Lenguajes de Programación

Encapsulamiento y Abstracción

Ma. Laura Cobo

Universidad Nacional del Sur
Departamento de Ciencias e Ingeniería de la Computación
2019

Abstracción

La **abstracción** es una vista o representación de una entidad que incluye los atributos mas significativos

En un sentido general permite agrupar entidades en grupos, teniendo en mente atributos esenciales.

En lenguajes de programación es una herramienta para reducir la complejidad de la programación (simplificando el proceso de programación)

Los lenguajes contemporáneos proveen dos tipos fundamentales de abstracción:

- Abstracción de procesos
- Abstracción de datos

2

Evolución del concepto de tipo

- Tipos básicos
- Tipos definidos por el usuario
- Tipo de dato abstracto

- Un tipo determina la interpretación de un valor almacenado y permite chequeos (Fortran)
- Un tipo es el conjunto de valores que puede tomar un dato (Pascal)
- Un tipo es un conjunto de valores y un conjunto de operaciones (SIMULA)
- Un tipo de dato abstracto es una entidad que encapsula valores y operaciones (solo acceso a través de las operaciones)
- Un TDA genérico es una entidad que encapsula un conjunto de valores (debe incluir las operaciones) y un conjunto de valores de manera independiente.

3

Abstracción de datos

Casi todos los lenguajes de programación soportan *abstracción de procesos* a través de las unidades.

Los lenguajes de programación diseñados desde 1980 soportan en general *abstracción de datos*

Un *tipo de dato abstracto* es un tipo de dato definido por el usuario que satisface las siguientes condiciones:

1. La representación de los objetos del tipo son definidos como unidades sintácticas simples.
2. La representación del tipo son escondidos del programa que utilizan estos objetos, por lo tanto las únicas operaciones posibles son aquellas provistas por la definición del tipo

Sebesta

1. MODIFICABILIDAD - ORGANIZACIÓN
2. CONFIABILIDAD

Abstracción de datos

Un *tipo de dato abstracto* tiene tres beneficios importantes:

- Reduce la cantidad de detalles que el programador debe conocer y tener en mente a cada momento.
- Previene el uso incorrecto de componentes del programa (contención de fallas), y
- Independencia significativa entre las componentes del programa.

La definición completa debería estar encapsulada de manera que el usuario del tipo solo necesite conocer el nombre del tipo, la semántica y las operaciones disponibles.

5

El concepto de abstracción

Una *abstracción* es una vista o representación de una entidad, la cual incluye sólo los atributos más significativos.

El concepto de *abstracción* es fundamental en programación y en ciencias de la computación.

La *abstracción* permite pensar a un lenguaje de programación como una máquina virtual más poderosa y simple que la máquina real. El lenguaje provee abstracciones primitivas y mecanismos para definir otras nuevas: *funcionales* o *basadas en datos*.

El aumento del nivel de *abstracción* favorece:

- Legibilidad
- Oscurecimiento
- Reusabilidad
- Verificación
- Confiabilidad
- Extensibilidad

6

El concepto de abstracción

El concepto central detrás del diseño de abstracciones definidas por el programados es el de *ocultamiento de información*.

Cuando la información es encapsulada en una abstracción, significa que el usuario de dicha abstracción:

- No necesita conocer la información oculta para poder hacer uso de la abstracción, y
- No está permitida la manipulación directa la información oculta aunque se desee hacerlo.

El *ocultamiento de información* es una cuestión de diseño de programas; es posible en cualquier programa que sea diseñado apropiadamente independientemente del lenguaje de programación utilizado.

El *encapsulamiento* es una cuestión de diseño del lenguaje; una abstracción resulta efectivamente encapsulada solo cuando el lenguaje prohíbe el acceso a la información oculta en la abstracción.

7

El concepto de abstracción

El oscurecimiento es un objetivo de diseño de suma importancia para lograr un buen encapsulamiento.

A partir del encapsulamiento se incorpora la programación modular.

Logrando:

- mejorar la programación y compilación.
- La idea centrar es descomponer el problema es módulos de propósito simple, interfaz clara e implementación eficiente

Para lograr estos objetivos, es necesario dividir el módulo en dos partes: *interface* e *implementación*.

Al usuario, ni le interesa cómo se implementa el módulo, ni tampoco puede accederlo. Modula 2 fue uno de los primeros lenguajes en soportar encapsulamiento

8

Constructores de encapsulamiento

Los programas grandes, tienen dos necesidades especiales:

- Necesidad de organización, más que simple división en unidades.
- Necesidad de compilación parcial (unidades de compilación más pequeñas que el programa completo)

La solución obvia es agrupar las unidades que están lógicamente relacionadas en una unidad que pueda ser compilada separadamente (*unidad de compilación*)

Esta solución obvia conduce al encapsulamiento.

9

Constructores de encapsulamiento

Los programas se pueden organizar anidando la definición de unidades (Ada – Fortran 95)

En C cada archivo que contenga una o mas unidades puede ser compilado independientemente. La interface se coloca en un archivo *header*.

Los paquetes de Ada pueden incluir la declaración de cualquier número de datos y unidades.

Estos paquetes pueden ser compilados en forma separada; la declaración y el cuerpo del paquete también pueden compilarse separadamente

10

Diseño de tipos de datos abstractos

- Unidad sintáctica para definir el tipo de dato abstracto.
- Operaciones built-in
  - Asignaciones
  - Comparación
- Operaciones comunes
  - Iteradores
  - Constructores
  - Destructores
- Tipo de datos abstractos parametrizados
- Extensiones de tipos

Diseño de tipos de datos abstractos

La facilidad para definir tipos de datos abstractos en un lenguaje debe proveer una *unidad sintáctica* que encierre la definición del tipo y la definición de unidades para las operaciones de la abstracción.

ADA

El constructor de encapsulamiento se llama *packages*

- Especificación del paquete (la interface)
- Cuerpo del paquete (implementación de las entidades que son nombradas en la especificación)

La representación del tipo aparece en una parte de la especificación, llamada *private* part.

12

Diseño de tipos de datos abstractos: ADA

```
package Stack_Pack is
    type stack_type is limited private;
    max_size: constant := 100;
    function empty(stk: in stack_type) return Boolean;
    procedure push(stk: in out stack_type; elem:in Integer);
    procedure pop(stk: in out stack_type);
    function top(stk: in stack_type) return Integer;

    private  -- hidden from clients
    type list_type is array (1..max_size) of Integer;
    type stack_type is record
            list: list_type;
            topsub: Integer range 0..max_size) := 0;
    end record;
end Stack_Pack
```

13

Diseño de tipos de datos abstractos: C++

La clase es el mecanismo de encapsulamiento. Está basado en el *struct* de C y la clase de Simula 67.

Todas las instancias de la clase comparten una sola copia de las funciones miembro, pero tienen su propia copia de los datos.

La información se oculta a través de los modificadores:

- *Private*: para entidades ocultas
- *Public*: para entidades de interface
- *Protected*: para herencia

14

Diseño de tipos de datos abstractos: C++

Constructores:

- Funciones para inicializar los datos de las instancias (no crean el objeto).
- Pueden alocar almacenamiento, si parte de los objeto es dinámico en heap.
- Pueden incluir parámetros para proveer parametrización de los objetos.
- Son llamados implícitamente cuando se crea un objeto.
- Pueden ser llamados en forma explícita.
- El nombre es el mismo que el de la clase.

15

Diseño de tipos de datos abstractos: C++

Destructores:

- Funciones de limpieza luego de que una instancia es destruida: usualmente solo para reclamar el almacenamiento del heap.
- Llamadas implícitamente cuando el tiempo de vida de un objeto termina.
- El nombre, es el nombre de la clase precedido por el símbolo ~.

16

Diseño de tipos de datos abstractos: C++

```
class stack {
    private:
        int *stackPtr, maxLen, topPtr;
    public:
        stack() { // a constructor
            stackPtr = new int [100];
            maxLen = 99;
            topPtr = -1;
        };
        ~stack () {delete [] stackPtr;};
        void push (int num) {…};
        void pop () {…};
        int top () {…};
        int empty () {…};
}
```

17

Tipos de datos abstractos parametrizados

Los tipos de datos parametrizados permiten el diseño de una tipo de dato abstracto que puede almacenar elementos de cualquier tipo.

Estos tipos de datos abstactos son también conocidos como *clases genéricas*.

C++ y ADA proveen soporte para tipos de datos abstractos parametrizados.

Java 5.0 provee una forma restringida de tipo de dato abstracto parametrizado.

C# no provee soporte para este tipo de dato abstracto.

18

Tipos de datos abstractos parametrizados

Ada Generic Packages: Hace que tipo stack sea mas flexible haciendo que el tipo elemento y el tamaño de la pila genéricos

```
generic
Max_size: Positive;
type Elem_Type is Private;
package Generic_Stack is
…
function Top(Stk: in out StackType) return Elem_type;
…
end Generic_Stack;
```

```
Package Integer_Stack is new Generics_Stack(100,Integer);
Package Float_Stack is new Generics_Stack(100,Float);
```

19

Tipos de datos abstractos parametrizados

C++: las clases pueden ser de alguna manera genéricas escribiendo funciones constructoras parametrizadas

```
template <class type>
class stack {
…
stack (int size) {
stk_ptr = new int [size];
max_len = size - 1;
 top = -1;
};
        …
}

stack stk(100);
```

20

Extensiones de tipos

Información que es conocida es una parte del programa generalmente es requerida y utilizada en otra parte. Eso sucede:

- *Explícitamente*: pasaje de parámetros, pasaje de información de los parámetros actuales a los formales. La ligadura se hace en la llamada explicita.
- *Implícitamente*: generalmente llamada *herencia*. Sucede cuando una componente del programa, recibe propiedades o características de otra componente siguiendo una relación especial, existente entre ambas componentes.

Observación: Una primera forma de herencia aparece en las reglas de alcance de los datos en los programas estructurados por bloques.

A pesar de esto, el término *herencia* se utiliza mas frecuentemente para indicar pasaje de datos y unidades entre módulos independientes de un programa (ejemplo las clases de C++)

Herencia

Si tenemos una relación entre A y B (A => B)

- A es la clase padre o superclase de B
- B es la clase hija o subclase de A

Si una clase tiene un solo padre diremos que la relación es de *herencia simple*. Si una clase tiene más de un padre diremos que tiene *herencia múltiple*.

Ada permite una forma de herencia a través de los llamados *tagged types* (los objetos declarados *tagged* pueden ser utilizados en un nuevo *package* con componentes adicionales)

```
type Calendario is tagged private;
   ….
private
type Calendario is tagged record
    Day: integer;   Month: integer;
end record
```

```
type CalendarioAnual is new Calendario
  with record
   Year: integer;
  end record
```

22

Herencia

Vimos que el concepto de encapsulamiento, suele verse como un mecanismo para *dividir y conquistar* el control del programa. El programador solo tiene acceso a aquellos datos que son parte de la especificación del segmento de programa a ser desarrollado. La especificación de cualquier otra parte del programa está mas allá del dominio de conocimiento del programador

La abstracción relacionada con la herencia, puede verse como más que una pared que evita que el programado vea los contenidos de objetos de datos impropios.

La herencia provee el mecanismo para pasar información entre objetos de clases relacionadas. Hay cuatro relaciones que resumen el uso de la herencia en los lenguajes

23

Implementación: programación OO

Los aspectos más importantes de la programación orientada a objetos son una técnica de diseño dirigida a la determinación y delegación de responsabilidades. En general los lenguajes orientados a objetos proveen:

- Abstracción de datos (encapsulamiento de un estado con operaciones)
- Encapsulamiento (ocultamiento de información)
- Polimorfismo (pasaje de mensajes)

Los lenguajes orientados a objetos también implementan:

- Herencia
- Ligadura dinámica

24

Implementación: programación OO

Abstracción de datos

La representación de la estructura de datos y operaciones es fija; indicando que el tipo de dato abstracto es implementado. Desde el punto de vista teórico, un tipo de dato abstracto es:

- un conjunto de objetos de datos. Comúnmente se utilizan una o mas definiciones de tipo.
- un conjunto de operaciones para manipular esos objetos de datos.

En las soluciones no orientadas a objetos, las estructuras de datos y las operaciones son piezas diferentes; en contraste las implementaciones orientadas a objetos requieren encapsulamiento (debido a la naturaleza dual de los objetos)

25

Implementación: programación OO

Encapsulamiento

Al contar con encapsulamiento el usuario de la abstracción:

- no necesita conocer la información oculta en la abstracción
- no es posible utilizar o manipular la información oculta aunque se desee hacerlo.

Polimorfismo

En el contexto de la programación orientada a objetos implica que en una misma jerarquía de herencia, se puede responder al mismo mensaje en forma diferente.

26

Implementación

Representación de objetos

La representación del objeto, debería mantener la información que cambia de instancia en instancia.
La información que se mantiene igual para todas las instancias, se mantiene en la representación de la clase. La información guardada en la representación de la clase incluye los métodos de clase e instancia

La representación del objeto es básicamente un registro que contiene los valores de las variables de instancia y una referencia a la representación de la clase de la cual es instancia (solución más simple)

27

Implementación

Representación de clases

Es necesario mantener la siguiente información:

1. El nombre de la clase.
2. La superclase (que en general es object en caso que no se especifique).
3. Los nombres de las variables de instancia.
4. Un diccionario con los métodos de clase.
5. Un diccionario con los métodos de instancia

28

Implementación

Representación del registro de activación

Se mantiene un registro de activación, de manera tal que es posible mantener la información relevante a la activación de un método.

El registro sigue manteniendo tres partes como en su versión para lenguajes imperativos:

a. **Entorno**: contexto utilizado para la ejecución del método (entorno local y no local)
b. **Instrucciones**: la instrucción a ejecutarse cuando el método se reanude.
c. **Emisor**: el registro de activación del método que envió el mensaje invocando éste método (enlace dinámico)

29

Implementación

Enviando y retornando mensajes

A la hora de enviar un mensaje, debemos:

a. Crear el registro de activación para el método llamado.
b. Identificar el método invocado extrayendo el patrón del objeto receptor o superclase.
c. Transmitir los parámetros al registro de activación del receptor.
d. Suspender al llamador, salvando su estado en el registro de activación.
e. Establecer el camino de regreso al emisor y colocar el registro de activación como el registro activo.

Al retornar, debemos:

a. Transmitir el objeto resultante (si lo hay) al emisor.
b. Reanudar la ejecución del emisor restaurando su estado con la información en su registro de activación

30

Implementación

Implementación de constructores Orientados a Objetos

Hay al menos dos partes de la programación orientada a objetos que provocan preguntas interesantes a los implementadores de lenguajes de programación.

- Las estructuras de almacenamiento para las variables de instancia.
- La ligadura dinámica de los mensajes a los métodos.

Almacenamiento de los datos de una instancia:
La estructura de almacenamiento tiene claras similitudes al registro. Se llama **CIR** (**C**lass **I**nstance **R**ecord).
La estructura del **CIR** es estática, se construye en tiempo de compilación y es utilizada como template para la creación de instancias de clase.
Los accesos a los atributos de una instancia se resuelven de igual manera que los accesos a las componentes del registro y con la misma eficiencia.

31

Implementación: herencia

¿Qué sucede si agregamos herencia?

El contar con herencia no agrega demasiado trabajo al traductor, ya que en una clase derivada, sólo los nombres heredados de la clase padre, son agregados al espacio de nombres local y solo los públicos tienen visibilidad para los usuarios de la clase.
La determinación del **CIR** sigue siendo estática, debido a las declaraciones en la definición de la clase.

Hay dos aproximaciones a la implementación de herencia:

- Aproximación basada en copia: es la más simple y directa de implementar. La objetos instancia de la clase derivada tiene todos los detalles de su implementación.
- Aproximación basada en delegación: es este modelo, cada objeto instancia de una clase derivada utiliza el almacenamiento de la clase padre. Las propiedades heredadas no son duplicadas en el objeto derivado.

32

Implementación: Herencia

Aproximación basada en delegación

Para la implementación del segundo modelo, es necesario contar con alguna forma para compartir datos, de esta manera los cambios en el objeto base generaran cambios en el objeto derivado

Métodos:
Los métodos virtuales pueden ser implementados de manera similar a los registros de activación. Cada método virtual en una clase derivada reserva un slot en el registro que define la clase. El procedimiento constructor, simplemente rellena la locación con la información que necesita.

33

Implementación

Ligadura dinámica de los mensajes a los métodos

Los métodos en una clase están estáticamente ligados y no es necesario incluirlos en el CIR de la clase. Sin embargo, los métodos puede ligarse dinámicamente deberían mantenerse en esta estructura.
La entrada podría consistir simplemente de un puntero al código del método, puntero que puede ser seteado al momento de creación del objeto. Es aspecto negativo de esta solución es que todos los objetos instancias de una clase deben mantener los punteros de todas los métodos que se liguen dinámicamente.

La desventaja radica en que el conjunto de estos métodos es el mismo para todas las instancias de la clase (depende de la clase no de la instancia) por lo tanto, pueden almacenarse una sola vez. Este almacenamiento puede ser referenciado por todas las instancias de la clase.

Implementación

Ligadura dinámica de los mensajes a los métodos

De ésta manera el **CIR** sólo necesita mantener un solo puntero a una lista para poder encontrar el método llamado.

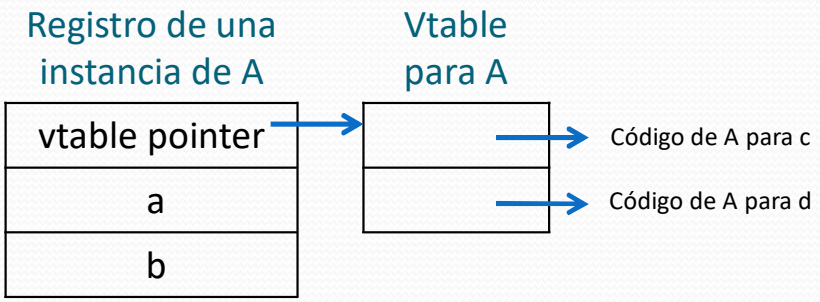
La estructura para almacenar esta lista es a veces llamada tabla virtual de métodos (**vtable**).

Las llamadas a los métodos pueden ser representadas por *offset* desde el comienzo de **vtable**.

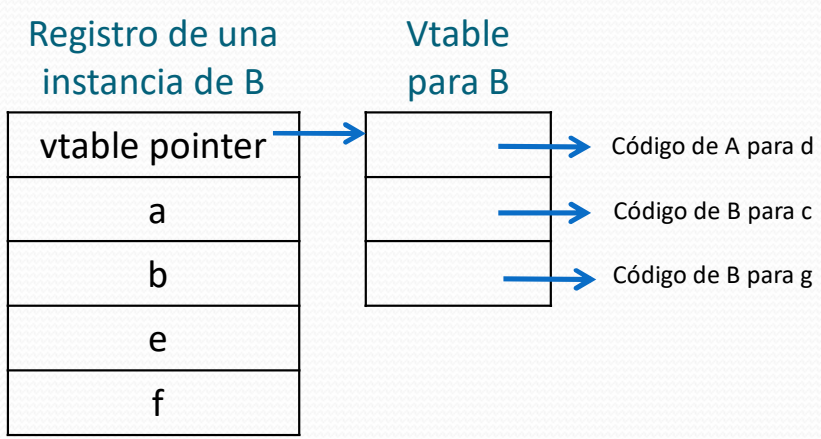
35

Implementación

```
Public class A {
 public int a,b;
 public void c() {.....}
 public int d() {....}
}
```




```
Public class B extends A {
 public int e,f;
 public void c() {.....}
 public void g() {....}
}
```



36

Implementación

Genericidad

Un tipo de dato abstracto genérico, en principio, tiene una implementación directa. Los parámetros genéricos deben darse cuando se instancia el paquete.

El compilador usa la clase genérica, como un molde al cual una vez que se le insertan los parámetros adecuados funciona como una clase tradicional.

Una vez que se insertan los valores para los parámetros, se compila la definición cómo si se tratase de una unidad tradicional ***sin*** parámetros.

37

Implementación

Polimorfismo

Para los lenguajes tipados estáticamente, el polimorfismo no agrega complejidad.
En cambio para los que permiten polimorfismo dinámico aparecen dificultades, ya que la cantidad de argumentos de una función polimórfica debe determinarse durante la ejecución.

Hay dos maneras de pasar los argumentos a una función polimórfica:

a) A través de un descriptor inmediato (se utiliza un bit para indicar cual es el tipo del parámetro).
b) A través de un descriptor. Se otorga la información completa del tipo, es como dar la estructura completa de los objetos de datos compuestos.

38

Implementación

Polimorfismo

La ejecución de funciones polimórficas será claramente más lenta que para lenguajes con tipado estático, debido a que el programa debe interrogar los argumentos antes de obtener sus valores.

Sin embargo, para muchas aplicaciones en poder de crear funciones polimórficas le quita peso a la ineficiencia en ejecución.

Encapsulamiento de nombres

Problema:

¿Como pueden los desarrolladores que trabajan en forma independiente crear nombres para las entidades utilizadas sin reutilizar accidentalmente los mismos nombres?

La extensión del uso de librería incrementó está necesidad de que los lenguajes provean soporte para encapsular nombres.

La idea es resolver los problemas de nombres, cada parte lógica del software puede crear una encapsulación de nombres. Dado que el encapsulamiento es lógico, los códigos afectados puede estar almacenados en diferentes lugares.

40

Encapsulamiento de nombres

Los programas grandes, definen muchos nombres globales, los cuales es necesario agrupar de forma lógica.

El *encapsulamiento de nombres* se utiliza para crear un nuevo ambiente o espacio de nombres.

Namespaces en C++: puede colocar cada librería en su propio espacio de nombres y cualificar las llamadas cuando se esta fuera.

Paquetes Java: pueden contener mas de una definición de clase. Los clientes del paquete pueden usar llamadas cualificadas o utilizar la declaración *import*.

Ada package: los paquetes se definen en jerarquías que corresponden a la jerarquía de archivos. La visibilidad es otorgada mediante la clausula *with*.

41