Prueba de aplicaciones orientadas a objetos

Una mirada rápida

¿Quién lo hace?

¿Por qué es importante?

¿Cuáles son los pasos?

¿Cuál es el producto final?

¿Cómo me aseguro de que lo hice bien?

Ampliación de la definición de pruebas

La construcción de software orientado a objetos comienza con la creación de modelos de requerimientos (análisis) y de diseño. Debido a la naturaleza evolutiva del paradigma de ingeniería del software OO, dichos modelos comienzan como representaciones relativamente informales de los requisitos de sistema y evolucionan hacia modelos detallados de clases, relaciones de clase, diseño y asignación de sistema, y diseño de objetos. En cada etapa, los modelos pueden "probarse" con la intención de descubrir errores previamente a su propagación hacia la siguiente iteración.

Puede argumentarse que la revisión de los modelos de análisis y de diseño OO es especialmente útil, pues los mismos constructos semánticos (por ejemplo, clases, atributos, operaciones, mensajes) aparecen en los niveles de análisis, diseño y código. Por tanto, un problema en la definición de los atributos de clase que se descubra durante el análisis soslayará los efectos colaterales que puedan ocurrir si el problema no se descubriera hasta el diseño o el código (o incluso en la siguiente iteración de análisis).

Modelos de prueba AOO y DOO

- Exactitud de los modelos AOO y DOO
- Consistencia de los modelos orientados a objetos

Exactitud de los modelos AOO y DOO

La notación y la sintaxis utilizadas para representar los modelos de análisis y diseño se ligarán a los métodos de análisis y diseño específicos que se elijan para el proyecto. Por tanto, la exactitud sintáctica se juzga mediante el uso adecuado de la simbología; cada modelo se revisa para garantizar que se mantienen las convenciones de modelado adecuadas.

Durante el análisis y el diseño, la exactitud semántica puede valorarse con base en la conformidad del modelo con el dominio de problemas del mundo real. Si el modelo refleja con precisión el mundo real , entonces es semánticamente correcto. Para determinar si el modelo verdaderamente refleja los requerimientos del mundo real, debe presentarse a expertos de dominio de problemas, quienes examinarán las definiciones y jerarquía de clase en busca de omisiones y ambigüedad. Las relaciones de clase se evalúan para determinar si reflejan con precisión conexiones de objetos en el mundo real.

Consistencia de los modelos OO

La consistencia de los modelos orientados a objetos puede juzgarse al “considerar las relaciones entre entidades en el modelo”.

Para valorar la consistencia, debe examinarse cada clase y sus conexiones con otras clase.

Para evaluar el modelo de clase, se recomienda seguir los siguientes pasos :

(1) Vuelva a consultar el modelo CRC y el modelo objeto-relación.

(2) Inspeccione la descripción de cada tarjeta índice CRC para determinar si una responsabilidad delegada es parte de la definición del colaborador.

(3) Invertir la conexión para garantizar que cada colaborador al que se solicita servicio recibe solicitud de una fuente razonable.

(4) Al usar las conexiones invertidas que se examinaron en el paso 3, se determina si es posible requerir otras clases o si las responsabilidades se agrupan de manera adecuada entre las clases.

(5) Determinar si las responsabilidades de amplia solicitud pueden combinarse en una sola responsabilidad.

CRC: clase-responsabilidad-colaboración

Estrategias de pruebas orientadas a objetos

La estrategia clásica de prueba de software comienza “probando en lo pequeño” y funciona hacia afuera, “probando en lo grande”.

Prueba de unidad

Prueba de integración

Pruebas de validación y sistema

Prueba de unidad en el contexto OO

Esto significa que cada clase y cada instancia de una clase encapsulan los atributos y las operaciones que manipulan dichos datos. En lugar de probar un módulo individual, la unidad comprobable más pequeña es la clase encapsulada. Puesto que una clase puede contener algunas operaciones diferentes y una operación particular puede existir como parte de un número de clases diferentes, el significado de prueba de unidad cambia dramáticamente.

La prueba de clase para el software OO es el equivalente de la prueba de unidad para software convencional.

Prueba de integración en el contexto OO

En el software orientado a objetos no tiene una estructura de control jerárquica, las estrategias de integración tradicionales, descendente y ascendente, tienen poco significado.

Existen dos diferentes estrategias para la prueba de integración de los sistemas OO:

- prueba basada en hebra
- enfoque de integración

Prueba de validación en un contexto OO

En el nivel de validación o de sistema, desaparecen los detalles de las conexiones de clase. Como la validación convencional, la del software OO se enfoca en las acciones visibles para el usuario y en las salidas del sistema reconocibles por él mismo.

Los métodos convencionales de prueba de caja negra pueden usarse para activar pruebas de validación.

Métodos de prueba orientada a objetos

1. Cada caso de prueba debe identificarse de manera única y explícita asociado con la clase que se va a probar.
2. Debe establecerse el propósito de la prueba.
3. Debe desarrollarse una lista de pasos de prueba para cada una de ellas, que debe contener:
    1. Una lista de estados especificados para la clase que se probará
    2. Una lista de mensajes y operaciones que se probarán como consecuencia de la prueba
    3. Una lista de excepciones que pueden ocurrir conforme se prueba la clase
    4. Una lista de condiciones externas
    5. Información complementaria que ayudará a comprender o a implementar la prueba

Implicaciones del diseño de casos de prueba de los conceptos OO

Conforme una clase evoluciona a través de los modelos de requerimientos y diseño, se convierte en un blanco para el diseño de casos de prueba.

Como anota Binder: “las pruebas requieren reportar el estado concreto y abstracto de un objeto”

Aplicabilidad de los métodos convencionales de diseño de casos de prueba

Los métodos de prueba de caja blanca descritos en el tema anterior pueden aplicarse a las operaciones definidas para una clase. Las técnicas de ruta básica, prueba de bucle o flujo de datos pueden ayudar a garantizar que se probaron todos los enunciados en una operación. Sin embargo, la estructura concisa de muchas operaciones de clase hace que algunos argumenten que el esfuerzo aplicado a la prueba de caja blanca puede redirigirse mejor para probar en un nivel de clase.

Los métodos de prueba de caja negra son tan apropiados para los sistemas OO como para los sistemas desarrollados, usando métodos de ingeniería del software convencional. Como se observó en el tema anterior, los casos de uso pueden proporcionar entrada útil en el diseño de las pruebas de caja negra y en las basadas en estado.

Prueba basada en fallo

El objeto de la prueba basada en fallo dentro de un sistema OO es diseñar pruebas que tengan una alta probabilidad de descubrir fallos plausibles.

El examinador busca aspectos de la implementación del sistema que pueden resultar en defectos. Para determinar si existen dichos fallos, los casos de prueba se diseñan a fin de testear el código.

Si los fallos reales en un sistema OO se perciben como improbables, entonces este enfoque realmente no es mejor que cualquier técnica de prueba aleatoria.

Casos de prueba y jerarquía de clase

La herencia no dispensa la necesidad de pruebas amplias de todas las clases derivadas. De hecho, en realidad puede complicar el proceso de prueba. Considere la siguiente situación. Una clase Base contiene operaciones inherited( ) y redefined( ). Una clase Derived redefine redefined( ) para servir en un contexto local. Hay poca duda de que Derived::redefined( ) tiene que probarse porque representa un nuevo diseño y un nuevo código. Pero, ¿Derived::inherited( ) debe probarse nuevamente?

Diseño de pruebas basadas en escenario

Las pruebas basadas en fallo pierden dos tipos principales de errores:

1) especificaciones incorrectas e

2) interacciones entre subsistemas.

Cuando ocurren errores asociados con una especificación incorrecta, el producto no hace lo que el cliente quiere. Puede hacer lo correcto u omitir funcionalidad importante.

Los errores asociados con la interacción de subsistemas ocurren cuando el comportamiento de un subsistema crea circunstancias que hacen que otro subsistema falle.

Diseño de pruebas basadas en escenario

Caso de uso: corrección del borrador final

Antecedentes: No es raro imprimir el borrador “final”, leerlo y descubrir algunos errores desconcertantes que no fueron obvios en la imagen de la pantalla. Este caso de uso describe la secuencia de eventos que ocurren cuando esto sucede.

1. Imprimir todo el documento.
2. Moverse en el documento, cambiar ciertas páginas.
3. Conforme cada página cambia, imprimirla.
4. En ocasiones se imprime una serie de páginas.

Diseño de pruebas basadas en escenario

Caso de uso: imprimir una nueva copia

Antecedentes: Alguien pide al usuario una copia reciente del documento. Debe imprimirla.

1. Abrir el documento.

2. Imprimirlo.

3. Cerrar el documento.

De nuevo, el enfoque de las pruebas es relativamente obvio. Excepto que este documento no aparece de la nada. Se creó en una tarea anterior. ¿Dicha tarea afecta a la actual?

Diseño de pruebas basadas en escenario

Caso de uso: imprimir una nueva copia

1. Abrir el documento.
2. Seleccionar “Imprimir” en el menú.
3. Comprobar si se imprime un rango de páginas; si es así, dar clic para imprimir todo el documento.
4. Dar clic en el botón Imprimir.
5. Cerrar el documento.

Pruebas de las estructuras superficial y profunda

Cuando se habla de estructura superficial se hace referencia a la estructura observable externamente de un programa OO. En lugar de realizar funciones, a los usuarios de muchos sistemas OO se les pueden dar objetos para manipular en alguna forma. Pero las pruebas se basan todavía en tareas de usuario. Capturar estas tareas involucra comprensión, observación y hablar con usuarios representativo.

Cuando se habla de estructura profunda, se hace referencia a los detalles técnicos internos de un programa OO. La prueba de estructura profunda se diseña para ejercitar dependencias, comportamientos y mecanismos de comunicación que se establezcan como parte del modelo de diseño para el software OO.

Métodos de prueba en el nivel de clase

- Prueba aleatoria para clases OO
- Prueba de partición en el nivel de clase

Prueba aleatoria para clases OO

Account
open() setup() deposit() withdraw() balance() sumaries() creditLimit() close()

La historia de vida de comportamiento mínima de una instancia de Account incluye las siguientes operaciones:

open•setup•deposit•withdraw•close

Sin embargo, dentro de esta secuencia puede ocurrir una amplia variedad de otros comportamientos:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]n•withdraw•close

Prueba de partición en el nivel de clase

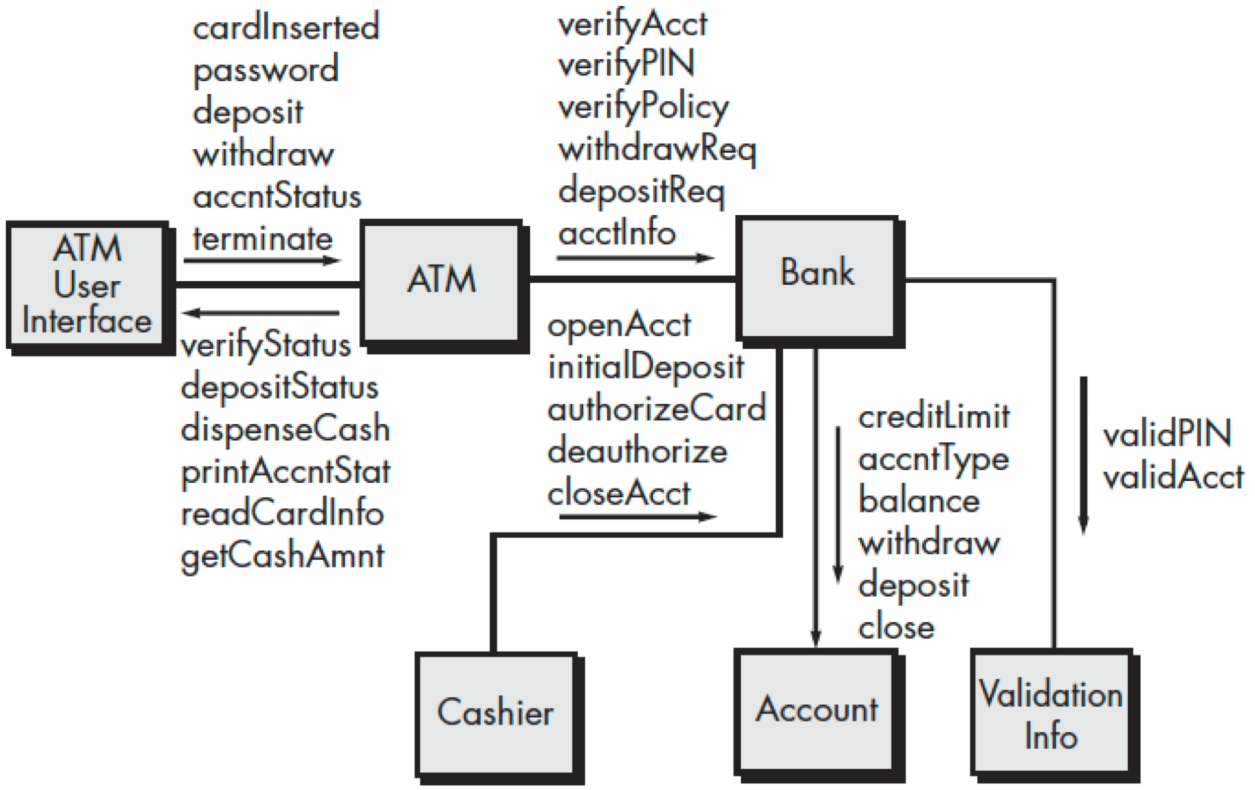
Las entradas y salidas se categorizan y los casos de prueba se diseñan para probar cada categoría. ¿Pero cómo se derivan las categorías de partición?

La partición con base en estado categoriza las operaciones de clase a partir de su capacidad para cambiar el estado de la clase. Considere de nuevo la clase Account, las operaciones de estado incluyen deposit( ) y withdraw( ), mientras que las operaciones de no estado incluyen balance( ), sumaries( ) y creditLimit( ). Las pruebas se diseñan para que ejerciten por separado las operaciones que cambian el estado y aquellas que no lo cambian. En consecuencia,

Caso de prueba p1: open•setup•deposit•deposit•withdraw•withdraw•close

Caso de prueba p2: open•setup•deposit•summarize•creditLimit•withdraw•close

Diseño de casos de prueba interclase

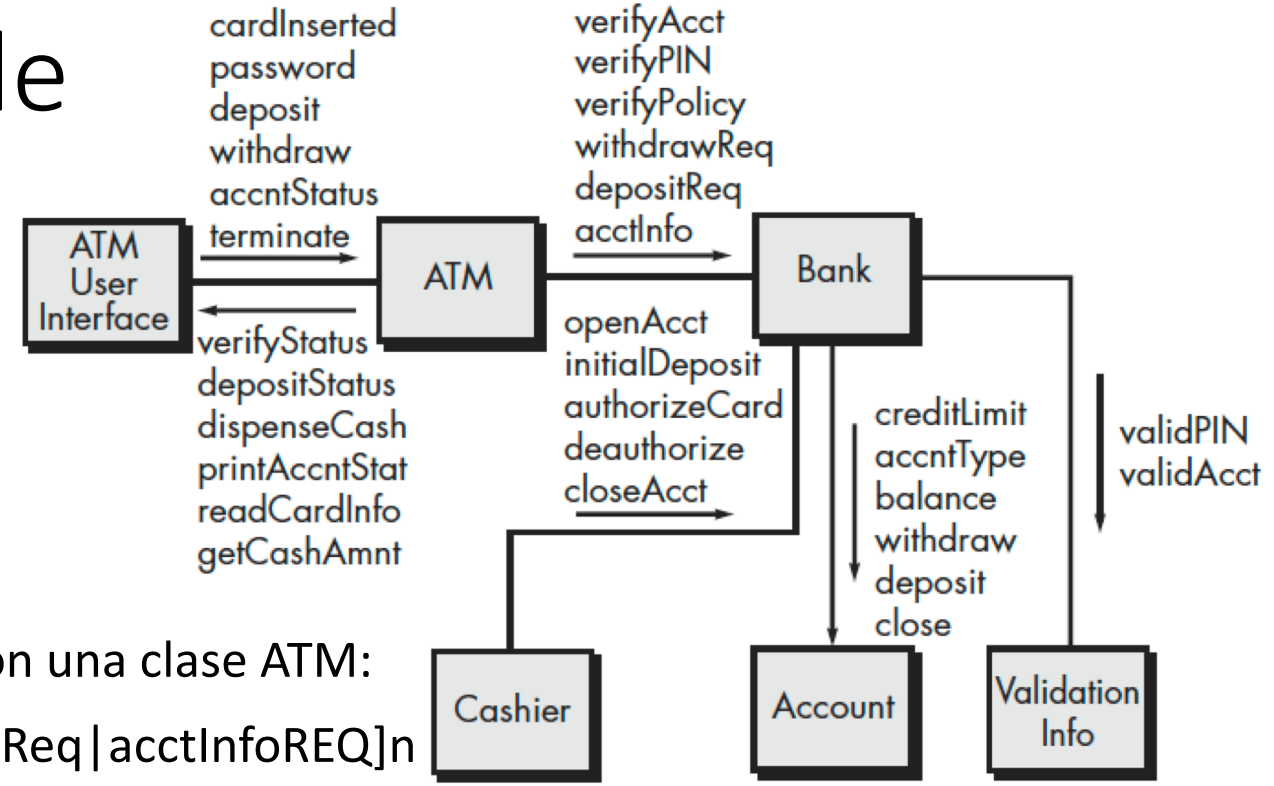


Prueba de clase múltiple

Pasos para generar casos de prueba aleatorios de clase múltiple:

1. Para cada clase cliente, use la lista de operaciones de clase a fin de generar una serie de secuencias de prueba aleatorias. Las operaciones enviarán mensajes a otras clases servidor.
2. Para cada mensaje generado, determine la clase colaborador y la correspondiente operación en el objeto servidor.
3. Para cada operación en el objeto servidor determine los mensajes que transmite.
4. Para cada uno de los mensajes, determine el siguiente nivel de operaciones que se invocan e incorpore esto en la secuencia de prueba.

Prueba de clase múltiple



secuencia de operaciones para la clase Bank en relación con una clase ATM:

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]n

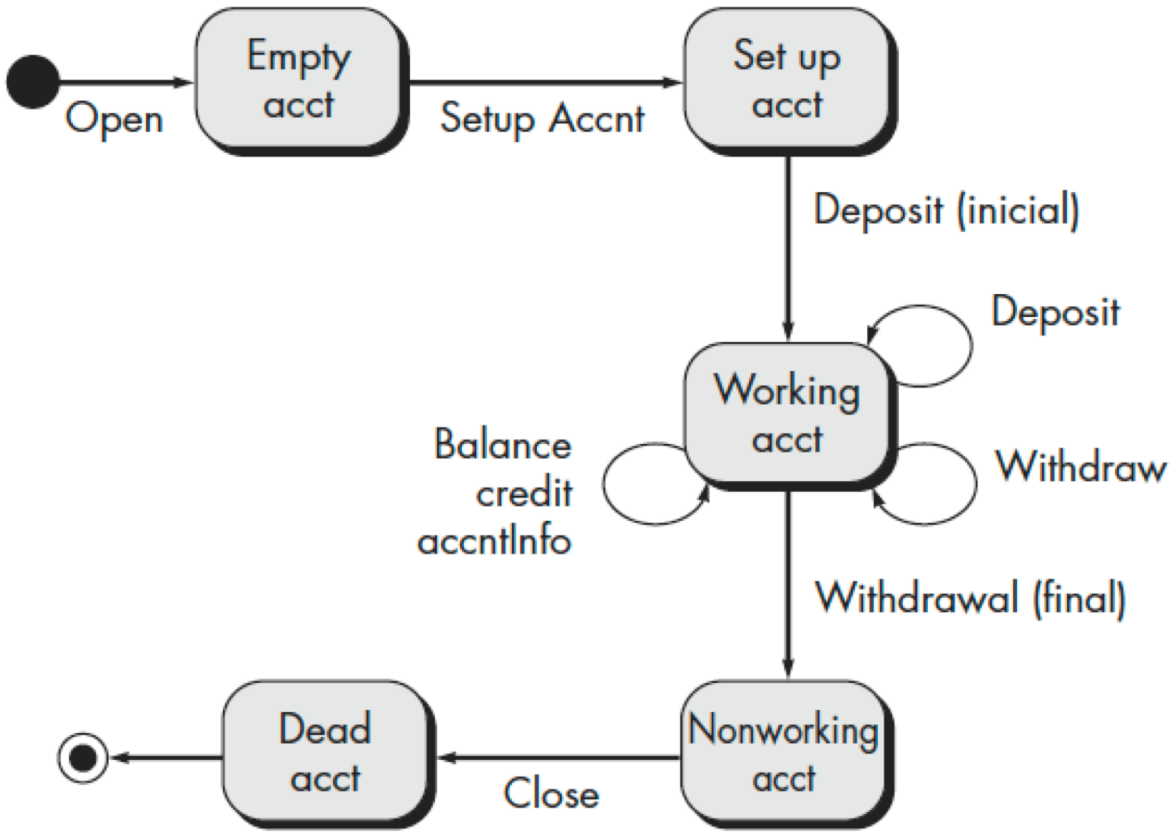
Un caso de prueba aleatorio para la clase Bank puede ser

Caso de prueba r3 verifyAcct•verifyPIN•depositReq

Caso de prueba r4 verifyAcct [Bank:validAcctValidationInfo]•verifyPIN

[Bank: validPinValidationInfo]•depositReq [Bank: depositaccount]

Pruebas derivadas a partir de modelos de comportamiento



Resumen

El objetivo global de las pruebas orientadas a objetos es idéntico al de la prueba de software convencional. La estrategia y las tácticas de la prueba OO difieren significativamente.

Una vez disponible el código, la prueba de unidad se aplica para cada clase. El diseño de pruebas para una clase usa varios métodos: prueba basada en fallo, prueba aleatoria y prueba de partición. Las secuencias de prueba se diseñan para garantizar que se pruebas las operaciones relevantes.

La prueba de integración puede lograrse usando una estrategia basada en hebra o en uso.

La prueba de validación del sistema OO está orientada a caja negra y puede lograrse al aplicar los mismos métodos de caja negra estudiados para el software convencional.