

Practica 1 - SOLID, Estrategia y Análisis de un algoritmo

Universidad de La Laguna
Grado en Ingeniería Informática
Diseño y Análisis de Algoritmos
Autor: Adal Díaz Fariña
Contacto: alu0101112251@ull.edu.es

Índice

Índice	1
1. ¿En qué consisten estos principios? ¿En qué consiste el principio de responsabilidad única?	2
2. Repaso del concepto de Polimorfismo	4
3. Patrón estrategia	10

1. ¿En qué consisten estos principios? ¿En qué consiste el principio de responsabilidad única?

Los principios SOLID son una serie de reglas o directrices a seguir para mejorar el diseño orientado a objetos. Existen 5 principios SOLID:

1. Principio de la responsabilidad única: "Nunca debe haber una razón para cambiar una clase". - Robert Martin.

El principio de la responsabilidad única lo podemos traducir como que cada clase debe hacer una única tarea. Es decir, una clase debe tener una sola responsabilidad. Por ejemplo: si tenemos una clase de números complejos, esta clase debe ser capaz de representar números complejos, de realizar operaciones con números complejos, etc. Su responsabilidad es la de poder tratar con números complejos. Cualquier método que no tenga que ver con los números complejos queda exento de estar en esta clase.

2. Principio abierto cerrado: "Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para la modificación". — Robert Martin parafraseando a Bertrand Meyer.

La idea principal de este principio es que no hay necesidad de modificar una clase que ya hemos creado, si necesitamos hacer uso de esa clase modificando algunos métodos podemos hacer uso de técnicas orientadas a objetos como la herencia y la composición para modificarla o aumentarla.

3. Principio de sustitución de Liskov: "Las funciones que usan punteros o referencias a clases base deben poder usar objetos de clases derivadas sin saberlo". — Robert Martin

El principio de sustitución de Liskov nos indica que deberíamos poder sustituir la instancia de una subclase por la clase principal y todo debería seguir funcionando correctamente.

4. Principio de segregación de interfaz: "Los clientes no deben verse obligados a depender de interfaces que no utilizan". — Robert Martin**

Hay que mantener las interfaces pequeñas y cohesivas. Esto es debido a que si una interfaz es muy amplia, se está poniendo una enorme carga de implementación en cualquiera que quiera adherirse a ese contrato. Ejemplo: "cuando un cliente depende de una clase que contiene interfaces que el cliente no usa, pero que otros clientes sí usan, entonces ese cliente se verá afectado por los cambios que esos otros clientes fuerzan en la clase"

5. Principio de inversión de dependencia:

- A. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones.
- B. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones". — Robert Martin

Básicamente lo que nos dice este principio es que si una clase tiene dependencias de otra clase, debe basarse en las interfaces de las dependencias en vez de sus tipos concretos. La idea es que nuestra clase dependa de abstracciones. Así si todos los detalles de nuestras abstracciones cambian, nuestra clase seguirá estando a salvo.

2. Repaso del concepto de Polimorfismo

En este apartado vamos a repasar sobre Polimorfismo comentando los ejemplos que nos han facilitado en la asignatura.

1. Ejemplo polimorfismo 0

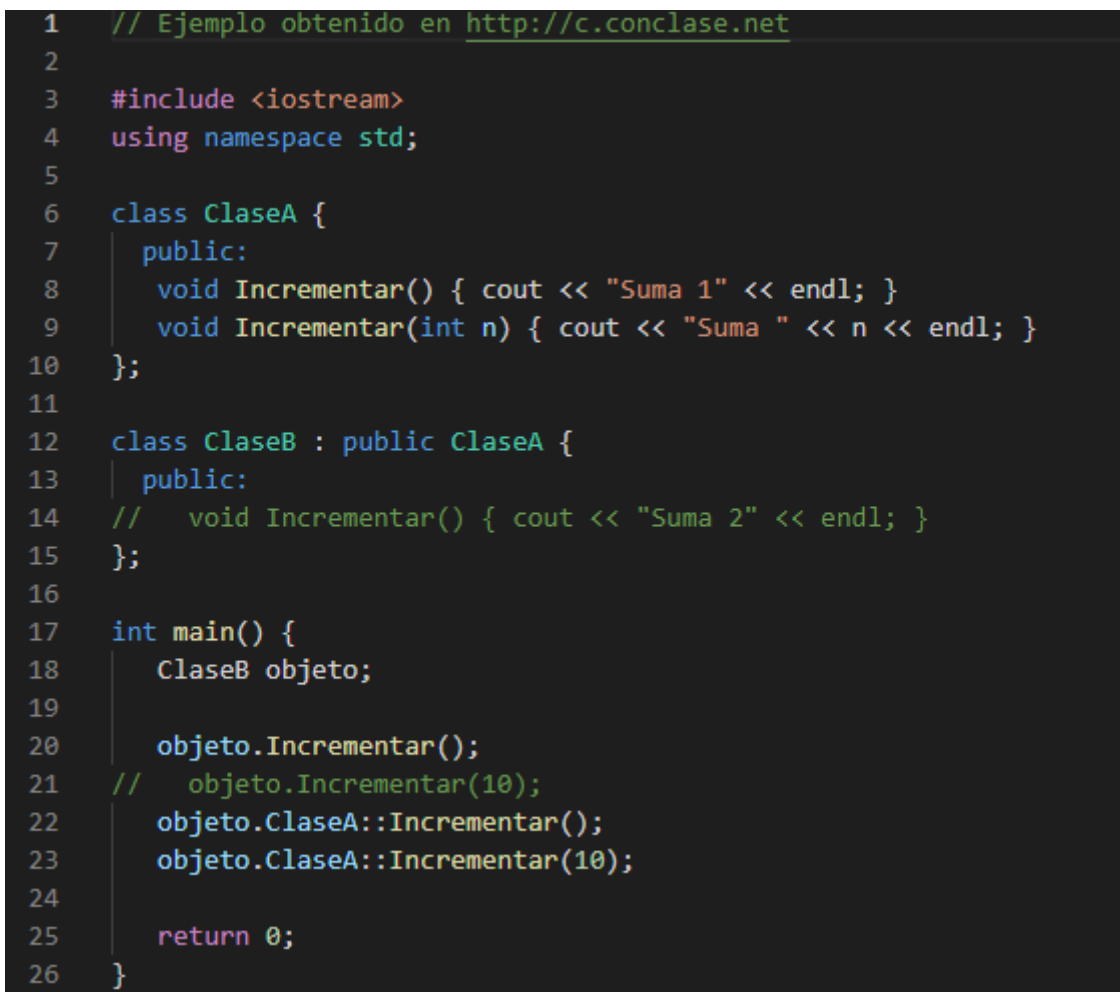
```
1  #include <iostream>
2  using namespace std;
3
4  class ClaseA {
5  public:
6      ClaseA() : datoA(10) {}
7      int LeerA() const { return datoA; }
8      void Mostrar() {
9          cout << "a = " << datoA << endl; // (1)
10     }
11     protected:
12         int datoA;
13 };
14
15 class ClaseB : public ClaseA {
16 public:
17     ClaseB() : datoB(20) {}
18     int LeerB() const { return datoB; }
19     void Mostrar() {
20         cout << "a = " << datoA << ", b = "
21         << datoB << endl; // (2)
22     }
23     protected:
24         int datoB;
25 };
26
27 int main() {
28     ClaseB objeto;
29
30     objeto.Mostrar();
31     objeto.ClaseA::Mostrar();
32
33     return 0;
34 }
```

Como podemos ver en la imagen en este ejemplo tenemos dos clases: Una clase base o padre que es la clase A y una clase hija o derivada que es la clase B. Como podemos ver en el main en la línea 28, se define un objeto de la clase B y luego en la línea 30 se llama al método mostrar por lo que esperamos como resultado $a = 10$, $b = 20$. Esto es debido a que estamos llamando al método mostrar de la clase B. En la línea a continuación se vuelve a llamar al método mostrar pero haciendo uso del polimorfismo esta vez se llama al método mostrar de la clase A y el resultado que esperamos es $a = 10$. En la primera llamada al método mostrar sabemos que a es igual 10 porque así se declara en la clase padre (la clase A) y al ser una atributo de tipo protected se hereda como un atributo privado en la clase hija (la clase B).



```
./Polimorfismo
→ Polimorfismo git:(main) ./ejemploPolimorfismo0
a = 10, b = 20
a = 10
```

2. Ejemplo polimorfismo 1



```
1 // Ejemplo obtenido en http://c.conclase.net
2
3 #include <iostream>
4 using namespace std;
5
6 class ClaseA {
7 public:
8     void Incrementar() { cout << "Suma 1" << endl; }
9     void Incrementar(int n) { cout << "Suma " << n << endl; }
10 };
11
12 class ClaseB : public ClaseA {
13 public:
14     // void Incrementar() { cout << "Suma 2" << endl; }
15 };
16
17 int main() {
18     ClaseB objeto;
19
20     objeto.Incrementar();
21     // objeto.Incrementar(10);
22     objeto.ClaseA::Incrementar();
23     objeto.ClaseA::Incrementar(10);
24
25     return 0;
26 }
```

En este ejemplo, da igual que estemos usando un objeto de la clase A o la clase B. Al la clase B no tener implementado estos métodos va a utilizar los de la clase padre (la clase A), por lo que los resultados esperados son: Suma 1, Suma 1, y Suma 10

```
→ Polimorfismo git:(main) x ./ejemploPolimorfismo1
Suma 1
Suma 1
Suma 10
```

Si descomentamos las líneas de código de la línea 14 ahora cuando el objeto llama a incrementar sin aplicar el polimorfismo llamamos al incrementar de la clase B.

```
→ Polimorfismo git:(main) x ./ejemploPolimorfismo1
Suma 2
Suma 1
Suma 10
```

Lo interesante ocurre cuando descomentamos la línea 21. Cuando descomentamos la línea 21 nos devolverá un error esto es debido a que no se puede acceder a las funciones superpuestas de la clase padre, aunque tengan diferentes número de parámetros

3. Ejemplo polimorfismo 2

```
#include <iostream>
#include <cstring>
using namespace std;

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;
    }
};

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;
}
```

Como podemos ver en la imagen tenemos cuatro punteros y a través de esos punteros estamos llamando a los métodos VerNombre. Lo curioso es que aunque digamos que Pepito es un nuevo estudiante al puntero ser de tipo Persona, cuando llamamos al método VerNombre se está llamando al del Persona y no al de estudiante. Es decir se ejecuta el método VerNombre que se definio a la clase base. Entonces la salida es la correspondiente:

```
→ Polimorfismo git:(main) x ./ejemploPolimorfismo2
Persona:Carlos
Empleado: Carlos
Persona:Jose
Estudiante: Jose
```

4. Ejemplo polimorfismo 3

```
3  #include <iostream>
4  #include <cstring>
5  using namespace std;
6
7  class Persona {
8  public:
9      Persona(char *n) { strcpy(nombre, n); }
10     virtual void VerNombre() { cout << "Persona: " << nombre << endl; }
11 protected:
12     char nombre[30];
13 };
14
15 class Empleado : public Persona {
16 public:
17     Empleado(char *n) : Persona(n) {}
18     void VerNombre() {
19         cout << "Empleado: " << nombre << endl;
20     }
21 };
22
23 class Estudiante : public Persona {
24 public:
25     Estudiante(char *n) : Persona(n) {}
26     void VerNombre() {
27         cout << "Estudiante: " << nombre << endl;
28     }
29 };
30
31 int main() {
32
33     Persona *Pepito = new Estudiante((char *) "Jose");
34     Persona *Carlos = new Empleado((char *) "Carlos");
35     Estudiante *EPepito = new Estudiante((char *) "Jose");
36     Empleado *ECarlos = new Empleado((char *) "Carlos");
37     Carlos->VerNombre();
38     ECarlos->VerNombre();
39     Pepito->VerNombre();
40     EPepito->VerNombre();
41     delete Pepito;
42     delete Carlos;
43     delete EPepito;
44     delete ECarlos;
45
46     return 0;
47 }
```


Si comparamos el ejemplo anterior con el actual es el mismo menos que en la línea 10 le añadimos virtual delante. Gracias al virtual nos da un resultado totalmente diferente. Lo interesante cuando definimos antes a Pepito como un estudiante y llamamos al método VerNombre era que llamará al método de la clase hija estudiante pero no era así. Ahora con virtual en vez de llamar al método de la clase padre llamamos al método de la clase hija por lo que el resultado nos queda así:

```
→ Polimorfismo git:(main) x ./ejemploPolimorfismo3
Empleado: Carlos
Empleado: Carlos
Estudiante: Jose
Estudiante: Jose
```

5. Ejemplo polimorfismo 4

```
2
3 #include <iostream>
4 #include <cstring>
5 using namespace std;
6
7 class Persona {
8 public:
9     Persona(const char *n) { strcpy(nombre, n); }
10 virtual void VerNombre() {
11     cout << "Persona: " << nombre << endl;
12 }
13 protected:
14     char nombre[30];
15 };
16
17 class Empleado : public Persona {
18 public:
19     Empleado(const char *n) : Persona(n) {}
20 void VerNombre() {
21     cout << "Empleado: " << nombre << endl;
22 }
23 };
24
25 class Estudiante : public Persona {
26 public:
27     Estudiante(const char *n) : Persona(n) {}
28 void VerNombre() {
29     cout << "Estudiante: " << nombre << endl;
30 }
31 };
32
33 int main() {
34     Estudiante Pepito("Pepito");
35     Empleado Carlos("Carlos");
36     Persona &rPepito = Pepito; // Referencia como Persona
37     Persona &rCarlos = Carlos; // Referencia como Persona
38
39     rCarlos.VerNombre();
40     rPepito.VerNombre();
41
42     return 0;
43 }
```

En este caso funciona igual que el caso anterior pero con referencias. Como sigue siendo un método virtual VerNombre, lo seguirá siendo en las clase heredadas así que la solución será la siguiente:

```
→ Polimorfismo git:(main) x ./ejemploPolimorfismo4
Empleado: Carlos
Estudiante: Pepito
```

3. Patrón estrategia

En este apartado vamos a repasar los patrones de estrategia utilizando los ejemplos que nos han facilitado en la asignatura.

1. Ejemplo estrategia plantilla

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

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

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

template<class STRATEGY>
class Stat {
public:
    void readVector( int v[], int n ) {
        imp_.sort( v, n );
        min_ = v[0];    max_ = v[n-1];
        med_ = v[n/2];
    }
    int getMin() { return min_; }
    int getMax() { return max_; }
```

```

    int getMed() { return med_; }
private:
    int      min_, max_, med_;
    STRATEGY imp_;
};

int main( void ) {
    const int NUM = 9;
    int      array[NUM];
    time_t   t;
    srand((unsigned) time(&t));
    cout << "Vector: ";
    for (int i=0; i < NUM; i++) {
        array[i] = rand() % 9 + 1;
        cout << array[i] << ' ';
    }
    cout << endl;

    Stat<SortBubble>  obj;
    obj.readVector( array, NUM );
    cout << "min is " << obj.getMin()
        << ", max is " << obj.getMax()
    << ", median is " << obj.getMed()
        << endl;

    Stat<SortShell>  two;
    two.readVector( array, NUM );
    cout << "min is " << two.getMin()
        << ", max is " << two.getMax()
        << ", median is " << two.getMed()
        << endl;
};

```

Esta estrategia tiene 3 puntos claves. El primero sería la clase Stat que contiene que es lo que queremos hacer que en este caso es obtener el máximo, mínimo y el valor medio de un array. El otro sería que definimos una template para ver qué estrategia se utiliza para ordenar el array en la clase Stat y por último nos encontramos con las clases SortBubble y SortShell que son las dos estrategias de ordenación que podemos utilizar en este ejemplo. Ese sería resumido lo que hace este patrón. A través de la template indicamos que estrategia vamos a utilizar para obtener lo que queremos con la clase Stat.

```
→ Estrategia git:(main) x ./estrategiaplantilla
Vector: 8 8 4 5 6 8 9 5 8
SortBubble
min is 8, max is 8, median is 6
SortShell
min is 8, max is 8, median is 6
```

2. Ejemplo estructura

```
// Strategy pattern -- Structural example

#include <iostream>
using namespace std;

// 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;
    }
};
```

```

// The 'Context' class
class Context {
private:
    Strategy *_strategy;
public:
    // Constructor
    Context(Strategy *strategy) {
        _strategy = strategy;
    }

    void ContextInterface() {
        _strategy -> AlgorithmInterface();
    }
};

int main() {
    Context *context_a, *context_b, *context_c;

    // Three contexts following different strategies

    context_a = new Context(new ConcreteStrategyA());
    context_a->ContextInterface();

    context_b = new Context(new ConcreteStrategyB());
    context_b->ContextInterface();

    context_c = new Context(new ConcreteStrategyC());
    context_c->ContextInterface();
}

```

En esta estrategia tenemos 3 partes importantes. La primera sería el contexto que nos permite utilizar las diferentes funcionalidades que necesitamos sin conocer cómo funcionan por detrás. Es decir, nos abstrae de la lógica de la aplicación. Luego tenemos la clase Strategy que es el padre de todas las estrategias y por último tenemos a las diferentes clases hijas que componen el conjunto de estrategias disponibles. Es decir, a través de la herencia podemos usar una misma clase cambiando la interfaz que queremos en cada caso. Usando context basta con definir

la estrategia que vamos a utilizar y la clase context ya nos permite utilizar las funcionalidades que necesitamos.

El resultado de la ejecución es el siguiente:

```
→ Estrategia git:(main) x ./estructura
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyC.AlgorithmInterface()
```

3. Ejemplo estructura 1

```
// Strategy pattern -- Structural example

#include <iostream>
using namespace std;

enum TYPESTRATEGY{A, B, C};

// 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;
    }
};

// The 'Context' class
class Context {
private:
    Strategy *_strategy;
public:
    // Constructor
    Context() {
        _strategy = NULL;
    }
    // Constructor
    Context(Strategy *strategy) {
        _strategy = strategy;
    }

    void setstrategy(TYPESTRATEGY type ) {
        delete _strategy;
        if (type == A)
            _strategy = new ConcreteStrategyA();
        else if (type == B)
            _strategy = new ConcreteStrategyB();
        else if (type == C)
            _strategy = new ConcreteStrategyC();
        else
            cout << "ERROR: Stratey not known" << endl;
    }

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

    void ContextInterface() {
        _strategy -> AlgorithmInterface();
    }
};

int main() {

```

```

Context *pcontext;
Context context;

pcontext = new Context(new ConcreteStrategyA());
context.setstrategy(A);

pcontext->ContextInterface();
context.ContextInterface();

pcontext -> setstrategy(new ConcreteStrategyB());
context.setstrategy(B);

pcontext->ContextInterface();
context.ContextInterface();

}

```

El ejemplo de estructura 1 es una continuación del ejemplo de estructura. Lo único que añade es que ahora podemos enumerar las estrategias. Como podemos ver en el main queda mucho más legible si enumeramos las estrategias. Pero la lógica es exactamente la misma.

El resultado de la ejecución es el siguiente:

```

→ Estrategia git:(main) x ./estructural
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()

```

4. Ejemplo estructura 2

```

// Strategy pattern -- Structural example

#include <iostream>
using namespace std;

enum TYPESTRATEGY{A, B, C, D};

// 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;
    }
};

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

// The 'Context' class
class Context {
protected:
    Strategy *_strategy;
public:
    // Constructor
    Context() {
        _strategy = NULL;
    }
    // Constructor
    Context(Strategy *strategy) {

```

```

        _strategy = strategy;
    }

    void setstrategy(TYPESTRATEGY type ) {
delete _strategy;
if (type == A)
    _strategy = new ConcreteStrategyA();
else if (type == B)
    _strategy = new ConcreteStrategyB();
else if (type == C)
    _strategy = new ConcreteStrategyC();
else {
cout << "ERROR: Strategy not known" << endl;
_strategy = NULL;
}
}

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

    void ContextInterface() {
if (_strategy)
    _strategy -> AlgorithmInterface();
else
cout << "ERROR: Strategy not set" << endl;
}

};

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

int main() {

```

```

Context *pcontext;
Context context;
Newcontext newcontext;

pcontext = new Context(new ConcreteStrategyA());
context.setstrategy(A);
newcontext.setstrategy(A);

pcontext->ContextInterface();
context.ContextInterface();
newcontext.ContextInterface();

pcontext -> setstrategy(new ConcreteStrategyD());
pcontext->ContextInterface();

context.setstrategy(D);
context.ContextInterface();

newcontext.setstrategy(D);
newcontext.ContextInterface();
}

```

En este último ejemplo disponemos de una última estrategia y formas de acceder a ella. Con el puntero no va a ver ningún problema, mientras el puntero a esa estrategia exista va a funcionar. Con el objeto context depende. Al no estar todos los tipos definidos en context como es el caso de D, si decimos que la estrategia va a ser la D va a fallar porque no contempla ese tipo. Para poder hacer uso de la estrategia D necesitamos usar el puntero o el objeto newcontent que está en la clase NewContent. Esta clase si contempla el tipo D aparte de los tres tipos previos de estrategia que hemos visto.

El resultado de la ejecución es el siguiente:

```

→ Estrategia git:(main) x ./estructura2
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyD.AlgorithmInterface()
ERROR: Strategy not known
ERROR: Strategy not set
Called ConcreteStrategyD.AlgorithmInterface()

```