/ Design Patterns / Behavioral patterns / Iterator

Iterator in C++

Back to **Iterator** description

Iterator design pattern

Take traversal-of-a-collection functionality out of the collection and promote it to "full object status". This simplifies the collection, allows many traversals to be active simultaneously, and decouples collection algorithms from collection data structures.

"Every 'container-like' class must have an iterator." It may seem like a violation of encapsulation for a Stack class to allow its users to access its contents directly, but John Lakos' argument is that the designer of a class inevitably leaves something out. Later, when users need additional functionality, if an iterator was originally provided, they can add functionality with "open for extension, closed for modification". Without an iterator, their only recourse is to invasively customize production code. Below, the original Stack class did not include an equality operator, but it did include an iterator. As a result, the equality operator could be readily retrofitted.

1. Design an "iterator" class for the "container" class
2. Add a `createIterator()` member to the container class
3. Clients ask the container object to create an iterator object
4. Clients use the `first()`, `isDone()`, `next()`, and `currentItem()` protocol

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

class Stack
{
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack()
    {
        sp =  - 1;
    }
    void push(int in)
    {
        items[++sp] = in;
    }
    int pop()
    {
        return items[sp--];
    }
    bool isEmpty()
    {
        return (sp ==  - 1);
    }
    StackIter *createIterator()const; // 2. Add a createIterator() member
};

class StackIter
{
    // 1. Design an "iterator" class
    const Stack *stk;
    int index;
  public:
    StackIter(const Stack *s)
    {
        stk = s;
    }
    void first()
    {
        index = 0;
    }
    void next()
    {
        index++;
    }
    bool isDone()
    {
        return index == stk->sp + 1;
    }
    int currentItem()
    {
        return stk->items[index];
    }
};

StackIter *Stack::createIterator()const
{
  return new StackIter(this);
}

bool operator == (const Stack &l, const Stack &r)
{
  // 3. Clients ask the container object to create an iterator object
  StackIter *itl = l.createIterator();
  StackIter *itr = r.createIterator();
  // 4. Clients use the first(), isDone(), next(), and currentItem() protocol
  for (itl->first(), itr->first(); !itl->isDone(); itl->next(), itr->next())
    if (itl->currentItem() != itr->currentItem())
      break;
  bool ans = itl->isDone() && itr->isDone();
  delete itl;
  delete itr;
  return ans;
}

int main()
{
  Stack s1;
  for (int i = 1; i < 5; i++)
    s1.push(i);
  Stack s2(s1), s3(s1), s4(s1), s5(s1);
  s3.pop();
  s5.pop();
  s4.push(2);
  s5.push(9);
  cout << "1 == 2 is " << (s1 == s2) << endl;
  cout << "1 == 3 is " << (s1 == s3) << endl;
  cout << "1 == 4 is " << (s1 == s4) << endl;
  cout << "1 == 5 is " << (s1 == s5) << endl;
}
```

Output

```
1 == 2 is 1
1 == 3 is 0
1 == 4 is 0
1 == 5 is 0
```

Read next

Code examples

Java	Iterator in Java: Before and after	Iterator in Java
C++	Iterator in C++	Iterator in C++: Using operators instead of methods
PHP	Iterator in PHP	
Delphi	Iterator in Delphi	

UML

About us

Terms / Privacy policy