Principios SOLID y patrón Estrategia

David Arteaga Sánchez

Índice

Introducción

En este informe de la asignatura *Diseño y Análisis de Algoritmos* de la *Universidad de La Laguna* veremos una breve descripción de los principios SOLID así como una explicación del patrón Estrategia. También se comentarán y explicarán brevemente los algoritmos proveídos por los profesores para esta práctica.

Principios SOLID

Los principios SOLID son una serie de palabras sobre los fundamentos de la arquitectura y el desarrollo del software. Define una serie de reglas que se recomienda seguir a la hora de crear un código orientado a objetos. Sus siglas significan:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

En este informe solo se explicará el *Single Responsibility Principle*.

Principio de Responsabilidad Única

Este principio establece que cada módulo o clase de nuestro código debería ser responsable únicamente de una parte de la funcionalidad que proveerá nuestro programa y esta responsabilidad debe estar encapsulada en su totalidad por la clase.

Polimorfismo

En programación orientada a objetos, el polimorfismo se refiere a una relajación del sistema de tipos de tal manera que una referencia a una clase acepta direcciones de objetos de dicha clase y de sus clases derivadas. Esto se consigue mediante la herencia.

Código:

ejemploPolimorfismo0.cpp

En este código se crean dos clases *ClaseA* y *ClaseB* que hereda de esta primera. ClaseA cuenta con un atributo protected *datoA* que heredará la *ClaseB* y esta a su vez contará con otro llamado *datoB*.

```
class ClaseA {
  public:
   ClaseA() : datoA(10) {}
   int LeerA() const { return datoA; }
   void Mostrar() {
      cout << "a = " << datoA << endl; // (1)
   }
  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {}
   int LeerB() const { return datoB; }
   void Mostrar() {
      cout << "a = " << datoA << ", b = "
           << datoB << endl; // (2)
   }
  protected:
   int datoB;
};
```

ClaseA y ClaseB del programa 0

En el programa principal se crea un objeto de la clase *ClaseB* y se llama al método Mostrar() que imprime los valores de *datoA* y *datoB*. A continuación se llama al método mostrar de la clase padre ClaseA y vemos que este solo imprime *datoA*.

```
PS C:\Users\David\Documents\ULL\DAA\P1\polimorfismo> .\ejemplo0.exe
a = 10, b = 20
a = 10
```

Salida del programa ejemploPolimorfismo0.cpp

ejemploPolimorfismo1.cpp

Contamos de nuevo con dos clases ClaseA y Clase B, que hereda de A. Ambas tienen un método Incrementar, el de la ClaseA imprime un mensaje y el de la clase B otro.

```
class ClaseA {
  public:
   void Incrementar() { cout << "Suma 1" << endl; }
   void Incrementar(int n) { cout << "Suma " << n << endl; }
};

class ClaseB : public ClaseA {
  public:
   void Incrementar() { cout << "Suma 2" << endl; }
};
```

ClaseA y ClaseB el programa 1

En el programa principal se crea un objeto de la ClaseB y se llama al método incrementar desde el objeto. Posteriormente se llama desde el objeto al método Incrementar() pero esta vez al de la ClaseA. En la salida podemos ver cómo ambos métodos imprimen cosas distintas aunque se llamen desde un mismo objeto.

ejemploPolimorfismo2.cpp

En este caso contamos con tres clases. Una clase padre *Persona* y dos clases derivadas *Empleado* y *Estudiante*.

```
class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   void VerNombre() { cout << "Persona:" << nombre << endl; }
  protected:
   char nombre[30];
};

class Empleado : public Persona {
  public:
   Empleado(char *n) : Persona(n) {}
   void VerNombre() {
      cout << "Empleado: " << nombre << endl;
   }
};

class Estudiante : public Persona {
  public:
   Estudiante(char *n) : Persona(n) {}
   void VerNombre() {
      cout << "Estudiante: " << nombre << endl;
   }
};
```

Clases Persona Empleado y Estudiante

En el programa principal se crean distintas instancias de las clases. Algunas con punteros a la clase *Persona* y otros con punteros a su clase específica.

```
int main() {

   Persona *Pepito = new Estudiante((char *) "Jose");
   Persona *Carlos = new Empleado((char *) "Carlos");
   Estudiante *EPepito = new Estudiante((char *) "Jose");
   Empleado *ECarlos = new Empleado((char *) "Carlos");
   Carlos->VerNombre();
   ECarlos->VerNombre();
   Pepito->VerNombre();
   EPepito->VerNombre();
   delete Pepito;
   delete Carlos;
   delete EPepito;
   delete ECarlos;

   return 0;
}
```

Función main del programa ejemploPolimorfismo2.cpp

Curiosamente los objetos creados con el puntero *Persona* llaman a los métodos de estos en vez de a los métodos del objeto que están instanciando. Para evitar esto habría que usar métodos virtual. Esto lo veremos en el siguiente ejemplo.

```
PS C:\Users\David\Documents\ULL\DAA\P1\polimorfismo> .\ejemplo2.exe
Persona:Carlos
Empleado: Carlos
Persona:Jose
Estudiante: Jose
```

Salida de ejemplo2.cpp

ejemploPolimorfismo3.cpp

Este programa es una ampliación del anterior. Esta vez el método *VerNombre()* se ha declarado virtual. Esto permitirá al compilador saber a qué método nos referimos cuando hagamos uso de polimorfismo.

```
PS C:\Users\David\Documents\ULL\DAA\P1\polimorfismo> .\ejemplo3.exe
Empleado: Carlos
Empleado: Carlos
Estudiante: Jose
Estudiante: Jose
```

Salida de ejemplo3.cpp

Como se puede ver en la imagen, aunque los objetos se han instanciado con un puntero a la clase *Persona*, sabe que estamos intentando llamar al método de la clase derivada.

ejemploPolimorfismo4.cpp

De nuevo este programa no es más que una modificación del anterior. En este caso se han creado los objetos y se han inicializado unas referencias de la clase *Persona* para ver cómo se comporta al llamar al método *VerNombre()*.

```
int main() {
    Estudiante Pepito("Jose");
    Empleado Carlos("Carlos");
    Persona &rPepito = Pepito; // Referencia como Persona
    Persona &rCarlos = Carlos; // Referencia como Persona

    rCarlos.VerNombre();
    rPepito.VerNombre();

    return 0;
}
```

Función main del programa ejemploPolimorfismo4.cpp

Al ejecutar el programa vemos que aún así no se confunde y llama a los métodos de las clases derivadas.

```
PS C:\Users\David\Documents\ULL\DAA\P1\polimorfismo> .\ejemplo4.exe
Empleado: Carlos
Estudiante: Jose
```

Salida del programa ejemploPolimorfismo4.cpp

Patrón Estrategia

El patrón Estrategia es un conocido patrón de diseño orientado al desarrollo de software. Se trata de un patrón de comportamiento ya que determina cómo se debe realizar el intercambio de mensajes entre diferentes *objetos* para resolver una tarea. El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el *objeto cliente* puede elegir aquel que le conviene e *intercambiarlo dinámicamente* según sus necesidades.

Código:

estrategiaplantilla.cpp

En este programa podemos observar que tenemos tres clases: SortBubble, SortShell y la clase Stat. Estas dos primeras tienen un método llamado *sort()* que simplemente imprime por pantalla el nombre de la clase que lo ha llamado.

```
class SortBubble {
public:
   void sort( int v[], int n ) {
   cout << "SortBubble" << endl;
   }
};

class SortShell {
public:
   void sort( int v[], int n ) {
   cout << "SortShell" << endl;
   }
};
```

Clases SortBubble y SortShell

A continuación tenemos la clase Stat que tiene el método readVector() que toma como parámetro un vector y se encarga de almacenar el primer, el último y el intermedio elemento del vector en las variables min_, max y med_ respectivamente.

Finalmente el programa inicializa un vector al azar y crea dos objetos de la clase Stat, cada uno rellenando la plantilla con una de las dos clases disponibles: SortBubble y SortShell. Al llamar al método readVector() que a su vez llama al método sort() podemos observar como cada uno de los objetos tiene una salida distinta, en función de la estrategia que hayamos escogido.

```
PS C:\Users\David\Documents\ULL\DAA> .\estrategiaplantilla.exe
Vector: 5 8 8 8 9 5 4 8 3
SortBubble
min is 5, max is 3, median is 9
SortShell
min is 5, max is 3, median is 9
```

Salida del programa estrategiaplantilla.cpp

estructura.cpp

En este caso contamos con una clase abstracta llamada *Strategy* y una serie de clases concretas que en este documento llamaremos *A*, *B* y *C* para abreviar. Cada clase concreta tiene la implementación de un método común abstracto llamado *AlgortihmInterface()* que simplemente imprime por pantalla el nombre de la clase que lo ha llamado.

```
// The 'Strategy' abstract class
class Strategy {
    public:
    virtual void AlgorithmInterface() = 0;
 };


// A 'ConcreteStrategy' class
class ConcreteStrategyA : public Strategy  {
    void AlgorithmInterface() {
     cout <<  "Called ConcreteStrategyA.AlgorithmInterface()" << endl;
    }
};


// A 'ConcreteStrategy' class
class ConcreteStrategyB : public Strategy  {
    void AlgorithmInterface() {
     cout <<  "Called ConcreteStrategyB.AlgorithmInterface()" << endl;
    }
};

// A 'ConcreteStrategy' class
class ConcreteStrategyC : public Strategy  {
    void AlgorithmInterface() {
     cout <<  "Called ConcreteStrategyC.AlgorithmInterface()" << endl;
    }
};
```

Clase abstracta Strategy y sus implementaciones concretas

Después el código cuenta con la clase Context que tiene como atributo un vector a una clase Strategy que apuntará a una implementación concreta de las que hablamos anteriormente. También cuenta con un método que simplemente llama al método *AlgorithmInterface()* de las clases anteriores.

En el programa principal se crean tres contextos que contarán cada uno con una estrategia diferente de las tres disponibles. Finalmente todas las clases *Context* llaman al método *ContextInterface()* y vemos como cada uno imprime el nombre de cada una de las clases que mencionamos anteriormente.

```
PS C:\Users\David\Documents\ULL\DAA> .\estructura.exe
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyC.AlgorithmInterface()
```

estructura1.cpp

Este programa es una modificación del anterior. En él vemos como se ha añadido un enum llamado TYPESTRATEGY que se usará para seleccionar el tipo de estrategia en tiempo de ejecución. También se ha creado el método *setstrategy* para eliminar la estrategia seleccionada anteriormente y cambiarla por otra que introduzca el usuario.

```
void setstrategy(Strategy *strategy ) {
   delete _strategy;
   _strategy = strategy;
}
```

Método setstrategy

Finalmente en el programa principal, se inicializan los contextos de las dos maneras posibles: mediante un puntero usando el operador new o mediante el nuevo método *setstrategy()*. Al ejecutarlo vemos que las dos formas de seleccionar una estrategia funcionan.

```
PS C:\Users\David\Documents\ULL\DAA> .\estructura1.exe
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
```

Salida del programa estructura1.cpp

estructura2.cpp

Una vez más contamos con una ampliación de los anteriores programas. En este caso se ha añadido una nueva clase concreta llamada *ConcreteStrategyD*. También se crea la nueva clase *NewContext* derivada de la clase Context. La existencia de esta nueva estructura de datos se justifica en que ahora el método *setstrategy(TYPESTRATEGY)* también incluirá la opción de seleccionar la estrategia D en la nueva clase *NewContext*.

```
// The 'Context' class
class Newcontext: public Context {
    public:
        void setstrategy(TYPESTRATEGY type ) {
            delete _strategy;
            if (type == D)
                _strategy = new ConcreteStrategyD();
            else
            Context::setstrategy(type);
        }
};
```

Clase NewContext

En el programa principal se inicializan dos clases Context y una NewContext. Vemos que el método setstrategy(D) solo funciona con la clase NewContext que es el que tiene la opción disponible.

```
PS C:\Users\David\Documents\ULL\DAA> .\estructura2.exe
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyD.AlgorithmInterface()
ERROR: Stratey not known
ERROR: Strategy not set
Called ConcreteStrategyD.AlgorithmInterface()
```

Salida del programa estructura2.cpp