Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 20: Chain of Responsibility Design Pattern

Contents

20.1 Introduction

The design and implementation of the chain of responsibility design pattern is fairly straight forward. An example that is often used to illustrate the intent of this pattern is the implementation of a cash dispenser as is commonly found in automatic teller machines (ATM) and coin operated machines capable of returning change. In this lecture we discuss this pattern and explain it at the hand of a simulation of the cash dispensing mechanism in an ATM.

20.2 Chain of Responsibility Design Pattern

20.2.1 Identification

Name	**Classification**	**Strategy**
Chain of Responsibility	Behavioural	Delegation
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. ([1]:223)		

20.2.2 Problem

There is a potentially variable number of handler objects, and a stream of requests that must be handled. We need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings [2].

20.2.3 Structure

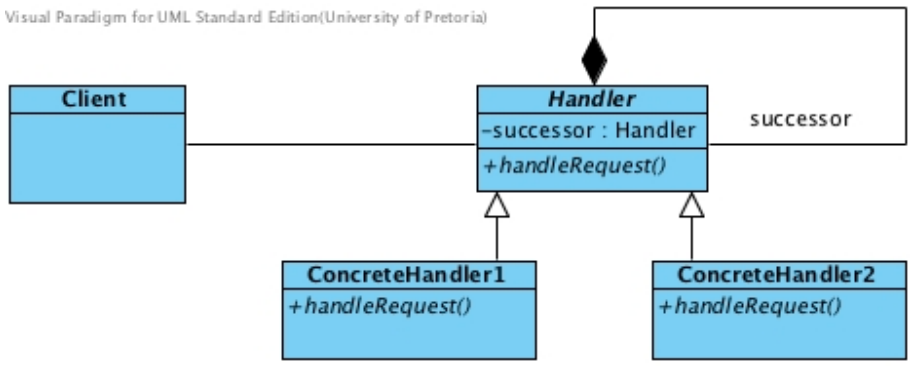


Figure 1: The structure of the Chain of Responsibility Design Pattern

20.2.4 Participants

Handler

- Defines an interface for handling requests.
- Implements the successor link.

Concrete Handler

- Handles requests it is responsible for.
- Can Access its successor.
- If the Concrete Handler can handle the request, it does so; otherwise it delegates the request to its successor via Handler.

Client

- Initiates the request to a ConcreteHandler object in the chain.

20.3 Chain of Responsibility Pattern Explained

20.3.1 Design

A simple way to implement the pattern is to declare the `handleRequest()` in the interface as a virtual method, but not to make it pure virtual. Also provide a default implementation that delegates the request to its successor if it exists. This way if the concrete handler does not implement `handleRequest()`, the request will automatically be delegated to its successor who might be able to handle it. One may also provide a default action that can be taken if there is no successor to prevent the situation that an un-handled request go undetected. The following is the code and related figure (figure 2) for a generic handler interface that applies the chain of responsibility design pattern:

```
class Handler
{
    public:
        Handler(Handler* s) : successor(s) { }
        virtual void handleRequest();
    private:
        Handler* successor;
};

void Handler::handleRequest()
{
    if (successor)
        successor->handleRequest();
    else
        //define action in case of no successor here
}
```

Note the difference in representation between the `Handler` class given in figure 1 with that given in figure 2. The association is implementation dependent. In figure 'refChainStructure the successor is placed in the stack, while in the next implementation is it placed on the heap. When the successor is on the stack, it is tightly coupled to the Handler class

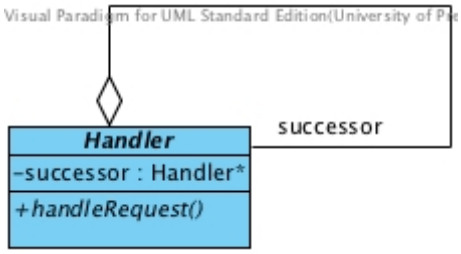


Figure 2: Handler class of the Chain of Responsibility Design Pattern

and will be destroyed when the Handler class goes out of scope. On the heap, it is the responsibility of the Handler class when it destructs.

The implementation of the concrete handlers contains the "intelligence". It can contain code with heuristics to decide to handle the request, ignore the request or pass the request to its successor. The following is the code for a generic concrete handler that applies the chain of responsibility design pattern. If A is true this concrete handler will handle the request. If B is true it will delegate to its successor by calling the implementation of `handleRequest()` in its parent class. If both conditions A and B are false it will ignore the request.

```
class ConcreteHandler : public Handler
{
    public:
        void handleRequest();
}

void ConcreteHander::handleRequest()
{
    if (A)
        //define action to handle the request here
    else if (B)
       Handler :: handleRequest();
}
```

It is important to apply this pattern only if the solution requires multiple handlers for a request that may differ dynamically. Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows at compile time which service object should handle the request [3].

20.3.2 Improvements achieved

- **Reduced coupling**
  The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled. Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure. As a result, Chain of Responsibility can simplify object inter-connections. Instead of objects maintaining references to all candidate receivers, each object keeps a single reference to its successor.

- **Added flexibility in assigning responsibilities to objects**
  Chain of Responsibility gives you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialise handlers statically.

20.3.3 Disadvantage

- **Receipt isn't guaranteed.**
  Since a request has no explicit receiver, there's no guarantee that it will be handled – the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

20.3.4 Real world example

source: `http://sourcemaking.com/design_patterns/chain_of_responsibility`

In an ATM there is physical handler for each kind of money note inside the machine. One for R200 notes, one for R100 notes, down to one for R10 notes. While the amount that still needs to be dispensed can be handled by a specific handler, it will dispense a note and reduce the amount. If the amount can not be handled by a specific handler it will simply delegate to the next handler.

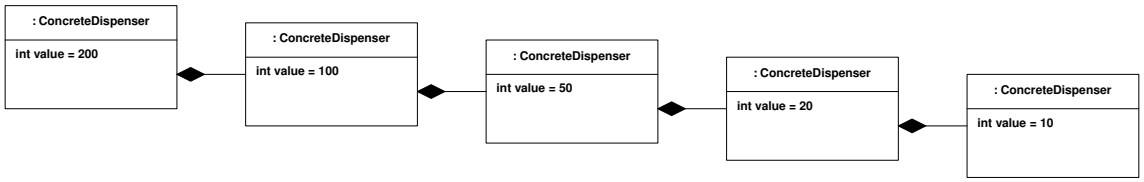


Figure 3: An object diagram showing a chain of ConcreteDispenser objects

An example implementation simulation an ATM can be found in `L25_atm.tar.gz`. The program instantiates five ConcreteDispenser objects and arrange them in a chain as shown

in Figure 3. The following are two sample test runs of this simulation that illustrates the the chain of events for dispensing cash in this simulation:

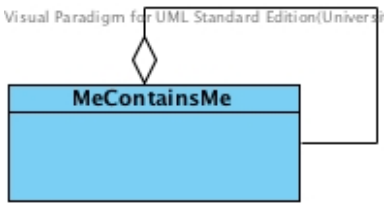
```
Amount to be dispensed: R80
R80 to small for R200 dispenser - pass  on
R80 to small for R100 dispenser - pass  on
R50 dispenser dispenses R50
R30 to small for R50 dispenser - pass  on
R20 dispenser dispenses R20
R10 to small for R20 dispenser - pass  on
R10 dispenser dispenses R10
R0 to small for R10 dispenser - pass  on
Required amount was dispensed

Amount to be dispensed: R605
R200 dispenser dispenses R200
R200 dispenser dispenses R200
R200 dispenser dispenses R200
R5 to small for R200 dispenser - pass  on
R5 to small for R100 dispenser - pass  on
R5 to small for R50 dispenser - pass  on
R5 to small for R20 dispenser - pass  on
R5 to small for R10 dispenser - pass  on
R5 can not be dispensed
```

20.3.5 Related Patterns

Composite

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

Composite and Decorator



The Chain of Responsibility, Composite and Decorator patterns all have recursive composition, i.e. they have a pointer to an object of its own kind as an instance variable.

Command, Mediator, and Observer

Chain of Responsibility, Command, Mediator, and Observer, address how you can **decouple senders and receivers**, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

20.4 Implementation Issues

20.4.1 Implementing the successor chain

When implementing the successor chain the link to a concrete handler's successor is defined in the handler interface. This link can be defined private to obligate the concrete handlers to delegate responsibility to their successors through the handler interface.

It is also possible to implement a successor chain through re-use of existing links. When the concrete handlers are already in a structure, for example being elements of an application of the composite pattern or the decorator pattern, the existing links may be re-used to form a successor chain. Using existing links works well when the links support the chain you need. However, if such existing structure doesn't reflect the chain of responsibility your application requires this is not an option.

20.5 Example

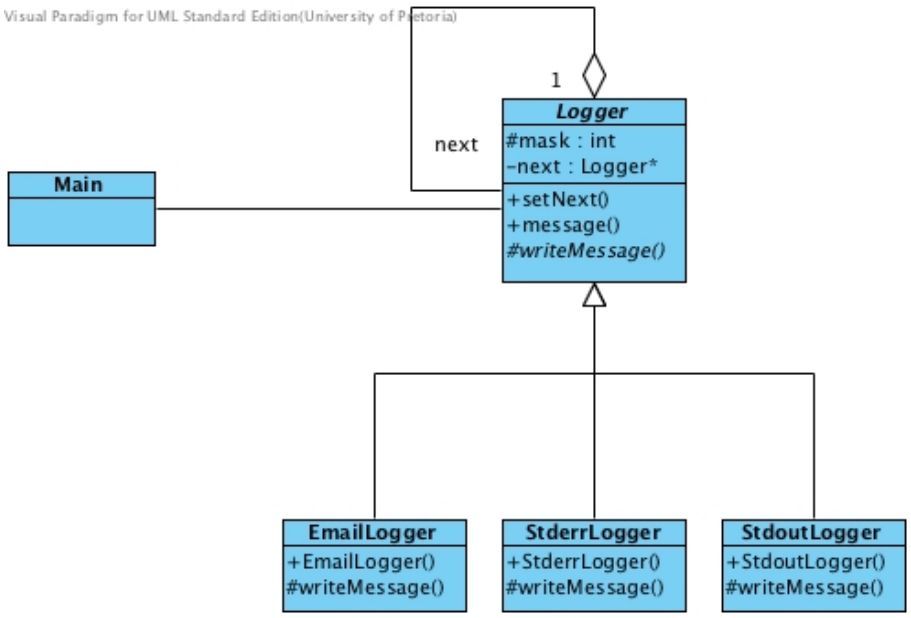


Figure 4: Class Diagram of an Implementation of a Logger

Figure 4 is a class diagram of a system illustrating the implementation of the Chain of Responsibility design pattern. It is a prototype of a system that can be used to log events in a log file. The following table summarises how the implementation relates to the participants of this pattern:

Participant	**Entity in application**
Handler	Logger
Concrete Handlers	StdoutLogger, EmailLogger, StderrLogger
handleRequest()	writeMessage(: string)

Handler

- `Logger` acts as the `Handler` interface.
- It defines an interface for handling and delegating requests.
- It has a private instance variable `next` and a public setter for this variable. They are needed to be able to link Loggers in a chain of responsibility.
- `message()` acts as a template method. It implements the intelligence to execute the request and/or to delegate to the next Logger in the chain.
- It has a protected instance variable `mask` this is set when a concrete Logger is created. It is used to indicate the level of detail that needs to be logged by the specific concrete logger.

Concrete Handler

- The concrete handlers that are implemented are `StdoutLogger, EmailLogger` and `StderrLogger`. Their priorities are set on creation in their respective constructors.
- The implementation of the `writeMessage(:string)` of each of these loggers simply writes a fixed string to stdout. In a real application more detail can be gathered and be written to a log file.

Client

- In this application the client is implemented in the main routine.
- This main routine is a simple test harness to illustrate how the chain of responsibility acts in different situations.
- It sets up a chain with four loggers and then triggers the chain in three different conditions.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[2] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].

[3] Alexander Shvets. Design patterns simply. `http://sourcemaking.com/design_patterns/`, n.d. [Online; Accessed 29-June-2011].