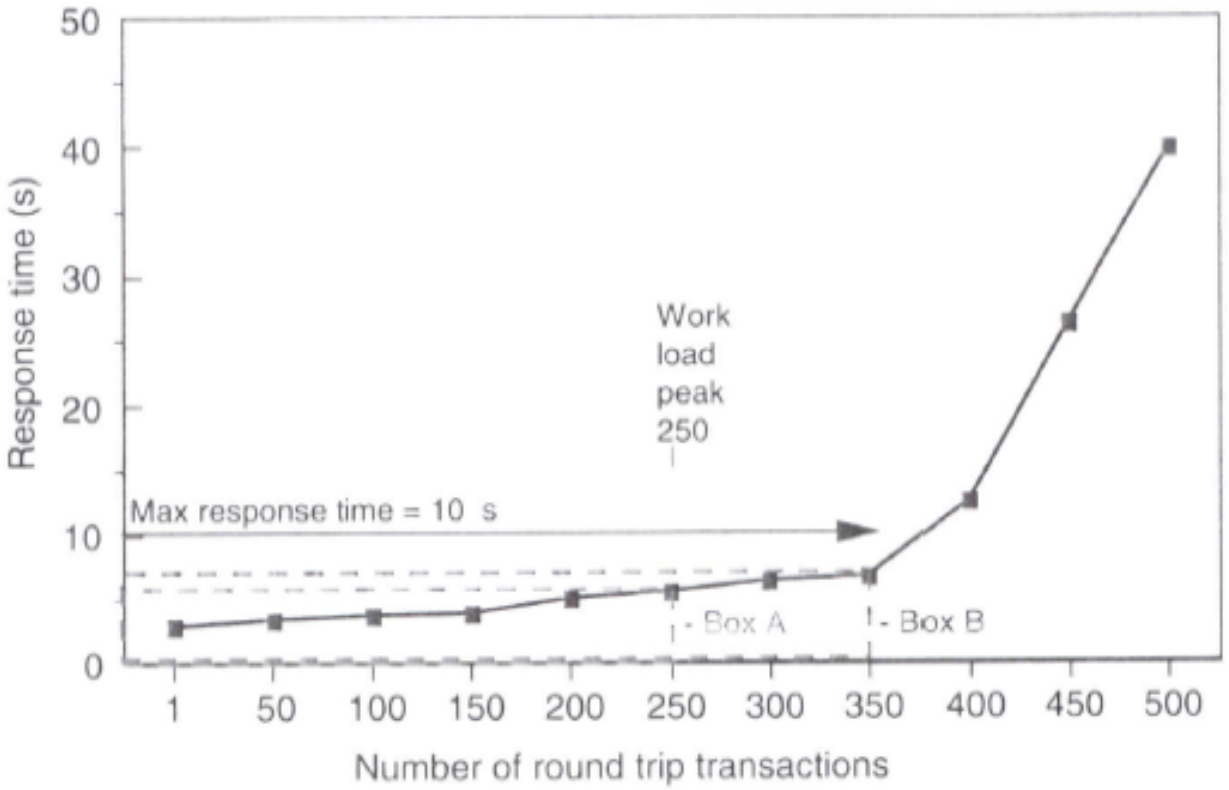


Figura: Rendimiento - Transacciones de búsqueda en catálogo

Caso de Estudio: "*Compras del Sábado*"

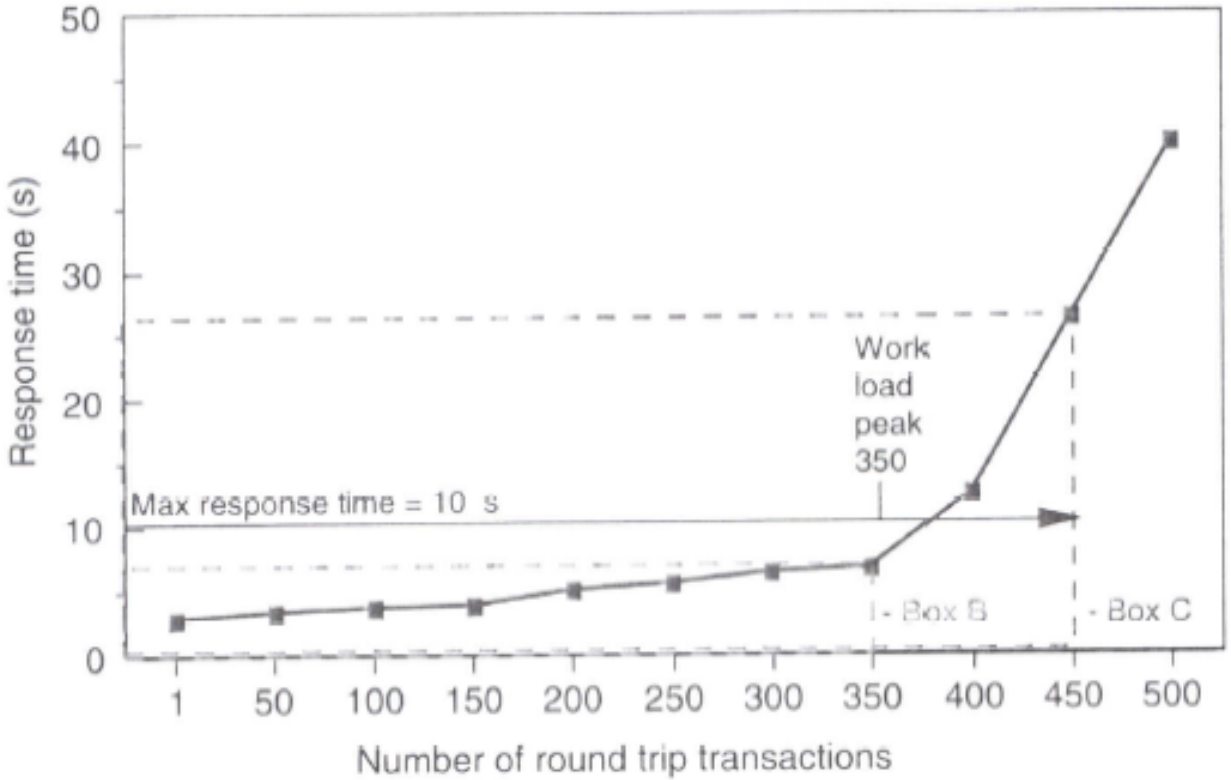


Figura: Rendimiento - Transacciones de búsqueda en catálogo

Caso de Estudio: "*Compras del Sábado*"

Gráficas del t.respuesta para la búsqueda en catálogo

- "*Box A*" incluye el tiempo de respuesta de peor caso para el *pico de carga de trabajo* que se observa con 250 transacciones activas (5.6 s)
- "*Box B*" extiende el test hasta las 350 transacciones activas, obteniéndose una gráfica lineal y valores debajo del máximo requerido (6.8s en el pico)
- "*Box C*" extiende el test hasta las 450 transacciones , lo cual hace aparecer el codo de la gráfica que eleva el tiempo de respuesta en el pico de carga a 26.4 s
- Al llegar a las 350 transacciones activas al mismo tiempo, la ejecución de la aplicación comenzará a ser muy lenta

Caso de Estudio: "*Compras del Sábado*"

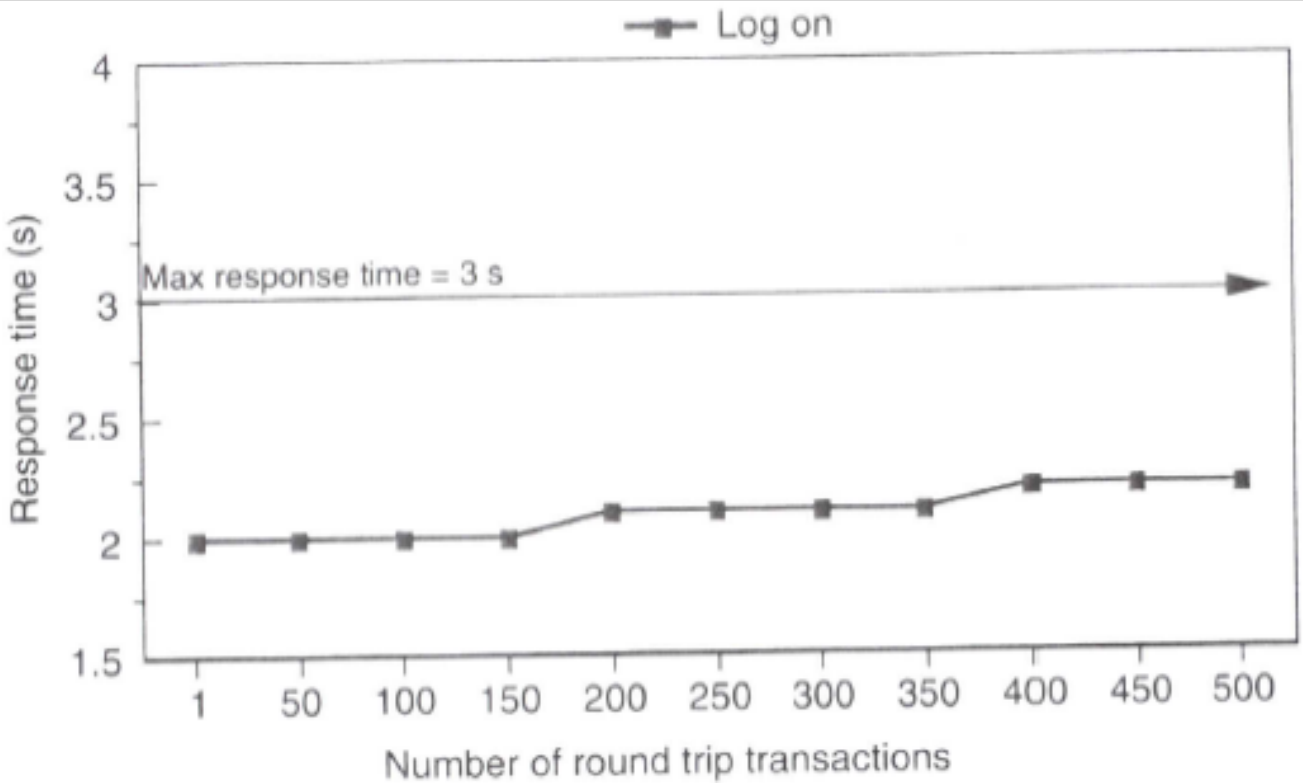


Figura: Pico de Trabajo - Transacciones de conexión (logon)

Caso de Estudio: "*Compras del Sábado*"

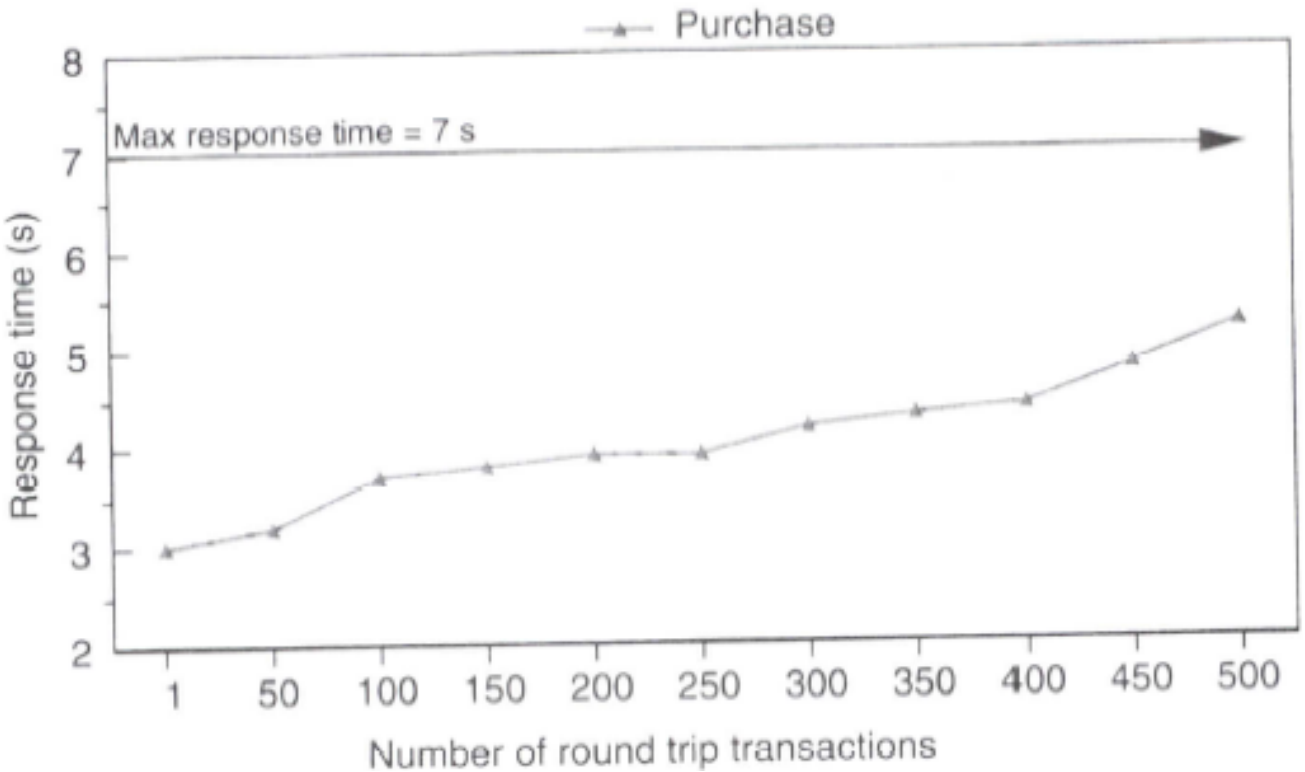


Figura: Pico de Trabajo - Transacciones de compra artículo

Caso de Estudio: "*Compras del Sábado*"

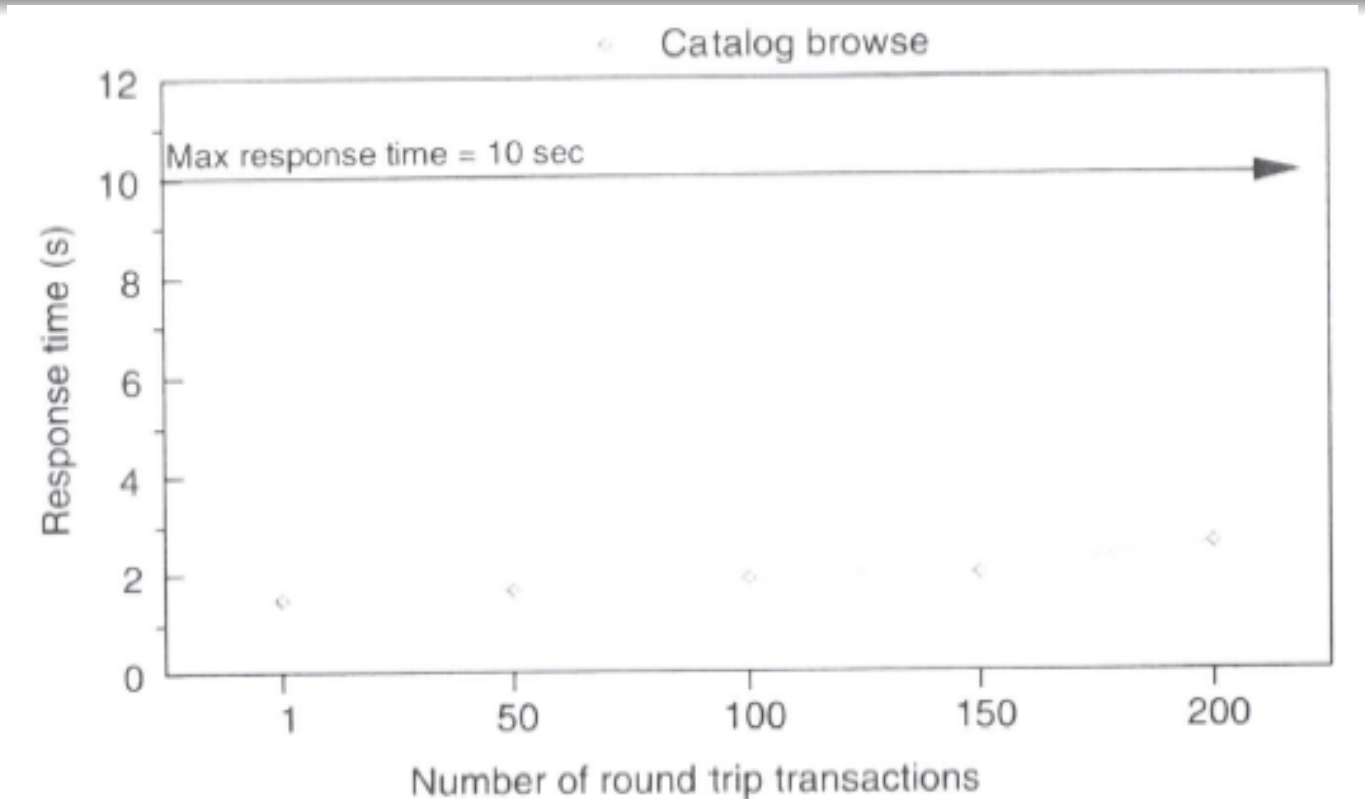


Figura: Pico de Trabajo - Transacciones de búsqueda en catálogo

Caso de Estudio: "*Compras del Sábado*"

Gráficas del t.respuesta para cada transacción individual

- Por simplicidad, se estudian sólo 3 grupos de transacciones (*conexión*, *búsqueda en catálogo*, *compra*)
- Se elije la planificación de carga de trabajo de las *compras de los sábados* porque necesitan simular menos transacciones (500) hasta el pico de carga que la de la planificación de los días laborables (2000 transacciones)
- Se encontró que los tiempos de respuesta de las 3 transacciones consideradas individualmente están bastante por debajo de máximo requerido para el pico de carga de los sábados de cada transacción

Caso de Estudio: "*Compras del Sábado*"

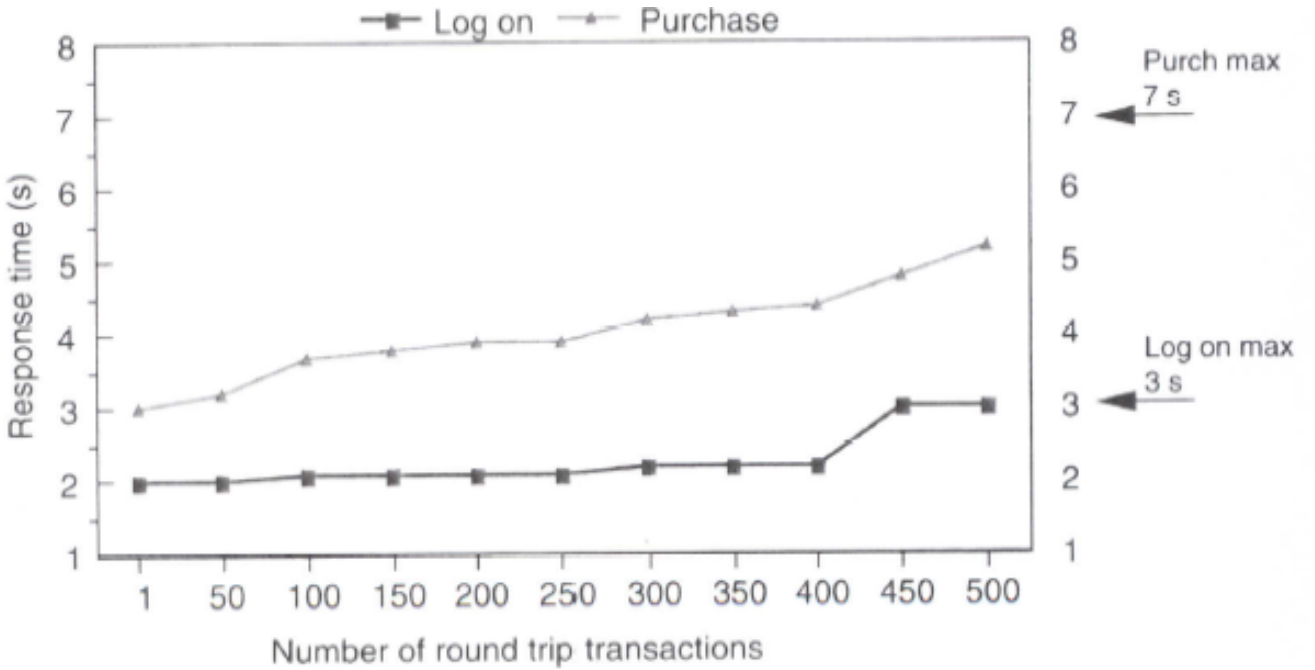


Figura: Pico de Trabajo - Transacciones de conexión+compra

Caso de Estudio: "*Compras del Sábado*"

Gráficas del t.respuesta para mezclas de transacciones

- Las mezclas de las transacciones han de hacerse gradualmente (*conexión+ compra*, *conexión+ compra+ búsqueda*) para no introducir indeterminación respecto de la que causa la pérdida de rendimiento
- El tiempo de respuesta de la transacción de conexión se convierte en un elemento marginal para el cálculo del rendimiento si se superan las 400 transacciones activas
- Por consiguiente, existe un conflicto en el acceso a recursos entre la transacción de compra y la de búsqueda

Caso de Estudio: "*Compras del Sábado*"

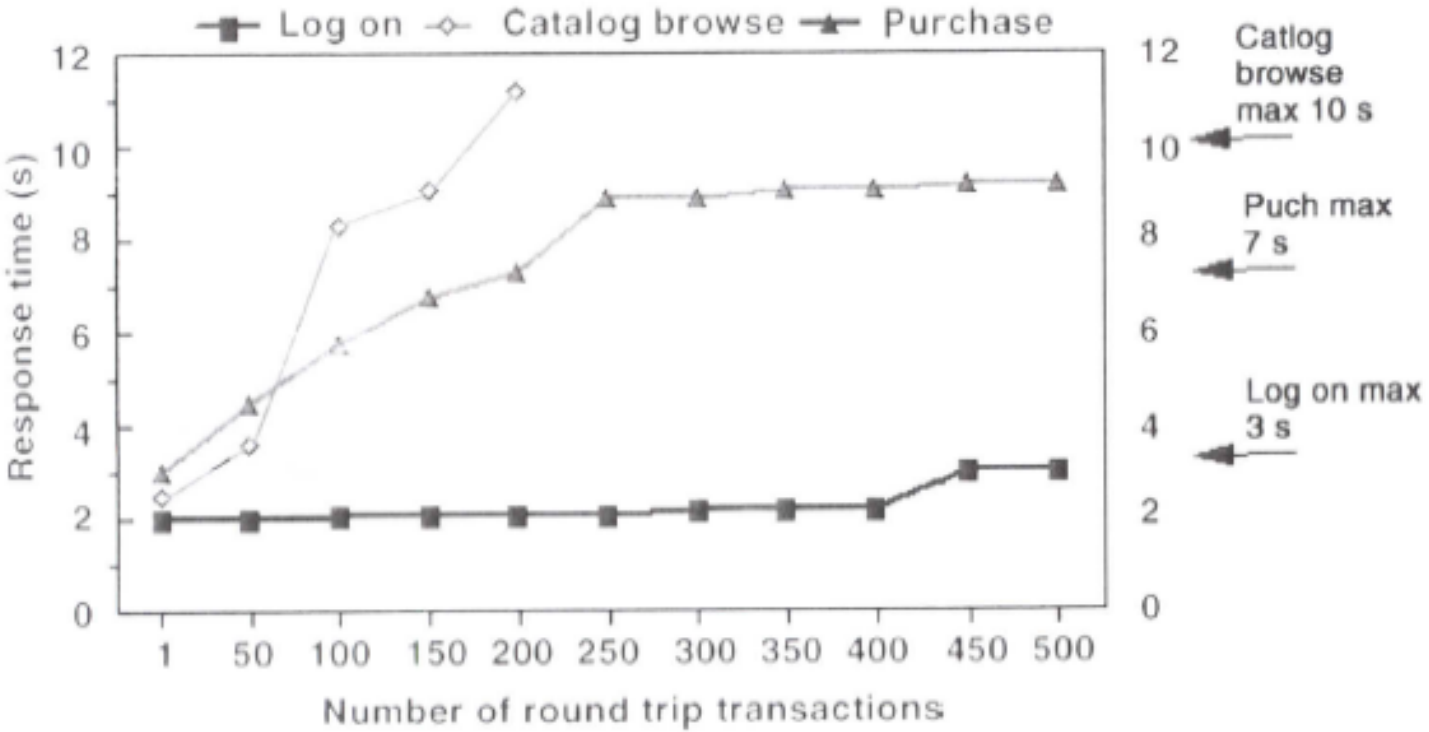


Figura: Transacciones de conexión+compra+búsqueda

Caso de Estudio: "*Compras del Sábado*"

Gráficas del t.respuesta para mezclas de transacciones (2)

- Para la búsqueda en catálogo por encima de las 150 transacciones, la gráfica que contempla la mezcla de las 3 transacciones (conexión+compra+búsqueda) muestra un tiempo de respuesta inaceptablemente alto
- Se deben detener las pruebas de rendimiento (situación denominada "*showstopper*") y devolver el software a equipo de desarrollo

Caso de Estudio: "*Compras del Sábado*"

Gráficas para cada transacción individual *corregida*

- Se recibió el software de la aplicación corregido y con las pruebas funcionales de regresión realizadas:
  - El módulo de *conexión* de los usuarios interfería con el módulo de compra por motivo de pérdida de memoria asignada al primero, que le retiraba el segundo módulo a partir de 400 transacciones activas
  - Degradación del rendimiento de una librería dinámica compartida entre el módulo de compra y el de búsqueda en catálogo cuando se cargan en memoria rutinas de utilidades para su ejecución

Caso de Estudio: "*Compras del Sábado*"

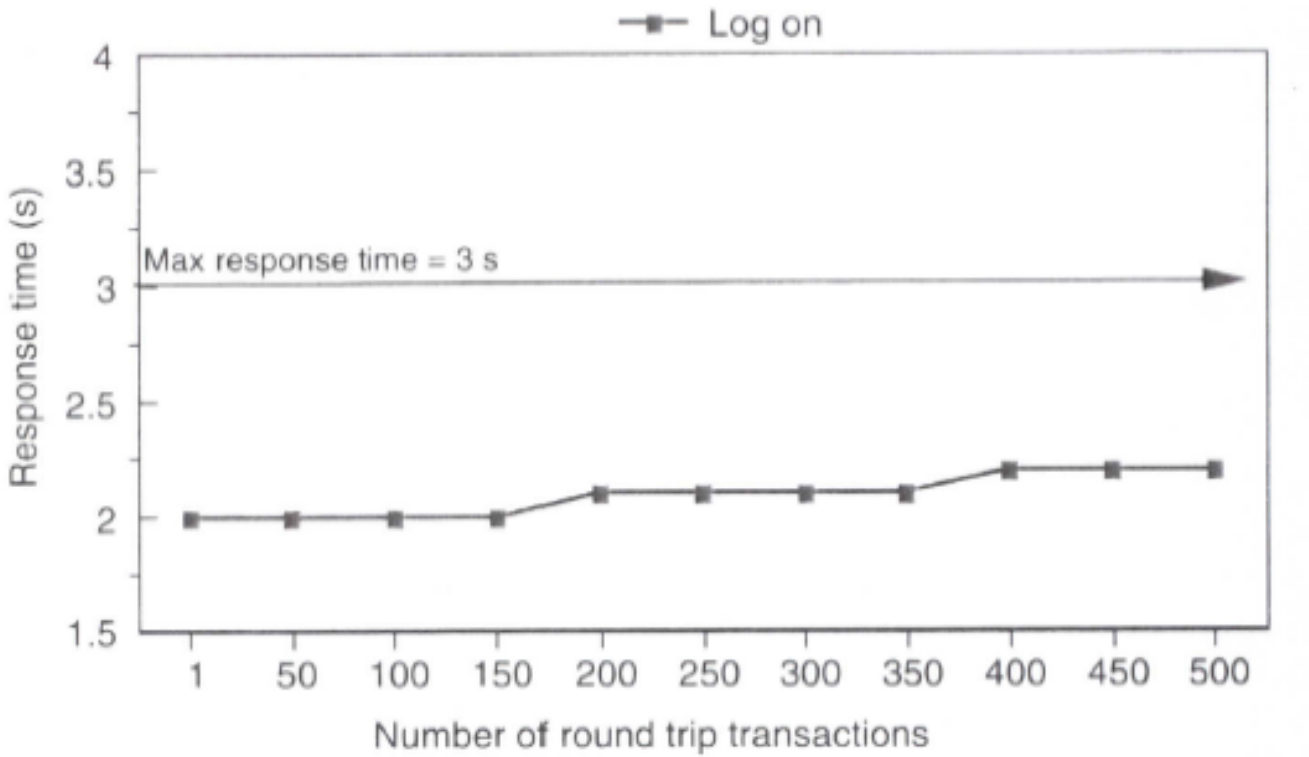


Figura: Transacciones de conexión después de correcciones

Caso de Estudio: "*Compras del Sábado*"

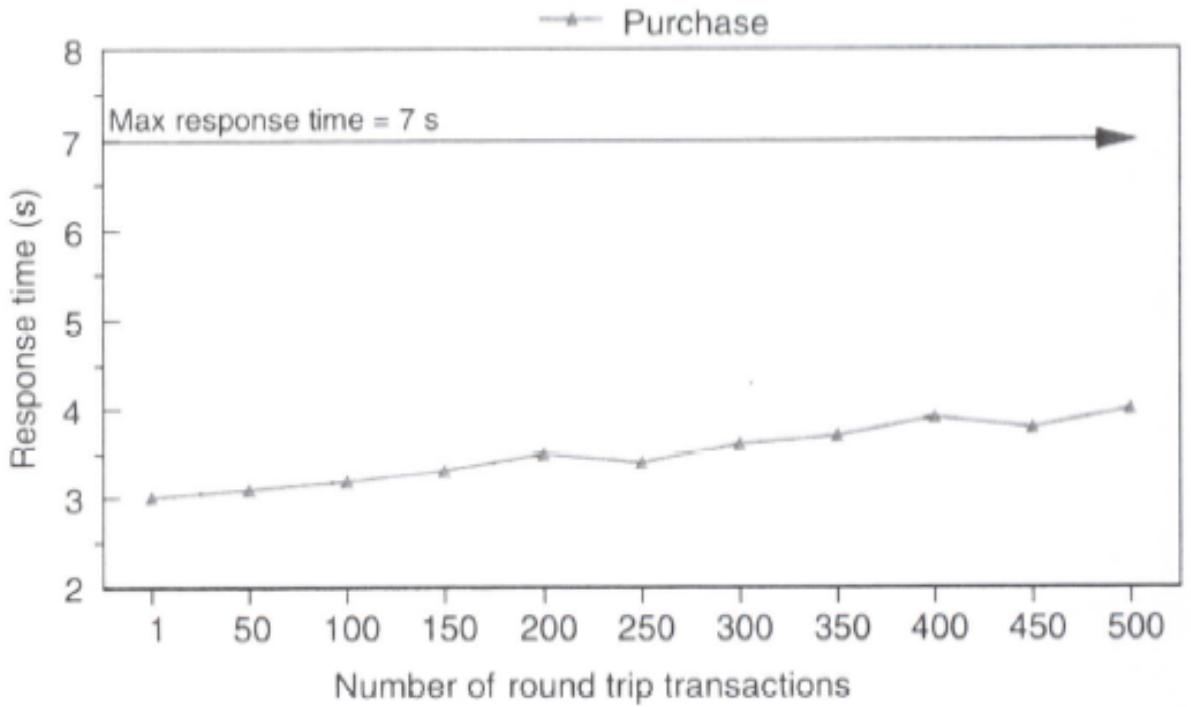


Figura: Transacciones de compra después de correcciones

Caso de Estudio: "*Compras del Sábado*"

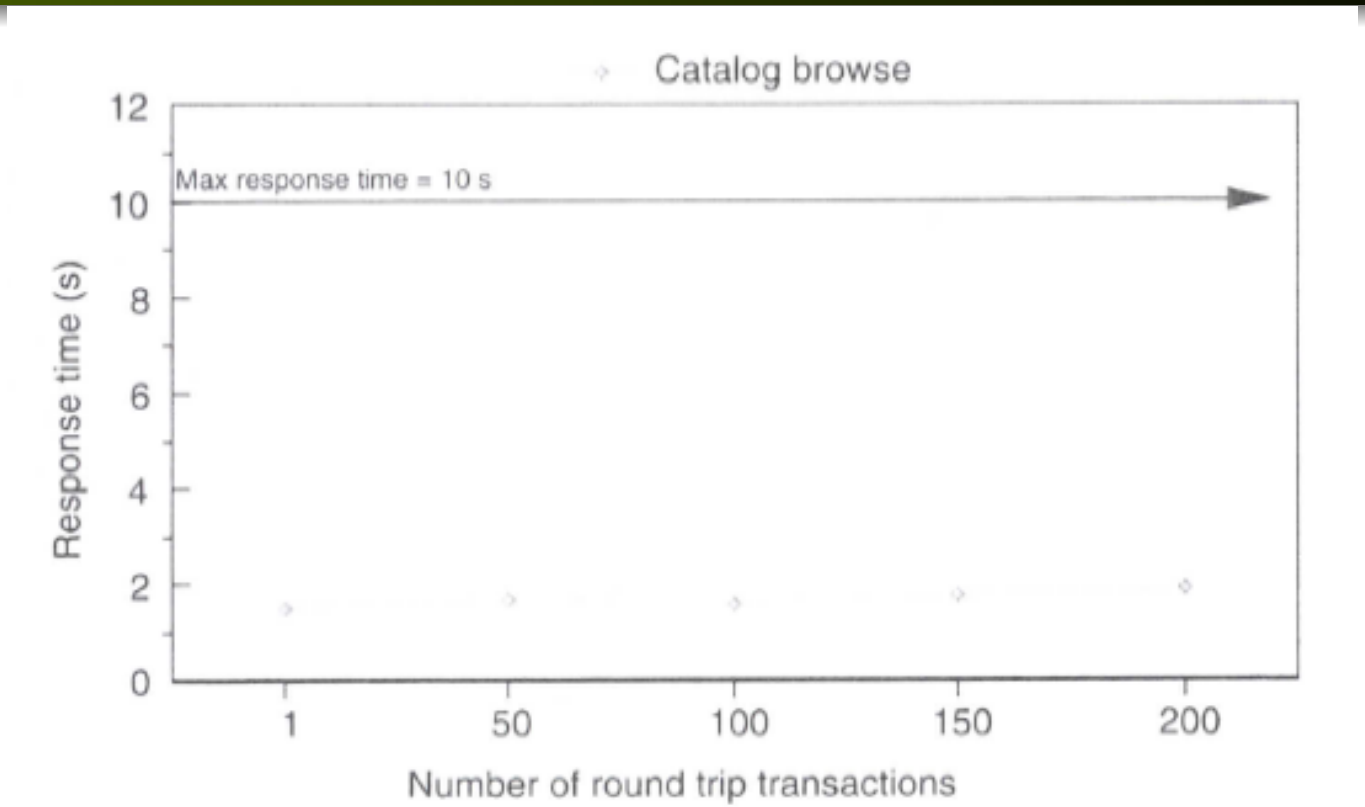


Figura: Transacciones de búsqueda en catálogo después de correcciones

Caso de Estudio: "*Compras del Sábado*"

Gráficas del para cada transacción individual *corregida*

- El rendimiento del código del módulo de conexión no se ha visto afectado por los cambios
- Los módulos de compra y búsqueda en catálogo muestran un ligero aumento de su rendimiento para cada transacción individual
- Aunque los cambios del código se centraron en corregir la degradación del rendimiento debido a una mala implementación de librerías compartidas, también mejoró el comportamiento individual de algunos módulos

Caso de Estudio: "*Compras del Sábado*"

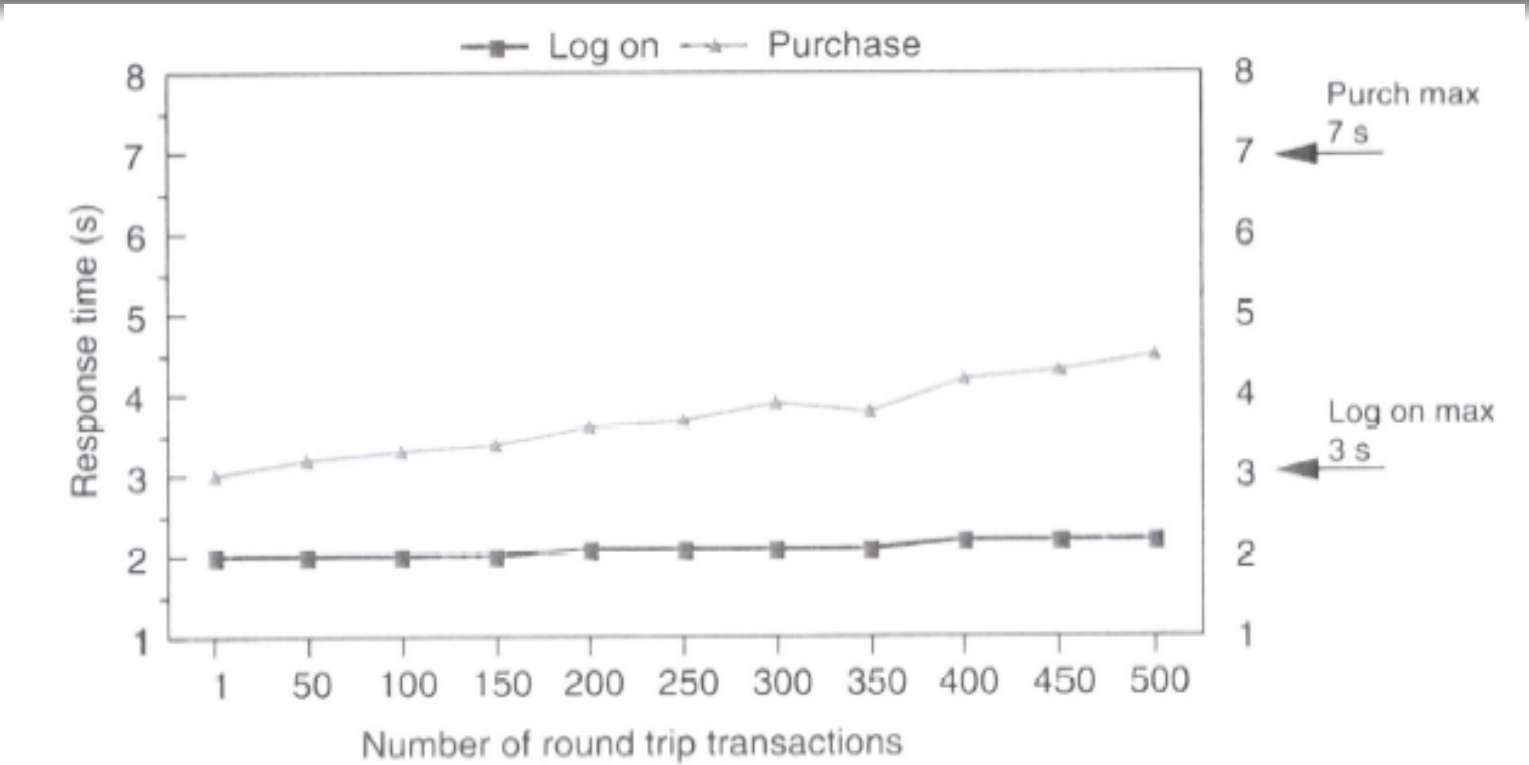


Figura: Transacciones de conexión+búsqueda después de correcciones

Caso de Estudio: "*Compras del Sábado*"

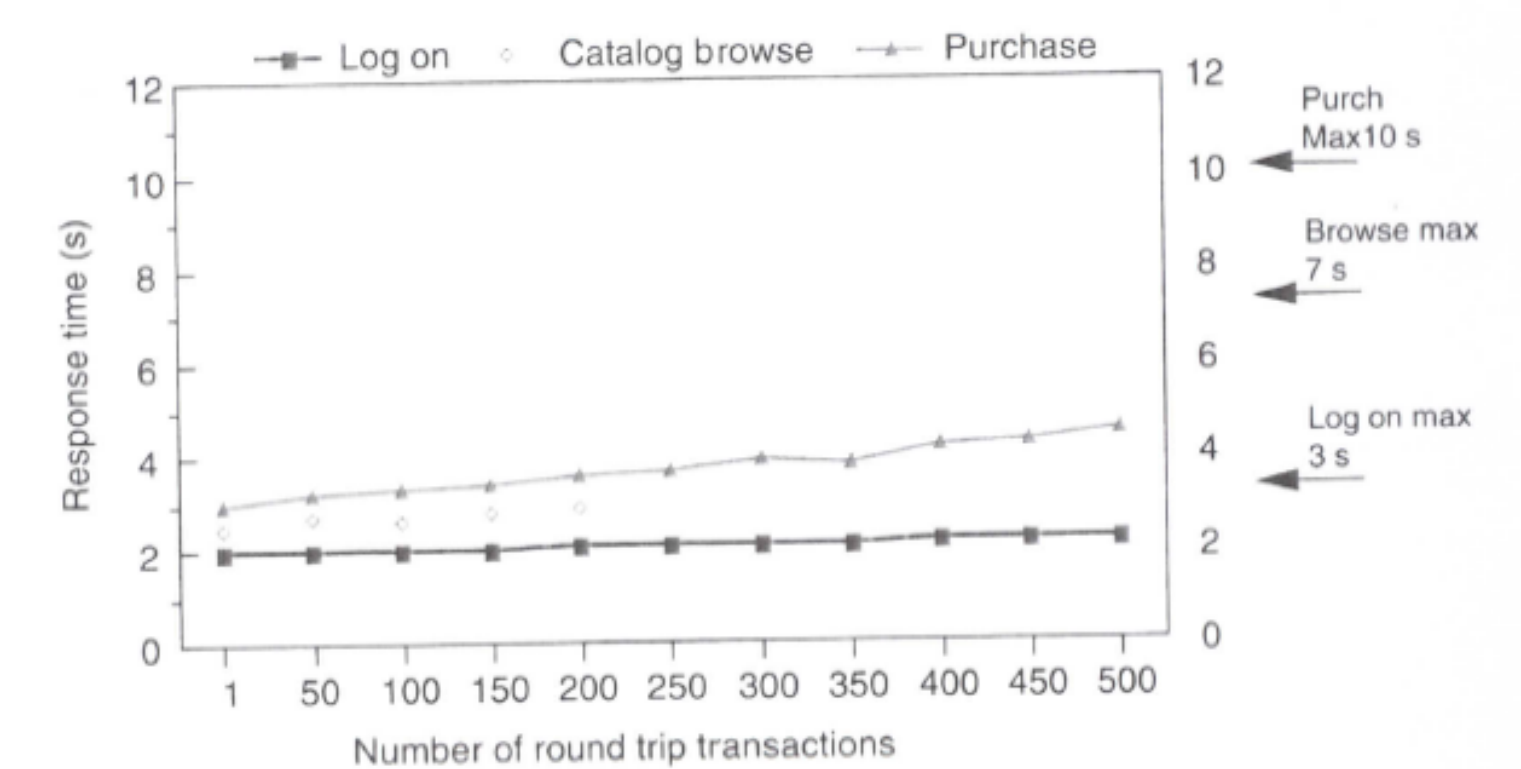


Figura: Transacciones de conexión+búsqueda+compra después de correcciones

Caso de Estudio: "'*Compras del Sábado*"

Gráficas del t.respuesta para mezclas corregidas

- Se resolvió el problema de la interferencia con la compra (cuando existen $>$ 400 transacciones-compra activas)
- Ambos grupos de transacciones (*conexión* y *compra*) producen un peor tiempo de respuesta muy por debajo del máximo tiempo en pico (en los requerimientos)
- Añadir *búsqueda* en catálogo a las anteriores no afecta ahora al rendimiento de ninguna trasacción cuando compiten por recursos
- Los tests de rendimiento han de formar parte de las pruebas de regresión para demostrar que la inclusión de nueva funcionalidad no degrada el rendimiento global

Software Orientado a Objetos

Cómo probarlo

- ¡La naturaleza del software OO modifica la *estrategia* y la *táctica* de la prueba del software!
- Cambia el concepto de *unidad* en la pruebas unitarias de componentes software
- Ya no se pueden probar una sola operación aisladamente, sino como parte de una clase
- La prueba de clases en el software OO es el equivalente de la prueba unitaria en el software convencional
- La prueba de clases en software OO es conducida por sus operaciones encapsuladas y el comportamiento del estado

Diseño de tests para pruebas de software OO

Características de los métodos de prueba

- Las pruebas unitarias se aplican clase por clase
- Se han de ejercitar todas las operaciones (*métodos*) que encapsula la clase
- Se diseñan *secuencias de tests* para asegurar que las operaciones importantes son ejercitadas
- Se ha de examinar el estado de cada clase (representado por los valores de sus atributos) para determinar si existe algún error todavía no identificado

Diseño de tests para pruebas de software OO (II)

Desafíos

- La encapsulación propia de las clases puede hacer difícil el extraer el estado concreto de cualquier objeto durante un momento de la ejecución
- El mecanismo de herencia puede complicar la realización de las pruebas de una clase
- La herencia múltiple complicaría aún más las pruebas, ya que incrementa el número de contextos de utilización

Diseño de tests para pruebas de software OO (III)

Lista común de pasos de los tests

1. Los estados de la clase que se va a probar
2. Los mensajes y las operaciones que serán ejercitados como consecuencia de aplicar el test
3. Las excepciones que pueden suceder cuando la clase esté siendo probada
4. Las condiciones externas
5. Información suplementaria, que ayudará a entender o implementar el test que se está diseñando

Diseño de tests para pruebas de software OO (IV)

Categorias de métodos de prueba

- Pruebas basadas en fallos
- Pruebas aleatorias
- Pruebas de partición

Pruebas basadas en fallos

Características

- Partiendo del modelo de análisis OO, el comprobador trata de identificar posibles fallos
- Se diseñan casos de prueba para ejercitar el diseño del sistema o su codificación
- Se buscan posibles fallos en las llamadas a los métodos y las comunicaciones a través de mensajes:
  - llamadas a operaciones: hay que examinar su comportamiento
  - tipos de errores: *resultado inesperado*, *operación/mensaje incorrecto*, *invocación incorrecta*
  - Se trata de determinar si existen errores en el código que llama, no en el llamado

Prueba aleatoria para clases OO

Características

- Este tipo de prueba se utiliza para ejercitar una clase
- Existen restricciones en el orden de invocación de las operaciones de una clase, debido al problema
- Incluso con dichas restricciones, existen muchas posibles permutaciones de la secuencia de llamadas a operaciones
- Se generan aleatoriamente diferentes secuencias para ejercitar distintas *historias* de la *vida* de las instancias

Prueba aleatoria para clases OO (II)

Ejemplo

`class CuentaCorriente:`

- métodos:{`abrir()`, `iniciar()`, `ingresar()`, `retirar()`, `saldo()`, `movimientos()`, `limiteCredito()`, `cerrar()`}
- atributos:{`saldo`, `limiteCredito`}
- Historia mínima: `abrir()` · `iniciar()` · `ingresar()` · `retirar()` · `cerrar()`
- Todos los comportamientos legales: `abrir()` · `iniciar()` · `ingresar()` · [`ingresar()`|`retirar()`|`saldo()`|`movimientos()`|`limiteCredito`]^n · `retirar()` · `cerrrar()`
- Test 1: `abrir()` · `iniciar()` · `ingresar()` · `ingresar()` · `saldo()` · `movimientos()` · `retirar()` · `cerrar()`
- Test 2: `abrir()` · `iniciar()` · `ingresar()` · `retirar()` · `ingresar()` · `saldo()` · `limiteCredito()` · `retirar()` · `cerrar()`
- ...

Prueba de Partición

Características

- Respecto de la prueba aleatoria, reduce el número de tests necesarios para realizar la prueba de una clase
- Se categorizan las entradas y las salidas de las operaciones y se diseñan tests para ejercitarlas
- *Particionamiento por estados*: clasifica las operaciones según su habilidad para cambiar el estado de la clase
- *Particionamiento basado en los atributos*: clasifica las operaciones según los atributos de la clase que utilizan
- *Particionamiento basado en categorías*: clasifica las operaciones de la clase según las funciones genéricas

Prueba de Partición para clases OO (II)

Ejemplo

`class CuentaCorriente`:

- Partición por estados: {{`ingresar()`, `retirar()`}}, {{`saldo()`, `movimientos()`, `limiteCredito()`}}
  - Test 1: `abrir()` · `iniciar()` · `ingresar()` · `ingresar` · `retirar()` · `retirar()` `saldo()` · `cerrar()`
  - Test 2: `abrir()` · `iniciar()` · `depositar()` · `movimientos()` · `limiteCredito()` · `saldo()` · `retirar()` · `cerrar()`
- Partición por atributos:
  - Operaciones que usan `limiteCredito`
  - Operaciones que usan `saldo`
  - Operaciones que no usan o modifican los atributos anteriores

Prueba de Partición para clases OO (III)

Ejemplo

`class CuentaCorriente`:

- Partición por categorías:
  - operaciones de inicialización:{`abrir`(), `iniciar`()}
  - operaciones computacionales:{`ingresar`(), `retirar`()}
  - consultas:{`saldo`(), `movimientos`(), `limiteCredito`()}
  - operaciones de terminación:{`cerrar`()}
- Construir tests como secuencias distintas que incluyan operaciones de cada categoría

Pruebas de Integración

Antecedentes

- Dado que el software OO no posee una estructura de control jerárquica obvia, las estrategias de integración top–down y bottom–up tradicionales fracasan
- El integrar operaciones de una en una en una clase es, a menudo, imposible

Pruebas de Integración II

Estrategias

- El integrar operaciones de una en una, en una sola clase, es, a menudo, imposible
- (1) Pruebas *basadas en hebras*: cada hebra de la aplicación es probada individualmente
- Hay que aplicar la prueba de regresión (muy importante)
- (2) Pruebas *basadas en la utilización*
- Después de probar las clases *independientes*, se prueba la siguiente capa de clases
- La secuencia de prueba de capas de clases dependientes continúa hasta se construye el sistema completo

Pruebas de Integración III

Estrategias

- Se suelen utilizar *drivers* y *stubs* en las pruebas del software OO: permiten probar la funcionalidad del sistema antes de la implementación
- (3) *Cluster Testing*: se intenta descubrir los posibles errores en las colaboraciones entre clases

Pruebas de Integración IV

Pruebas de múltiples clases

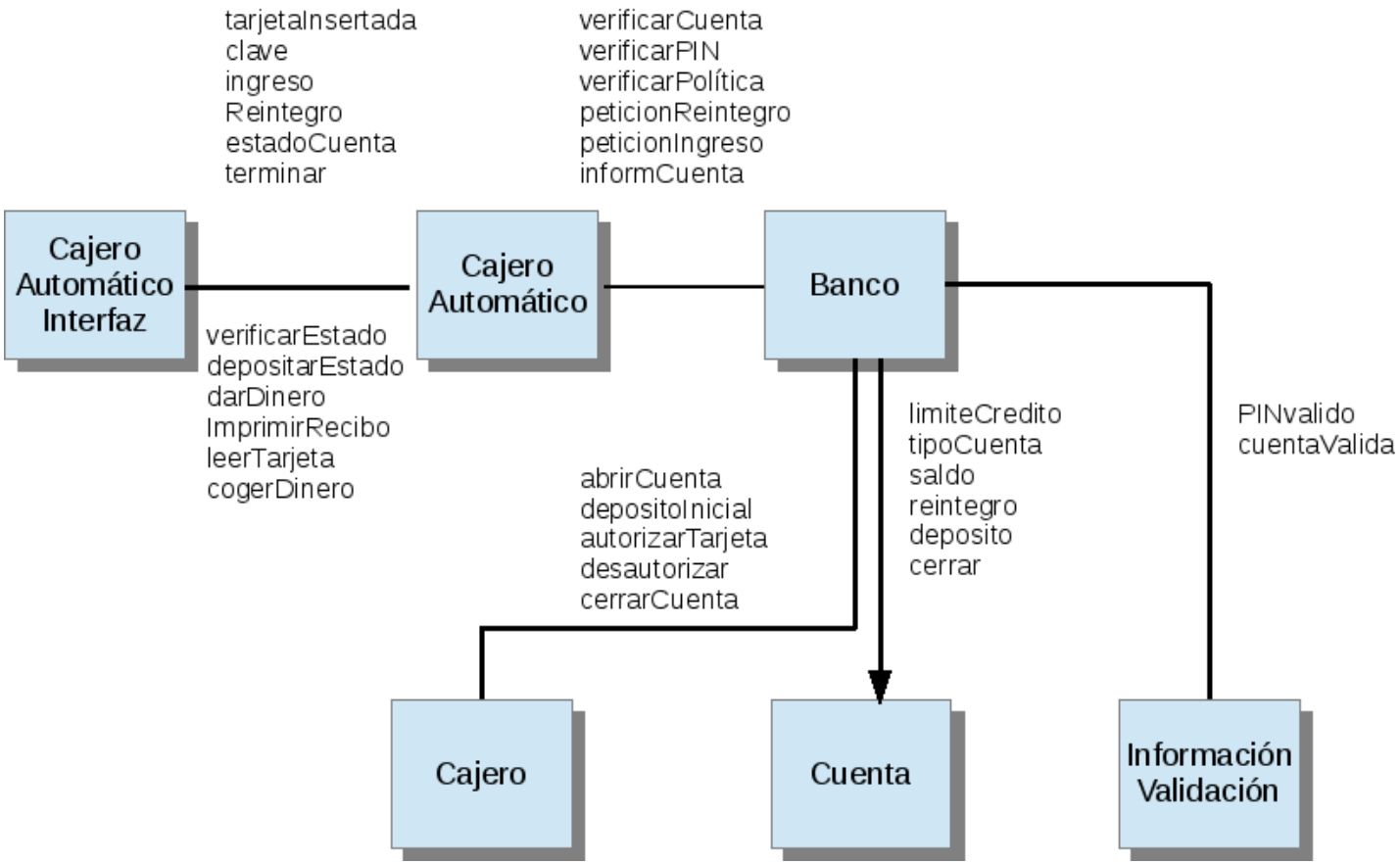
1. Para cada clase cliente: usar la lista de operaciones para generar secuencias de prueba aleatorias (las operaciones enviarán mensajes a otras clases)
2. Para cada mensaje: determinar la clase colaboradora y la operación correspondiente en el objeto servidor
3. Para cada operación en el objeto servidor: determinar los mensajes que transmite
4. Para cada uno de estos mensajes: determinar el siguiente nivel de operaciones que son invocadas e incluirlas en la secuencia inicial de prueba

Pruebas para múltiples clases

Ejemplo

- La prueba de partición para múltiples clases es similar a la de 1 clase
- Ahora, la secuencia es ampliada para que incluya las operaciones que se activan en las clases colaboradoras, después de recibir los mensajes:
  - secuencia para*Banco* : *verificarCuenta*() · *verificarPIN*() · [[*verificarPolitica*() · *peticionReintegro*()]|*peticionIngreso*()|*inforCuenta*()]
  - Una secuencia aleatoria:Test 3: *verificarCuenta*() · *verificarPIN*() · *peticionIngreso*()
  - Secuencia con colaboraciones:Test 4: *verificarCuenta*() · [*Banco* : *InformacionValidacion*.*inforCuenta*()] ·*verificarPIN*()[*Banco* : *InformacionValidacion*.*validarPIN*()] ·*peticionIngreso*()[*Banco* : *Cuenta*.*deposito*()]

Diagrama de Colaboración de Clases

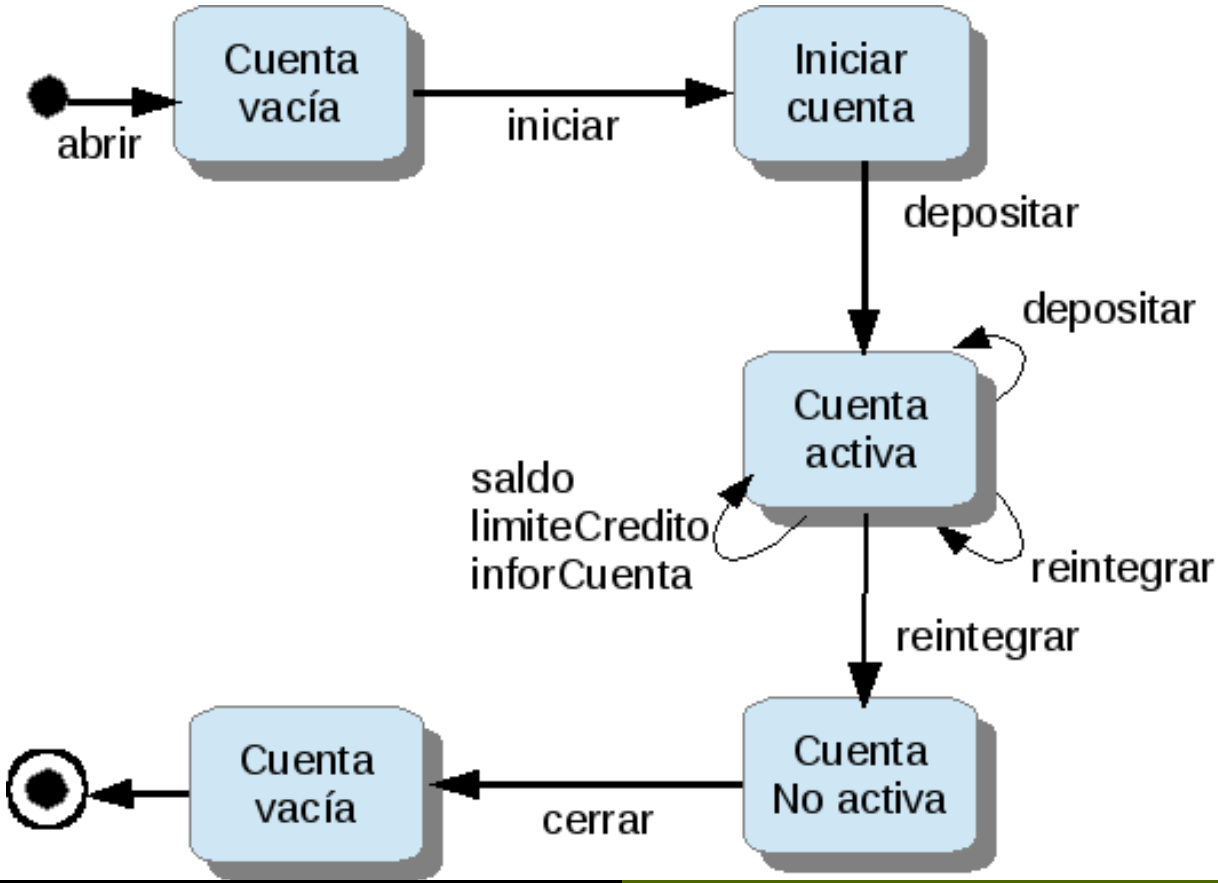


Pruebas derivadas de los modelos–UML de comportamiento

Diagramas Estados–UML

- Se utilizan para representar el comportamiento dinámico de una clase
- Ahora, lo usamos para derivar una secuencia de pruebas que ejercite el comportamiento dinámico de la clase y sus colaboradoras
- Los tests diseñados han de incluir todos los estados del diagrama–UML
- El modelo de estados puede ser atravesado de forma *breadth–first*

Diagrama de Estados para la Clase `CuentaCorriente`

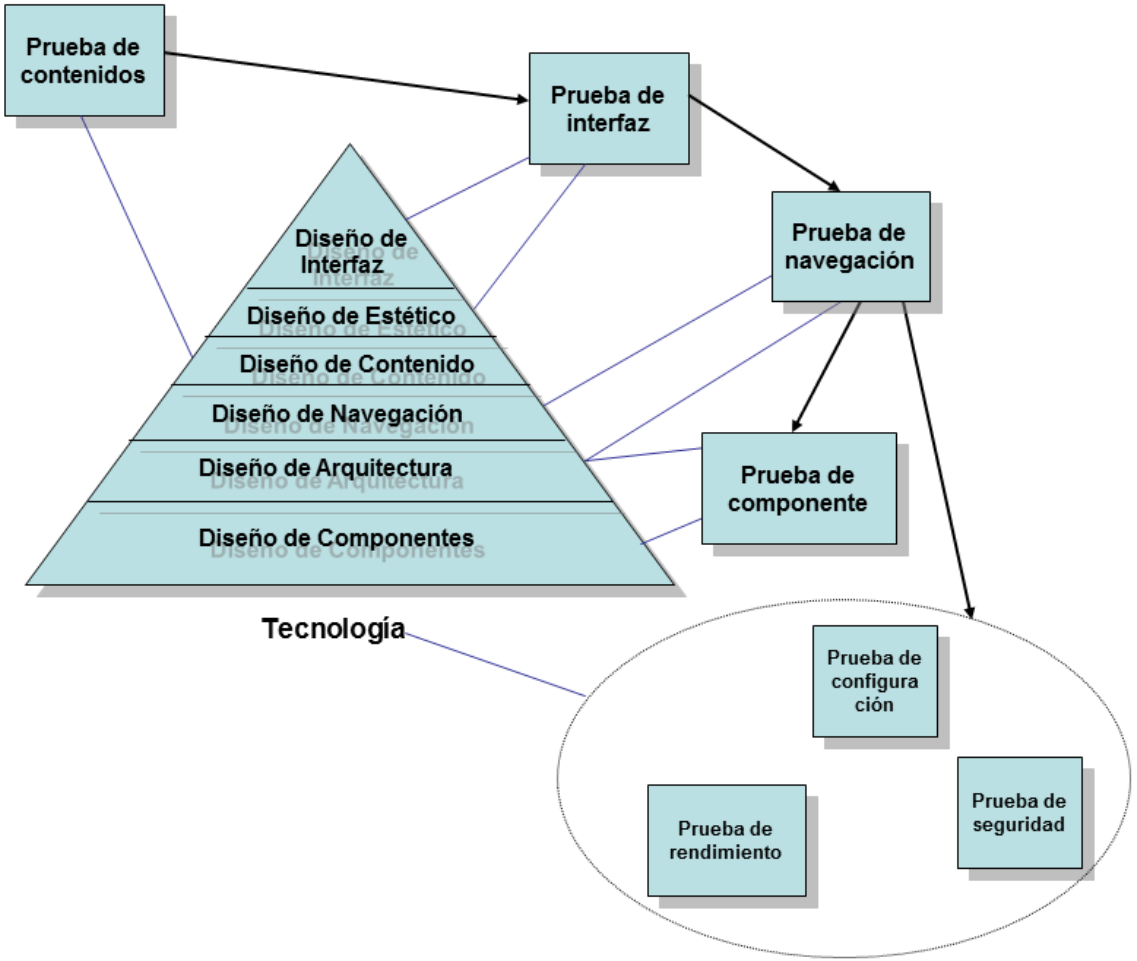


Pruebas para Aplicaciones Web

Antecedentes

- Se siguen los principios básicos y los pasos de la pruebas de software en general: unitarias, integración, regresión, etc.
- Se aplican las tácticas de la prueba de software OO
- Lo que se quiere llegar a probar:
  - Exclusión de errores de navegación
  - Preocupación por la *usabilidad*
  - Compatibilidad con diferentes plataformas y configuraciones
  - Confiabilidad (= seguridad+robustez+fiabilidad+disponibilidad)
  - Rendimiento

Pruebas para Aplicaciones Web (II)



Pruebas para Aplicaciones Web III

Pasos del Proceso de Prueba

1. Revisar el modelo de contenidos
2. Comprobar que todos los casos de uso son tratados en el modelo de interfaz
3. Descubrir cualquier error de navegación en el modelo de diseño
4. Exclusión de errores de la mecánica de navegación o de la presentación en la interfaz de usuario
5. Prueba unitaria de cada componente fucnional
6. Navegación a través de toda la arquitectura software de la aplicación
7. Tests de compatibilidad con plataformas y configuraciones
8. Pruebas de seguridad, exclusión de vulnerabilidades de la aplicación o de su entorno
9. Tests de rendimiento

Pruebas de Contenidos

Caracteríticas

- Combina las revisiones con la generación de pruebas ejecutables
- Objetivos:
  - Descubrir errores sintácticos
  - Errores semánticos
  - Errores en la organización de la página o en la estructura de los contenidos que se presentan al usuario

Pruebas de Contenidos II

Semántica

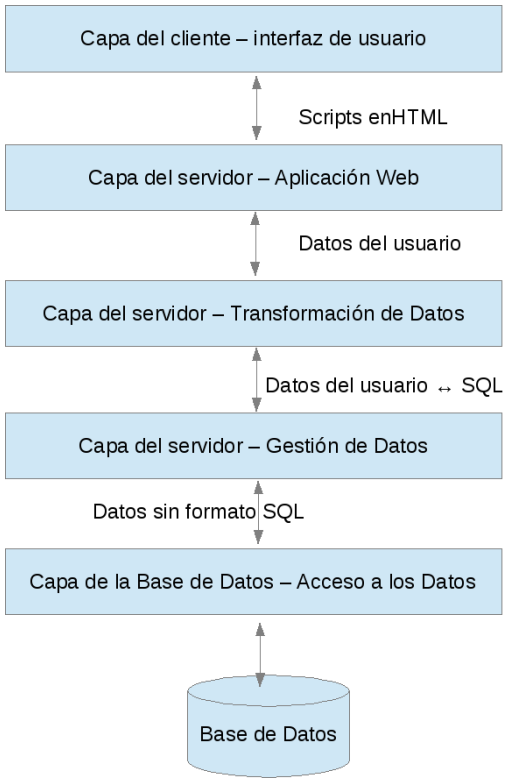
- ¿La información que se presenta es precisa?
- ¿Es el diseño de los contenidos fácil de comprender?
- ¿La información embebida en el contenido es fácil de encontrar?
- ¿Las referencias y fuentes son las apropiadas?
- ¿Infringe algún (c) o marcas registradas?
- ¿Contiene enlaces internos que complementan los contenidos?
- ¿Coordina estéticamente el estilo del contenido con el resto de la interfaz?

Interfaz entre Aplicación Web–SGBD

Características

- Descubrir errores producidos en la traducción de las peticiones del usuario al lenguaje de consulta del SGBD
- Identificar errores de comunicación entre la aplicación–Web y la BD remota
- Demostrar la validez de los datos sin formato recibidos por el servidor de la aplicación Web
- Los contenidos dinámicos han de ser transmitidos al cliente de una forma que puedan ser mostrados (comprobar compatibilidad con diferentes plataformas y configuraciones del usuario)

Capas de Interacción entre Aplicación y Base de Datos



Prueba de la Interfaz de Usuario

Ubicación en el ciclo de vida

- Durante el Análisis de Requerimientos
- Durante el Diseño
- Durante la Prueba del Software
- Se trataría de descubrir errores relacionados con mecanismos específicos de la interfaz
- y los errores relacionados con la semántica de la navegación, funcionalidad de la aplicación Web o contenidos mostrados

Prueba de la Interfaz de Usuario (II)

Actividades

- Probar mecanismos de la Interfaz
- Comprobación de *cookies* en el lado del servidor y del cliente
- Prueba de la Semántica de la Interfaz
- Pruebas de Usabilidad
- Pruebas de Compatibilidad

Pruebas a Nivel de Componentes

Concepto

- También se llama prueba de funcionalidad: un conjunto de tests que intentan descubrir errores en las funciones de la aplicación
- Cada función es un componente software y puede ser probado utilizando técnicas de caja negra
- Se pueden automatizar utilizando formularios

Pruebas a Nivel de Componentes (II)

Métodos de diseño de pruebas concretas

- Particionamiento en equivalencias
- Análisis de valores frontera
- Prueba de caminos (técnica de *caja blanca*)
- Prueba de *error forzado*, para descubrir errores que ocurren durante el tratamiento de errores
- Cada prueba específica de este tipo especifica todos los valores de entrada y la salida esperada que el componente ha de proporcionar

Pruebas de Navegabilidad

Antecedentes

- Se trata de asegurar que los mecanismos que permiten al usuario navegar a través de la aplicación Web son todos funcionales
- Validar que cada *unidad semántica de navegación* (NSU) puede ser alcanzada por la categoría adecuada de usuarios
- Se hacen pruebas de *Sintaxis de Navegación* y de su Semántica

Pruebas de Sintaxis

- Enlaces de navegación
- Redirecciones
- Marcas de libro (*bookmarks*)
- Frames y framesets
- Tabla de contenidos de sitio
- Motores de búsquedas internos

Pruebas de Semántica

Definición de NSU

- Un conjunto de información y estructuras de navegación relacionadas que colaboran para que se puedan satisfacer un subconjunto relacionado de requerimientos del usuario

Pruebas de Semántica II

- La prueba de navegabilidad ejercita cada NSU para asegurarse que los requerimientos se alcanzan:
  - ¿Todos los nodos de navegación se alcanzan desde el contexto definido para un NSU?
  - ¿Todos los caminos importantes hacia/desde el NSU se han probado?
  - ¿Funcionan adecuadamente los mecanismos de navegación dentro de los nodos de navegación grandes?
  - ¿Son los NSU tolerantes a fallos y errores?

Pruebas de Configuración

Tipos de pruebas

- Pruebas específicas de configuración en el lado del servidor para probar que puede soportar a la aplicación Web sin errores ni pérdida inaceptable de rendimiento
- Pruebas en el lado del cliente: compatibilidad con configuraciones que contienen los siguientes componentes:
  - hardware
  - Sistemas operativos
  - Navegadores
  - Componentes (activos) en la intefaz de usuario
  - Plugings
  - Connectividad

Pruebas de Seguridad

Tipos de amenazas

- Desde el cliente: errores en navegadores, correo electrónico, software de comunicaciones, acceso a cookies, *spoofing*
- Desde el servidor: ataques de denegación de servicio, scripts maliciosos, robo de datos

Elementos de Seguridad

- Firewalls
- Autenticación de todos los clientes y servidores
- Encriptación (y certificados digitales)
- Autorización
- Las pruebas de seguridad ha de diseñarse para probar que todas las tecnologías anteriores funcionan; se intenta descubrir agujeros de seguridad

Pruebas de Rendimiento

Problemática

Se utilizan para descubrir problemas que pueden llevar a la falta de recursos del lado del servidor, ancho de banda inadecuado, capacidades de bases de datos mal dimensionadas, funcionalidad pobre de la aplicación Web, problemas con el sistema operativo, etc.

Pruebas de Rendimiento II

Objetivos

1. Comprender cómo responde el sistema a una situación de *carga* creciente: número de usuarios, transacciones, excesivo volumen de datos
2. Reunir métricas que nos ayuden a diseñar modificaciones para mejorar el rendimiento

Prueba de Carga

Concepto

- Examina la carga que puede experimentar el sistema en el mundo real, a varios niveles y en una variedad de combinaciones
- Prueba de *Stress*: fuerza el incremento de la carga hasta el punto de *ruptura* del sistema para determinar la capacidad máxima que puede aguantar

Prueba de Carga II

Conjunto de condiciones del test:

- N: número de usuarios concurrentes de la aplicación–Web
- T: número dede transacciones en línea por unidad de tiempo
- D: datos procesados por el servidor por transacción

Prueba instantánea

Determinar si una combinación de las tres medidas (N, T, D) puede asociarse a un decrecimiento importante del rendimiento en un momento:

$$Throughput = N \times T \times D \quad (1)$$

Prueba de Stress

Condiciones

Las variables N, T, D de la *prueba de carga* son obligadas a llegar a los límites operacionales del sistema y, después, a excederlos

Prueba de Stress (II)

Cosas a Determinar

- ¿Se degrada el sistema suavemente?
- ¿El software del sistema genera mensajes “server not available"?
- ¿Almacena peticiones de los usuarios y deja vacía la cola una vez que se supera el ímite?
- ¿Se pierden transacciones?
- ¿Qué valores de N, T, D ocasionan que el servidor falle?
- ¿La integridad de los datos se ve afectada?
- ¿Cuando tardará el sistema en recuperarse?

Para ampliar


Black (2007).
Pragmatic Software Testing.
Wiley.


Everett and Raymond (2007).
Software Testing.
Wiley.


Lewis (2004).
Software Testing and Continuous Quality Improvement.
Auerbach.


Perry (2005).
Effective Methods for Software Testing.
Wiley.


Pressman, R. (2010).
Software engineering: a practitioners approach.
McGraw-Hill.


Spiller (2007).
Softwre Testing Process: Test Management.
Rocky Nook.