Bridge

Linda Marshall

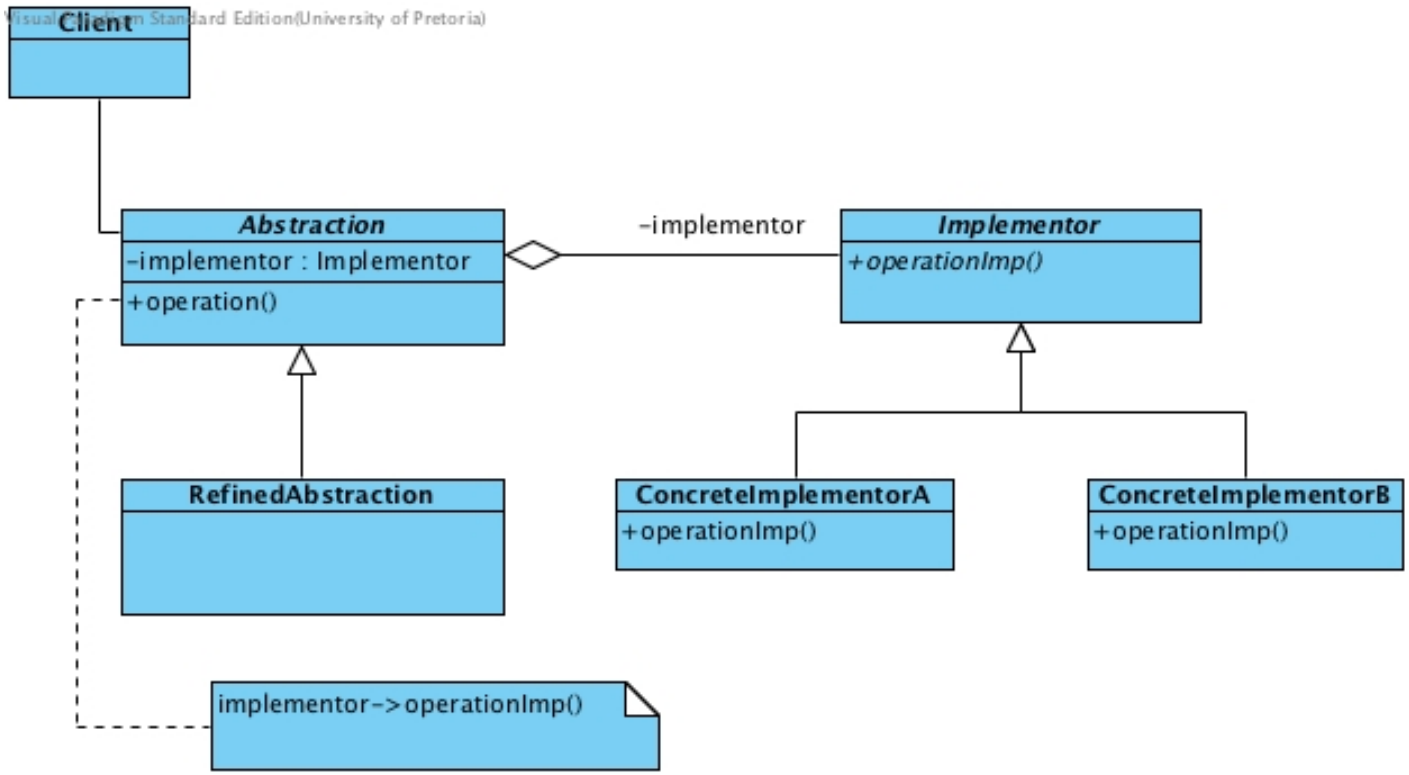
Department of Computer Science
University of Pretoria

28 October 2021

Name and Classification: Bridge, Object Structural

Intent: "Decouple an abstraction from its implementation so that the two can vary independently." (GoF:151)

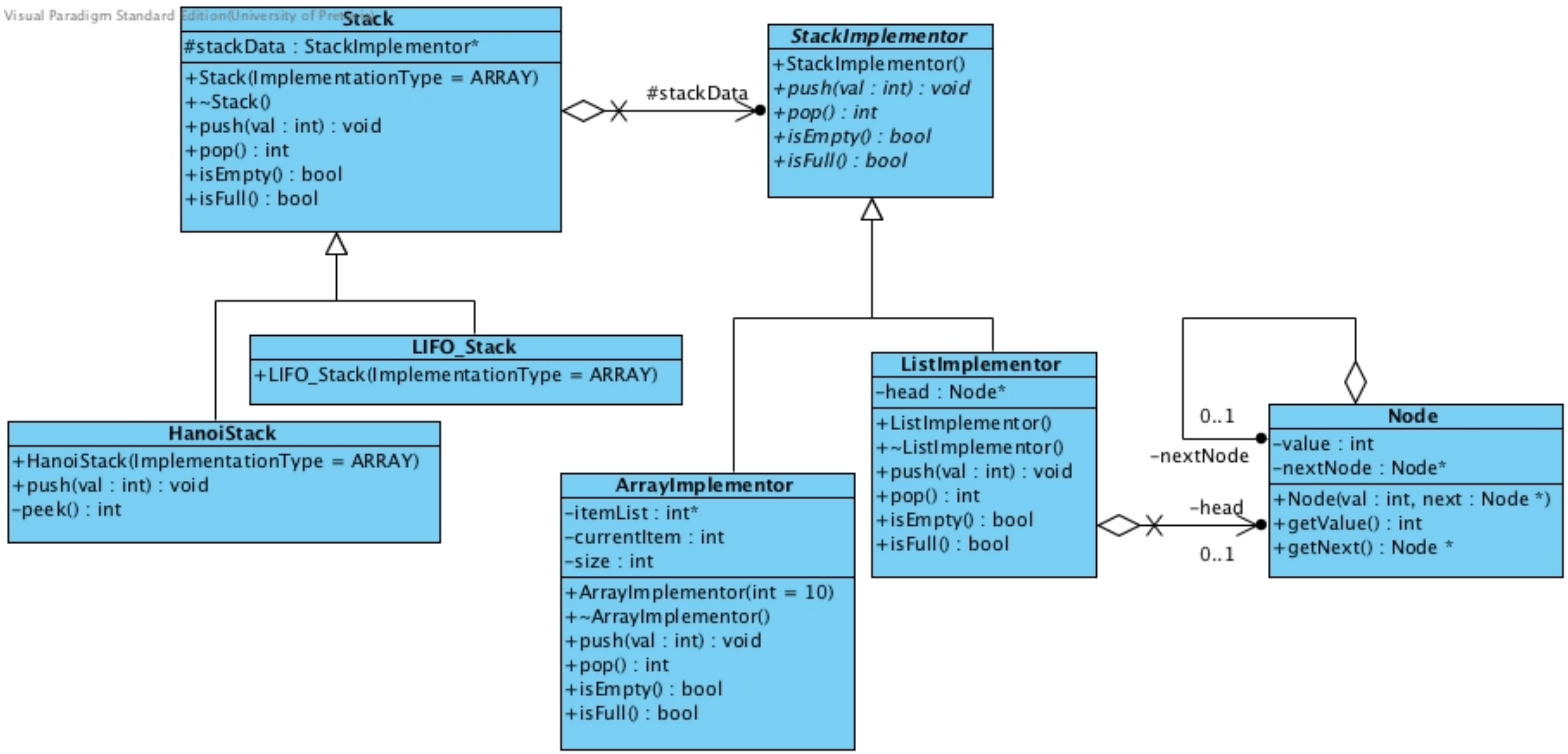
"Decouple an abstraction from its implementation so that the two can vary independently." (GoF:151)


Client
Abstraction
-implementor : Implementor
+operation()
-implementor
Implementor
+operationImp()
RefinedAbstraction
ConcreteImplementorA
+operationImp()
ConcreteImplementorB
+operationImp()
implementor->operationImp()

- **Abstraction**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **Refined Abstraction**
  - extends the interface defined by Abstraction.

- **Implementor**
  - defines an independent interface
  - This interface usually provides primitive operations while Abstraction provides higher-level operations based on these primitives.
- **Concrete Implementor**
  - implements the Implementor interface and defines its concrete implementation

- **Adapter** and Bridge implement co-operation between classes.
- **Strategy** and Bridge use an abstract interface to concrete implementations performing operations. With Strategy the operations are interchangeable, in Bridge the operations are common, acting on interchangeable 'systems' e.g. operating systems, data structures etc.


Visual Paradigm Standard
Stack
#stackData : StackImplementor*
+Stack(ImplementationType = ARRAY)
+~Stack()
+push(val : int) : void
+pop() : int
+isEmpty() : bool
+isFull() : bool
#stackData
StackImplementor
+StackImplementor()
+push(val : int) : void
+pop() : int
+isEmpty() : bool
+isFull() : bool
LIFO_Stack
+LIFO_Stack(ImplementationType = ARRAY)
HanoiStack
+HanoiStack(ImplementationType = ARRAY)
+push(val : int) : void
-peek() : int
ArrayImplementor
-itemList : int*
-currentItem : int
-size : int
+ArrayImplementor(int = 10)
+~ArrayImplementor()
+push(val : int) : void
+pop() : int
+isEmpty() : bool
+isFull() : bool
ListImplementor
-head : Node*
+ListImplementor()
+~ListImplementor()
+push(val : int) : void
+pop() : int
+isEmpty() : bool
+isFull() : bool
0..1
-nextNode
-head
0..1
Node
-value : int
-nextNode : Node*
+Node(val : int, next : Node *)
+getValue() : int
+getNext() : Node *

Participant	**Entity in application**
Abstraction	Stack
Refined Abstraction	LIFO_Stack, HanoiStack
Implementation	StackImplementor
Concrete Implementation	ArrayImplementor, ListImplementor
operation()	push(), pop(), isEmpty(), isFull()
implementation()	push(), pop(), isEmpty(), isFull()

```
class StackImplementor {
    public:
        StackImplementor() { };
        virtual void push( int val ) = 0;
        virtual int  pop() = 0;
        virtual bool  isEmpty() = 0;
        virtual bool  isFull() = 0;
        virtual ~StackImplementor() { };
};
```

```
class ListImplementor : StackImplementor {
  public:
     ListImplementor();
     ~ListImplementor();
     void push( int val );
     int  pop();
     bool  isEmpty();
     bool  isFull();
  private:
     Node* head;
};
```

```
ListImplementor :: ListImplementor() {
        head = 0;
}

ListImplementor :: ~ListImplementor()  {
        Node  *current, *previous;
        current = head;
        while (current)  {
                previous = current;
                current = current->getNext();
                delete previous;
        }
}
```

```
void ListImplementor :: push( int val ) {
    Node* temp = new Node( val, head );
    head = temp;
}

int ListImplementor :: pop() {
    if (isEmpty()) {
            return 0;
    } else {
            Node* temp = head;
            int val = head->getValue();
            head = head->getNext();
            delete temp;
            return val;
    }
}
```

```
bool ListImplementor :: isEmpty() {
        return (head == 0);
}

bool ListImplementor :: isFull() {
        return false;
}
```

```
class ArrayImplementor : StackImplementor {
  public:
    ArrayImplementor( int = 10 );
    ~ArrayImplementor();
    void push( int val );
    int  pop();
    bool  isEmpty();
    bool  isFull();
  private:
    int* itemList;
    int  currentItem;
    int  size;
};
```

```
enum ImplementationType {ARRAY, LIST};

class Stack {
  public:
        Stack( ImplementationType = ARRAY );
        virtual ~Stack();
        virtual void push( int val ) ;
        virtual int  pop() ;
        virtual bool isEmpty();
        virtual bool isFull();
  protected:
        StackImplementor* stackData;
};
```

```
Stack :: Stack(ImplementationType type) {
    switch (type) {
        case ARRAY:
            stackData=(StackImplementor*) new ArrayImplementor();
            break;
        case LIST:
            stackData=(StackImplementor*) new ListImplementor();
            break;
        default:
            stackData=(StackImplementor*) new ArrayImplementor();
    }
}

Stack :: ~Stack() {
        delete stackData;
}
```

```
void Stack :: push( int val ) {
        stackData->push( val );
}

int Stack :: pop() {
        return stackData->pop();
}

bool Stack :: isEmpty() {
        return stackData->isEmpty();
}

bool  Stack :: isFull() {
        return stackData->isFull();
}
```

```
class HanoiStack : public Stack {
public:
        HanoiStack( ImplementationType = ARRAY );
        void push( int val );
private:
        int peek();
};

class LIFO_Stack : public Stack {
public:
        LIFO_Stack( ImplementationType = ARRAY );
};
```

```
void HanoiStack :: push( int value ) {
        if ( stackData->isEmpty() || value < peek() ) {
                stackData->push( value );
        }
}

int HanoiStack :: peek() {
        int value = stackData->pop();
        stackData->push( value );
        return value;
}
```

```
int main ()  {
        Stack* stack1 = new HanoiStack( ARRAY );
        Stack* stack2 =new LIFO_Stack( LIST );
        for (int i=1; i < 16; i++) {
                int value;
                cout << i << ":";  cin >> value;
                stack1->push( value );
                stack2->push( value );
        }
        cout << "Array stack:\n";
        for (int i=1; i < 18 ; i++) {
                cout << stack1->pop() << "\t";
        }
        cout << endl;  cout << "List stack:\n";
        for (int i=1; i < 18 ; i++) {
           cout << stack2->pop() << "   ";
        }
        cout << endl;
        delete stack1; delete stack2;
        return 0;
}
```