Clases y objetos

Universidad de Chile

Departamento de Ciencias de la Computación

Prof.: Nancy Hitschfeld Kahler

Contenido

- Clases y objetos
- Objetos: estado, comportamiento e identidad
- Tipos de datos abstractos (TDA)
- De TDA a clases
- Ejemplo de TDA genérico en C++
- Ejemplo de TDA genérico en Java

Clases y Objetos

- Objeto:
  - Elemento que surge del análisis del dominio del problema a resolver
  - Instancia de una clase (programación)
- Clase:
  - Tipo de molde o plantilla que define los valores que almacena un objeto en sus variables de instancia, y acciones u operaciones que se le pueden aplicar a través de sus métodos

Clases y objetos

- Puede constar de:
  - Variables de instancia
  - Métodos
  - Variables de la clase (static)
- Ejemplo de una clase en Java:

```
public class Point {
        private int x;   // variable  de instancia
        private int y;
        public Point(){x=0; y=0;} // constructor
        public Point(int _x, int _y){ x=_x;  y= _y;} // constructor
        public int getX(){ return x; }  // método
        public int getY(){ return y; }
        ...
}
```

Clases y objetos

- **Declaración y uso de un objeto de la clase Punto**
  - `Point point = new Point(5,8);`
  - `int x = point.getX();`
- **Características de un objeto**
  - Estado: definido a través de las variables de instancia
  - Comportamiento: definido a través de las operaciones o métodos
  - Identidad: es lo que se preserva a pesar que el estado cambie

Objetos: definición de clases (c++)

```
class Point{
  private:
     int x,y;
  public:
    Point(){ x=0; y=0; }
    Point(int _x, int _y) {
         x = _x; y = _y;
    }
    int getX(){return x;}
    ...
}
class Color{
     float  rgb[3];
  public:
     Color(){ rgb[0] = 1;
              rgb[1] = rgb[2] = 0;
      }
    Color(float r, float g,
          float b){...}
    ...
}
```

```
class Figure{
       Point center;
       Color color;
       ...
   public:
        Figure(){}
        Figure(Point _center,
               Color _color){
           center =_center;
           color = _color;
        }
        void move(Point  point){
            center = point;
        }
        Point where(){
            return center;
        }
        ...
}
```

Objetos: declaración, estado y comportamiento

Uso:

```
Color blue(0,0,1);
Point position_a(1,1);
Point position_b(2,3);
Figure rectangle;
Figure circle(position_a,blue);
rectangle.move(position_b);
...
position_a = rectangle.where();
```

Objetos:

- blue
- position_a
- position_b,
- rectangle
- circle

Estado:

- blue: (0,0,1)
- position_a: (1,1)
- position_b: (2,3)
- rectangle: (0,0) (1,0,0)
- circle: (1,1) (0,0,1)
- rectangle: (2,3) (1,0,0)

- position_a: (2,3)

- Comportamiento de rectangle:
  - move
  - where

Objetos: identidad

C++

```
Point p1(1,2);
Point *p2 = new Point(3,4);
Point *p3 = new Point(1,2);
Point *p4;
p4 = p3;
```

p1, *p2 y *p3 son objetos distintos

*p4 y *p3 son el mismo objeto

Java

```
Point p1 = new Point(1,2);
Point p2 = new Point(3,4);
Point p3 = new Point(1,2);
Point p4 = p3;
```

p1, p2 y p3 contienen referencias a objetos distintos

p3 y p4 contienen referencias al mismo objeto

Tipos de datos abstractos (TDA)[Libro (Mey97)]

- ¿Cómo definir buenos objetos? Con un buen TDA.
- ¿Qué característica debe cumplir un buen TDA? Su descripción debe ser:
  - precisa y no ambigua
  - completa
  - no sobre-especificada
  - independiente de la implementación

Tipos de datos abstractos (TDA)

- Ejemplo: Stack (almacenamiento de datos en una pila)
- Típicas operaciones:
  - push
  - pop
  - top
  - empty
  - new (creador)

Tipos de datos abstractos (TDA)

Contratos (contracts): Forma de definir los derechos y deberes de los clientes y proveedores

- pre-condición (require): precondición que impone los deberes de el cliente
- post-condición(promise): postcondición que impone los deberes del desarrollador de software
- invariante(invariant): expresa propiedades semánticas del TDA, y restricciones de integridad de la clase que lo implementa. Estas deben ser mantenidas por todas sus rutinas

Tipos de datos abstractos (TDA)

Elementos de una especificación formal:

- tipo
- funciones: definen las operaciones (interfaz) del TDA
- axiomas: definen el comportamiento del TDA
- precondiciones: definen parte del contrato

Tipos de datos abstractos (TDA)

Ejemplo: Stack

- Tipo
  - STACK[G], donde G es un tipo arbitrario
- Funciones
  - push: STACK[G]xG ---> STACK[G]
  - pop: STACK[G] -/-> STACK[G]
  - top: STACK[G] -/-> G
  - empty: STACK[G] ---> boolean
  - new: ---> STACK[G]

Tipos de datos abstractos (TDA)

¿Por qué funciones y no procedimientos?

- TDA es un modelo matemático
- Concepto de función ya existe

Clasificación de funciones:

- Creación (new)
- Consulta (top, empty): Obtención de propiedades
- Comando (pop, push): Obtención de nuevas instancias a partir de las ya existentes

Tipos de datos abstractos (TDA)

- Axiomas:
  - top(push(s,x)) = x
  - pop(push(s,x)) = s
  - empty(new) = true
  - not empty(push(s,x)) = true

- Precondiciones (funciones parciales)
  - pop(s:STACK[G]) require not empty(s)
  - top(s:STACK[G]) require not empty(s)

De TDA's a clases

¿Qué es una clase? TDA equipado con una implementación (variables de instancia y algoritmos), posiblemente parcial

- Clase efectiva: especifica completamente su implementación
- Clase abstracta: puede esta implementada parcialmente o nada

¿Cuál es la forma de definir una clase efectiva?

- Especificar un TDA
- Escoger una implementación
- Implementar las funciones, axiomas y precondiciones

De TDA's a clases

¿Cómo se implementa una clase efectiva?

- Parte pública: funciones del TDA
- Parte privada: parte dependiente de la implementación

El TDA se implementa como sigue:

- Comandos procedimientos
- Consultas funciones
- Axiomas postcondicones o invariantes
- Precondiciones precondiciones

De TDA's a clases (en c++)

Aspectos de C++ que permiten la implementación de un TDA

- Inicialización y limpieza de objetos (constructor y destructor)
- Asignación e inicialización (redefinir operador de asignación y redefinir constructor de copia)
- Redefinición de operadores
- Parte pública y privada
- Llamada implícita y explícita de destructores
- Tipos parametrizados (templates)

De TDA's a clases (c++)

```
template<class T>
class Stack{
  public:
    Stack();
    Stack(int);
    Stack(Stack<T>& s);
    ~Stack();
    void push(T elemento);
    void pop();
    T top();
    int empty();
  private:
    T* contenedor;
    int tamano;
    int tope;

    int operator==(Stack<T>& s);
    int invariante(){
        return tamano > 0 && tope >=-1 && tope < tamano && contenedor != NULL;
    }
};
```

De TDA's a clases (c++)

```
template<class T> Stack<T>::Stack(){
    assert(0);
}
template<class T> Stack<T>::Stack(int t){
    assert(t>0);
    tamano = t;
    tope = -1;
    contenedor = new T[t];
    assert(invariante());
}
template<class T> Stack<T>::Stack(Stack<T>& s){
    assert(s.invariante());
    tamano = s.tamano;
    tope = s.tope;
    contenedor = new T[s.tamano];
    for(int i=0; i<=tope; i++)
         contenedor[i] = s.contenedor[i];
    assert(invariante() && s.invariante() && s == *this);
}
```

De TDA's a clases (c++)

```
template<class T> Stack<T>::~Stack(){
    assert(invariante());
    delete contenedor[];
}
template<class T> void Stack<T>::push(T elemento){
    assert(invariante() && (tope+1) < tamano);
    contenedor[tope+1] = elemento;
    tope +=1;
    assert(contenedor[tope] == elemento &&
           invariante());
}
template<class T> void Stack<T>::pop(){
    assert( !empty() && invariante());
    tope -=1;
}
```

De TDA's a clases (c++)

```
template<class T> T Stack<T>::top(){
    assert( !empty() && invariante());
    return contenedor[tope];
}

template<class T> int Stack<T>::operator==(Stack<T>& s){
    assert( invariante() || s.invariante() );

    /* solo el contenido debe der igual */
    if( tope != s.tope ) return 0;

    int i;
    for(i=0; i <= tope;  i++ )
        if( contenedor[i] != s.contenedor[i]) return 0;
    return 1;
}
```

De TDA's a clases (c++)

File: main.C

```
#include <iostream>
#include “stack.h"
main(){
    Stack<int> s(10);
    s.push(5);
    s.push(8);
    s.push(3);
    s.push(-1);
    s.push(20);
    Stack<int> nuevo_stack=s; // inicializacion
    std::cout << "Contenido del stack:\n";
    while( !s.empty() ){
        std::cout << s.top() << "\n";
        s.pop();
    }
    std::cout << "Contenido del stack inicializado:\n";
    while( !nuevo_stack.empty() ){
            std::cout << nuevo_stack.top() << "\n";
            nuevo_stack.pop();
    }
}
```

De TDA's a clases (c++)

File: main.C

```
    Stack<float> *puntero_a_stack  = new Stack<float>(20);

    puntero_a_stack->push(5.8);
    puntero_a_stack->push(1.22);

    delete puntero_a_stack; // destruccion explícita
    // destrucción implícita de s y nuevo_stack
}
```

De TDA´s a clases (java)

Aspectos importantes en Java

- Parte pública y privada
- Inicialización (constructor)
- Definición de objetos genéricos a través del tipo Object y templates
- Recolección automática de basura

De TDA's a clases (java)

Stack generico en java: (version simple usando Object)

file: My_stack.java

```
class My_stack{

    private Object  v[];
    private int top;
    private int max_size;
    public My_stack(int size){
        max_size = size;
        v = new Object[size];
        top = -1;
    }
    public Object top(){
        return v[top];
    }
    public void pop(){ top--;}
    public void push(Object item){
        v[++top] = item;
    }
    public boolean empty(){
         return top == -1;
    }
}
```

De TDA's a clases (java)

Uso de la clase My_stack genérico en java: (version simple)

file: Example.java

```
import My_stack;

public class Example {

    public static void main(String args[]){

           My_stack s1 = new My_stack(100);

           s1.push(new Integer(1));
           s1.push(new Integer(2));
           s1.push(new Integer(3));

           while( !s1.empty() ){
                System.out.print(s1.top() + " ");
                s1.pop();
           }
    }
}
```

De TDA´s a clases (java)

```
class My_stack{  // Stack genérico con contrato

    private Object  v[];
    private int top;
    private int max_size;

    private boolean invariant(){ return -1 <= top && top < max_size && v !=null;}
    public My_stack(int size){
        assert size > 0 : “size must be greater than 0”;
        max_size = size;
        v = new Object[size];
        top = -1;
        assert invariant() : “invalid invariant”;
    }
    public Object top(){
        assert invariant() : “invalid invariant”;
        assert !empty() : “stack empty”;
        return v[top];
    }
    public void pop(){
        assert invariant() : “invalid invariant”;
        assert !empty() : “stack empty”;
        top--;
       assert invariant() : “invalid invariant”;
    }
```

De TDA's a clases (java)

```
 public void push(Object item){
        assert (top + 1 < max_size) : “stack full”;
        assert invariant() : “invalid invariant ”;
        v[++top] = item;
        assert invariant() : “invalid invariant”;
    }
 public boolean empty(){
        assert invariant() : “invalid invariant”;
        return top == -1;
    }
}
```

De TDA's a clases (java)

```
class My_stack<T>{  // Stack genérico con templates  y contrato

    private T  v[];
    private int top;
    private int max_size;

    private bolean invariant(){ return -1 <= top && top < max_size && v !=null;}
    public My_stack(int size){
        assert size > 0 : “size must be greater than 0”;
        max_size = size;
        v = new T[size];
        top = -1;
        assert invariant() : “invalid invariant”;
    }
    public T top(){
        assert invariant() : “invalid invariant”;
        assert !empty() : “stack empty”;
        return v[top];
    }
    public void pop(){
        assert invariant() : “invalid invariant”;
        assert !empty() : “stack empty”;
        top--;
        assert invariant() : “invalid invariant”;
    }
```

De TDA's a clases (java)

```
 public void push(T item){
        assert (top+1 < max_size) : “stack full”;
        assert invariant() : “invalid invariant ”;
        v[++top] = item;
        assert invariant() : “invalid invariant”;
    }
 public boolean empty(){
        assert invariant() : “invalid invariant”;
        return top == -1;
    }
}
```