Chain of Responsibility

Linda Marshall

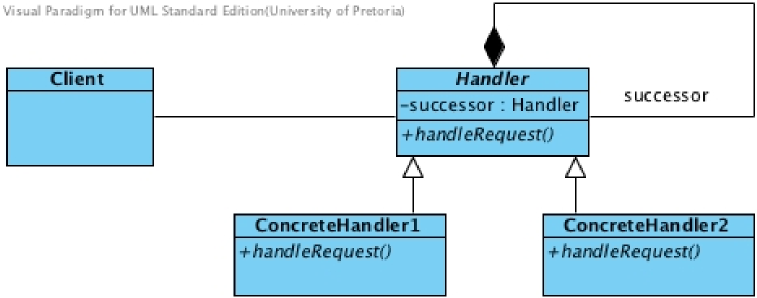
Department of Computer Science
University of Pretoria

22 October 2021

Name and Classification: Chain of Responsibility

Intent: “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.” (GoF:223)

“Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.” (GoF:223)


Visual Paradigm for UML Standard Edition(University of Pretoria)
Client
Handler
-successor : Handler
+handleRequest()
successor
ConcreteHandler1
+handleRequest()
ConcreteHandler2
+handleRequest()

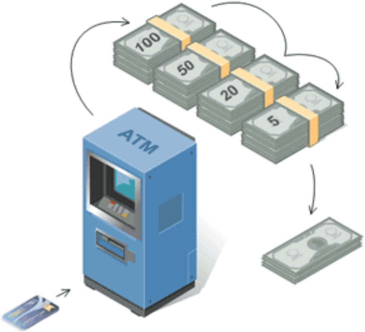
- The client does not need to know which other object is going to handle the request.
- Handling responsibilities is flexible, objects can be added to the chain.

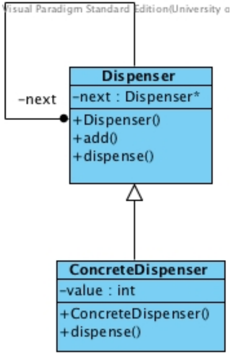
- **Handler**: Defines the interface for handling requests and implements the successor links.
- **ConcreteHandler**: Handles requests it is responsible for and may handle the successor link.
- **Client**: Initiates the request to a ConcreteHandler object in the chain.

- **Composite** - A component's parent can act as a successor. Has recursive composition.
- **Decorator** - Has recursive composition.
- **Command, Mediator and Observer** - Also decouple senders from receivers.

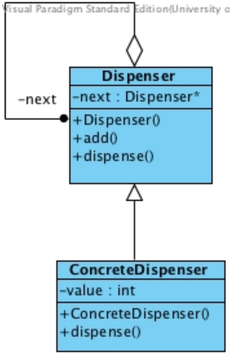
All a baby needs is to be fed, loved and changed. Granny's naturally love the baby. Dad's feed the baby and Mom's are left to change the baby. Model the needs of a baby using the Chain of Responsibility design pattern.

All a baby needs is to be fed, loved and changed. Granny's naturally love the baby. Dad's feed the baby and Mom's are left to change the baby. Model the needs of a baby using the Chain of Responsibility design pattern.


ATM
100
50
20
5



C++ Reverse

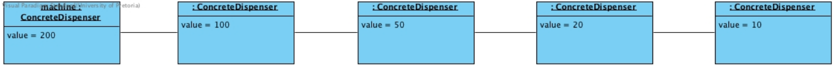


Updated

```
int main()
{
  //Assemble the chain:
  Dispenser* machine = new ConcreteDispenser(200);
  machine->add(new ConcreteDispenser(100));
  machine->add(new ConcreteDispenser(50));
  machine->add(new ConcreteDispenser(20));
  machine->add(new ConcreteDispenser(10));

  int n;
  cout << "Amount to be dispensed: R";
  cin >> n;
  machine->dispense(n);
  cout << endl;

  return 0;
}
```


machine : ConcreteDispenser
value = 200
: ConcreteDispenser
value = 100
: ConcreteDispenser
value = 50
: ConcreteDispenser
value = 20
: ConcreteDispenser
value = 10

```
Amount to be dispensed: R285
R200 dispenser dispenses R200
R85 to small for R200 dispenser — pass  on
R85 to small for R100 dispenser — pass  on
R50 dispenser dispenses R50
R35 to small for R50 dispenser — pass  on
R20 dispenser dispenses R20
R15 to small for R20 dispenser — pass  on
R10 dispenser dispenses R10
R5 to small for R10 dispenser — pass  on
R5 can not be dispensed
```

```
class Dispenser{
  public:
    Dispenser(): next(0){ };
    void add(Dispenser *n) {
      if (next)
        next->add(n);
      else
        next = n;
    };
    virtual void dispense(int i) {
      if(i > 0) {
        if(next)
          next->dispense(i);
        else
          cout << "R" << i << " can not be dispensed" << endl;
      } else
        cout << "Required amount was dispensed" << endl;
    };
  private:
    Dispenser* next;
};
```

```
class ConcreteDispenser: public Dispenser {
  public:
    ConcreteDispenser(int v): Dispenser(), value(v){};
    void dispense(int i) {
      while(i >= value) {
        cout << "R" << value << " dispenser dispenses R"
            << value << endl;
        i -= value;
      }
      cout << "R" << i << " to small for R"  << value
          << " dispenser - pass on" << endl;
      Dispenser::dispense(i);
    }
  private:
    int value;
};
```