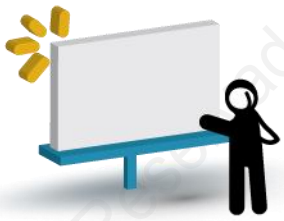
UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

Professional Testing Master

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Unidad 3: Testing de aplicaciones orientadas a objetos



Presentación:

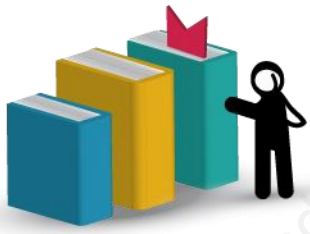
En esta Tercera Unidad del curso, nos centraremos en las estrategias de testing para el caso de aplicaciones orientadas a objetos.

Objetivos:

Al terminar la Unidad los participantes:

Habrán ensanchado la visión del testing para incluir la revisión de los modelos de requerimientos y de diseño.

Estarán en condiciones de diseñar pruebas efectivas aplicadas a clases y objetos.



Bloques temáticos:

1. Testing de modelos de análisis y diseño orientados a objetos
2. Estrategias de testing orientadas a objetos
3. Métodos de testing orientados a objetos
4. Métodos de testing aplicables a nivel de clase
5. Diseño de casos de prueba inter-clase

Contenido

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

**** El MEC es el modelo de E-learning colaborativo de nuestro Centro.***

Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1 Introducción

Nota importante: Se recomienda a los lectores que no estén familiarizados con el paradigma orientado a objetos, leer primero el apéndice que se encuentra al final de esta Unidad (sección 7) antes de continuar con estos párrafos.

En la Unidad 1 se señaló que el objetivo de las pruebas, dicho de manera simple, es encontrar la mayor cantidad posible de errores con una cantidad manejable de esfuerzo aplicado durante un lapso realista. Aunque este objetivo fundamental permanece igual para el software orientado a objetos (OO), la naturaleza de los programas OO cambia las estrategias y las tácticas de las pruebas. Podría pensarse que, conforme las bibliotecas de clases reutilizables crecen en tamaño, un reuso mayor mitigará a los sistemas OO en su necesidad de pruebas pesadas. Pero no. Cada reuso es un nuevo contexto de uso y es prudente una nueva comprobación. Probablemente se necesitarán más pruebas, no menos, para obtener alta confiabilidad en los sistemas OO.

Para probar adecuadamente estos sistemas, deben realizarse tres cosas: 1) ampliar la definición de prueba para incluir las técnicas de descubrimiento de errores aplicadas al análisis OO y a modelos de diseño, 2) cambiar significativamente la estrategia para prueba unitaria e integración y 3) contemplar las características únicas del software OO durante el diseño de casos de prueba.

La construcción de software OO comienza con la creación de modelos de requerimientos (**análisis**) y de **diseño**[1]. Debido a la naturaleza evolutiva del paradigma OO, dichos modelos comienzan como representaciones relativamente informales de los requerimientos del sistema y evolucionan hacia modelos detallados de clases, relaciones entre clases, y diseño de objetos (que incorpora conectividad entre objetos mediante mensajes). En cada etapa, los modelos pueden "probarse" con la intención de descubrir errores previamente a su propagación hacia la siguiente iteración.

La revisión de los modelos de análisis y de diseño OO es especialmente útil, pues las mismas construcciones semánticas (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 evitará efectos colaterales que podrían ocurrir si el problema no se descubriera hasta el diseño o el código (o incluso en la siguiente iteración de análisis).

[1] Habitualmente se dice que el análisis define el **"qué"** hace el software y diseño define el **"cómo"** lo hace.

Por ejemplo, considere una clase en la que se definen atributos durante la primera iteración de análisis. Suponga que debido a una mala interpretación del dominio del problema se incluye un atributo innecesario. Entonces se especifican dos operaciones para manipular el atributo (por ejemplo get y set[2]). Se lleva a cabo una revisión y un experto en el dominio puntualiza el problema. Al eliminar el atributo innecesario durante el análisis pueden evitarse los siguientes problemas y un esfuerzo innecesario:

1. Se podrían haber generado subclases especiales para alojar el atributo innecesario. Se evita el trabajo involucrado en la creación de subclases innecesarias.
2. Una mala interpretación de la definición de clase puede conducir a relaciones de clase incorrectas o innecesarias.
3. El comportamiento del sistema o sus clases puede caracterizarse de manera inadecuada para alojar el atributo innecesario.

Si el problema no se descubre durante el análisis y se propaga aún más, podrían ocurrir los siguientes problemas durante el diseño (que se evitarían con la revisión temprana):

1. Durante el diseño del sistema, podría ocurrir la asignación inadecuada de la clase a un subsistema.
2. Puede invertirse trabajo de diseño innecesario a fin de crear el diseño de algoritmos para las operaciones que manejan el atributo innecesario.
3. El modelo de mensajería será incorrecto (mensajes para las operaciones que son innecesarias).

Si el problema sigue sin detectarse durante el diseño y pasa a la actividad de codificación, se empleará esfuerzo considerable para generar código que implemente un atributo innecesario, dos operaciones innecesarias (get y set), mensajes que activen la comunicación entre objetos y muchos otros conflictos relacionados. Además, la prueba de la clase tomará más tiempo que el necesario. Una vez que finalmente se haya descubierto

[2] get y set (que se pueden traducir como obtener y establecer) son el típico par de operaciones para leer y escribir un atributo de un objeto, Por ejemplo, para una hipotética clase Producto, get_stock() y set_stock() permiten respectivamente leer y escribir la cantidad de unidades en stock del producto.

el problema, la modificación del sistema debe realizarse con la posibilidad siempre presente de **efectos colaterales debidos al cambio**.

Los modelos de análisis (AOO) y de diseño (DOO) orientado a objetos proporcionan información sustancial acerca de la estructura y comportamiento del sistema. Por esta razón, dichos modelos deben someterse a una rigurosa revisión, previa a la generación del código. Estos modelos deben probarse (en este contexto, el término prueba significa revisiones técnicas, es decir testing estático) en relación con su exactitud, completitud y consistencia semántica.

2 Prueba de modelos de análisis y diseño

Los modelos de análisis y diseño no pueden probarse de la manera convencional porque no pueden ejecutarse. Sin embargo, pueden usarse revisiones técnicas para examinar su exactitud y consistencia.

2.1 Exactitud de los modelos AOO y DOO

Durante el análisis y el diseño, la exactitud semántica puede evaluarse en base a la conformidad del modelo con el dominio de problemas del mundo real. Si el modelo refleja con precisión el mundo real (en un nivel de detalle que sea apropiado para la etapa de desarrollo en la que se revisó el modelo), entonces es semánticamente correcto. Para determinar si el modelo verdaderamente refleja los requerimientos del mundo real, debe presentarse a expertos de dominio del problema, quienes examinarán las definiciones y jerarquía de clases en busca de omisiones y ambigüedad. Las relaciones de clase (conexiones de instancia) se evalúan para determinar si reflejan con precisión conexiones de objetos en el mundo real. Los casos de uso[3] [Deb16] pueden ser invaluables para comparar los modelos de análisis y diseño contra escenarios de uso del sistema OO en el mundo real.

[3] Caso de uso: Una secuencia de transacciones en la interacción entre un actor y un componente o sistema con un resultado tangible. Un actor puede ser un usuario o cualquier cosa que pueda intercambiar información con el sistema (por ej. un sistema externo). Son la base para especificar la funcionalidad/requerimientos de un sistema OO.

2.2 Consistencia de los modelos orientados a objetos

La consistencia de los modelos puede evaluarse al considerar las relaciones entre entidades. Un modelo de análisis o diseño inconsistente tiene representaciones en una parte del modelo que no se reflejan de manera correcta en otras porciones.

Para evaluar la consistencia, debe examinarse cada clase y sus conexiones con otras clases. A fin de facilitar esta actividad, puede usarse el modelo clase-responsabilidad-colaboración (CRC) o un diagrama de objeto-relación. El modelo CRC [Pre15] se compone de fichas CRC. Cada ficha CRC menciona el nombre de la clase, sus responsabilidades (operaciones) y sus colaboradores (otras clases a las que envía mensajes y de las que depende para lograr sus responsabilidades). Las colaboraciones implican una serie de relaciones (es decir, conexiones) entre clases del sistema OO. El modelo objeto-relación proporciona una representación gráfica de las conexiones entre clases. Toda esta información puede obtenerse a partir del modelo de análisis [Pre15].

El método para evaluar la consistencia del modelo se explicará mediante un ejemplo. Supongamos un sistema de facturación para una empresa de servicios cuya clase principal es **SistemaFacturacion** que ofrece las operaciones *verFacturaOnline(),* que permite visualizar una factura previamente emitida y *facturarCliente()*, que emite la factura para un cliente determinado de acuerdo a los servicios consumidos por éste y opcionalmente la imprime.

Esto se muestra en el diagrama de la Figura 2-1, el cual utiliza un diagrama de comunicación UML[4] versión 2 para representar el modelo objeto-relación surgido del análisis o diseño.

El objetivo de este test es determinar la consistencia del modelo de análisis o diseño del sistema OO, el cual está formado por clases que colaboran entre sí.

[4] El lenguaje unificado de modelado (UML, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de software más conocido y utilizado. Consiste en diferentes tipos de diagramas para especificar y documentar sistemas.

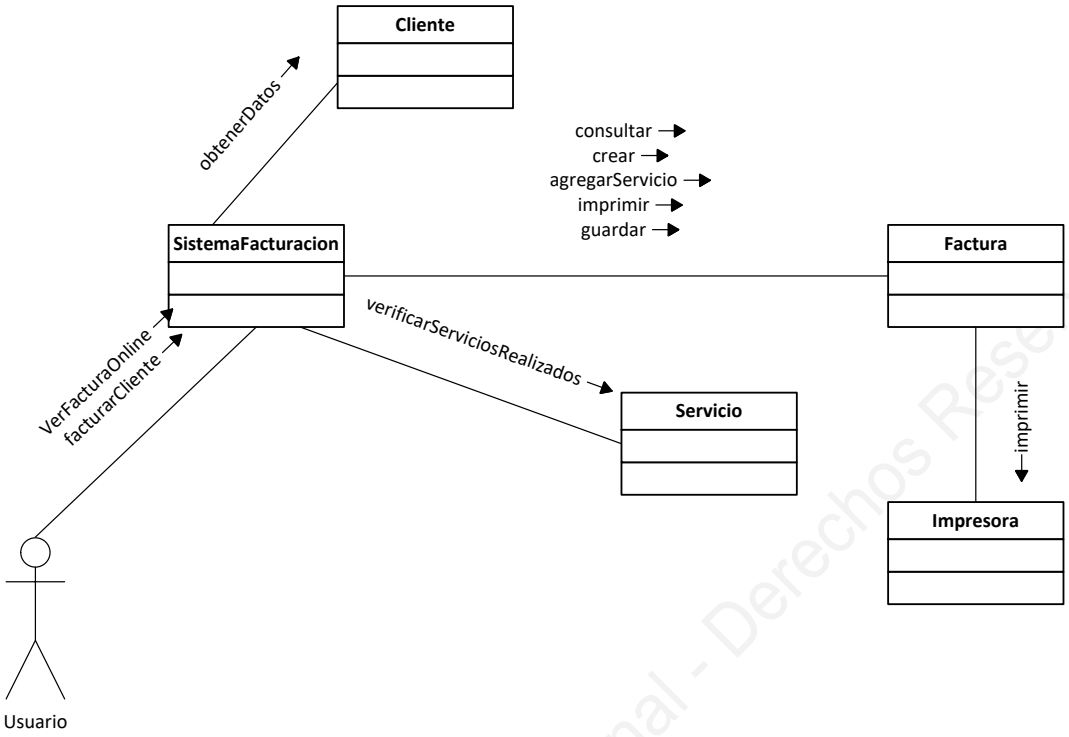


Figura 2-1. Modelo objeto-relación del sistema

A partir del modelo objeto-relación, se genera el modelo CRC explicado en los párrafos anteriores, llegando a las siguientes fichas para cada clase:

Nombre de la clase: SistemaFacturacion	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
verFacturaOnline	Factura
facturarCliente	Cliente
	Servicio
	Factura

Nombre de la clase: Cliente	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
obtenerDatos	

Nombre de la clase: Factura	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
consultar	
crear	
agregarServicio	
imprimir	Impresora
guardar	

Nombre de la clase: Servicio	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
verificarServiciosRealizados	

Nombre de la clase: Impresora	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
imprimir	

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

1. Revisar el modelo CRC para garantizar que todas las colaboraciones implicadas por el modelo de requerimientos se reflejan de manera adecuada en el CRC.

- ✓ OK: Vemos que cada clase tiene su ficha CRC, todas las operaciones (responsabilidades) y colaboraciones (clases utilizadas) están representadas.

2. Inspeccionar la descripción de cada ficha CRC para determinar si una responsabilidad delegada es parte de la definición del colaborador.
   - ✓ OK: Por ejemplo, vemos que la responsabilidad ***imprimir*** de la clase **Factura** está delegada a la clase **Impresora**, la cual tiene como responsabilidad ***imprimir***.

Nombre de la clase: Factura	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
consultar	
crear	
agregarServicio	
imprimir	Impresora
guardar	

Nombre de la clase: Impresora	
Tipo de la clase: No relevante para este ejemplo	
Características de la clase: No relevante para este ejemplo	
Responsabilidades	**Colaboradores**
imprimir	

3. Invertir la conexión para garantizar que cada colaborador al que se solicita servicio recibe solicitud de una fuente razonable. Por ejemplo, si la clase **Factura** recibe una solicitud para ***crear*** una factura desde la clase **Impresora**, habría un problema, dado que **Impresora** no maneja facturas sino que sólo se limita a tareas de impresión.
   - ✓ OK: En este ejemplo todas las responsabilidades (servicios ofrecidos por las clases) son requeridos por clases razonables.
4. Usando las conexiones invertidas que se examinaron en el paso 3, determinar si podrían hacer falta otras clases o si las responsabilidades están agrupas de manera adecuada entre las clases.
   - ✓ OK: En este ejemplo todas las responsabilidades son resueltas por alguna clase y parecen correctamente distribuidas entre las clases.

5. Determinar si las responsabilidades muy solicitadas (de una misma clase) pueden combinarse en una sola responsabilidad. Por ejemplo si fuera obligatorio imprimir siempre la factura, entonces las responsabilidades ***guardar*** e ***imprimir*** de la clase **Factura** se podrían unificar en una sola operación.
   - ✓ OK: En este ejemplo, no se da ese caso.

Los pasos del 1 al 5 deben aplicarse de manera iterativa a cada clase y a lo largo de cada evolución del modelo de requerimientos.

3 Estrategias de prueba orientadas a objetos

Como se vio en las dos Unidades anteriores, la estrategia clásica de prueba de software comienza “probando lo pequeño” y funciona hacia afuera, “probando lo grande”. Dicho en el lenguaje de testing (Unidad 2), comienza con la prueba unitaria, luego avanza hacia la prueba de integración y culmina con las pruebas de validación y sistema. En aplicaciones convencionales, la prueba unitaria se enfoca en la unidad de programa compatible más pequeña: el subprograma (por ejemplo, componente, módulo, subrutina, procedimiento, función). Una vez que cada una de estas unidades se prueba de manera individual, se integra en una estructura de programa mientras se aplica una serie de pruebas de integración para descubrir errores debidos a la puesta en interfaz de los módulos y los efectos colaterales que se generan al sumar nuevas unidades. Finalmente, el sistema como un todo se prueba para garantizar que se descubren los eventuales errores en los requerimientos.

3.1 Prueba unitaria

Cuando se piensa en software OO, cambia el concepto de unidad. El encapsulamiento impulsa la definición de clases y objetos. Esto significa que cada clase y cada instancia (objeto individual) de una clase encapsulan los atributos (datos) y las operaciones (también conocidas como métodos o servicios) 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 varias operaciones diferentes y una operación particular puede existir en clases diferentes, el significado de prueba unitaria cambia drásticamente.

Ya no es posible probar una sola operación aislada (la visión convencional de la prueba unitaria) sino, como parte de una clase. Para ilustrar lo anterior, considere una jerarquía de clase en la que se define una operación *X( )* para la superclase y la heredan algunas subclases. Cada subclase usa la operación *X( )*, pero se aplica dentro del contexto de los atributos y operaciones privados que se definieron para cada subclase. Puesto que el contexto donde se usa la operación *X( )* varía en formas sutiles, es necesario probarla en el contexto de cada una de las subclases. Esto significa que probar la operación *X( )* en el vacío (el enfoque tradicional de la prueba unitaria) no es efectivo en el contexto OO.

La prueba de clase para el software OO es el equivalente de la prueba unitaria para software convencional. A diferencia de la prueba unitaria convencional, que tiende a enfocarse en el detalle algorítmico de un módulo y en los datos que fluyen a través de la interfaz del módulo, la prueba de clase para el software OO depende de las operaciones encapsuladas por la clase y del comportamiento del estado de la misma.

3.2 Prueba de integración

Integrar las operaciones de a una en una clase (la integración incremental convencional) suele ser imposible debido a las interacciones directas e indirectas de los componentes que constituyen la clase [Ber93]. En otras palabras, **la unidad mínima de integración es la clase**, al igual que se explicó en la sección anterior para la prueba unitaria.

Las clases se pueden integrar en forma ascendente o descendente, como se vio en la Unidad 1.

3.3 Prueba de validación

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. Para

auxiliar en la generación de pruebas de validación, el tester debe recurrir a casos de uso que son parte del modelo de requerimientos. El caso de uso proporciona un escenario que tiene una alta probabilidad de descubrir errores en los requerimientos de interacción con el usuario.

Los métodos convencionales de prueba de caja negra (Unidad 2) pueden usarse para hacer pruebas de validación.

4 Métodos de prueba orientados a objetos

La arquitectura del software OO da como resultado una serie de subsistemas en capas que encapsulan clases colaboradoras. Cada uno de estos elementos de sistema (subsistemas y clases) realiza funciones que ayudan a lograr requerimientos de sistema. Es necesario probar un sistema OO en varios niveles diferentes con la intención de descubrir errores que puedan ocurrir conforme las clases colaboran unas con otras y conforme los subsistemas se comunican a través de las capas arquitectónicas.

Berard [Ber93] sugiere un enfoque global en el diseño de casos de prueba OO:

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 objetivo de la prueba.
3. Debe desarrollarse una lista de pasos para cada prueba, que debe contener:
   a. Una lista de estados especificados para la clase que se probará
   b. Una lista de mensajes y operaciones que se ejercitarán como consecuencia de la prueba
   c. Una lista de excepciones que pueden ocurrir conforme se prueba la clase
   d. Una lista de condiciones externas (es decir, cambios en el entorno externo al software que debe existir para poder realizar las pruebas)
   e. Información complementaria que ayudará a comprender o implementar la prueba

A diferencia del diseño convencional de casos de prueba, que se activan mediante una visión entrada-proceso-salida del software o con el detalle algorítmico de módulos individuales, la prueba orientada a objetos se enfoca en el diseño de secuencias apropiadas de operaciones para probar los estados de una clase.

4.1 Implicaciones de la OO en el diseño de casos de prueba

Puesto que los atributos y las operaciones están encapsulados, no es posible probar operaciones afuera de la clase. Aunque el encapsulamiento es un concepto de diseño esencial para OO, puede crear un obstáculo menor cuando se prueba. Las pruebas requieren reportar el estado de un objeto. No obstante, el encapsulamiento puede hacer que esta información sea un poco difícil de obtener. A menos que se proporcionen operaciones internas a fin de reportar los valores para los atributos de clase, puede ser difícil adquirir una foto del estado de un objeto.

La herencia también puede presentar retos adicionales durante el diseño de casos de prueba. Ya se indicó que cada nuevo contexto de uso requiere un nuevo examen, aun cuando se haya logrado la reutilización.

4.1.1 Casos de prueba y jerarquía de clase

Como se dijo, la herencia no exceptúa de 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 llamada **Base** contiene las operaciones *heredada( ) y redefinida( )*. Otra clase llamada **Derivada** redefine la operación *redefinida( )* para servir en un contexto local. No hay duda de que **Derivada::redefinida( )**[5] tiene que probarse porque representa un nuevo diseño y un nuevo código. Pero, ¿**Derivada::heredada( )** debe probarse nuevamente?

Si **Derivada::heredada( )** llama a *redefinida( )* y el comportamiento de *redefinida( )* cambió, entonces **Derivada::heredada( )** puede manejar mal el nuevo comportamiento. Por lo tanto, necesita nuevas pruebas aun cuando el diseño y el código no hayan cambiado. No obstante, es importante observar que es posible que sólo se ejecute un subconjunto de todas las pruebas para **Derivada::heredada( )**. Si parte del diseño y código para *heredada( )* no depende *de redefinida( )* (es decir, no lo llama ni llama a

[5] La notación A::b() significa la operación b de la clase A.

código que lo llame de manera indirecta), dicho código no necesita probarse de nuevo en la clase derivada.

Base::redefinida( ) y **Derivada::redefinida( )** son dos operaciones diferentes con diferentes especificaciones e implementaciones. Cada una tendrá un conjunto de requerimientos de prueba salidos de la especificación y la implementación. Pero es probable que las operaciones sean similares. Sus conjuntos de requerimientos de prueba se solaparán. Mientras mejor sea el diseño OO, mayor es la superposición. Es necesario derivar nuevas pruebas sólo para aquellos requerimientos de **Derivada::redefinida( )** que no se satisfagan con las pruebas de **Base::redefinida( )**.

Para resumir, las pruebas de **Base::redefinida( )** se aplican a objetos de la clase **Derivada**. Las entradas de prueba pueden ser adecuadas tanto para la clase base como para la derivada, pero los resultados esperados pueden diferir en la clase derivada.

4.2 Aplicabilidad de métodos convencionales de diseño de casos de prueba

Los métodos de prueba de caja blanca descritos en la Unidad 2 pueden aplicarse a las operaciones definidas para una clase. Las técnicas de ruta básica y condición múltiple, pueden ayudar a garantizar que se probaron todas las instrucciones 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 a nivel de clase.

Los métodos de prueba de caja negra son tan apropiados para los sistemas OO como para los sistemas convencionales. Como se observó en la Unidad 2, la especificación (en este caso los casos de uso) proporciona información útil en el diseño de las pruebas de caja negra y en las de partición basada en estado (tratadas más adelante).

4.3 Diseño de pruebas basadas en escenario

A veces, las pruebas, fracasan en detectar dos tipos principales de errores: 1) especificaciones incorrectas y 2) interacciones entre subsistemas. Cuando ocurren errores asociados con una especificación incorrecta, el producto no hace lo que el cliente quiere. Puede hacer algo incorrecto u omitir funcionalidad importante. Pero en ambos casos, la calidad (conformidad con los requerimientos) se resiente. Los errores asociados con la interacción de subsistemas ocurren cuando el comportamiento de un subsistema crea circunstancias (por ejemplo eventos o flujo de datos) que hacen que otro subsistema falle.

La prueba basada en escenario se concentra en lo que hace el usuario, no en lo que hace el producto. Esto significa capturar las tareas (por medio de casos de uso) que el usuario tiene que realizar y luego aplicar éstas y sus variantes como pruebas.

Los escenarios descubren errores de interacción. Pero, para lograr esto, los casos de prueba deben realistas. La prueba basada en escenario tiende a probar múltiples subsistemas en una sola prueba (los usuarios no se limitan al uso de un subsistema a la vez).

Como ejemplo, considere el diseño de pruebas basadas en escenario para un editor de texto al revisar los casos de uso que siguen:

Escenario: corrección del borrador final

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

1. Imprimir todo el documento.
2. Moverse en el documento, modificar ciertas páginas.
3. A medida que se modifica cada página, imprimirla.
4. En ocasiones se imprime un intervalo de páginas.

Este escenario describe dos cosas: una prueba y necesidades específicas del usuario. Las necesidades del usuario son obvias: 1) un método para imprimir páginas individuales y 2) un método para imprimir un rango de páginas. Mientras avanzan las pruebas, hay

necesidad de probar la edición después de imprimir (así como lo inverso). Por lo tanto, se trabaja para diseñar pruebas que descubrirán errores en la función de edición que fueron provocados por la función de impresión, es decir, errores que indicarán que las dos funciones de software no son independientes.

Escenario: imprimir una nueva copia

Antecedentes: Alguien pide al usuario una impresión actualizada del documento.

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?

En muchos editores modernos, los documentos recuerdan cómo se imprimieron la última vez. Por defecto, imprimen de la misma forma la siguiente ocasión. Después del escenario **Corrección del borrador final**, seleccionar solamente “Imprimir” en el menú y dar clic en el botón Imprimir en el cuadro de diálogo hará que se imprima solamente la última página (o intervalo) corregido. De manera que, de acuerdo con el editor, el escenario correcto debe verse del modo siguiente:

Escenario: imprimir una nueva copia

1. Abrir el documento.
2. Seleccionar “Imprimir” en el menú.
3. Comprobar si se imprimirá 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.

Pero este escenario indica una potencial especificación de error. El editor no hace lo que el usuario razonablemente espera que haga. Los usuarios con frecuencia pasan por alto la comprobación indicada en el paso 3. Entonces quedarán desconcertados cuando vayan

a la impresora y encuentren una página cuando querían 100. Los clientes desconcertados señalan errores de especificación.

Esta dependencia puede perderse cuando se diseñan pruebas, pero es probable que el problema salga a la luz durante las pruebas con el usuario. Entonces tendría que lidiar con el probable reclamo: “¡se supone que debe funcionar así!”

5 Métodos de prueba aplicables a nivel clase

La prueba “en lo pequeño” se enfoca en una sola clase y en los métodos que encapsula. La prueba aleatoria y la partición de equivalencia son métodos que pueden usarse para probar una clase durante la prueba OO. Sin embargo, el número de posibles permutaciones para la prueba aleatoria puede volverse muy grande. Para mejorar la eficiencia de la prueba, puede usarse una estrategia similar a la partición de equivalencia.

5.1 Prueba aleatoria para una clase

Para ofrecer breves ilustraciones de estos métodos, considere una aplicación bancaria en la que una clase **Cuenta** tiene las siguientes operaciones: *abrir( ), depositar( ), extraer( ), saldo( ), resumen( ), limiteCredito( ) y cerrar( )*. Cada una de estas operaciones puede aplicarse a **Cuenta**, pero ciertas restricciones están implícitas por la naturaleza del problema (por ejemplo, la cuenta debe abrirse antes de que otras operaciones puedan aplicarse y debe cerrarse después de que todas las operaciones se completen). Incluso con estas restricciones, existen muchas permutaciones de las operaciones. La historia de vida de comportamiento mínima de una instancia de **Cuenta** incluye las siguientes operaciones:

abrir • depositar • extraer • cerrar

Esto representa la secuencia de prueba mínima para Cuenta. Sin embargo, dentro de esta secuencia puede ocurrir una amplia variedad de otros comportamientos:

abrir • depositar • [depositar | extraer | saldo | resumen | limiteCredito]^{n} • extraer • cerrar

En el comportamiento anterior, el superíndice (^{n}) significa que la o las operaciones encerradas entre corchetes pueden repetirse una cantidad indefinida de veces. La barra vertical (|) separando las operaciones significa una u otra.

Varias secuencias diferentes de operaciones pueden generarse al azar. Por ejemplo:

Caso de prueba a1: abrir • depositar • depositar • saldo • resumen • extraer • cerrar

Caso de prueba a2:

abrir • depositar • extraer • depositar • saldo • limiteCredito • extraer • cerrar

Éstas y otras pruebas de tipo aleatorio se realizan para verificar diferentes historias de vida de las instancias de clase.

5.2 Prueba de partición en el nivel de clase

La prueba de partición reduce el número de casos de prueba requeridos para verificar la clase, en una forma muy similar a la partición de equivalencia (Unidad 2) para el software tradicional. 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 basada en estado** (state-based partitioning) categoriza las operaciones de clase a partir de su capacidad para **cambiar el estado** de la clase. Considere de nuevo la clase **Cuenta**. Las operaciones de estado incluyen *depositar( )* y *extraer( )*, mientras que

las operaciones de no estado incluyen *saldo( ), resumen( )* y *limiteCredito( )*. Las pruebas se diseñan para que verifiquen por separado las operaciones que cambian el estado y aquellas que no lo cambian. En consecuencia,

Caso de prueba p1: abrir • depositar • **depositar • extraer** • extraer • cerrar

Caso de prueba p2: abrir • depositar • **resumen • limiteCredito** • extraer • cerrar

El caso de prueba p1 cambia el estado, mientras que el p2 verifica las operaciones que no cambian el estado (más allá de las que están en la secuencia de prueba mínima).

La **partición basada en atributo** (attribute-based partitioning) categoriza las operaciones de clase en base a los atributos que usan. Para la clase **Cuenta**, los atributos *_saldo* y *_limiteCredito* pueden usarse para definir particiones. Por ejemplo, usando *_limiteCredito*, las operaciones se dividen en tres particiones: 1) operaciones que usan *_limiteCredito*, 2) operaciones que modifican *_limiteCredito* y 3) operaciones que no usan ni modifican *_limiteCredito*. Entonces se diseñan secuencias de prueba para cada partición.

La **partición basada en categoría** (category-based partitioning) jerarquiza las operaciones de clase con base en la función genérica que cada una realiza. Por ejemplo, las operaciones en la clase **Cuenta** pueden categorizarse en operaciones de inicialización (*abrir*), de cálculo (*depositar, extraer*), consultas (*saldo, resumen, limiteCredito*) y de terminación (*cerrar*).

6 Diseño de casos de prueba interclase

El diseño de casos de prueba se vuelve más complicado a medida que comienza la integración del sistema OO. En esta etapa debe comenzar la prueba de las colaboraciones entre clases. Para ilustrar la generación de casos de prueba interclase, se modifica el ejemplo bancario presentado en la sección 5 a fin de incluir las clases y colaboraciones indicadas en la Figura 6-1 (diagrama de colaboración UML 2), donde la clase **ATM** representa un cajero automático y la clase **Cajero** representa la ventanilla manual. La dirección de las flechas en la figura indica la dirección de los mensajes y las

etiquetas indican las operaciones que se involucran como consecuencia de cada uno de los mensajes.

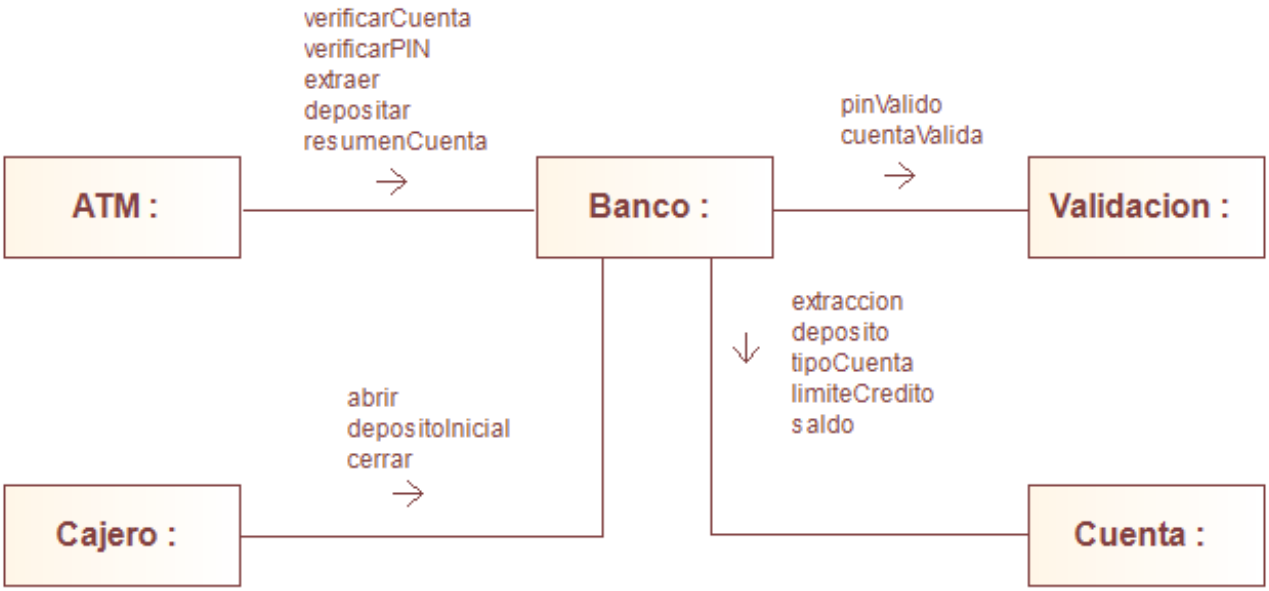


Figura 6-1. Diagrama de colaboración de clases para aplicación bancaria

6.1 Prueba de clase múltiple

Kirani y Tsai [Kir94] sugieren la siguiente secuencia de pasos para generar casos de prueba aleatorios de clase múltiple:

1. Para cada clase cliente (consumidora de un servicio), use la lista de operaciones de clase a fin de generar una serie de secuencias de prueba aleatorias. Las operaciones enviarán mensajes a clases servidoras.
2. Para cada mensaje generado, determine la clase colaboradora y la correspondiente operación en la clase servidora.
3. Para cada operación en la clase servidora (invocada por los mensajes enviados desde la clase cliente), 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.

Para ilustrar, considere una secuencia de operaciones ofrecidas por la clase **Banco** a la clase **ATM** (Figura 6-1):

verificarCuenta • verificarPIN • [extraer | depositar | resumenCuenta]^{n}

Un caso de prueba aleatorio para la clase Banco puede ser:

Caso de prueba a3: verificarCuenta • verificarPIN • depositar

Para considerar los colaboradores involucrados en esta prueba, se consideran los mensajes asociados con cada una de las operaciones indicadas en el caso de prueba a3. **Banco** debe colaborar con **Validacion** para ejecutar *verificarCuenta( )* y *verificarPIN( )*. **Banco** debe colaborar con **Cuenta** para ejecutar *depositar( )*. Por lo tanto, un nuevo caso de prueba que verifica estas colaboraciones es:

Caso de prueba a4: verificarCuenta [**Banco**: cuentaValida **Validacion**] • verificarPIN [**Banco**: pinValido **Validacion**] • depositar [**Banco**: deposito **Cuenta**]

El enfoque de prueba de partición de clase múltiple es similar al que se usó para la prueba de partición de clases individuales. Una sola clase se divide, como se estudió en la sección 5.2. Sin embargo, la secuencia de prueba se expande para incluir aquellas operaciones que se invocan mediante mensajes a clases que colaboran. Un enfoque alternativo divide las pruebas con base en las interfaces en una clase particular. Por ejemplo en la Figura 6-1, la clase **Banco** recibe mensajes de las clases **ATM** y **Cajero**. Por lo tanto, los métodos dentro de **Banco** pueden probarse al dividirlos en los que sirven

a **ATM** y los que sirven a **Cajero**. La partición basada en estado (sección 5.2) puede usarse para refinar aún más las particiones.

7 Apéndice: conceptos de orientación a objetos

¿Qué es un punto de vista orientado a objetos (OO)? ¿Por qué un método se considera orientado a objetos? ¿Qué es un objeto? Este apéndice se diseñó para ofrecer un breve panorama de este tema e introducir los conceptos y terminología básicos.

Para entender el punto de vista orientado a objetos, considere un ejemplo de un objeto del mundo real: el objeto sobre el cual se sienta en este momento, una silla. **Silla** es una ***subclase*** de una ***clase*** mucho más general que puede llamar **Mueble**. Las sillas individuales son miembros llamados ***instancias*** de la clase **Silla**. Se puede asociar un conjunto de ***atributos*** genéricos con cada objeto en la clase **Mueble**. Por ejemplo, todos los muebles tienen un costo, dimensiones, peso, ubicación y color, entre muchos posibles atributos. Lo mismo se aplica si se habla de una mesa, una silla, un sofá o un armario. Dado que la clase **Silla** es una subclase (o clase ***derivada***) de **Mueble**, **Silla** hereda todos los atributos definidos para la clase **Mueble**.

Recién se buscó una definición informal de una clase al describir sus atributos, pero algo falta. Todo objeto de la clase **Mueble** puede manipularse de varias formas. Puede comprarse y venderse, modificarse físicamente (por ejemplo, puede serruchar una pata o pintar el objeto de morado) o moverlo de un lugar a otro. Cada una de estas ***operaciones*** (otros términos son *servicios* o ***métodos***) modificarán uno o más atributos del objeto. Por ejemplo, si el atributo ubicación es un dato compuesto definido como

ubicación = edificio + piso + habitación

entonces una operación denominada *mover()* modificaría uno o más de los ítems de datos (**edificio, piso** o **habitación**) que forman el atributo **ubicación**. Para hacer esto, *mover()* debe tener "conocimiento" de dichos ítems de datos. La operación *mover()* podría usarse para una silla o una mesa, en tanto ambos sean instancias de la clase **Mueble**. Las operaciones válidas para la clase **Mueble** [*comprar()*, *vender()*, *peso()*] son especificadas como parte de la definición de clase y son heredadas por todas las instancias de la clase.

La clase **Silla** (y todos los objetos en general) encapsulan datos (los valores de atributo que definen la silla), operaciones (las acciones que se aplican para cambiar los atributos de silla), otros objetos incrustados, constantes y otra información relacionada. *Encapsulamiento* significa que toda esta información se empaqueta bajo un nombre y puede reutilizarse como un componente de especificación o programa.

Ahora que se introdujeron algunos conceptos básicos, una definición más formal de *orientado a objetos* resultará más significativa. Coad y Yourdon [Coa91] definen el término de esta forma:

Orientado a objetos = objetos + clasificación (en clases) + herencia + comunicación

Tres de los conceptos ya se introdujeron. La comunicación se estudia más tarde, en este apéndice.

7.1 Clases y objetos

Una clase es un concepto OO que encapsula los datos y los procedimientos (métodos) requeridos para describir el contenido y el comportamiento de alguna entidad del mundo real. Las abstracciones de datos que describen la clase se encierran mediante una "pared" de abstracciones procedurales (representadas en la Figura 7-1) que son capaces de manipular los datos de alguna forma. En una clase bien diseñada, **la única forma de llegar a los atributos (y operar sobre ellos) es ir a través de uno de los métodos** que forman la "pared" que se ilustra en la figura. Por tanto, la clase encapsula datos (dentro de la pared) y el procesamiento que los manipula (los métodos que constituyen la pared). Esto logra que la información se oculte y que se reduzca el impacto de los efectos colaterales asociados con el cambio. Puesto que la comunicación ocurre solamente a través de los métodos que constituyen la "pared", la clase tiende a estar menos fuertemente acoplada[6] a otros elementos de un sistema. Imagine que se modifica el tipo de datos de un atributo, basta con adaptar los métodos de la propia clase que lo manipulan. Las otras clases que interactúan no requieren modificarse dado que no se "enteran" del cambio gracias a este encapsulamiento.

[6] En sistemas, se dice que dos componentes están fuertemente acoplados cuando un cambio en la implementación interna de uno tiene fuerte impacto en el otro. En caso contrario se llaman débilmente acoplados

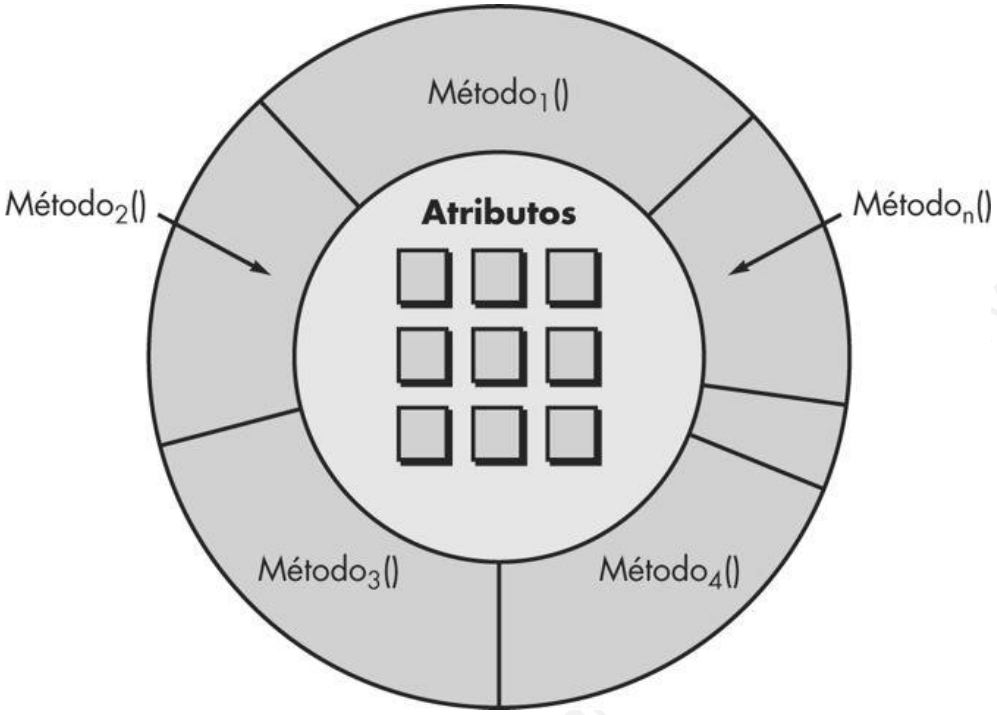


Figura 7-1. Representación esquemática de una clase

Dicho de otra forma, una clase es una descripción generalizada (como una plantilla) de una colección de objetos similares (instancias). Por definición, los objetos son instancias de una clase específica y heredan sus atributos y las operaciones que están disponibles para manipular esos atributos. Una ***superclase*** (también llamada clase ***base***) es una generalización de un conjunto de clases que se relacionan con ella. Una *subclase* (también llamada clase ***derivada*** o ***hija***) es una especialización de la superclase. Por ejemplo, la superclase **VehículoAutomotor** es una generalización de las clases **Camión**, **SUV**[7], **Automóvil** y **Van**. La subclase **Automóvil** hereda todos los atributos de **VehículoAutomotor**, pero además puede incorporar atributos adicionales que son específicos solamente de automóviles.

Estas definiciones implican la existencia de una jerarquía de clase en la que los atributos y operaciones de la superclase se heredan por parte de la subclase, que puede agregar atributos y métodos "propios" adicionales. Por ejemplo, las operaciones *sentarseEn()* y *voltear()* pueden ser exclusivas de la subclase **Silla**.

[7] Sport Utility Vehicle, automóvil todo terreno o camioneta 4x4

7.2 Atributos

Usted aprendió que los atributos describen la clase en alguna forma. Un atributo puede tomar un valor definido por un *dominio* enumerado. En la mayoría de los casos, un dominio es simplemente un conjunto de valores específicos. Por ejemplo, suponga que una clase **Automóvil** tiene un atributo **color**. El dominio de valores para **color** es {**blanco, negro, plata, gris, azul, rojo, amarillo, verde**}. En situaciones más complejas, el dominio puede ser una clase. Continuando con el ejemplo, la clase **Automóvil** también tiene un atributo **trenPotencia** que en sí mismo es una clase. La clase **TrenPotencia** contendría atributos que describen el motor y la transmisión específicos del vehículo.

Se puede asignar un valor por defecto a un atributo. Por ejemplo, el atributo **color** por defecto es **blanco**.

7.3 Operaciones, métodos o servicios

Un objeto encapsula datos (representados como una colección de atributos) y los algoritmos que los procesan. Dichos algoritmos se llaman *operaciones*, *métodos* o *servicios* y pueden verse como componentes de procesamiento.

Cada una de las operaciones que se encapsula mediante un objeto proporciona una representación de uno de los comportamientos del objeto. Por ejemplo, la operación *obtenerColor()* para el objeto de la clase **Automóvil** extraerá e informará el color almacenado en el atributo **color**. La implicación de la existencia de estas operaciones es que la clase **Automóvil** se diseñó para recibir un estímulo (a los estímulos se les llama ***mensajes***) que solicitan el color de la instancia particular de una clase. Siempre que un objeto recibe un estímulo, inicia cierto comportamiento. Esto puede ser tan simple como recuperar el color del automóvil o tan complejo como el inicio de una cadena de estímulos que pasan entre varios objetos diferentes. En el último caso, considere un ejemplo donde el estímulo inicial recibido por el **objeto_1** da como resultado la generación de otros dos estímulos que se envían al **objeto_2** y al **objeto_3**. Las operaciones encapsuladas por el segundo y tercer objeto actúan sobre los estímulos, y regresan información necesaria al primer objeto. Entonces el **objeto_1** usa la información devuelta para satisfacer el comportamiento demandado por el estímulo inicial.

7.4 Conceptos de análisis y diseño orientado a objetos

El modelado de requerimientos (también llamado ***análisis***) se enfoca principalmente en clases que se extraen directamente del enunciado del problema. Luego, durante el diseño pueden crearse clases adicionales.

7.4.1 Herencia

La herencia es uno de los diferenciadores clave entre sistemas convencionales y orientados a objetos. Una subclase **Y** hereda todos los atributos y operaciones asociadas con su superclase **X**. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para **X** están disponibles de inmediato para **Y**; no se necesita hacer más trabajo. La reutilización se logra directamente.

Cualquier cambio a los atributos u operaciones contenidos dentro de una superclase se hereda inmediatamente para todas las subclases. Por tanto, la jerarquía de clase se convierte en un mecanismo mediante el cual los cambios (en niveles superiores) pueden propagarse inmediatamente a través de un sistema.

Es importante observar que en cada nivel de la jerarquía de clase pueden agregarse nuevos atributos y operaciones a las que se heredaron de niveles superiores en la jerarquía. De hecho, siempre que se crea una nueva clase, tendrá algunas opciones:

- La clase puede diseñarse y construirse desde cero, es decir, no se usa herencia.
- La jerarquía de clase puede revisarse para determinar si una clase superior en la jerarquía contiene la mayoría de los atributos y operaciones requeridos. La nueva clase hereda de la clase superior y entonces pueden agregarse adiciones, según se requiera.
- La jerarquía de clase puede reestructurarse de modo que los atributos y operaciones requeridos puedan heredarse en la nueva clase.
- Las características de una clase existente pueden ser reemplazadas. Entonces se implementan diferentes versiones de todas o algunas operaciones para la nueva clase (derivada).

Como concepto fundamental de diseño, la herencia puede proporcionar beneficios significativos para el diseño, pero si se usa de manera inadecuada[8], puede complicar un diseño de manera innecesaria y conducir a software proclive a errores, que es difícil de mantener.

7.4.2 Mensajes

Las clases deben interactuar unas con otras para lograr los objetivos del diseño. Un mensaje genera algún comportamiento en el objeto receptor y se logra ejecutando una operación en dicho objeto receptor.

La interacción entre los objetos se ilustra de manera esquemática en la Figura 7-2. Una operación dentro de **ObjetoEmisor** genera un mensaje de la forma *mensaje*(<parámetros>) donde los parámetros identifican **ObjetoReceptor** como el objeto que se va a estimular mediante el mensaje; la operación dentro de **ObjetoReceptor** consiste en recibir el mensaje y los datos que proporcionan información requerida por la operación. La colaboración se define como parte del modelo de requerimientos.

Cox [Cox86] describe el intercambio entre clases en la forma siguiente:

A un objeto [miembro de una clase] se le solicita realizar una de sus operaciones al enviarle un mensaje que le diga qué hacer. El objeto responde al mensaje ejecutando la operación correspondiente y luego regresando el control al solicitante. Los mensajes conectan el sistema orientado a objetos. Proporcionan comprensión acerca del comportamiento de objetos individuales y del sistema como un todo.

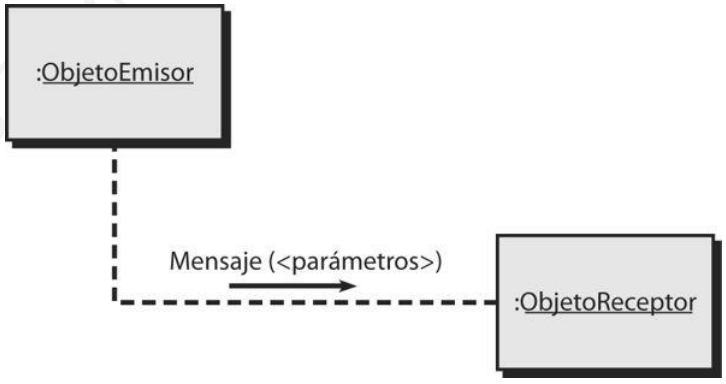


Figura 7-2. Mensaje que pasa entre objetos

[8] Por ejemplo, diseñar una subclase que herede atributos y operaciones de más de una superclase (herencia múltiple) es mal vista por la mayoría de los diseñadores.

Bibliografía utilizada y sugerida

Libros y otros manuscritos

- [Arl02] Arlow, J. y I. Neustadt. UML and the Unified Process. 2002.
- [Bin94a] Binder, R. Testing Object-Oriented Systems: A Status Report. American Programmer, vol. 7, Nº 4, abril 1994, pp. 23-28.
- [Boo05] Booch, G., Rumbaugh, J. & Jacobsen, I. The Unified Modeling Language User Guide. Second Edition. 2005.
- [Coa91] Coad, P. y E. Yourdon. Object-Oriented Analysis, 2a. ed. 1991.
- [Cop04] Copeland, Lee. A Practitioner's Guide to Software Test Design. 2004.
- [Cox86] Cox, Brad. Object-Oriented Programming. 1986.
- [Deb16] Debrauwer, L y F. Van Der Heyde. UML 2.5 Iniciación, ejemplos y ejercicios corregidos, 4ta ed. 2016.
- [Eve07] Everett, Gerald & McLeod, Raymond. Software Testing - Testing Across The Entire Software Development LifeCycle. 2007.
- [Far08] Farrell-Vinay, Peter. Manage Software Testing. 2008
- [Kir94] Kirani, S. y W. T. Tsai. Specification and Verification of Object-Oriented Programs, Technical Report TR 94-64, Computer Science Department, University of Minnesota. Diciembre 1994.
- [Pre15] Pressman, Roger y Bruce Maxim. Software Engineering: A Practitioner's Approach. Eighth Edition. 2015.

Lo que vimos:

En esta Unidad cubrimos las particularidades del testing de aplicaciones orientadas a objetos (OO). Mencionamos las limitaciones de los métodos tradicionales y propusimos métodos especializados.

Sin embargo, pudimos marcar ciertas similitudes entre los métodos OO y algunos métodos tradicionales.

Lo que viene:

En las próximas Unidades nos enfocaremos en el testing de aplicaciones Web, y luego completaremos el conocimiento con conceptos, métodos y herramientas que aún no han sido cubiertos.