1 Pruebas de unidad usando el Método de rebanadas

En el software orientado a objetos la unidad mínima que tiene sentido es la clase, ya que los métodos individuales no son independientes, sino que interactúan entre sí a través de los atributos. Así pues, las pruebas de unidad deberán aplicarse a clases completas. La prueba puede dirigirse con varios métodos, como el de rebanadas y el de máquinas de estados. Para clases de complejidad simple o media resulta más sencillo emplear el método de rebanadas, que se describe en este capítulo.

El método de rebanadas ha sido adaptado de la obra de Bashir y Goel, buscando adecuarlo a Java y eliminando ciertos aspectos que lo obscurecen. Sin embargo, se recomienda revisar el material de dichos autores.

1.1 Concepto de rebanadas

Cuando se habla de objetos se considera una abstracción definida por un nombre, un estado y un conjunto de operaciones o métodos que definen su comportamiento. El estado usualmente es dinámico, afectado por la realización de los diversos métodos. Todos los objetos de una misma clase comparten el con junto de métodos y la existencia de un conjunto de atributos que representarán el estado, el cual puede caracterizarse como un conjunto de atributos con sus respectivos valores en un instante dado.

Cuando se habla de probar una clase, en realidad no se puede hacer directamente; en vez de ello se prueba a través de instancias de la misma, es decir objetos, así que en lo que sigue todo lo que se diga de clase se refiere a objetos pertenecientes a la misma.

Cuando se realizan pruebas, se busca encontrar situaciones que rompan los supuestos que dan validez a la pieza de software que se está probando (o que confirmen que sí los cumple). En el caso de los objetos, además de la función que debe satisfacer cada método, se tiene el supuesto de que el estado es consistente sin importar el orden de ejecución de los métodos. Piense por ejemplo en una clase Cuenta que represente una cuenta bancaria con un atributo saldo y dos métodos deposita y retira; no importando cuántas veces se invoque a los métodos ni el orden en que se haga, el saldo debe ser consistente, es decir, que no se pierda ni aparezca dinero y que no se pueda retirar dinero no depositado. Así pues, las pruebas de unidad tendrán que verificar de alguna manera que esa consistencia existe para todas las combinaciones de ejecución de los métodos.

Ahora bien, el ejemplo mencionado es extremadamente simple y su estado consiste de un solo atributo; en clases más complejas el estado tendrá más

atributos y también existirán muchos métodos, volviendo impráctica la prueba de todas las combinaciones de ejecución de métodos. Por lo tanto, se buscarán formas de reducir el problema.

Al considerar el estado con detenimiento, puede observarse que no todos los métodos interactúan con todos los atributos, sino que cada método se relaciona con un subconjunto de éstos. Cambiando el enfoque, si se fija la atención en un atributo se tendrá que sólo un subconjunto de los métodos de la clase se relacionando una manera u otra con dicho atributo. De ahí surge la idea de "cortar" la clase de tal manera que en cada rebanada exista sólo un atributo y los métodos que se relacionan con éste. Claramente, cada rebanada será más fácil de probar que la clase entera y la combinación de rebanadas aún será inferior a la complejidad de probar toda la clase de una vez.

Formalmente podemos escribir que una clase está formada por una pareja

C = <A,M>,

donde **A** es un conjunto de atributos y **M** un conjunto de métodos. Si tomamos un atributo $\mathbf{a_i} \in \mathbf{A}$, tendremos que $\mathbf{M_i} \subseteq \mathbf{M}$ será el subconjunto de métodos que forman parte de la rebanada de $\mathbf{a_i}$.

El método de rebanadas consiste de dos grandes pasos:

1) Generar las rebanadas correspondientes a una clase
2) Para cada rebanada generar secuencias de métodos

Para la aplicación práctica del método se requieren algunos elementos adicionales, que se verán en la Sección 4, como el uso de cabos y manejadores para automatizar la ejecución.

1.2 Generación de rebanadas

Para generar las rebanadas se emplean dos estructuras de datos auxiliares: un Grafo de Llamadas Mejorado (GLLM) y una Matriz de Uso Mínima (MUM). A continuación se describen ambos y su relación.

1.2.1 Grafo de Llamadas Mejorado

Un Grafo de Llamadas Mejorado (GLLM) de una clase es un grafo constituido por uno o más árboles, donde cada árbol tiene como raíz un método de la clase y de ella parten ramas a los atributos a que hace referencia y a los métodos de la misma clase que invoca. Los métodos se representan como nodos circulares y los atributos como nodos cuadrados. Mientras sea posible se siguen expandiendo los nodos correspondientes a métodos. Nótese que los atributos son pasivos y por tanto siempre estarán en las hojas del árbol. En la Figura 7.1 se muestra uno de tales árboles.

El nombre de "grafo de llamadas" viene del hecho de representar las llamadas entre métodos; lo de mejorado porque incluye los atributos.

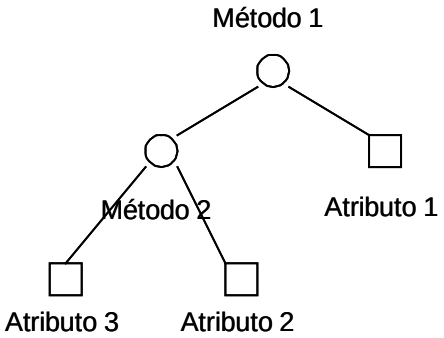

Figura 1 Grafo de llamadas mejorado

En los siguientes ejemplos se puede apreciar mejor la estructura del GLLM.

Ejemplo 1. Clase Cuenta.

Suponga que se tiene una clase que representa una cuenta bancaria un poco más desarrollada que la descrita en la sección anterior, como se muestra en la Figura 2.

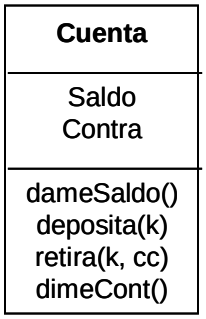

Figura 2 Clase Cuenta en UML

Brevemente descritos, los métodos tienen el siguiente comportamiento:

Cuenta()	El constructor, inicia saldo en cero
dimeSaldo()	Regresa el monto de saldo
dimeContra()	Regresa el valor de la contraseña (contra)
deposita(k)	Aumenta el saldo por una cantidad k
retira(k, cc)	Si la contraseña es igual a cc, entonces si el saldo es mayor o igual a k, retira dicha cantidad disminuyéndolo; regresa la cantidad retirada, cero si no alcanza y menos uno si la clave no corresponde.

En la Figura 3 se muestra el grafo de llamadas mejorado para esta clase.

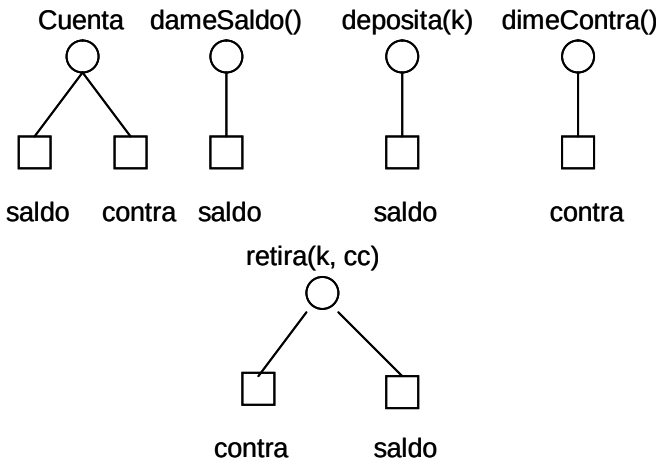

Figura 3 GLLM para la clase Cuenta

Ejemplo 2 Clase PoligonoRegular

En la Figura 4 se muestra la estructura de esta clase en UML y luego en la Figura 5 se muestra su GLLM. La descripción de los métodos es como sigue:

PoliginoRegular(a,l,n)	constructor; inicializa alto, lado y numlados
perimetro()	regresa el valor del perímetro, calculado como lado * numlados
area()	regresa el valor del área, calculada como (alto*lado/2)*numlados
areaCirc()	regresa el valor del área del círculo inscrito en el polígono, como: π * alto * alto / 2
dimeAlto()	regresa alto
dimeLado()	regresa lado
dimeNum()	regresa numlados

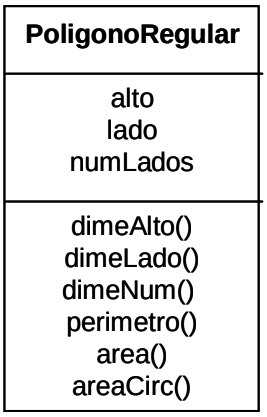

Figura 4 Clase PoligonoRegular en UML

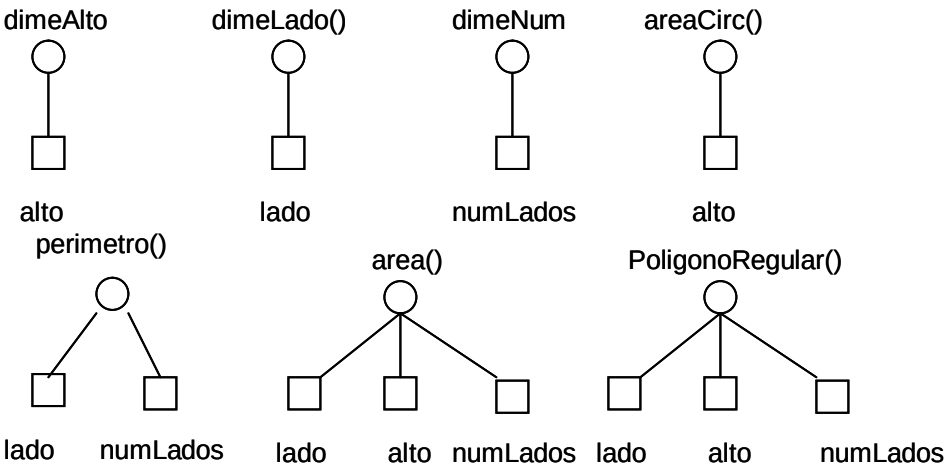

Figura 5 GLLM de la clase PoligonoRegular

Ejemplo 3 Clase CalcRacional

La clase CalcRacional representa una calculadora que opera sobre números racionales, es decir fracciones o quebrados. Así, cada número requiere un numerador y un denominador. Sólo se ilustran la suma y la multiplicación, para no complicar el ejemplo. Los métodos simplifica, maxcomdiv (máximo común divisor) y mincommult (mínimo común múltiplo) son auxiliares. En la Figura 6 se muestra la estructura de la clase y en la Figura 7 el GLLM correspondiente. Los métodos operan así:

CalcRacional()	Constructor, deja numerador y denominador indefinidos
carga(n,d)	Carga un número en forma de numerador y denominador, actualizando nume y deno
suma(n,d)	Suma el valor actualizando nume y deno; usa mincommult para calcular el nuevo denominador y simplifica para reducir la fracción
multiplica(n,d)	Multiplica el número almacenado por el nuevo; usa simplifica para reducir la fracción
mincommult(x,y)	Obtiene el mínimo común múltiplo de los números enteros x e y
maxcomdiv(x,y)	Obtiene el máximo común divisar de los números enteros x e y
simplifica()	Reduce la fracción a los menores valores posibles de numerador y denominador usando la función maxcomdiv; deja el resultado en nume y deno
dimeNume()	Regresa el numerador
dimeDeno()	Regresa el denominador
limpia()	Deja nume y deno en cero

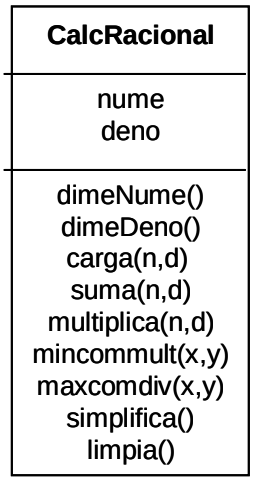

Figura 6 Clase CalcRacional en UML

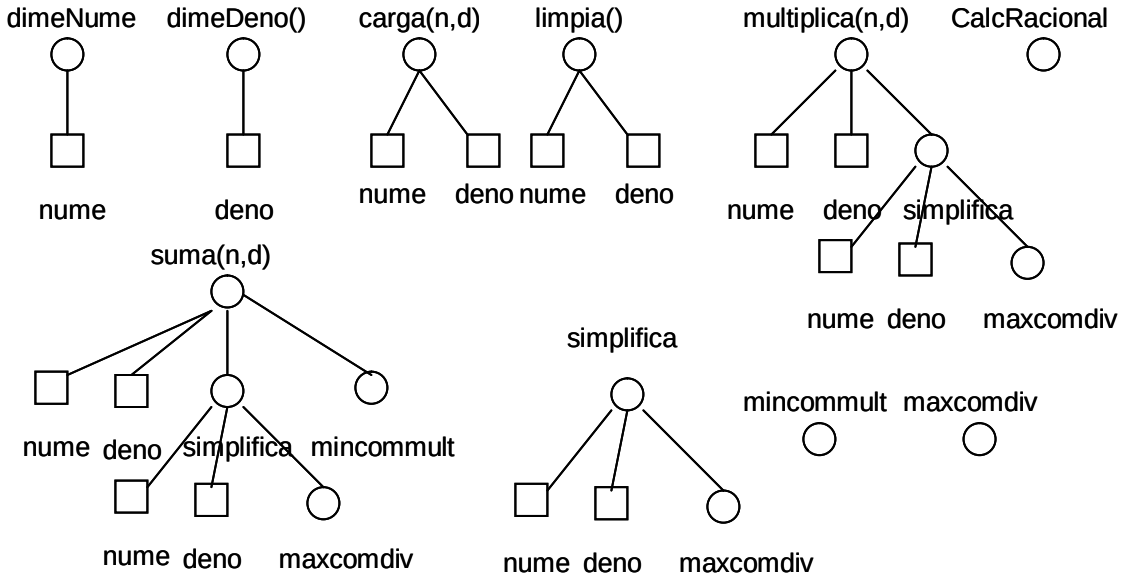

Figura 7 GLLM de la clase CalcRacional

1.2.2 Matriz de uso mínima (MUM)

La Matriz de Uso Mínima es una matriz que tiene una fila por cada atributo de una clase y una columna por cada método de la misma, incluyendo los constructores. La celda tendrá un contenido que depende de la relación entre el método j y el atributo i, como sigue:

MUM(i,j)

t	si el método j transforma o modifica el valor del atributo i
r	si el método j reporta o informa el valor del atributo i
o	si el método j utiliza el valor del atributo i en alguna operación, sin cambiarlo
Vacío	si no hay relación entre ellos

La MUM se puede construir directamente observando cada método y marcando el uso que hace de los atributos. Esta forma es útil para clases muy sencillas, con pocos atributos. Para clases más complejas y que incluyen invocación de métodos de la propia clase, es más conveniente construir primero el grafo GLLM.

1.2.3 Construcción de la MUM a partir del GLLM

Si se cuenta con el GLLM se construye la MUM como sigue:

Para cada árbol del grafo:
Para cada hoja atributo:
se marca la celda correspondiente a su fila y la columna del método que está en la raíz y junto se anota un número que indica el número de arcos entre la hoja y la raíz. Si el mismo atributo se conecta en más de un camino, se anotan todos.

Ejemplo de anotaciones: t1 (el método transforma al atributo y el camino es de un arco), o2 (el método usa el atributo y el camino tiene dos arcos); o1,o2 (existen dos usos del atributo, uno a un arco de distancia o directo, y otro a dos, es decir a través de una llamada a otro método).

Ejemplos completos.

Ejemplo 1 Clase Cuenta
A partir del grafo de la Figura 3, se construye la matriz de la Tabla 1, como sigue:

a) en el primer árbol los atributos saldo y contra son inicializados por el constructor Cuenta, en un solo arco, por lo que se anota t1 en cada uno.
b) en el segundo árbol, el valor de saldo es reportado por el método dameSaldo, por lo cual se anota r1.
c) en el tercero, el atributo saldo es afectado por el método deposita, anotándose t1.
d) en el cuarto, contra es reportado por el método dimeContra, anotándose r1.
e) en el quinto, saldo es modificado por el método retira, anotándose t1; contra es usado para comprobación, por lo que se marca o1.

Tabla 1 MUM de la clase Cuenta

Atributo	Cuenta	dameSaldo	deposita	dimeContra	retira
saldo	t1	r1	t1		t1
contra	t1			r1	o1

Ejemplo 2 Clase PoligonoRegular
En forma similar al ejemplo anterior, a partir del grafo de la Figura 5 se construye la matriz presentada en la Tabla 2:

Tabla 2 MUM de la clase PoligonoRegular

Atributo	Polígono Regular	dime Alto	dime Lado	dime Num	area Circ	perimetro	area
alto	t1	r1			o1		o1
lado	t1		r1			o1	o1
numLados	t1			r1		o1	o1

Ejemplo 3 Clase CalcRacional
A partir del grafo de la Figura 7 construimos la MUM que se muestra en la Tabla 3, para lo cual la mayoría de los árboles son similares a los de los ejemplos anteriores. Sin embargo, los árboles de los métodos suma y multiplica son más complejos. En el árbol de suma, el atributo nume es transformado por el método directamente (distancia 1) y también indirectamente por el método simplifica, con distancia 2; así pues habrá de anotarse t1,t2. Otro tanto ocurre con el atributo deno. La misma situación ocurre en el árbol del método multiplica.

Tabla 3 MUM de la clase CalcRacional

Atri-buto	Calc Racional	Dime Nume	Dime Deno	carga	Limpia	suma	multiplica	min com mult	max com div	simplifica
nume		r1		t1	t1	t1,t2	t1,t2			t1
deno			r1	t1	t1	t1,t2	t1,t2			t1

En la Tabla 3 note que la primera columna, correspondiente al constructor, está vacía, ya que no carga ningún dato.

1.2.4 Minimización de la matriz

La matriz MUM indica ser mínima, pero la matriz que obtuvimos aún no lo es, al menos no siempre lo es. Cuando todos los caminos son de longitud uno, la matriz es mínima; en cambio, cuando existen caminos de mayor longitud, es posible que se puedan eliminar algunos elementos, haciendo más sencilla la matriz y reduciendo el número de pruebas necesarias.

Si existen dos árboles como en la Figura 8, con la matriz correspondiente en Tabla 4, tenemos la siguiente situación:

- el atributo a2 es utilizado por el método me2, directamente, por lo cual se anota o1 en su columna;
- también es usado por me1, a través de la llamada a me2, es decir en dos pasos, por lo que se anota o2 en su columna
- si comparamos las dos columnas de la fila de a2, veremos que sólo difieren en el número; esto significa que una de ellas es redundante.

En consecuencia, dejamos únicamente la que corresponde a me2. La otra es una llamada repetida, ya que me1 no accede directamente a a2.

En cambio, para a1 las dos entradas son o1,o2 y o1; como se ve, son diferentes y deben ser probadas por separado. En este caso, me1 accede a a1 directamente y también a través de me2.

De aquí se puede establecer como regla:

- Para la fila de un atributo, si dos columnas tienen el mismo contenido en cuanto a tipo (o,r, t) y sólo difieren en el número, elimine aquella de número mayor.

Finalmente la matriz quedaría como en Tabla 5.

Para los ejemplos presentados no existe ninguna simplificación, por lo que ya son matrices mínimas.

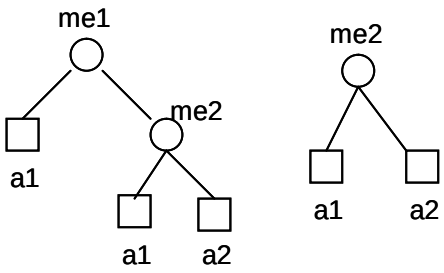

Figura 8 GLLM de ejemplo

Tabla 4 MUM para el ejemplo de Figura 8

Atributo	me1	me2
a1	o1,o2	o1
a2	o2	o1

Tabla 5 MUM minimizada correspondiente al ejemplo de Figura 8

Atributo	me1	me2
a1	o1,o2	o1
a2		o1

1.2.5 Revisión de la clase usando la MUM

La matriz MUM, además de su función en las pruebas, nos brinda una herramienta para revisar el diseño de la clase que representa. Algunos elementos que podemos extraer de ella son los siguientes:

Situación	Qué representa
Fila sin entradas	Muy posiblemente se viola el principio de ocultamiento de la información, ya que no se requiere un método para acceder al atributo. Otra posibilidad es que el atributo está sobrando. En todo caso conviene revisar el diseño.
Columna sin entradas	El método correspondiente sólo es un auxiliar de cálculo o bien no pertenece a la clase y debería acomodarse en otra parte. Conviene revisar diseño.
Las filas y columnas pueden reordenarse de modo de formar dos submatrices disjuntas	La clase en realidad contiene dos conceptos mezclados, uno con un grupo de atributos y métodos y otro separado, sin conexión. Debe rediseñarse separando en dos. Dejarla así es un error de Cohesión
Fila con más de 5 entradas de transformadores	Posiblemente se descompuso excesivamente alguno de los métodos. Revisar el diseño para balancear el número de métodos que operan sobre el atributo y la longitud de los mismos

A partir de esta información se puede diseñar mejores clases, que no caigan en los problemas mencionados. Los ejemplos de arriba no sufren de tales problemas.

1.3 Secuencias de métodos para prueba

Una vez que se cuenta con la matriz mínima de uso, se procederá a generar una serie de secuencias de prueba. Para esto se procede como sigue:

a) Se categoriza cada método como sigue: constructores, reporteros, transformadores y otros. La categoría transformadores incluye los métodos que realizan modificaciones sobre los atributos y también los que operan con sus valores, es decir los que se marcaron con “t” y con “o”. En otros se dejan los que no caen en ningún caso anterior, que se supone son excepciones, como métodos que reportan errores y mandan avisos auxiliares, sin relación con el estado del objeto.
b) Se prueba a los métodos reporteros. En este paso sólo interesa que reporten adecuadamente el valor del atributo que les corresponde. Si existe un atributo sin reportero, se tratará más abajo. Pueden generarse secuencias de reporteros al azar.
c) Se prueban los diversos constructores, con diferentes valores y se verifican éstos usando los reporteros. Debe cuidarse que los reporteros cubran todas las rebanadas.
d) Se prueban los transformadores como sigue:
    a. Se elige un atributo y los transformadores que corresponden a su rebanada y el reportero del atributo.
    b. Se forman las diversas permutaciones de los diferentes transformadores, colocando siempre el reportero al final
    c. Cada una de las secuencias anteriores será una secuencia de prueba; para cada una de ellas deberán considerarse varios casos de prueba, correspondientes a los diferentes tipos de valores que pueden tomar sus parámetros. Ésto se trata en la Sección 7.4.

e) Una vez probados los métodos anteriores, sólo quedan los métodos que no caen en ninguna de las categorías anteriores, pero que resultan más sencillos al no afectar ningún atributo.

Ejemplos

Clase Cuenta.

categoría	
reporteros	dameSaldo() y dimeContra()
constructores	Cuenta(k, c)
Rebanadas de transformadores	Rebanada de saldo: {deposita(k), retira(k,cc)} Rebanada de contra: {retira(k,cc)}
Secuencias de transformadores	deposita(k), retira(k,cc),dimeSaldo() retira(k,cc), deposita(k), dimeSaldo() retira(k,cc), dimeContra()
Otros	No hay

Clase PoligonoRegular.

categoría	
reporteros	dimeAlto(), dimeLado(), dimeNum()
constructores	PoligonoRegular(a, l, n)
Rebanadas de transformadores	Rebanada de alto: {areaCirc(), area()} Rebanada de lado: {perímetro(), area()} Rebanada de numLados: {perímetro(), area()}
Secuencias de transformadores	areaCirc(), area(), dimeAlto() area(), areaCirc(), dimeAlto() perimetro(), area(), dimeLado() area(), perimetro(), dimeLado() perimetro(), area(), dimeNum() area(), perimetro(), dimeNum()
Otros	No hay

Clase CalcRacional.

{Como ambas rebanadas son iguales y siempre se opera sobre los dos, usamos un solo juego de permutaciones con los dos reporteros al final.}

categoría	
reporteros	dimeNume() y dimeDeno()
constructores	CalcRacional()
Rebanadas de transformadores	Rebanada de nume: {carga(n,d), limpia(), suma(n,d), mult(n,d), simplifica()} Rebanada de deno: {carga(n,d), limpia(), suma(n,d), mult(n,d), simplifica()}
Secuencias de transformadores(¡)	carga(n,d), limpia(), suma(n,d), mult(n,d), simplifica(), dimeNume(), dimeDeno() carga(n,d), limpia(), suma(n,d), simplifica(), mult(n,d), dimeNume(), dimeDeno() carga(n,d), limpia(), simplifica(), suma(n,d), mult(n,d), dimeNume(), dimeDeno() carga(n,d), simplifica(), limpia(), suma(n,d), mult(n,d), dimeNume(),

	dimeDeno() simplifica(), carga(n,d), limpia(), suma(n,d), mult(n,d), dimeNume(), dimeDeno() carga(n,d), limpia(), mult(n,d), suma(n,d), simplifica(), dimeNume(), dimeDeno() carga(n,d), mult(n,d), limpia(), suma(n,d), simplifica(), dimeNume(), dimeDeno() mult(n,d), carga(n,d), limpia(), suma(n,d), simplifica(), dimeNume(), dimeDeno() carga(n,d), suma(n,d), limpia(), mult(n,d), simplifica(), dimeNume(), dimeDeno() suma(n,d), carga(n,d), limpia(),mult(n,d), simplifica(), dimeNume(), dimeDeno() limpia(), carga(n,d), suma(n,d), mult(n,d), simplifica(), dimeNume(), dimeDeno()....{hasta completar 120 secuencias, ya que corresponden a las permutaciones de los métodos que las forman}
Otros	No hay

El procedimiento anterior suena largo, pero sin complicaciones. Sin embargo, la programación orientada a objetos introduce algunos problemas que afectan a la prueba. Por el momento se tratarán dos:

1. Carencia de reportero. Un atributo puede carecer de reportero, lo cual acarrea problemas para aplicar las pruebas como se ha descrito. Si el atributo es privado, será imposible probar adecuadamente la clase, a menos que se modifique el código. Si el atributo es protegido (friend), se puede generar una clase derivada de la clase que se está probando, agregando los reporteros necesarios, lo cual no altera la clase original.
2. Conexiones con otras clases. Una clase no existe de manera aislada; generalmente es parte de un conjunto que realizan una cierta funcionalidad deseada por un usuario. Esto hace que existan un gran número de interacciones en la forma de invocaciones a métodos de diversas clases. Sin embargo, en la prueba de unidad se busca probar unidades aisladas, sin interacción. Así pues, se requiere sustituir las clases invocadas y también las que invocan a la clase que se prueba. Esto se hace con clases de utilería, que contienen métodos auxiliares, pero sin código específico del sistema. Los métodos de clase invocadas se llaman cabos (stubs) y su única función es regresar un resultado (y valor) que resulte típico. Las clases que llaman a la clase bajo prueba se llaman Manejadoras (drivers) y se concretan a invocarla. En la sección 7.5 se describen auxiliares automáticos para estos trabajos.

Para garantizar que no ocurran problemas como el de la carencia de reporteros, existen una serie de recomendaciones de diseño que se conocen generalmente como diseño facilitador de pruebas (“design for testability”), entre las cuales podemos citar:

Siempre que escriba una clase, haga privados todos los atributos y escriba un reportero para cada uno.

1.4 Generación de casos de prueba

Una vez que se tiene una secuencia de pruebas, con la forma que sigue, se debe ejecutar.

Transformador 1 – Transformador 2 - … - Transformador n – Reportero

En este momento surge una complicación: un método transformador usualmente tiene parámetros y por tanto debe decidirse qué valor se dará a éstos. En casos simples, cualquier valor que se de al parámetro origina el mismo tipo de respuesta; por ejemplo, un método ***deposita(k)*** esperamos que agregue el valor de k al de saldo, no importa cuál sea el valor de k. Sin embargo, en otros casos no es así; por ejemplo, un método que recibe un valor entero positivo y regresa el logaritmo de ese número, en caso de recibir un número negativo deberá regresar un error; así pues hay dos casos.

Para asegurarse de haber probado adecuadamente cada transformador, deben cubrirse todos los casos. Para eso existen varios métodos, estudiados en otra parte. Por el momento, recomendamos usar el de dominios (particiones) y valores a la frontera. Los diferentes casos se reemplazarán donde aparezca el transformador en cada secuencia de prueba. Note que ésto aumenta considerablemente los casos de prueba que deben ejecutarse.

Como el proceso es muy laborioso y bastante aburrido, se debe preferir el uso de herramientas automáticas, como se describe en la sección siguiente.

Aplicación a los ejemplos.

Clase Cuenta:

Aquí los dos métodos son indiferentes al número que se use, así que bastará emplear un valor típico y, para más seguridad, un valor negativo.

Clase PoligonoRegular:

Aquí no hay problema pues no hay parámetros en los métodos transformadores y por tanto bastará un caso de prueba para cada uno.

Clase CalcRacional:

En carga, suma y mult se pueden presentar dos casos: el de un número racional normal (una fracción x) y un número con denominador cero, así que las 120 secuencias se convierten en muchas más. Es posible que se puedan reducir algunas, pues el caso del denominador cero es una excepción.

1.5 Herramientas auxiliares

De los elementos presentados anteriormente, se tiene que para probar una clase se requiere:

a) clases de utilería con cabos para los métodos que puedan ser llamados, capaces de regresar un valor típico
b) una clase manejadora que dirija la prueba

Además se requiere generar las permutaciones de secuencias de métodos, lo cual puede hacerlo una un método dentro de la clase manejadora.

De todo ésto lo más difícil es la clase manejadora, quien deberá llevar control de secuencias, reportar fallas, etc.

En el mercado existen diversas herramientas que automatizan parcialmente el proceso y también existen herramientas de software libre de muy buena calidad. Entre éstas se tiene a JUnit, creada para programas escritos en Java y que se puede bajar libremente; además se encuentra integrada en la plataforma Eclipse, también de software libre, así como en NetBeans, que se puede bajar gratis, aunque sea propiedad de SUN.

A partir de JUnit se han desarrollado herramientas equivalentes para otros lenguajes, como DUnit, que se adapta a la plataforma Delphi, CUnit para probar programas escritos en C++, etc.

Estas herramientas proporcionan clases básicas para probar clases, permitiendo preparar la prueba (crear objetos, preparar el estado, etc.), aplicar la prueba, reportar resultados y limpiar estado al final

Si no se cuenta con una de estas herramientas y no se desea obtenerla, deberá desarrollarse software auxiliar que la reemplace. Los autores Bashir y Goel [BAS 2000] recomiendan crear una clase auxiliar que hereda de la clase bajo prueba y realiza varias funciones auxiliares. Para más detalles, ver su obra.

1.6 Prueba de clases derivadas (subclases)

Un concepto de gran importancia en la programación orientada a objetos es la herencia. Las clases forman árboles de jerarquía donde unas clases son más generales y otras más particulares. Las clases particulares heredan los atributos y comportamiento generales de una clase base y lo extienden o modifican para darle características diferentes.

Para probar una clase derivada de otra que ya se probó, podría hacerse como si se careciera de información, aplicando el método como se vio en las secciones 7.2 a 7.4. Sin embargo, una de las ventajas de la herencia es el no repetir cosas que ya están hechas. Así pues, se busca aprovechar el trabajo anterior sin repetirlo.

En una clase derivada se presentan cuatro posibles escenarios:

i) métodos nuevos[1] o redefinidos que afectan atributos de la clase base
ii) métodos de la clase base que afectan atributos de la clase base
iii) métodos de la clase base que afectan atributos nuevos
iv) métodos nuevos o redefinidos que afectan atributos nuevos

De los cuatro casos sólo el segundo se puede considerar libre de nuevas pruebas. Los otros dependen de la existencia de algún cambio que corresponda a ellos. El más improbable, pero no imposible, es el caso tres, ya que los métodos preexistentes no pueden actuar directamente sobre datos nuevos, a menos que exista una llamada indirecta a través de un método redefinido.

El procedimiento de prueba para una clase derivada utiliza las mismas ideas de las rebanadas, con algunas modificaciones. Puede resumirse así:

1. construir el Grafo de Llamadas Mejorado (GLLM)
2. construir la matriz MUM modificada
3. probar las rebanadas correspondientes a atributos nuevos
4. probar las rebanadas correspondientes a atributos originales que hayan sufrido cambios

Para ejemplificar utilizaremos una clase derivada de la clase Cuenta definida anteriormente, como se muestra en la Figura 9 y que se define abajo.

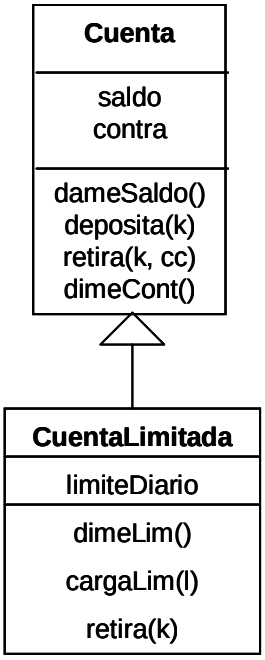

Figura 9 Clase CuentaLimitada

La clase CuentaLimitada es en todo similar a Cuenta, excepto que tiene un límite de retiro por día. Cada día se carga el limiteDiario usando el método cargaLim(l).

[1] Se entiende como nuevo método y nuevo atributo aquellos que son propios de una clase hijo o subclase, es decir, que heredan de otra previamente definida y probada.

Durante el día, si se retira una cantidad menor o igual al límite y aceptable por el saldo, se realiza y se disminuye el límite. Si se pretende retirar más del límite, se rechaza. Así pues, se debe modificar el método retira y supondremos que se reescribió completamente. También se agregará un método dimeLim() que reporta el límite en ese momento.

1.6.1 Grafo de llamadas mejorado

En forma similar al método básico, se construye el GLLM, con los siguientes cambios: cuando se invoca a un atributo de la clase original, se marcará con un cuadrado doble y cuando se invoque a un método de la clase base, se marcará con un círculo doble. En la Figura 10 se muestra el GLLM para CuentaLimitada.

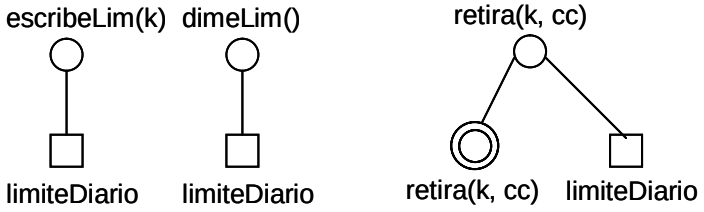

Figura 10 GLLM para la clase CuentaLimitada

Puede observarse en la Figura 10 que el método retira (nuevo) hace una llamada al método antiguo (con doble círculo).

1.6.2 Construir MUM modificada

El siguiente paso es construir la MUM, para lo cual se formará en cuatro cuadrantes, como se muestra en la Tabla 4, indicados con números romanos. El cuadrante II es idéntico a la MUM de la clase base y el cuadrante IV corresponde a las interacciones entre métodos y atributos nuevos. Los otros dos corresponden a interacciones entre métodos nuevos y atributos antiguos y viceversa.

Tabla 4 MUM modificada

	Métodos de clase base	Métodos nuevos
Atributos de clase base	**II**	**I**
Atributos nuevos	**III**	**IV**

El llenado de la MUM debe realizarse como sigue:

a) primero se llena el cuadrante II con los datos de la clase base;
b) después se llena el cuadrante IV, que es totalmente nuevo
c) después se llena el cuadrante III que probablemente esté vacío
d) finalmente se llena el cuadrante I

Para el ejemplo de la clase CuentaLimitada se obtendrá una MUM como la de la Tabla 5. Las líneas obscuras sirven únicamente para indicar los cuatro cuadrantes y en aplicaciones prácticas no se usan.

Tabla 5 MUM de la clase CuentaLimitada

	CuentaLimitada	**dameSaldo**	**deposita**	**dimeContra**	**retira**	**dimeLim**	**cargaLim**	**retira**
saldo	t1	r1	t1		t1			t2
contra	t1			r1	o1			o2
limite Diario						r1	t1	o1

En la Tabla 5 se puede observar que el cuadrante III quedó vacío, lo cual es el caso más común.

1.6.3 Generación de rebanadas y secuencias

Teniendo la MUM de la clase derivada, se procede a generar las rebanadas de los atributos agregados y a partir de ellos las secuencias de prueba. Para esto se usan los cuadrantes inferiores (III y IV).

Una vez terminado el paso anterior, se procede a analizar las rebanadas de la clase base y observar si hubo cambios. Si los hay, entonces se deben generar las secuencias de prueba igual que antes. Para las rebanadas que no sufrieron cambio no es necesario hacer nada. En esta etapa se usan los cuadrantes superiores (I y II).

Para la clase de ejemplo, se tiene:

Con los sectores III y IV:
Para el atributo limiteDiario sólo hay un método transformador, cargaLim(), así que bastará probarlo seguido del reportero dimeLim().

Con los sectores I y II:
Sólo hay cambios de transformadores en el atributo saldo y sólo es un nuevo método, que reemplaza al anterior, por lo cual podemos reemplazarlo en las secuencias que ya se habían generado en la clase original.