Tecnología de la Programación

Diseño por contrato

Departamento de Computación

Facultad de Informática
Universidade da Coruña

Curso 2006/2007

Introducción

- Promotor: Bertrand Meyer (*ver Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997*
- Busca mejorar la calidad del código (reducir errores)

Introducción

- Promotor: Bertrand Meyer (*ver Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997*
- Busca mejorar la calidad del código (reducir errores)

Diseño por contrato

- En OO: clases y sus clientes tienen un contrato:
  - Condiciones que garantiza el cliente (precondiciones).
  - Condiciones que garantiza el proveedor (postcondiciones).
- Las entradas del API quedan caracterizadas por su especificación.
- Anotaciones (Hoare) = contrato.
- En general, un contrato se puede implementar de varias formas.
- Separamos el contrato (qué/intención) de la implementación (cómo).
- El contrato se hace ejecutable.
- Se comprueba el contrato en tiempo de ejecución.

Diseño por contrato

- En OO: clases y sus clientes tienen un contrato:
  - Condiciones que garantiza el cliente (precondiciones).
  - Condiciones que garantiza el proveedor (postcondiciones).
- Las entradas del API quedan caracterizadas por su especificación.
- Anotaciones (Hoare) = contrato.
- En general, un contrato se puede implementar de varias formas.
- Separamos el contrato (qué/intención) de la implementación (cómo).
- El contrato se hace ejecutable.
- Se comprueba el contrato en tiempo de ejecución.

Diseño por contrato

- En OO: clases y sus clientes tienen un contrato:
  - Condiciones que garantiza el cliente (precondiciones).
  - Condiciones que garantiza el proveedor (postcondiciones).
- Las entradas del API quedan caracterizadas por su especificación.
- Anotaciones (Hoare) = contrato.
- En general, un contrato se puede implementar de varias formas.
- Separamos el contrato (qué/intención) de la implementación (cómo).
- El contrato se hace ejecutable.
- Se comprueba el contrato en tiempo de ejecución.

Diseño por contrato

- En OO: clases y sus clientes tienen un contrato:
  - Condiciones que garantiza el cliente (precondiciones).
  - Condiciones que garantiza el proveedor (postcondiciones).
- Las entradas del API quedan caracterizadas por su especificación.
- Anotaciones (Hoare) = contrato.
- En general, un contrato se puede implementar de varias formas.
- Separamos el contrato (qué/intención) de la implementación (cómo).
- El contrato se hace ejecutable.
- Se comprueba el contrato en tiempo de ejecución.

Documentar con contratos

- Ventajas:
  - Más abstracto que el lenguaje natural.
  - Menos ambiguo que el lenguaje natural.
  - En general, es declarativo.
  - Ejecutable.
  - Asignación de culpa.
  - Eficiencia. Evita la programación defensiva y la “compilación” de los contratos es opcional.

DPC: ejemplo

```
/* requiere   X >= 0
 * asegura    X ~= resultado * resultado
 */
FUNCTION raíz_cuadrada(float:X):float { ... }
```

	Obligaciones	Derechos
Cliente	Pasa un número no negativo	Obtiene una aproximación de la raíz cuadrada
Implementador	Calcula y devuelve la raíz cuadrada	Asume que el argumento es no negativo

Ejemplo: Eiffel

- Lenguaje Orientado a Objetos, “Tipado estático”, herencia, polimorfismo, excepciones, paquetes, tipos genéricos, aserciones, ...
- Origen: Bertrand Meyers, 1986-92.
- Las anotaciones forman parte del lenguaje.

```
deposit(sum: INTEGER) is
require
  sum >= 0
do
  add(sum)
ensure
  balance = old balance + sum
end;
```

DPC en Java

- Java 1.4, 1.5: aserciones.
  - `assert Expression1 ;`
  - `assert Expression1 : Expression2 ;`
  - `javac -source 1.4 MyClass.java`
  - Limitado (precondiciones, fórmulas lógicas, invariantes)

    > *Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?*

- JML.
- OCL (UML).
  `http://ocl4java.org/`

DPC en Java

- Java 1.4, 1.5: aserciones.
  - `assert Expression1 ;`
  - `assert Expression1 : Expression2 ;`
  - `javac -source 1.4 MyClass.java`
  - Limitado (precondiciones, fórmulas lógicas, invariantes)

    > *Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?*

- JML.
- OCL (UML).
  `http://ocl4java.org/`

DPC en Java

- Java 1.4, 1.5: aserciones.
  - `assert Expression1 ;`
  - `assert Expression1 : Expression2 ;`
  - `javac -source 1.4 MyClass.java`
  - Limitado (precondiciones, fórmulas lógicas, invariantes)

    > *Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?*

- JML.
- OCL (UML).
  `http://ocl4java.org/`

DPC en Java

- Java 1.4, 1.5: aserciones.
  - `assert Expression1 ;`
  - `assert Expression1 : Expression2 ;`
  - `javac -source 1.4 MyClass.java`
  - Limitado (precondiciones, fórmulas lógicas, invariantes)

    > *Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?*

- JML.
- OCL (UML).
  `http://ocl4java.org/`

DPC en Java

- Java 1.4, 1.5: aserciones.
  - `assert Expression1 ;`
  - `assert Expression1 : Expression2 ;`
  - `javac -source 1.4 MyClass.java`
  - Limitado (precondiciones, fórmulas lógicas, invariantes)

    > *Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?*

- JML.
- OCL (UML).
  `http://ocl4java.org/`

Sobre las especificaciones

- Consideramos un programa como correcto cuando cumple una especificación.
- ¿ Cómo comprobar que se cumple la especificación ?
  - Validación
  - Verificación
- La especificación puede ser erronea, incompleta, ...

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   dividendo == divisor * cociente + resto
 */
FUNCION dividir(int:dividendo, int:divisor):void { ...
```

- La siguiente implementación cumple la especificación:

```
int cociente = 0;
int resto = dividendo;
```

Sobre las especificaciones

- Consideramos un programa como correcto cuando cumple una especificación.
- ¿ Cómo comprobar que se cumple la especificación ?
  - Validación
  - Verificación
- La especificación puede ser erronea, incompleta, ...

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   dividendo == divisor * cociente + resto
 */
FUNCION dividir(int:dividendo, int:divisor):void { ...
```

- La siguiente implementación cumple la especificación:

```
int cociente = 0;
int resto = dividendo;
```

Sobre las especificaciones

- Consideramos un programa como correcto cuando cumple una especificación.
- ¿ Cómo comprobar que se cumple la especificación ?
  - Validación
  - Verificación
- La especificación puede ser erronea, incompleta, ...

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   dividendo == divisor * cociente + resto
 */
FUNCION dividir(int:dividendo, int:divisor):void { ...
```

- La siguiente implementación cumple la especificación:

```
int cociente = 0;
int resto = dividendo;
```

Sobre las especificaciones

- Consideramos un programa como correcto cuando cumple una especificación.
- ¿ Cómo comprobar que se cumple la especificación ?
  - Validación
  - Verificación
- La especificación puede ser erronea, incompleta, ...

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   dividendo == divisor * cociente + resto
 */
FUNCION dividir(int:dividendo, int:divisor):void { ...
```

- La siguiente implementación cumple la especificación:

```
int cociente = 0;
int resto = dividendo;
```

Sobre las especificaciones (cont.)

- Mejoramos la especificación:

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   dividendo == divisor * cociente + resto Y
 *           resto < divisor
 */
```

- Las especificaciones no son únicas:

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   divisor * cociente <= dividendo Y
 *           divisor * (cociente+1) > dividendo
 */
```

Sobre las especificaciones (cont.)

- Mejoramos la especificación:

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   dividendo == divisor * cociente + resto Y
 *           resto < divisor
 */
```

- Las especificaciones no son únicas:

```
/* requiere  dividendo >= 0 Y divisor > 0
 * asegura   divisor * cociente <= dividendo Y
 *           divisor * (cociente+1) > dividendo
 */
```