UNIVERSIDAD DE CHILE
DEPARTAMENTO DE CIENCIAS DE COMPUTACION

Diseño por Contratos y Aserciones

Bertrand Meyer. Construcción de Software Orientado a Objetos

EDUARDO JARA

Diseño por Contratos y Aserciones

El Diseño por Contrato ve las relaciones entre las clases y sus clientes como un acuerdo formal, que expresa los derechos y obligaciones de cada parte.

```
class ACCOUNT feature

  balance:INTEGER;
  owner:PERSON;
  minimun_balance:INTEGER is 100;

  open (who:PERSON) is
       do     owner:=who    end;

  deposit (sum:INTEGER) is
       do     add(sum)        end;

  withdraw (sum:INTEGER) is
      do      add(-sum)       end;

  may_withdraw (sum:INTEGER):BOOLEAN is
      do
         Result:=(balance >= sum + minimum_balance)
      end

feature {NONE}
  add (sum:INTEGER) is
     do     balance:= balance + sum     end

end
```

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Diseño por Contratos y Aserciones

Una *fórmula de corrección* (o tripleta de Horn) es una expresión de la forma:

$$\{P\}\ A\ \{Q\}$$

Donde *P* es la *precondición* y *Q* es la *postcondición*.

"Una ejecución de A que comience en un estado en el que se cumple P terminará en un estado en el que se cumple Q"

Ejemplo: $\{x \geq 9\}\ x := x + 5\ \{x \geq 14\}$

Diseño por Contratos y Aserciones

Condiciones fuertes y débiles

Se dice que *P1* es más fuerte que *P2* (y *P2* es más débil que *P1*), si *P1* implica a *P2* y no son iguales. Luego *Falso* es la proposición más fuerte de todas, y *Verdadero* es la proposición más débil.

Ejemplo :

Debilitar la postcondición $\{x \geq 9\}\ x := x + 5\ \{x \geq 13\}$

Fortalecer la precondición $\{x \geq 10\}\ x := x + 5\ \{x \geq 14\}$

Diseño por Contratos y Aserciones

Condiciones fuertes y débiles

Supongamos que un amigo está buscando trabajo y analiza varios anuncios, todos con salarios y beneficios similares, pero que difieren en su *P* y *Q*. Como cualquier otro, su amigo es perezoso, es decir, desea obtener el trabajo más fácil posible. Está pidiéndonos consejo. Qué es lo que se le debería recomendar para P. ¿Escoger un trabajo con una precondición *débil* o *fuerte*?

Diseño por Contratos y Aserciones

Panacea 1: {*Falso*} *A* {*Q*}
El mejor trabajo posible.¿Por qué?

Panacea 2: {*P*} *A* {*Verdadero*}
El segundo mejor trabajo posible.¿Por qué?

Este análisis ha sido hecho desde el punto de vista del empleado. ¿Qué pasa si cambiamos de “bando”?

Diseño por Contratos y Aserciones

Desde la perspectiva del empresario, todo se invierte:

- **➔ una precondición débil será una buena noticia, ya que significa un trabajo que tratará un amplio espectro de casos de entrada;**
- **➔una postcondición más fuerte quiere decir resultados más significativos.**

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Matemáticamente la noción más cercana a ***aserción*** es la de ***predicado***, aunque el lenguaje de aserciones tiene sólo parte de la potencia del cálculo de predicados completo.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Algunas consideraciones sintácticas.

Las expresiones:

n > 0 and x /= void ó

n > 0; x /= void ó

Positivo: n > 0
No_vacio: x /= void

son todas equivalentes.

La expresión:

balance = old balance - sum

significa que una vez terminado el procedimiento withdraw el valor de balance será el valor que tenía al iniciarse la ejecución del procedimiento menos el valor de sum.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Las precondiciones y postcondiciones se expresan como cláusulas de las declaraciones de las rutinas introducidas por las palabras clave requiere (requiere) y ensure (asegura) respectivamente.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Una **precondición** expresa las restricciones bajo las que una rutina funcionará correctamente. Por ejemplo:

➔ No se puede llamar a deposit si sum es menor a cero.

➔ No se puede crear una cuenta si el depósito inicial (initial) es menor en el balance mínimo exigido (minimum_balance).

La precondición se aplica a todas las llamadas a la rutina, tanto desde dentro de la clase como desde los clientes.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Una **postcondición** expresa proiedades del estado resultante de la ejecución de una rutina. Por ejemplo:

➔ después de ejecutar withdraw el balance es menor a su valor inicial en el monto sum.

La postcondición se aplica a todas las llamadas a la rutina, tanto desde dentro de la clase como desde los clientes.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

➔ La violación en tiempo de ejecución de una aserción es la manifestaciónn de un error (bug) en el software.

➔La violación de una precondición es la manifestación de un error en el cliente.

➔ La violación de una postcondición es la manifestación de un error en el proveedor.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

➔ Las **invariantes** expresan las propiedades globales de las instancias de una clase, que deben ser preservadas por todas las rutinas.

➔ Las **invariantes** capturan las propiedades semánticas más profundas y las restricciones de integridad que caracterizan a una clase.

```
class ACCOUNT creation
      make
  feature
  …Attributes as before:
  … balanc, minimum_balance, owner
  open … -- as before;
  deposit (sum:INTEGER) is
     require    sum >= 0
     do          add(sum)
     ensure    balance = old balance + sum
     end;

  withdraw (sum:INTEGER) is
     require    sum >= 0;
                sum <= balance - minimum_balance
     do          add(-sum)
     ensure   balance = old balance - sum
     end;

     may_withdraw … -- as before

  feature {NONE}
  add … -- as before

   make (initial:INTEGER) is
     require    initial >= minimum_balance
     do          balance := initial
     end

  invariant
     balance >= minimum_balance
end
```

Una clase es correcta respecto a sus aserciones si y sólo si:

➔ Para todo conjunto válido de argumentos x_p de un procedimiento de creación p:

$\{Por\_omisión_c$ **and** $pre_p(x_p)\}$

$Cuerpo_p$

$\{post_p\ (x_p)$ **and** $INV\}$

➔ Para cada rutina exportada r y cualquier conjunto x_r de argumentos válidos:

$\{pre_r(x_r)$ **and** $INV\}$

$Cuerpo_r$

$\{post_r\ (x_r)$ **and** $INV\}$

UNIVERSIDAD DE CHILE
DEPARTAMENTO DE CIENCIAS DE COMPUTACION

Diseño por Contratos y Aserciones

FIN

EDUARDO JARA