Lesson 6 Chain of Responsibility and Command

Chain of Responsibility

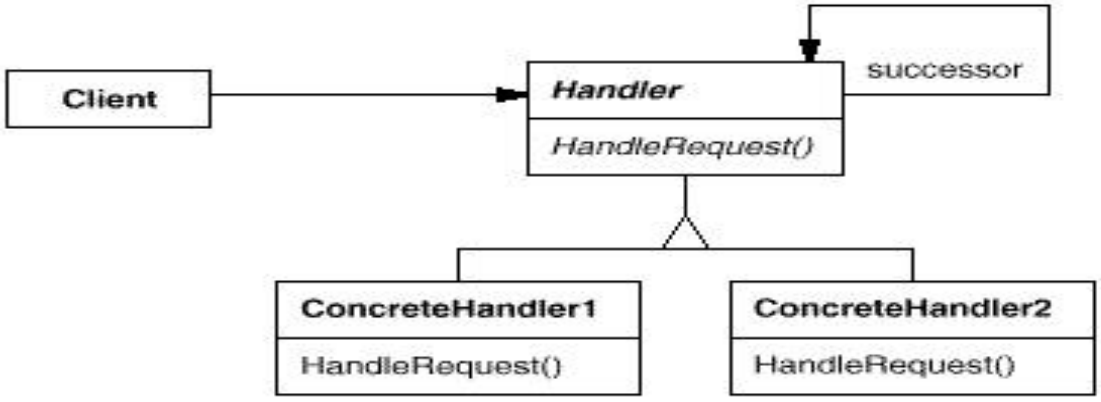
1. Intent

   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.

2. Motivation

   a. From the client's perspective, a request needs to be handled.
   b. From the handlers' perspective, there are multiple resources with different skills, different roles, or different responsibilities organized in a chain structure, from the most 'specific' to the most 'general'.
   c. These resources only know their successors (whom to pass the request to if they are not able to handle it). And each resource has only one successor or none if it is the last in the chain.
   d. Request processing always starts from the first resource in the chain. If request gets processed, the work for processing the request terminates right away. If not, the request is passed to its successor.
   e. The client always expects its requests get handled, even though it is possible that a request could fall through the chain without being handled.

3. Structure



4. Participants

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

b. ConcreteHandler
- Handles requests it is responsible for.
- Holds a reference to its successor.
- If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

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

5. Applicability

Use Chain of Responsibility:

a. When more than one object may handle a request and the actual handler is not known in advance.
b. You want to issue a request to one of several objects without specifying the receiver explicitly.
c. The set of objects that can handle a request should be specified dynamically.
d. When you process requests with a “handle or forward” model.

6. Consequences

a. Reduced coupling between the sender of a request and the receiver - the sender and receiver have no explicit knowledge of each other (except that the sender knows about the first one in the chain).
b. 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.

c. The chain does not consider how receipt is guaranteed - a request could fall off the end of the chain without being handled.

7. How to implement the CoR pattern?

a. Handler abstract class (AbstractAgent)

```
public abstract class AbstractAgent {
	protected AbstractAgent nextAgent;

	abstract public void handleRequest(Request req);
	boolean canHandleRequest(Request req){
		…
	}
}
```

b. Concrete Handler classes

```
public class JuniorAgent extends AbstractAgent{

	@Override
	public void handleRequest(Request req) {
		if ( this.canHandleRequest(req) ){
			//handle this request
			req.setAnswered(true);
			return;
		}else{
			if ( this.nextAgent != null )
				this.nextAgent.handleRequest(req);
		}
	}

}

public class SeniorAgent extends AbstractAgent{

	@Override
	public void handleRequest(Request req) {
		if ( this.canHandleRequest(req) ){
			//handle this request
			req.setAnswered(true);
			return;
		}else{
			if ( this.nextAgent != null )
				this.nextAgent.handleRequest(req);
		}
	}
}
```

c. Set up the chain and handle request

```
public class Client {
	private ChainBuilder chain; //can be injected or received
from a constructor

	public void sendRequest(Request request) {
		chain.getHandler().handleRequest(request);
	}
}


public class ChainBuilder {
	private AbstractAgent handler;

	//other necessary code here

	private void buildChain(){
		AbstractAgent junior = new JuniorAgent();
		AbstractAgent senior = new SeniorAgent();
		AbstractAgent supervisor = new AgentSupervisor();
		junior.nextAgent = senior;
		senior.nextAgent = supervisor;
		handler = junior;
	}

}
```

Lab 6-1

After a day's work by the agents in the above example, a large number of processed requests get stored in the database. Each night, 3 backend processes will start working on the data sequentially. The first one (Validator) will check if a request has the valid contact information provided (email, phone and address) and then separates the valid from the invalid ones. The invalid requests will be saved in a 'discarded-requests' file. The valid ones will be passed to the second process (Data Washer). The Data Washer will do some analysis on each request and generate sales leads before passing them to the third/last process, called Reporter. What the Reporter does is simply generate a report (sales leads) by State where the customer lives.

Design the application with the CoR pattern. You also create a 'client' program that sends call records to the Validator one by one, which checks if it is a valid one (to make it simple, we take all records that have non-empty address, phone and email as valid ones). Then the Validator sets the record's 'isValid' field to true or leaves it as false accordingly before sending it to the Data Washer. The Data Washer sets the record's 'isASalesLead' field to true before passing it to the Reporter. The Reporter simply generates a report for all the sales leads. You can use the following skeleton code in your implementation.

```
public class CallRecord {
	private Customer customer;
	private Agent agent;
	private String requestInformation;
	private boolean isValid;
	private boolean isASalesLead;

}

public class Customer{
	private String firstName;
	private String lastName;
	private Address address;
	private String phone;
	private String email;
}

public class Address{
	private String streetAddress;
	private String city;
	private String state;
	private String zipcode;
}

public class Agent{
	private String agentId;
	private Address workLocation;
}
```

Command

1. Intent

   Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

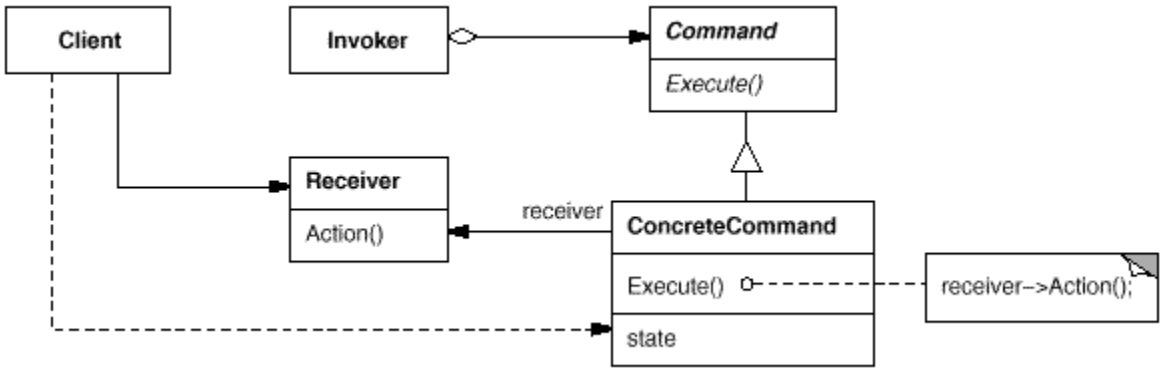
2. Motivation

   a. Sometimes it's necessary to issue requests without knowing anything about the operation being requested or the receiver of the request.
   b. For some applications, you have to store and pass requests around like other objects.
   c. For unsuccessfully executed requests, roll-back is required.

3. Applicability

   Use the Command pattern when you want to:
   a. Pass an object as an action to perform. The object has a callback method to be called by the invoker. Commands are an object-oriented replacement for procedural language callback functions.
   b. Specify, queue, and execute requests at different times.
   c. Support transactions.
   d. Support undo.
   e. Support logging and system recovery.

4. Structure



5. Participants

a. Command
   - declares an interface for executing an operation.

b. ConcreteCommand
   - defines a binding between a Receiver object and an action.
   - implements Execute by invoking the corresponding operation(s) on Receiver.

c. Invoker
   - asks the command to carry out the request.

d. Receiver
   - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

e. Client
   - creates a ConcreteCommand object and sets its receiver.

6. Consequences

The Command pattern has the following consequences:

a. Command decouples the object that invokes the operation from the one that knows how to perform it.
b. Commands are regular objects. They can be manipulated and extended like any other object.
c. It's easy to add new Commands, because you don't have to change existing classes.

7. There is a bad smell in the following code. How to fix it?

```
public void actionPerformed(ActionEvent e)
{
  Object o = e.getSource();
  if (o instanceof fileNewMenuItem)
    doFileNewAction();
  else if (o instanceof fileOpenMenuItem)
    doFileOpenAction();
  else if (o instanceof fileOpenRecentMenuItem)
    doFileOpenRecentAction();
  else if (o instanceof fileSaveMenuItem)
    doFileSaveAction();
  // and more ...
}
```

Solution: use the Command pattern.

```
public interface Command{
  public void execute();
}

public class FileOpenMenuItem extends JMenuItem implements Command{
  public void execute() {
    // your business logic goes here
  }
}
```

8. How to implement the Command pattern?

a. Command interface.

```
public interface Command {
      public boolean execute();
      public boolean undo();
}
```

b. Receiver class that executes a command.

```
public class Database {

	public boolean save(Person person){
		//save person object into the database
		try{
			return savePerson(person);
		}catch (Exception e){
			e.printStackTrace();
			return false;
		}

	}

	public boolean remove(Person person){
		// remove person from the database
		try{
			return removePerson(person);
		}catch (Exception e){
			e.printStackTrace();
			return false;
		}
	}

	private boolean savePerson(Person p) throws Exception{
		//send an 'insert' SQL query to the database and return a
true or false value;
		return result;
	}

	private boolean removePerson(Person p) throws Exception{
		//send a delete SQL query to the database and return a true
or false value;
		return result;
	}

}
```

c. Concrete command that implements the Command interface.

```
public class SaveCommand implements Command {
	private Database db = new Database();
	private Person person;

	public SaveCommand(){
	}
	public SaveCommand(Person person){
		this.person = person;
	}

	@Override
	public boolean execute() {
```

```
            return db.savePerson(person);
      }

      @Override
      public boolean undo() {
            return db.removePerson(person);
      }
```

d. Invoker class that asks a command to carry out requests

```
public class PersonSubSystemManager {
      private Command currentCommand = null;
      private Stack<Command> commandsExecuted = new Stack<Command>();

      public void saveParentAndChildren(Person parent){

            currentCommand = new SaveCommand(parent);

            //parent object saved in database
            if ( currentCommand.execute() == true ){
                  commandsExecuted.push(currentCommand);
                  if ( !parent.getChildren().isEmpty()){
                        for ( Person p: parent.getChildren() ){
                              currentCommand = new SaveCommand(p);
                              if ( currentCommand.execute() == true ){

      commandsExecuted.push(currentCommand);
                              }else{
                                    break;
                              }
                        }
                        //undo all executed commands
                        while ( !commandsExecuted.empty() ){
                              commandsExecuted.pop().undo();
                        }

                        System.out.println("operation failed!");
                        return;
                  }//both parent and children objects saved in database
                  System.out.println("operation completed!");

            } else { //parent object not saved
                  System.out.println("operation failed!");
            }
      }

}
```

Lab 6-2

Design/implement a simple program with Command Pattern, which creates, scales, and moves a square by issuing commands. For example, ‘Create 5’ will create a 5x5 square; ‘Scale 2’ will make the sides of a square twice as long; ‘Move right 3’ will move the position of a square to the right by 3, etc. Your program should also support ‘undo’ to cancel a previous operation/command.