SPRINT No. 1

Nombre: Greicy Belinth Hernández Oviedo

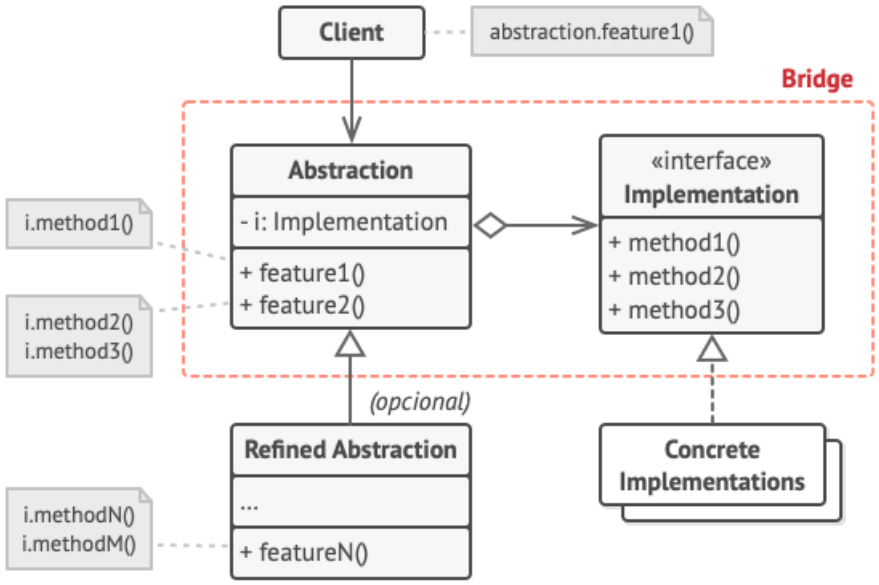
Ciclo: 4 - **Grupo:** G29

Punto 1: 2 patrones de diseño explicados con 2 ejemplos de la vida real y con código fuente.

Patrón estructural: Bridge.

En este patrón hay una alteración estructural en las clases principales y de implementador de interfaz sin tener ningún efecto entre ellas. Estas dos clases pueden desarrollarse de manera independiente y solo se conectan utilizando una interfaz como puente.

Estructura:



Ejemplo No. 1:

Gracias a este “puente” se puede configurar un producto que permite calcular el precio de este en cuotas dependiendo del método de pago con diferentes bancos emisores y diferentes tarjetas de crédito.

Ejemplo No. 2:

También se pueden crear algoritmos que permitan ordenar, según lo que desee el usuario en el momento de la ejecución, de manera descendiente o ascendente.

Código fuente en C++:

```
/**
 * The Implementation defines the interface for all implementation
classes. It
 * doesn't have to match the Abstraction's interface. In fact, the two
 * interfaces can be entirely different. Typically the Implementation
interface
 * provides only primitive operations, while the Abstraction defines
higher-
 * level operations based on those primitives.
 */

class Implementation {
 public:
  virtual ~Implementation() {}
  virtual std::string OperationImplementation() const = 0;
};

/**
 * Each Concrete Implementation corresponds to a specific platform and
 * implements the Implementation interface using that platform's API.
 */
class ConcreteImplementationA : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationA: Here's the result on the platform
A.\n";
  }
};
class ConcreteImplementationB : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationB: Here's the result on the platform
B.\n";
  }
};

/**
 * The Abstraction defines the interface for the "control" part of the
two class
 * hierarchies. It maintains a reference to an object of the
Implementation
 * hierarchy and delegates all of the real work to this object.
 */

class Abstraction {
  /**
   * @var Implementation
   */
 protected:
```

```
  Implementation* implementation_;

 public:
  Abstraction(Implementation* implementation) :
implementation_(implementation) {
  }

  virtual ~Abstraction() {
  }

  virtual std::string Operation() const {
    return "Abstraction: Base operation with:\n" +
           this->implementation_->OperationImplementation();
  }
};
/**
 * You can extend the Abstraction without changing the Implementation
classes.
 */
class ExtendedAbstraction : public Abstraction {
 public:
  ExtendedAbstraction(Implementation* implementation) :
Abstraction(implementation) {
  }
  std::string Operation() const override {
    return "ExtendedAbstraction: Extended operation with:\n" +
           this->implementation_->OperationImplementation();
  }
};

/**
 * Except for the initialization phase, where an Abstraction object gets
linked
 * with a specific Implementation object, the client code should only
depend on
 * the Abstraction class. This way the client code can support any
abstraction-
 * implementation combination.
 */
void ClientCode(const Abstraction& abstraction) {
  // ...
  std::cout << abstraction.Operation();
  // ...
}
/**
 * The client code should be able to work with any pre-configured
abstraction-
 * implementation combination.
 */
```

```
int main() {
  Implementation* implementation = new ConcreteImplementationA;
  Abstraction* abstraction = new Abstraction(implementation);
  ClientCode(*abstraction);
  std::cout << std::endl;
  delete implementation;
  delete abstraction;

  implementation = new ConcreteImplementationB;
  abstraction = new ExtendedAbstraction(implementation);
  ClientCode(*abstraction);

  delete implementation;
  delete abstraction;

  return 0;
}
```

Al ejecutar el código se obtiene el siguiente resultado:

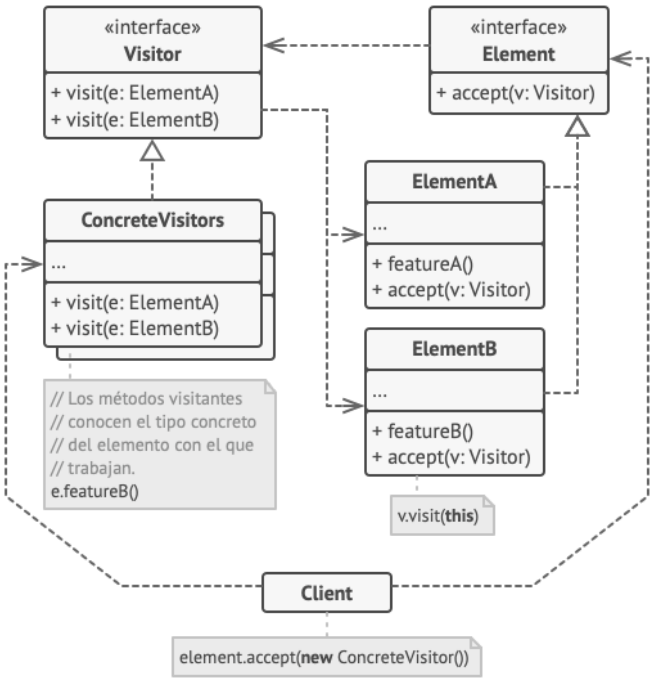
```
Abstraction: Base operation with:
ConcreteImplementationA: Here's the result on the platform A.

ExtendedAbstraction: Extended operation with:
ConcreteImplementationB: Here's the result on the platform B.
```

Patrón de comportamiento: Visitor.

El propósito de un patrón Visitor es definir una nueva operación sin introducir las modificaciones a una estructura de objeto existente.

Estructura:



Ejemplo No. 1:

Este patrón puede ser usado para identificar personas que pueden acceder a un descuento por cierto tipo de afiliación en un supermercado y dependiendo de la fecha en la que vayan. Si van en los primeros 10 días del mes, obtendrán 10% de descuento, si van en los siguientes 5 días obtendrán el 5”, después de estas fechas no obtendrán ningún tipo de descuento en sus compras.

Ejemplo No. 2:

También puede ser usado para distinguir entre tipos de jugadores de un juego. Si un jugador es de tipo Guerrero tendrá un tipo de arma con cierta fuerza, pero si es Mago tendrá otra diferente con diferentes funciones.

Código fuente en C++:

```
/**
 * The Visitor Interface declares a set of visiting methods that
correspond to
 * component classes. The signature of a visiting method allows the
visitor to
 * identify the exact class of the component that it's dealing with.
 */
class ConcreteComponentA;
class ConcreteComponentB;

class Visitor {
```

```
 public:
  virtual void VisitConcreteComponentA(const ConcreteComponentA *element)
const = 0;
  virtual void VisitConcreteComponentB(const ConcreteComponentB *element)
const = 0;
};

/**
 * The Component interface declares an `accept` method that should take
the base
 * visitor interface as an argument.
 */

class Component {
 public:
  virtual ~Component() {}
  virtual void Accept(Visitor *visitor) const = 0;
};

/**
 * Each Concrete Component must implement the `Accept` method in such a
way that
 * it calls the visitor's method corresponding to the component's class.
 */
class ConcreteComponentA : public Component {
  /**
   * Note that we're calling `visitConcreteComponentA`, which matches the
   * current class name. This way we let the visitor know the class of
the
   * component it works with.
   */
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentA(this);
  }
  /**
   * Concrete Components may have special methods that don't exist in
their base
   * class or interface. The Visitor is still able to use these methods
since
   * it's aware of the component's concrete class.
   */
  std::string ExclusiveMethodOfConcreteComponentA() const {
    return "A";
  }
};

class ConcreteComponentB : public Component {
  /**
```

```
   * Same here: visitConcreteComponentB => ConcreteComponentB
   */
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentB(this);
  }
  std::string SpecialMethodOfConcreteComponentB() const {
    return "B";
  }
};

/**
 * Concrete Visitors implement several versions of the same algorithm,
which can
 * work with all concrete component classes.
 *
 * You can experience the biggest benefit of the Visitor pattern when
using it
 * with a complex object structure, such as a Composite tree. In this
case, it
 * might be helpful to store some intermediate state of the algorithm
while
 * executing visitor's methods over various objects of the structure.
 */
class ConcreteVisitor1 : public Visitor {
 public:
  void VisitConcreteComponentA(const ConcreteComponentA *element) const
override {
    std::cout << element->ExclusiveMethodOfConcreteComponentA() << " +
ConcreteVisitor1\n";
  }

  void VisitConcreteComponentB(const ConcreteComponentB *element) const
override {
    std::cout << element->SpecialMethodOfConcreteComponentB() << " +
ConcreteVisitor1\n";
  }
};

class ConcreteVisitor2 : public Visitor {
 public:
  void VisitConcreteComponentA(const ConcreteComponentA *element) const
override {
    std::cout << element->ExclusiveMethodOfConcreteComponentA() << " +
ConcreteVisitor2\n";
  }
  void VisitConcreteComponentB(const ConcreteComponentB *element) const
override {
```

```
    std::cout << element->SpecialMethodOfConcreteComponentB() << " +
ConcreteVisitor2\n";
  }
};
/**
 * The client code can run visitor operations over any set of elements
without
 * figuring out their concrete classes. The accept operation directs a
call to
 * the appropriate operation in the visitor object.
 */
void ClientCode(std::array<const Component *, 2> components, Visitor
*visitor) {
  // ...
  for (const Component *comp : components) {
    comp->Accept(visitor);
  }
  // ...
}

int main() {
  std::array<const Component *, 2> components = {new ConcreteComponentA,
new ConcreteComponentB};
  std::cout << "The client code works with all visitors via the base
Visitor interface:\n";
  ConcreteVisitor1 *visitor1 = new ConcreteVisitor1;
  ClientCode(components, visitor1);
  std::cout << "\n";
  std::cout << "It allows the same client code to work with different
types of visitors:\n";
  ConcreteVisitor2 *visitor2 = new ConcreteVisitor2;
  ClientCode(components, visitor2);

  for (const Component *comp: components) {
    delete comp;
  }
  delete visitor1;
  delete visitor2;

  return 0;
}
```

Al ejecutar el código se obtiene el siguiente resultado:

```
The client code works with all visitors via the base Visitor interface:
A + ConcreteVisitor1
B + ConcreteVisitor1

It allows the same client code to work with different types of visitors:
```

```
A + ConcreteVisitor2
B + ConcreteVisitor2
```