CMPE 202

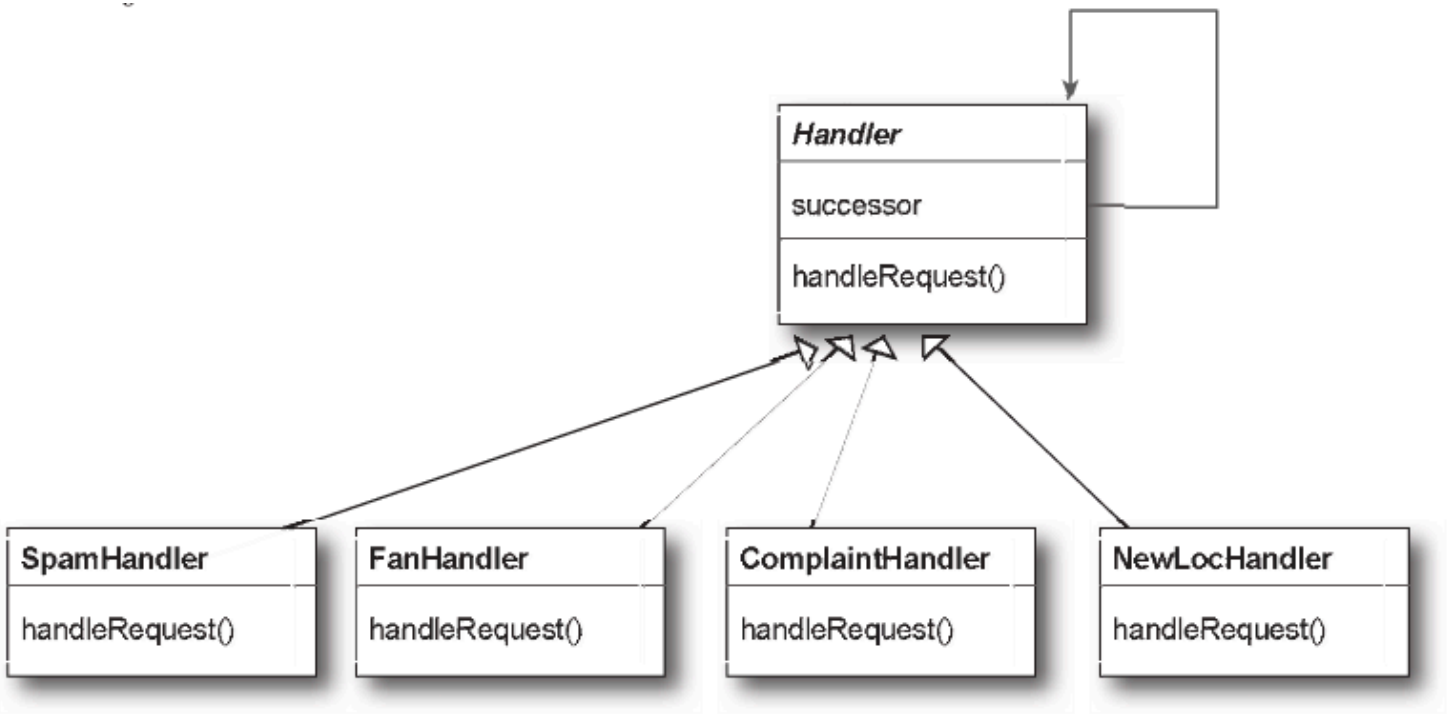
Gang of Four Design Patterns

Chain of Responsibility

Motivation

- Sender of a request does not know which object is the right one responsible for handling the request
- Want to decouple the senders and the receivers of a message (i.e. request)
- Would like to give multiple objects a chance to handle the request

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.

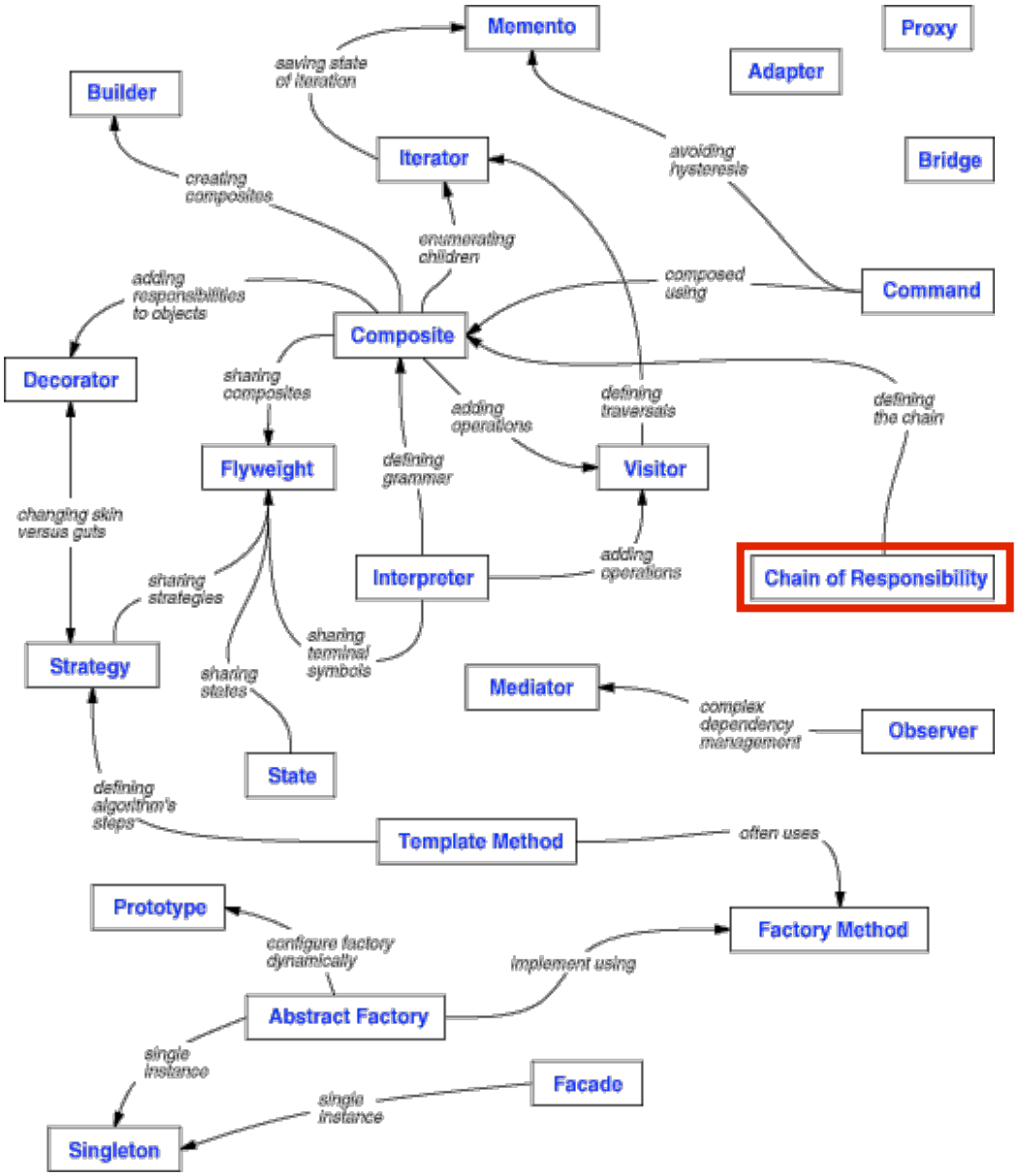


As email is received, it is passed to the first handler: the SpamHandler. If the SpamHandler can t handle the request, it is passed on to the FanHandler. And so on...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain – although, you can always implement a catch-all handler.

Chain of Responsibility



		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Design Pattern Catalog

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

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.

Applicability

Use Chain of Responsibility when

- more than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Participants

- **Handler** (Interface)
  - defines an interface for handling requests.
  - (optional) implements the successor link.

- **ConcreteHandler**
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

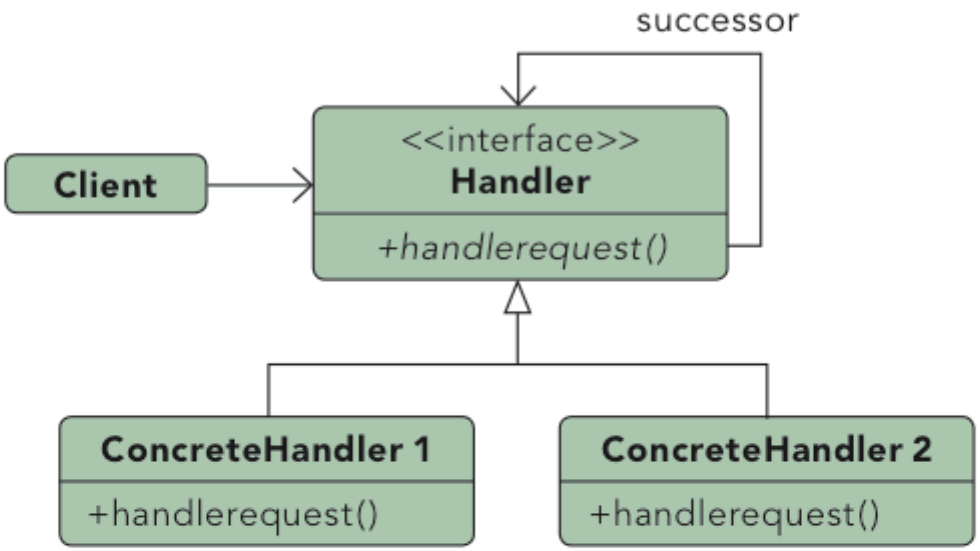
- **Client**
  - initiates the request to a ConcreteHandler object on the chain.

Collaborations

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

CHAIN OF RESPONSIBILITY

Object Behavioral

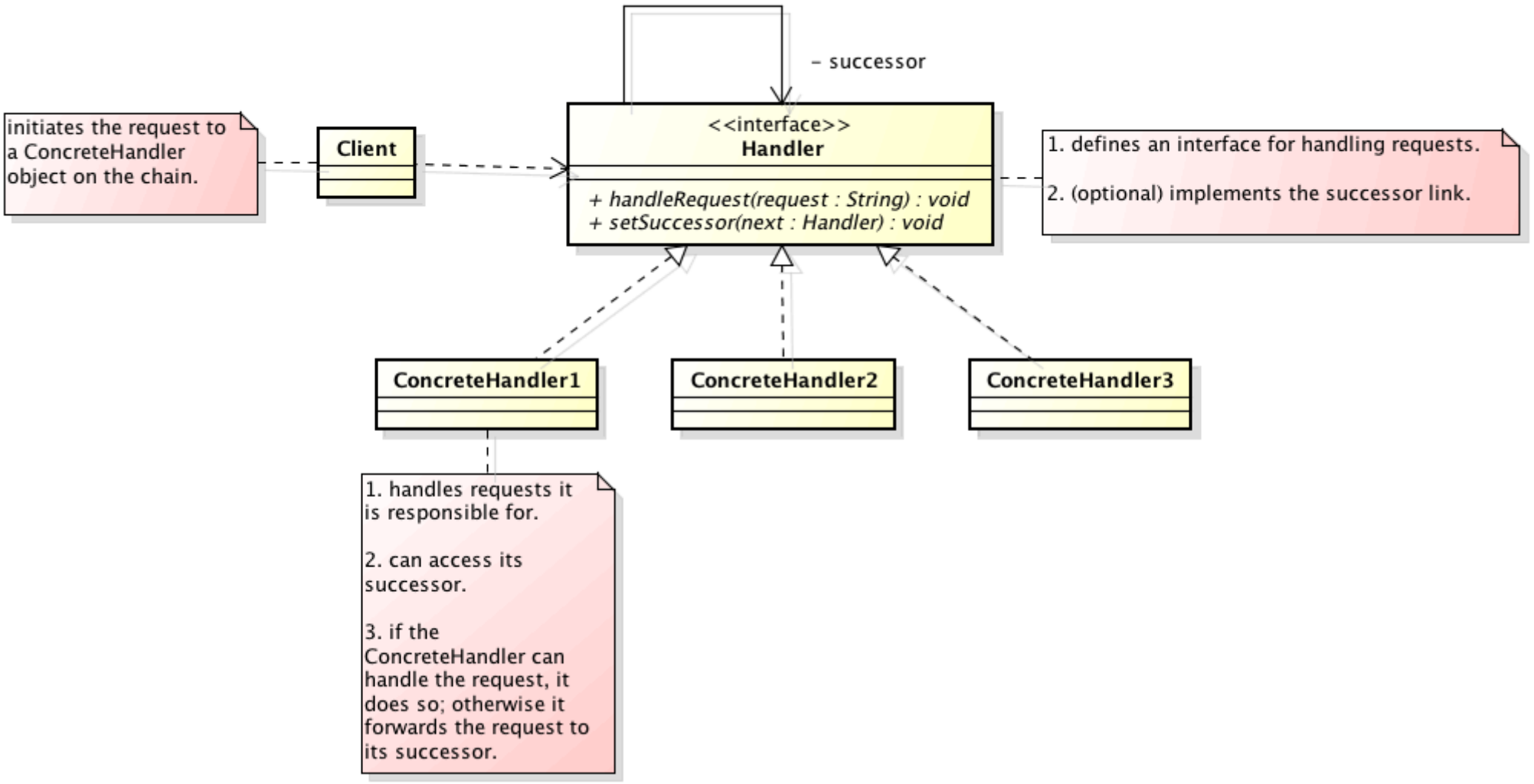


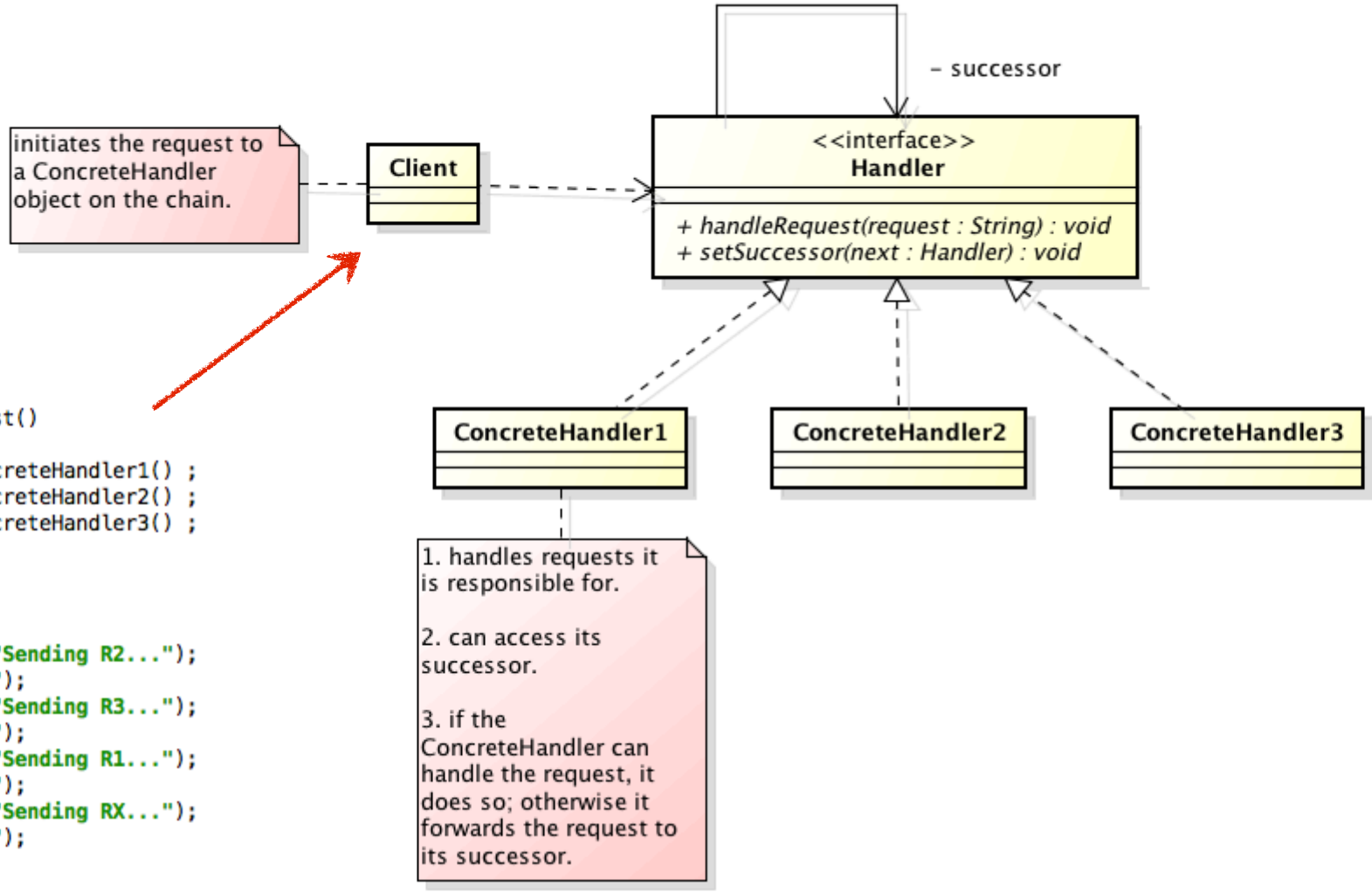
Purpose

Gives more than one object an opportunity to handle a request by linking receiving objects together.

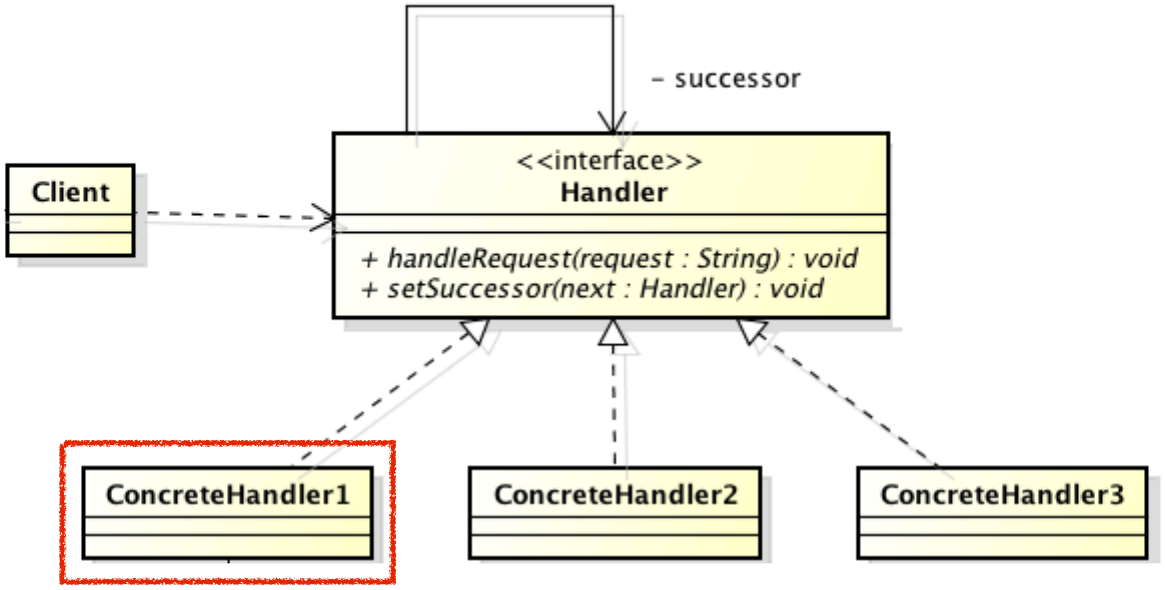
Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

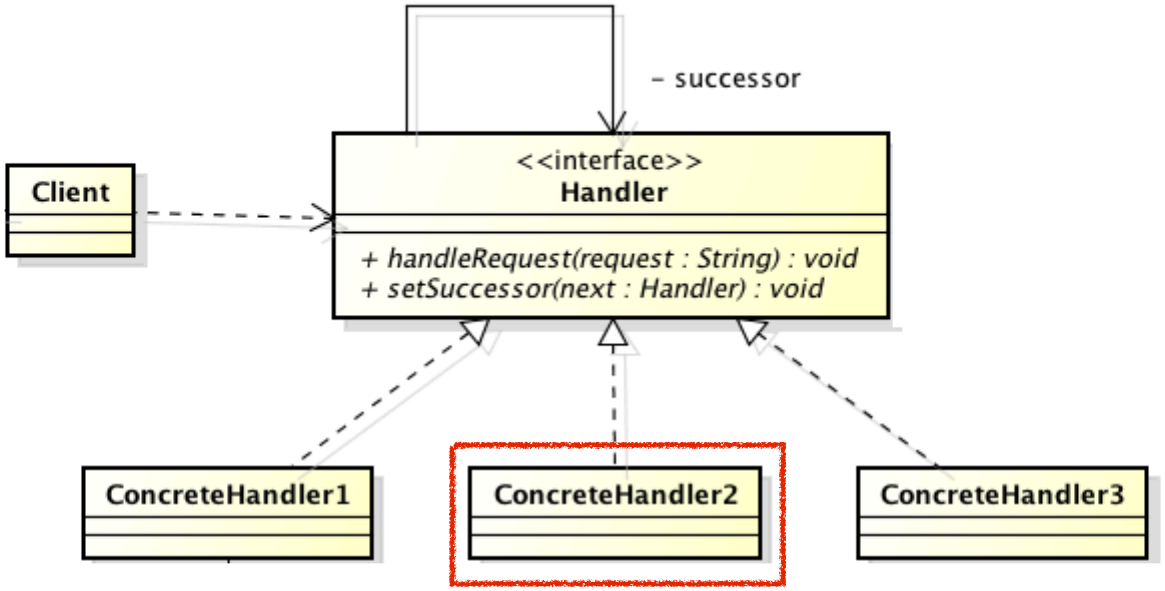

- successor
initiates the request to a ConcreteHandler object on the chain.
Client
<<interface>>
Handler
+ handleRequest(request : String) : void
+ setSuccessor(next : Handler) : void
1. defines an interface for handling requests.
2. (optional) implements the successor link.
ConcreteHandler1
ConcreteHandler2
ConcreteHandler3
1. handles requests it is responsible for.
2. can access its successor.
3. if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.



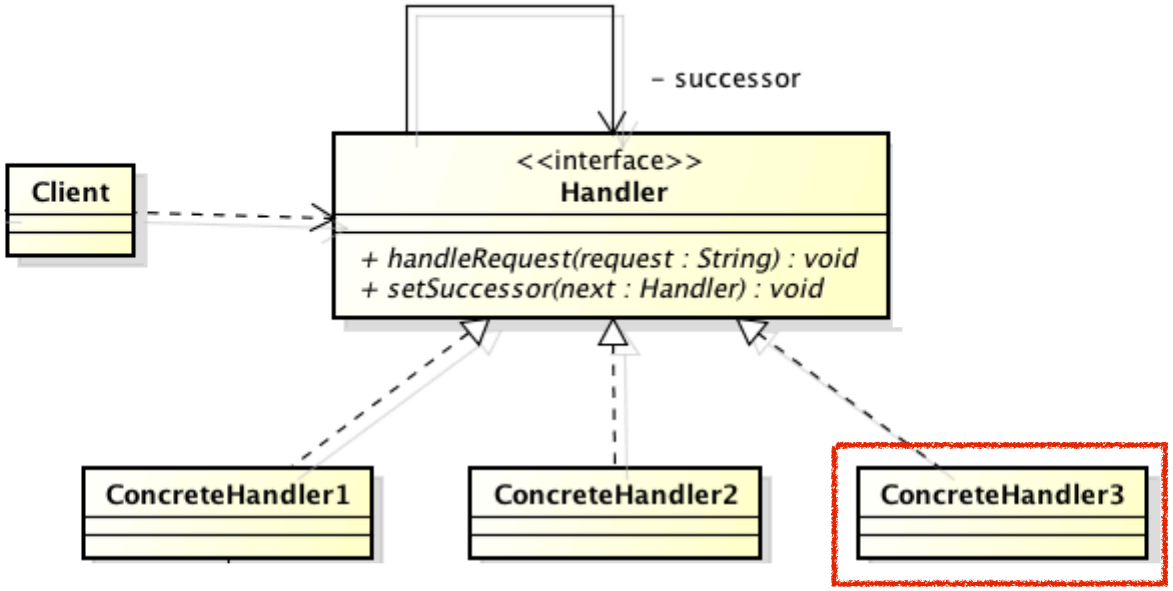

```java
public class Client {

    public static void runTest()
    {
        Handler h1 = new ConcreteHandler1() ;
        Handler h2 = new ConcreteHandler2() ;
        Handler h3 = new ConcreteHandler3() ;

        h1.setSuccessor(h2);
        h2.setSuccessor(h3);

        System.out.println( "Sending R2...");
        h1.handleRequest("R2");
        System.out.println( "Sending R3...");
        h1.handleRequest("R3");
        System.out.println( "Sending R1...");
        h1.handleRequest("R1");
        System.out.println( "Sending RX...");
        h1.handleRequest("RX");

    }

}
```




```java
public class ConcreteHandler1 implements Handler {

    private Handler successor = null;

    public void handleRequest( String request ) {
        System.out.println( "R1 got the request...");
        if ( request.equalsIgnoreCase("R1") )
        {
            System.out.println( this.getClass().getName() + " => This one is mine!");
        }
        else
        {
            if ( successor != null )
                successor.handleRequest(request);
        }

    }

    public void setSuccessor(Handler next) {
        this.successor = next ;
    }

}
```




```java
public class ConcreteHandler2 implements Handler {

    private Handler successor = null;

    public void handleRequest( String request ) {
        System.out.println( "R2 got the request...");
        if ( request.equalsIgnoreCase("R2") )
        {
            System.out.println( this.getClass().getName() + " => This one is mine!");
        }
        else
        {
            if ( successor != null )
                successor.handleRequest(request);
        }

    }

    public void setSuccessor(Handler next) {
        this.successor = next ;
    }


}
```




```java
public class ConcreteHandler3 implements Handler {


    private Handler successor = null;

    public void handleRequest( String request ) {
        System.out.println( "R3 got the request...");
        if ( request.equalsIgnoreCase("R3") )
        {
            System.out.println( this.getClass().getName() + " => This one is mine!");
        }
        else
        {
            if ( successor != null )
                successor.handleRequest(request);
        }

    }

    public void setSuccessor(Handler next) {
        this.successor = next ;
    }


}
```

Resources for this Tutorial