Paradigmas de Programación

Programación en Grande

Modularidad

- Un módulo es un grupo de componentes declarados para un propósito común. Estos componentes pueden ser tipos, variables, constantes, procedimientos, funciones etc.
- Un Módulo encapsula sus componentes.
- Permite una interfase con otros módulos y hace conocidos unos pocos componentes hacia fuera del mismo (exportados).
- Otros componentes quedan ocultos; asisten a la implementación de componentes exportados.

La complejidad del software

- **Tamaño del software**
- Hace dos/tres décadas: programas en lenguaje ensamblador en torno a centenares de líneas.
- Hoy: lenguajes de alto nivel con centenares de millares, o incluso millones de líneas de código.

- **Ámbitos de la Complejidad**
- *Complejidad del problema*: la implementación se descompone en centenares y a veces miles de módulos independientes que implica tener un equipo de desarrolladores.
- *Complejidad "humana"*: cuantos más desarrolladores, las comunicaciones entre ellos son más complejas, la coordinación es difícil: equipos dispersos geográficamente (proyectos grandes)

- **Sistemas orientados a objetos:**
- La programación se puede hacer con extensiones de lenguajes comerciales: Object-Pascal o C++, y lenguajes OO puros como Smalltalk y Eiffel.
- Aumentan la variedad de aplicaciones que se pueden programar ya que se liberan las restricciones de los tipos de datos predefinidos.
- Acomodan estructuras de datos heterogéneos y complejos: se pueden añadir nuevos tipos de datos sin modificar código existente.

- **Propuestas de reutilización (reusability) de componentes software:**
- Bloques iniciales para la construcción del programa, similares a la construcción de cualquier objeto complejo (tal como un automóvil) que se construye ensamblando sus partes.

> *POO no sólo son nuevos lenguajes de programación, sino un nuevo modo de pensar y diseñar aplicaciones.*

Factores de Calidad del software

Eficiencia: Capacidad de un software para hacer un buen uso de los recursos que manipula.
Transportabilidad (portabilidad)
Facilidad con la que un software puede ser transportado sobre diferentes sistemas físicos o lógicos.
Verificabilidad (facilidad de verificación)
Capacidad para soportar los procedimientos de validación y de aceptar juegos de pruebas o ensayo de programas.
Integridad Capacidad de un software a proteger sus propios componentes contra los procesos que no tengan derecho de acceso a ellos.
Facilidad de uso Un software es fácil de utilizar si se puede comunicar con él de manera cómoda.
Corrección Capacidad de los productos software de realizar exactamente las tareas definidas por su especificación.
Robustez Capacidad para funcionar incluso en situaciones anormales.
Extensibilidad Facilidad que tienen los productos de adaptarse a cambios en su especificación.

Dos principios fundamentales: diseño simple y descentralización.
Reutilización Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.
Compatibilidad Facilidad de los productos para ser combinados con otros.

Fundamentos de la Orientación a Objetos

- Causas que impulsan el desarrollo de la POO:
  - los mecanismos de encapsulamiento de POO soportan un alto grado de reutilización de código, que se incrementa por sus mecanismos de herencia
  - gran aumento de **L**enguajes de **P**rogramación **O**rientados a **O**bjetos (LPOO)
  - implementación de interfaces de usuario gráficos (por iconos) y visuales.
- El paradigma OO se revela como el más adecuado para la elaboración del diseño y desarrollo de aplicaciones.
- Se caracteriza por la utilización del diseño modular OO y la reutilización del software.
- Especial énfasis en la abstracción (capacidad para encapsular y aislar la información del diseño y la ejecución).
- *Los objetos pasan a ser los elementos fundamentales en este nuevo marco, en detrimento de los subprogramas que lo han sido en los marcos tradicionales*

Orientación a Objetos

Conjunto de disciplinas (ingeniería) que desarrollan y modelan software que facilita la construcción de sistemas complejos a partir de componentes.

Trata de cumplir las necesidades de los usuarios finales, y las propias de los desarrolladores de productos software mediante la modelización del mundo real.

El soporte fundamental es el *modelo de objetos*. Los cuatro elementos (propiedades) más importantes de este modelo son: ***abstracción***, ***encapsulamiento***, ***modularidad*** y ***jerarquía***. Además de éstas, suele incluirse el *polimorfismo*.

- Abstracción:
  - Una abstracción se centra en la vista externa de un objeto
  - Separar el comportamiento esencial de un objeto de su implementación
  - Una **clase** (elemento clave en la POO) es una descripción abstracta de un grupo de objetos.
- Encapsulamiento:
  - Propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior: el objeto A no conoce lo que hace el objeto B y viceversa.
  - También se conoce como *ocultación de la información.*
  - Es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales.
- Modularidad:
  - propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas *módulos*), cada una las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
  - Consiste en dividir un programa en módulos que se pueden compilar por separado, pero que tienen conexiones con otros módulos.
- Jerarquía:
  - Propiedad que permite una ordenación de las abstracciones.
  - Las dos jerarquías más importantes de un sistema complejo son: generalización/especialización y agregación.

    - Polimorfismo :
  - Propiedad que indica la posibilidad de que una entidad tome *muchas formas*.
  - Requiere ***ligadura dinámica* o *postergada***, que sólo se puede producir en lenguajes de programación orientados a objetos.
  - Ejemplo: la operación *calcular perímetro* de la clase *polígono*.

Encapsulación

Los conceptos estudiados hasta este momento caracterizan a los lenguajes diseñados para la construcción de programas en pequeña escala, dónde las unidades más grandes del programa son funciones y procedimientos.

Desde 1970 el diseño de lenguajes apuntó a soportar la *programación en grande*; construcción de grandes programas a partir de módulos.

Un módulo es una unidad de programa que puede implementarse como una entidad más o menos independiente. Un módulo bien diseñado tiene un propósito único, y presenta una interfase estrecha a otros módulos. Estos módulos pueden ser reusados, incorporados a otros programas y modificados sin necesidad de realizar mayores cambios a otros módulos.

Son importantes dos cuestiones respecto de los módulos:

- *Que* es lo que hace el módulo? Su propósito.
- *Cómo* se consigue ese propósito?

El "Qué" le interesa al usuario del módulo, el "Cómo" le interesa sólo al implementador del mismo.

Un módulo podría ser un simple procedimiento o función. Pero más comúnmente, un módulo es un grupo de varios componentes declarados con un propósito común. Estos componentes pueden ser tipos, constantes, variables, procedimientos, funciones y demás. Se dice que un módulo encapsula sus componentes.

Para lograr una interfase estrecha con otros módulos, un módulo típicamente hace solo visible unos pocos componentes al exterior. Tales componentes se dice que son exportados por el módulo. Hay otros componentes que quedan *ocultos* dentro del módulo y sirven sólo para asistir la implementación de los componentes exportados.

Existen varios conceptos importantes que soportan la modularidad: paquetes (*package*), tipos abstractos, objetos, clases de objetos y genéricos (*generics*).

Paquetes

Es un grupo de componentes declarados. En general, los componentes declarados pueden ser tipos, constantes, variables, procedimientos, funciones y sub-paquetes.

Un paquetes puede ser visto como un conjunto de enlaces encapsulados.

Por ejemplo, veamos estos dos ejemplos en Ada.

```
Package fisicos is
      c: constant float := 3.0e+8;
      g: constant float := 6.7e-11;
      h: constant float := 6.6e-34;
end fisicos;
```

El efecto de la declaración de este paquete es enlazar `fisicos` a un conjunto de enlaces encapsulados {`c` $\rightarrow 3.0x10^{8}$, `g` $\rightarrow 6.7x10^{-11}$, `h` $\rightarrow 6.6x10^{-34}$ }. Todos los componentes de este paquete son constantes. Fuera del paquete los mismos pueden accederse mediante `fisicos.c`, `fisicos.g`, `fisicos.h`.

```
Package tierra is
      Type continente is (Africa, Antartida, Asia, Australia,
                          Europa, America);
      Radio : constant float := 6.4e6;
      Area  : constant array (continente) of float :=
              (30.3e9, 13.0e9, 43.3e9, 7.7e9, 10.4e9, 42.6e9);
      poblacion : array (continente) of integer;
end tierra;
```

El efecto de la declaración de este paquete es enlazar `tierra` a un conjunto de enlaces encapsulados (`continente`, `radio`, `area` y `poblacion`)

Ocultamiento de Información

Los dos paquetes anteriores no son típicos. Todos sus componentes son exportados; no hay detalles de implementación de componentes ocultos. Comúnmente los paquetes contienen declaración de componentes exportados y ocultos, estos últimos sirven solo para dar soporte a la implementación de los componentes exportados.

En Ada, los componentes exportados y los ocultos son distinguidos al dividir el paquete en dos partes: la *declaración del paquete*, que declara sólo los componentes exportados; y el *cuerpo del paquete*, que contiene declaración de los componentes ocultos. Si algún procedimiento o función es exportada, su cuerpo es definido dentro del cuerpo del paquete.

Veamos un ejemplo de un paquete trigonométrico en Ada. El mismo exporta un par de funciones llamadas `sen` y `cos`, cada una definida de Real → Real.

```
Package trig is
      function sen (x: float) return float;
      function cos (x: float) return float;
End trig;
```

El cuerpo del paquete correspondiente completa los detalles necesarios de la implementación.

```
Package body trig is
      Pi : constant float := 3.1416;

      Function norm (x: float) return float is
      ... ; -- retorna x modulo 2*pi

      Function sen (x: float) return float is
      ... ; -- retorna el seno de norm(x)

      Function cos (x: float) return float is
      ... ; -- retorna el coseno de norm(x)
end trig;
```

El cuerpo del paquete define realmente las abstracciones de función a las cuales se enlazan `sen` y `cos`. El paquete también declara la constante `pi` y la función `norm` para asistir la implementación de `sen` y `cos`. Debido a que `pi` y `norm` no se declaran en la declaración del paquete, son componentes ocultos del paquete.

El efecto del paquete anterior es enlazar `trig` al siguiente conjunto de enlaces encapsulados:

{ `sen` → una función que aproxima la función seno,
  `cos` → una función que aproxima la función coseno }

Los componentes ocultos son excluidos debido a que no son visibles para el usuario del paquete.
Las funciones exportadas pueden llamarse mediante `trig.cos (theta/2.0)` por ejemplo.

Tipos abstractos

Un tipo abstracto es un tipo definido por un grupo de operaciones. Las operaciones son usualmente constantes, funciones y procedimientos. El conjunto de valores del tipo se definen indirectamente; consiste de todos los valores que pueden generarse por sucesivas aplicaciones de las operaciones, comenzando con las constantes.

El concepto de tipo abstracto es un tipo importante de módulo que está soportado en algunos lenguajes modernos. El programador elige la representación de los valores del tipo abstracto, e implementa las operaciones en términos de esa representación elegida. El punto clave es que la representación es oculta; el módulo exporta sólo el tipo abstracto y sus operaciones. Veamos un ejemplo en ML:

```
abstype racional = rac of (int * int)
with
      val cero = rac (0, 1)
      and uno  = rac (1, 1);
```

```
      fun op // (m: int, n: int) =
            if n <> 0
            then rac (m, n)
            else ... (* número racional inválido *)

      and op ++ (rac (n1, d1) : racional, rac (n2, d2) racional) =
            rac (n1*d2 + n2*d1, d1*d2)

      and op == (rac (n1, d1) : racional, rac (n2, d2) racional) =
            (n1*d2 = n2*d1)
end
```

La declaración del tipo abstracto produce el siguiente conjunto de enlaces:

{ `racional` → un tipo abstracto,
`cero` → un número racional igual a 0,
`uno` → un número racional igual a 1,
`//` → una abstracción de función que construye un racional,
`++` → una abstracción de función que suma dos racional,
`==` → una abstracción de función que compara dos racional por igualdad }

El siguiente código construye un número racional: `val h = 1//2`
El siguiente código también es factible de realizar: `if uno ++ h == 6//4 then .. else ..`

El *usuario* del módulo lo ve como el conjunto de enlaces que se muestra arriba. Por lo tanto para él *m//n* retornará un número racional matemáticamente igual a *m/n* (con n<>0); *r++s* retornará la suma de dos números racionales r y s; y *r==s* retornará verdadero si y solo si los números racionales r y s son matemáticamente iguales. La representación elegida para los valores del nuevo tipo están ocultas al usuario.

El *implementador* del módulo lo ve como la representación de cada número racional como el par rotulado rac (m, n).

La única manera para el usuario de generar valores del tipo `racional` es al evaluar expresiones que involucren las constantes `cero` y `uno`, y las funciones `//` y `++`. Valores del tipo `racional` pueden compararse solo llamando a la función `==`.

La representación de un tipo abstracto puede cambiarse en cualquier momento (por ejemplo para lograr mayor eficiencia en las operaciones), sin forzar ningún cambio fuera del módulo.

La definición de un tipo abstracto involucra trabajo extra, precisamente porque la representación del tipo está oculta al usuario del tipo. Esto implica que el tipo abstracto debe proveer de *constructores* para componer valores del tipo abstracto y de *destructores* para destruir dichos valores.

Por ejemplo, la función `//` permite construir valores del tipo racional, por lo tanto es un caso de constructor. Las constantes `cero` y `uno` también lo son. En el ejemplo visto no existen destructores.

Los tipos abstractos son similares a los tipos predefinidos tales como valores de verdad y enteros. Los valores predefinidos tienen una representación interna en términos de bits o bytes, pero están ocultas al programador. Por ejemplo una implementación particular de valores de verdad podrían ser cualquier byte con todos sus bits en cero para *falso* y *verdadero* en cualquier otro caso.

Clases y objetos

Otro tipo de módulo muy especial e importante son aquellos que consisten de datos ocultos junto con un conjunto de operaciones exportadas. Los datos ocultos sólo pueden accederse a través de las operaciones exportadas. Esto tiene la ventaja de que el interior del módulo puede cambiar (por ejemplo para lograr mayor eficiencia en las operaciones) sin forzar cambios hacia el exterior.

Algunos lenguajes permiten construir objetos independientes con estas características especificando todos los detalles de implementación dentro del cuerpo del objeto. Los mismos tendrán un tiempo de vida debido a que es un componente variable y se comporta como una variable común.

Lejos de considerar solo objetos únicos. Es importante poder crear ***clase*** de objetos similares. En Ada se implementa a través de un *generic package*. Y en C++ a través de una *class*.

Ejemplo 1: Supongamos que deseamos crear varios directorios telefónicos, primero debemos crear una estructura genérica para todos de ellos.

```
generic package clase_directorio is
procedure insertar (NombreNuevo : in Name; NumeroNuevo : in Number);
procedure buscar   (NombreBusc  : in Name; NumeroBusc  : out Number;
                    Encontrado  : out boolean);
end clase_directorio;

package body clase_directorio is
type Nodo;
type Punt is access Nodo;
type Nodo is record
               Nombre : Name;
               Numero : Number;
               Der, izq : Punt;
             End record;

Raiz : punt;

procedure insertar (NombreNuevo : in Name; NumeroNuevo : in Number) is ... ;
            -- Implementación
procedure buscar   (NombreBusc  : in Name; NombreBusc  : out Number;
                    Encontrado  : out boolean) is ... ;
            -- Implementación
end Clase_directorio;
```

Al elaborar este paquete genérico simplemente enlaza `clase_directorio` a una clase de objetos. No se crea ningún objeto realmente. Para crear objetos individuales, debemos *instanciar* el paquete genérico.

```
package dir_casa is new clase_directorio;
package dir_trabajo is new clase_directorio;
```

Estas declaraciones crean dos objetos similares pero distintos, denotados por `dir_casa` y `dir_trabajo`. Podremos acceder a los mismos utilizando la notación de punto.

```
dir_casa.insertar (yo, 452546);
dir_trabajo.insertar (yo, 479515);
dir_trabajo.buscar (yo, miNumero, Ok);
```

Ejemplo 2: Veamos otro ejemplo de estos conceptos implementados en C++. Supongamos una clase cola.

```
class cola {
  int c[100];
  int posIni, posFin;
public
  cola (void);    // constructor
  void ponc (int i);
  int quitac (void);
};

cola::cola (void)
  {  posIni = posFin = 0;  }

void cola::ponc (int i)
  {  // implementación de la función }

int cola::quitac (void)
  {  // implementación de la función }
```

Al definir esta clase se enlaza `cola` a una clase de objetos aunque no se crea ningún objeto. Acá también existen partes publicas y privadas es decir que son exportadas hacia el exterior o quedan ocultas dentro de la clase respectivamente.

Por defecto todos los elementos son privados. Por lo tanto `c`, `posIni`, `posFin` son privados. Sólo pueden accederse a través de funciones miembro de la clase como `cola`, `ponc` y `quitac`. Estas se definen dentro de la clase pero se implementan en general fuera de la misma.

Estas declaraciones crean dos objetos similares pero distintos denotados por `colaEnt1` y `colaEnt2`.

```
cola colaEnt1;
cola colaEnt2;
```

Entre los conceptos de tipo abstracto y clases tiene mucho en común. Cada uno nos permite crear varias variables del mismo tipo cuya representación es oculta y los accesos a esas variables solo puede hacerse a través de operaciones que se proveen para ese propósito. Sin embargo estos dos conceptos son sustancialmente diferentes.

Si se define un tipo abstracto similar a la clase clase_directorio del ejemplo anterior las operaciones insertar y buscar deberían definirse de la siguiente manera:

```
procedure insertar (dir          : in out Directorio;
                    NombreNuevo : in Name; NumeroNuevo : in Number);
procedure buscar   (dir          : in Directorio;
                    NombreBusc   : in Name; NumeroBusc  : out Number;
                    Encontrado   : out boolean);
```

En el caso de un tipo abstracto ambas funciones tienen un parámetro más del tipo abstracto `directorio`. Esto es necesario para que el procedimiento sepa a qué variable deberá insertar un nuevo elemento o en que variable buscar un elemento. En una clase, varias instanciaciones producen varios procedimientos distintos (por ej: `dir_casa.insertar`, `dir_trabajo.insertar`)

Podemos examinar la diferencia desde otro punto de vista; comparando las llamadas a los procedimientos. En un tipo abstracto, deberíamos escribir `insertar (dir_casa, yo, 452546)`. Aquí, obviamente, un directorio particular es un argumento del procedimiento. En el caso de una clase, deberíamos escribir `dir_casa.insertar (yo, 452546)`. Aquí, un directorio particular es un tipo de argumento implícito, pero es un argumento fijo cuando se crea el objeto, no cuando se llama al procedimiento.

Abstracciones genéricas

Hemos visto que las funciones son abstracciones sobre expresiones, y que los procedimientos son abstracciones sobre comandos. Podemos extender esta analogía y construir abstracciones sobre declaraciones.

Una *abstracción genérica* es una abstracción sobre una *declaración*. Quiere decir, una *abstracción genérica* tiene un cuerpo que es una *declaración* , y una *instanciación genérica* es una *declaración* que *producirá enlaces* al evaluar el *cuerpo de una abstracción genérica*.

En Ada el concepto aparece en la forma de *generic package*. Estos pueden ser parametrizados. Por ejemplo un paquete que encapsula una cola de caracteres, el cual es encapsulado respecto a la cantidad de caracteres que puede tener.

```
generic
  capacidad : in positive;
package clase_cola is
  procedure agregar (item : in Character);
  procedure quitar  (item : out Character);
end clase_cola;

package body clase_cola is
  items : array (1..capacidad) of Character;
  tamanio, ini, fin : integer range 0..capacidad;

  procedure agregar (item : in Character) is ...
  procedure quitar (item : out Character) is ...
```

```
begin ...
end clase_cola;
```

El parámetro formal de este paquete genérico es `capacidad`, el cual tiene varias ocurrencias de aplicación dentro del cuerpo del paquete.

Podemos, ahora, escribir una instanciación del paquete genérico de la siguiente manera:

```
Package buffer is new clase_cola (120)
```

Parametros Type

Hemos visto como una declaración puede hacer uso de valores definidos previamente a través de paquetes genéricos parametrizados. Una declaración también puede hacer uso de *tipos* definidos previamente como parámetros de los mismos. En Ada esto se implementa a través de parámetros Type.

Veamos un ejemplo: la clase_cola del ejemplo anterior está parametrizada respecto de la capacidad. Una generalización del mismo podría resultar al parametrizarlo respecto del tipo de items que puede manejar la cola.

```
generic
  capacidad : in positive;
  type item is private;
package clase_cola is
  procedure agregar (ItemNuevo :  in item);
  procedure quitar  (ItemViejo : out item);
end clase_cola;

package body clase_cola is
  items : array (1..capacidad) of item;
  tamanio, ini, fin : integer range 0..capacidad;

  procedure agregar (ItemNuevo :  in item) is
    begin
      ...; items (fin) := ItemNuevo; ...;
    end;

  procedure quitar  (ItemViejo : out item) is ...
    begin
      ...; ItemVievo := items (ini); ...;
    end;

begin
   ini := 1; fin := 0;
end clase_cola;
```

Los parámetros formales de este paquete genérico son `capacidad` (que denota un valor) e `Item` (que denota un tipo).

La frase **`type`** `item` **`is private`** es la manera en que se le dice a Ada que la asignación es una operación válida para el tipo denotado por `Item`.

En la siguiente porción de código:

```
Package buffer is new clase_cola (120, Character);
...
buffer.agregar ('*');
```

se elabora la instanciación de la siguiente manera: Primero, el parámetro formal `capacidad` se enlaza al primer argumento, el valor 120; y el parámetro formal `Item` se enlaza al segundo argumento, el tipo denotado por `Character`. Luego se elaboran la declaración y el cuerpo del paquete, produciendo un paquete que encapsula una cola con 120 caracteres. Finalmente `buffer` denota el paquete.

Indice