Departamento de Informática | Universidad de Valladolid

Programación III.I.T.I. de Sistemas

Programación bajo contrato

Félix Prieto

Curso 2007/08

Contenidos

Introducción

- Deseamos elaborar software de calidad ...
  - Diseñando y codificando con un enfoque adecuado
  - Aplicando los mecanismos básicos del lenguaje
  - Aplicando técnicas de Verificación Formal donde sea posible
  - Aplicando técnicas adecuadas de Validación
- ... sin embargo sabemos que *el software va a fallar*
- Estamos obligados a *minimizar los efectos* que provoca el fallo en el software que construimos

Normas para diseño y codificación

- Diseños simples, modulares y extensibles
  - El mayor enemigo de la fiabilidad es la complejidad
- Codificación elegante y legible
  - La claridad y simplicidad de las construcciones del lenguaje ayudan
  - El software se escribe para ser leído
- Respeto a los principios, reglas y criterios de modularidad
  - Tanto el lenguaje como el método de diseño y el programador deben atender a estos principios

Mecanismos básicos del lenguaje

- Delegación del manejo de la memoria en un mecanismo automático
  - El acceso manual a la memoria dinámica puede ser divertido, pero también peligroso
- Comprobación estática de tipos
  - Evita errores provocados por el mal uso de los tipos
  - Es importante disponer de un adecuado soporte para la genericidad
- Técnicas para facilitar la construcción de software correcto y robusto

Un ejemplo con dificultades

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el denominador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      create Result.make(denominador,numerador)
    end; -- inverso
...
```


Solución clásica

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el numerador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      if numerador /= 0 then
        create Result.make(denominador,numerador)
        else -- Pero ¿qué hacemos ahora?
      end
    end; -- inverso
```


Solución clásica: El silencioso

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el numerador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      if numerador /= 0 then
        create Result.make(denominador,numerador)
      end
    end; -- inverso
```

Los problemas del silencioso

- El objeto ignora la llamada cuando no es adecuada
- Es probable que sólo pospongamos el error, alejando el síntoma de su causa y dificultando el proceso de depuración
- El cliente deberá comprobar por su cuenta si su solicitud ha sido aceptada. En situaciones más complejas puede no conocer el motivo del rechazo
- La lectura del código se hace imprescindible para comprender la forma en que debemos usar el método

Solución clásica: El escándalo inútil

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el numerador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      if numerador /= 0 then
        create Result.make(denominador,numerador)
      else print("ERROR: mi numerador es cero")
      end
    end; -- inverso
```


Los problemas del escandaloso

- El cliente, el objeto que solicitó el servicio, se encuentra en la misma situación que antes pues no es notificado del error
- El código se hace dependiente del entorno de ejecución:¿Porqué escribir el mensaje en consola?
- El usuario recibe una información inútil: Es muy probable que se pregunte qué es el numerador del programa y porqué es un error que valga cero
- Como antes la lectura del código es imprescindible para ser cliente de este número racional

Solución clásica: El pánico

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el numerador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      if numerador /= 0 then
        create Result.make(denominador,numerador)
      else
          print("ERROR: mi numerador es cero")
          crash
          -- o bien die_with_code(exit_failure_code)

      end
    end; -- inverso
```


Los problemas del pánico

- La parada del sistema es sólo adecuada en situaciones realmente irrecuperables
- Seguimos dependiendo del contexto de ejecución. No siempre el usuario recibirá el mensaje de error
- La lectura del código sigue siendo imprescindible para ser cliente de esta versión del racional

Solución clásica:El código de error

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el numerador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      if numerador /= 0 then
        create Result.make(denominador,numerador)
      else ultimo_error:=codigo_division_por_cero
      end
    end; -- inverso
```


Los problemas del código de error

- El cliente debe consultar el código de error tras cada operación
- Si la solución se generaliza se complica el uso y mantenimiento de lo relacionado con los códigos de error
- Se requiere de la buena voluntad del programador del código cliente para evitar el problema original
- La lectura del código de racional sigue siendo muy recomendable para los programadores de sus clientes

Otra solución: Tratamiento de excepciones

```
class RACIONAL
...
  inverso: RACIONAL is
        -- Número que multiplicado por Current produce el uno.
        -- Evidentemente no tiene sentido si el numerador
        -- vale cero.
        -- ¿Cómo protegernos ante una llamada
        -- errónea como r.cero.inverso ?
    do
      if numerador /= 0 then
        create Result.make(denominador,numerador)
      else
          -- llamada al encargado de tratar esta excepción
      end
    end; -- inverso
```

Sobre el tratamiento de excepciones

- El lenguaje debe permitir que lancemos una excepción
- Hay que determinar claramente quién será el responsable de tratar la excepción y qué política de ejecución se sigue después
- En función del lenguaje podemos llegar a saber que el método puede lanzar una excepción, pero para conocer los motivos de la misma hay que leer el código del método

En resumen...

- Necesitamos compatibilizar el conocimiento del comportamiento del método con el principio de ocultación de información
  - Necesitamos saber *qué hace*, pero *no cómo lo hace*
- Debe existir un esquema claro de asignación de responsabilidades en situaciones de error
- La *metáfora del contrato* permite resolver ambos problemas de forma razonable
- La metáfora del contrato se basa en las técnicas de verificación de programas

Corrección del software

- Un algoritmo o programa es *parcialmente correcto* con respecto de una especificación si, comenzando en un estado que satisface su precondición, y terminando, lo hace en un estado que satisface su postcondición
  - La corrección es siempre relativa a la especificación
  - Nos interesa elegir la precondición más débil y la postcondición más fuerte que sea posible
- Un algoritmo o programa es *totalmente correcto* si es parcialmente correcto y finito.
  - La corrección parcial es un concepto intermedio que facilita las demostraciones
  - Podemos hacer correcto cualquier algoritmo sin mas que elegir adecuadamente la especificación

Equipar el software con su especificación

- Dotando al software de especificación podemos...
  - Mostrar los motivos que nos hacen pensar que el software es correcto
  - Facilitar la comprensión del problema y su solución
  - Mejorar la autodocumentación del software
  - Fijar la base para la posterior depuración
  - Proteger, en cierta medida, el software de usos inadecuados
- Pero...
  - La especificación no debe formar parte de las estructuras de control que utilizamos
  - Los contratos separan lo *correcto* de lo *robusto*

Programación bajo contrato

- Sistema para equipar al software con sus especificaciones mediante *asertos*
- Los asertos declaran las condiciones de corrección como parte del programa y de su documentación
- Un aserto es una expresión booleana que puede cumplirse o no en determinado punto del programa
- Los asertos no tienen porqué ralentizar la ejecución del programa, puesto que pueden ser compilados o no. La decisión es tomada en el momento de compilación (para más información ver `compile -help`)

La metáfora del contrato

- Dos partes implicadas: *Cliente* y *proveedor*
- Un *servicio*: Requerido por el cliente, proporcionado por el proveedor
- Dos tipos de asertos:
  - *Precondición:* Condiciones en que el proveedor puede prestar el servicio (`require`). Deben ser cumplidas por el cliente
  - *Postcondición:* Garantías que obtiene el cliente sobre la calidad del servicio (`ensure`). Deben ser cumplidas por el proveedor

El racional con su contrato

```
class RACIONAL
...
  inverso: RACIONAL is
      -- Número que multiplicado por Current produce el uno
    require
      no_nulo: not equal(Current, Current.cero)
    do
      create Result.make(denominador,numerador)
    ensure
      equal(Current.uno,Current*Result)
    end
  end; -- inverso
```


Consideraciones sobre los contratos

- Las clausulas `require` y `ensure` aparecen en la forma corta de la clase
- El incumplimiento del contrato puede activar o no una excepción (en función de las opciones de compilación)
- pero podemos intentar controlarlo procesando las excepciones
- El cuerpo de una rutina *no comprobará en ningún caso* que se cumple su precondición
- El cliente *no comprobará en ningún caso* que se cumple la postcondición de su proveedor
- *Los asertos no son*:
  - Un sistema de chequeo de la entrada
  - Una nueva estructura de control

Consideraciones sobre los contratos (y II)

- Cada precondición de una rutina debe satisfacer los siguientes requisitos:
  - Debe ser justificable únicamente en términos de la especificación
  - Cada característica que aparece en la precondición del método debe estar exportada a todos los clientes potenciales del método
- En las postcondiciones podemos utilizar la construcción `old()` que hace referencia a la versión original de la entidad (Por ejemplo `ensure i= old(i)+1`)
- Disponemos de expresiones booleanas como `and then`, `or else` e `implies`

Imperativo vs. Aplicativo

`do; i:=i+1`	`ensure; i=old(i)+1`
Cómo	Qué
Operacional	Denotacional
Implementación	Especificación
Orden	Consulta
Imperativo	Aplicativo
Instrucción	Aserto

Invariantes de clase

- Asertos situados al final de la clase tras la cláusula `invariant`
- Si el el invariante no se cumple *puede disparase* una excepción
- Son precondiciones para todos los métodos públicos de la clase
- Son postcondiciones para todos los métodos públicos de la clase
- Marcan las restricciones de *coherencia de la clase*
- Los *métodos de creación* tienen por objetivo llevar a un estado que satisface el invariante
- Todo método público que parte de un estado que satisface el invariante tiene la obligación de llevar el objeto a otro estado que satisface el invariante

Ejemplo de invariante

```
class RACIONAL
...
invariant
  denominador_no_nulo: denominador /= 0
  reducido1: denominador >0
  reducido2: numerador.abs.gcd(denominador)=1
  reducido3: numerador=0 implies denominador=1
end; -- class RACIONAL
```


Asertos en el código

```
x:=x^2+y^2
-- Código para despistar
check
  x>=0
  -- Es suma de cuadrados
end
y:=x.sqrt
```

- *Afirmamos* que se cumple un aserto
- Si el aserto no se cumple *puede dispararse* una excepción
- Requiere de un comentario explícito
- Facilita la comprensión del código
- Facilita la depuración del código

Invariante y variante de bucle

```
from
  -- Órdenes de inicialización
invariant
  -- Invariante del bucle
variant
  -- Sucesión de cota
until
  -- Condición de salida
loop
  -- Cuerpo de la iteración
end
```

- `invariant` y `variant` son optativos
- Si el aserto no se cumple *puede dispararse* una excepción
- Requiere de un comentario explícito
- Facilita la comprensión del código
- Facilita la depuración del código

Ejemplo de iteración

```
division(x,y: INTEGER): INTEGER is
  -- Cociente de la división entera de x entre y
  require
    parametros_positivos: x>0; y>0
  local
    q,r: INTEGER;
  do
    from r:= x
    invariant x=q*y+r; r>0
    variant r
    until r<y
    loop
      r:= r-y
      q:= q+1
    end -- loop
    Result:=q
  ensure
    resto_en_rango: r<y; r>0
    propiedad_cociente: x=q*y+r
  end
```

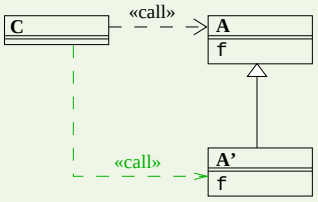

Otro ejemplo de iteración

```
mcd(x,y: INTEGER): INTEGER is
  -- Máximo común divisor de x e y
  require
    parametros_positivos: x>0; y>0
  local
    a,b: INTEGER;
  do
    from a:= x;b:= y
    invariant a>0; b>0; mcd(a,b)=mcd(x,y)
    variant a.max(b)
    until a=b
    loop
      if a>b then a:=a-b
        else b:=b-a
      end
    end
    Result:= a
  ensure -- ¿Sabemos si termina?
    result=mcd(x,y)
  end
```

Comentarios sobre los asertos

- Algunos asertos son difíciles o imposibles de escribir
  - Comentarios que explican el aserto que nos gustaría escribir
- Los asertos pueden contener llamadas a métodos, siempre que se trate de consultas que no modifiquen el objeto
- La ejecución de un método en la comprobación de un aserto no provoca comprobación de nuevos asertos
  - El peligro de la recursión infinita
  - ¿Quién vigila al vigilante?
- La mejor inspiración para construir un buen contrato es leer los contratos de buenas bibliotecas
- Existen opciones de compilación para activar, total o parcialmente, o incluso desactivar el chequeo de asertos (`compile -help`)

Clientes indirectos



- El polimorfismo permite que seamos clientes del hijo sin saberlo, luego...
- Al cumplir el contrato del padre debieramos satisfacer el del hijo
- El hijo debe garantizarnos al menos lo mismo que el padre

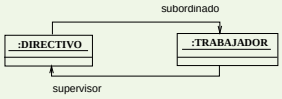
Contratos y herencia

- Todos los asertos son heredados por los hijos
- Los hijos sólo pueden hacer más débil la precondición (`require else`)
- Los hijos sólo pueden hacer más fuerta la postcondición (`ensure then`)
- Los hijos pueden ampliar el número de invariantes de clase

Fallos en los contratos

```
class DIRECTIVO
...
invariant
 --Si tengo un subordinado
 --soy su supervisor
 --Fallaré si él se escaquea
 subordinado /= void implies
   subordinado.supervisor=Current
end -- Class DIRECTIVO

class TRABAJADOR
...
feature
 escaqueo is
 do
  supervisor:=Void
 end
-- clase sin asertos.
end -- Class TRABAJADOR
```



Cuando fallan los contratos

- Una llamada a un método *tiene éxito* cuando termina su ejecución en un estado que satisface su contrato
- Una llamada a un método *fracasa* o *falla* cuando termina su ejecución en un estado que viola su contrato
- Una *excepción* es un evento en tiempo de ejecución que «puede» causar el fallo de una rutina
- No toda excepción provoca un fallo en la rutina (manejo de excepciones)
- El fallo de una rutina provoca una excepción en el cliente
- No solo el fallo de una rutina es capaz de provocar una excepción

Motivos de excepción

- Una llamada `a.f` en la que `a` está conectado a `Void`
- Un intento de conectar `Void` a un objeto expandido
- Una operación que provoca una situación anormal detectada en el «hardware» o el sistema operativo
- Una llamada a una rutina que falla
- El chequeo de una precondición falsa
- El chequeo de una postcondición falsa
- El chequeo de un invariante de clase falso
- El chequeo de un invariante de bucle falso
- El chequeo de un variante que no decrece
- El chequeo de un aserto falso en el código
- La ejecución de un disparo de excepción explícito (método `raise` de la clase `EXCEPTIONS`)

Gestión de excepciones

- Esquemas tradicionales
  - Gestión en un bloque de código centralizado
  - Gestión en el propio método que la provocó
  - Pero la ejecución del método original no se reintenta independientemente del éxito o fracaso del tratamiento realizado sobre la excepción
- Esquema de gestión segura:
  - Gestión en el propio método que la provocó
  - Reintentar utilizando la misma u otra estrategia (*retry*)
  - Reconocer el fallo disparando con ello una excepción (*Pánico, pero organizado*)
  - Los mensajes de error de los programas Eiffel muestran la *cascada de fallos* producidos

Excepciones en Eiffel

- Dos nuevos elementos del lenguaje:
  - Cada rutina puede contener una cláusula de rescate (`rescue`) con el código de tratamiento de las excepciones
  - La clausula de rescate puede contener una instrucción de reintento (`retry`)
- En caso de excepción se ejecuta la clausula de rescate. Si en ella no se ejecuta la instrucción de reintento la rutina falla, provocando una excepción en el cliente
- Existe una clase, `EXCEPTIONS`, que incorpora características especiales para el proceso de excepciones (ver su forma corta)

Ejemplo elemental

```
intentar_enviar(mensaje:STRING):BOOLEAN is
  -- Intento de transmisión del mensaje
  local
    fallos:INTEGER
  do
    if fallos < 50 then
      enviar(mensaje)
        -- Si llegamos aquí es porque
        -- se ha enviado el mensaje
      Result:=True
    else
      Result:=False
    end
  rescue
    intentos:=intentos+1
    retry
  end
```