Tema 8 - Diseño por contrato (DCA)

HISTORIAL DE REVISIONES			
NÚMERO	FECHA	MODIFICACIONES	NOMBRE

Índice

Logo DLSI

Tema 8 - Diseño por contrato
Curso 2018-2019

1. Preliminares (I)

- El *Diseño por Contrato* es una metodología de diseño de software. También se la conoce como *Programación por Contrato*.
- Trata de aplicar el concepto de las *condiciones* y *obligaciones* (a ambas partes) de un "contrato de negocios" a la implementación de un diseño de software.
- Un contrato es la especificación de las propiedades de un elemento software que afecta a su uso por parte de clientes potenciales.
- Para ello se dota al programador de una serie de nuevos elementos que permite llevar a cabo esta especificación: **precondiciones**, **postcondiciones** e **invariantes**.

2. Preliminares (II)

- El término "*Diseño por Contrato*" fue usado por primera vez por Bertrand Meyer al crear el lenguaje de programación Eiffel.
- Posteriormente registraron comercialmente dicho término (*DbC*).
- Está basado en la verificación y especificación formal de código, o también en la derivación de programas.
- Nosotros no vamos a entrar en la parte teórica de estos conceptos, sino que vamos a ir directamente a la práctica.

3. Precondiciones

- Son un elemento del "contrato".
- Se representan mediante una condición que al ser evaluada produce un valor **Verdadero** o **Falso**.
- Es una propiedad que un método o función "impone" a todos sus clientes.
- Un método o función puede tener varias precondiciones.
- Es responsabilidad de un cliente de este método o función asegurar que se satisfacen todas sus precondiciones cuando lo llama.
- Por tanto, para los clientes del método sus precondiciones son una obligación, mientras que para el método llamado representan lo que éste espera de los clientes que lo llaman.
- Son útiles cuando estamos depurando. Si no se cumple una precondición...el fallo está en el cliente del método que la ha definido.

4. Postcondiciones

- Son otro de los elementos del "contrato".
- Representan lo que obtendrán los clientes del método actual cuando este acabe...
- Evidentemente... si han cumplido las precondiciones del mismo.
- Son, por tanto, una obligación para el método actual que la define.
- Es este método el que debe asegurar que al terminar... todas sus postcondiciones se cumplen.
- Son útiles cuando estamos depurando. Si no se cumple una postcondición... el fallo está en el método que la ha definido.

5. Invariantes

- Son el último de los elementos del "contrato".
- *Precondiciones* y *postcondiciones* son propiedades lógicas que están asociadas con un método o función particular.
- Hay ocasiones donde queremos asociar una de estas propiedades lógicas con una "*clase*".
- A estas propiedades se las conoce como "**Invariantes de clase**". Una clase puede tener varios invariantes.
- Estos invariantes se deben cumplir **nada más crear un objeto** de la clase y **antes** y **después** de ejecutar cualquier método de la misma.

6. Lenguaje Eiffel

- Eiffel es un lenguaje de programación orientado a objetos.
- Diseñado por Bertrand Meyer. Basado en el cumplimiento de ciertos principios: design by contract, command-query separation, uniform-access principle, single-choice principle, open-closed principle, option-operand separation.
- Soporta herencia, herencia múltiple, genericidad, genericidad restringida, recolección de basura, etc...
- Disponemos del IDE EiffelStudio en plataformas *Windows*, como software libre disponemos de tecomp: The Eiffel Compiler.

7. Casos de uso: Eiffel (I):

Precondiciones, postcondiciones e invariantes:

```
class DATE
create  make
feature {NONE} -- Initialization
 make (a_day: INTEGER; a_hour: INTEGER)
                -- Initialize `Current' with `a_day' and `a_hour'.
        require
                valid_day: 1 <= a_day and a_day <= 31
                valid_hour: 0 <= a_hour and a_hour <= 23
        do
                day := a_day
                hour := a_hour
        ensure
                day_set: day = a_day
                hour_set: hour = a_hour
```



```
                end
        ...
        invariant
        valid_day: 1 <= day and day <= 31
        valid_hour: 0 <= hour and hour <= 23
        end
```

8. Casos de uso: Eiffel (II):

```
        class
                Account

        feature -- Access
          balance: INTEGER  -- Current balance

        feature -- Element change
                deposit (sum: INTEGER)
            -- Add `sum' to account.
        require
            non_negative: sum >= 0
        do
            ... As before ...
        ensure
            one_more_deposit: deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
        end
```

9. Lenguaje D (I)

- Es un lenguaje de la familia de **C**, trata de ser lo que **C++** no ha podido ofrecer por su intento de compatibilidad con **C**:

> "Can the power and capability of C++ be extracted, redesigned, and recast into a language that is simple, orthogonal, and practical? Can it all be put into a package that is easy for compiler writers to correctly implement, and which enables compilers to efficiently generate aggressively optimized code?
>
> — http://dlang.org/overview.html

- La versión del lenguaje a emplear es la **2**, puedes encontrar compiladores para la versión **1**, pero no se recomienda su uso.
- Su página web oficial la tienes en dlang.org.
- Características
- Puedes descargar el compilador de referencia desde aquí.
- Dispones de un tutorial recomendado.
- Dispone de análisis de cobertura del código y soporte de tests unitarios, además de compilación condicional y depuración llevadas un paso más allá de lo que estamos acostumbrados. Documentación generada a partir de comentarios.

10. Lenguaje D (II)

CONCEPTOS CLAVE EN EL DISEÑO DE **D**:

1. Make it easier to write code that is portable from compiler to compiler, machine to machine, and operating system to operating system. Eliminate undefined and implementation defined behaviors as much as practical.
2. Provide syntactic and semantic constructs that eliminate or at least reduce common mistakes. Reduce or even eliminate the need for third party static code checkers.
3. Support memory safe programming.
4. Support multi-paradigm programming, i.e. at a minimum support imperative, structured, object oriented, generic and even functional programming paradigms.
5. Make doing things the right way easier than the wrong way.
6. Have a short learning curve for programmers comfortable with programming in C, C++ or Java.
7. Provide low level bare metal access as required. Provide a means for the advanced programmer to escape checking as necessary.
8. Be compatible with the local C application binary interface.
9. Where D code looks the same as C code, have it either behave the same or issue an error.
10. Have a context-free grammar. Successful parsing must not require semantic analysis.
11. Easily support writing internationalized applications.
12. Incorporate Contract Programming and unit testing methodology.
13. Be able to build lightweight, standalone programs.
14. Reduce the costs of creating documentation.
15. Provide sufficient semantics to enable advances in compiler optimization technology.
16. Cater to the needs of numerical analysis programmers.
17. Obviously, sometimes these goals will conflict. Resolution will be in favor of usability.

11. Casos de uso: Lenguaje D (I)

Precondiciones y postcondiciones:

```
long square_root(long x)
in      {
        assert(x >= 0);
}
out (result)  {
        assert((result * result) <= x && (result+1) * (result+1) >= x);
}
body {
        return cast(long)std.math.sqrt(cast(real)x);
}
```


12. Casos de uso: Lenguaje D (II)

Invariantes

```
class Date
{
  int day;
  int hour;

  invariant() {
          assert(1 <= day && day <= 31);
          assert(0 <= hour && hour < 24);
  }
}
```

13. Caso de Uso: Vala

- Vala soporta *precondiciones* y *postcondiciones*, pero **no** *invariantes de clase*.

```
double method_name(int x, double d)
        requires (x > 0 && x < 10)
        requires (d >= 0.0 && d <= 1.0)
        ensures (result >= 0.0 && result <= 10.0)
{
        return d * x;
}
```

- Destacar la variable **result** que podemos ver en la postcondición.

14. Contracts Reading List

En el siguiente enlace puedes encontrar más información sobre diseño por contrato en diferentes lenguajes de programación.

15. Prácticas

GRUPO

- Existe otro tipo de *invariantes*, son los llamados *invariantes de bucles*. Investigad para qué sirven, de qué manera se pueden emplear en la verificación formal de programas y buscad ejemplos de lenguajes de programación con soporte sintáctico para ellos.

INDIVIDUALES

- Realiza una aplicación sencilla en Lenguaje-D que cree una o varias clases y haga uso de **pre** y **postcondiciones** en sus métodos, además de hacer uso de un **invariante de clase**. Comprueba qué ocurre cuando no se cumplen ni precondiciones ni postcondiciones ni invariante de clase. ¿Qué ocurre cuando un método *no-final* de una clase base no tiene precondiciones y cuando lo redefinimos en una clase derivada se las añadimos?

ENTREGA:

- La práctica se entregará en pracdlsi en las fechas allí indicadas.

Lenguaje D

```
El compilador de "D" instalado esta basado en "gcc" y se llama
"gdc". Si no esta instalado, prueba con alguna version concreta como
"gdc-4.6", "gdc-4.7".
```

- Haz lo mismo con el Lenguaje Vala, evidentemente sin los invariantes de clase.

Lenguaje Vala

```
Recuerda que el compilador de Vala se llama "valac". Tambien puedes
usar una version concreta del mismo como "valac-0.16", "valac-0.18",
etc...
```

Descarga tecomp: The Eiffel Compiler

Opcional:

```
Comprueba si se puede compilar el compilador de Eiffel en los
laboratorios. Si se puede generar el ejecutable del compilador,
trata de probar a compilar alguna demo en Eiffel con el.
```

---

16. Aclaraciones

EN NINGÚN CASO ESTAS TRANSPARENCIAS SON LA BIBLIOGRAFÍA DE LA ASIGNATURA.

- Debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.