

Tecniche di Progettazione: Design Patterns

GoF: Command

These top secret drop boxes have revolutionized the spy industry. I just drop in my request and people disappear, governments change overnight and my dry cleaning gets done. I don't have to worry about when, where, or how; it just happens!



Usual message/invoke

- ▶ When two objects communicate, often one object is sending a message to a receiver to perform a particular function
- ▶ The first object (the "sender") could hold a reference to the second (the "receiver")
 - ▶ or get it as a return value, or argument, or construct it
- ▶ The sender invokes a specific method of the receiver

The Command Pattern

- ▶ But what if the sender is not aware of, or does not care who the receiver is?
- ▶ The Command design pattern encapsulates the concept of a "Command" as an object
- ▶ The sender holds a reference to a Command object rather than to the specific receiver
 - ▶ The Command object encapsulates the receiver

The Command Pattern

- ▶ The sender sends a vanilla message (such as `actionPerformed`, `execute`, `doit`, or `undo`) to the Command object
- ▶ The Command object is then responsible for dispatching the correct messages to the specific receiver(s) to get the job done

Command Pattern in Java

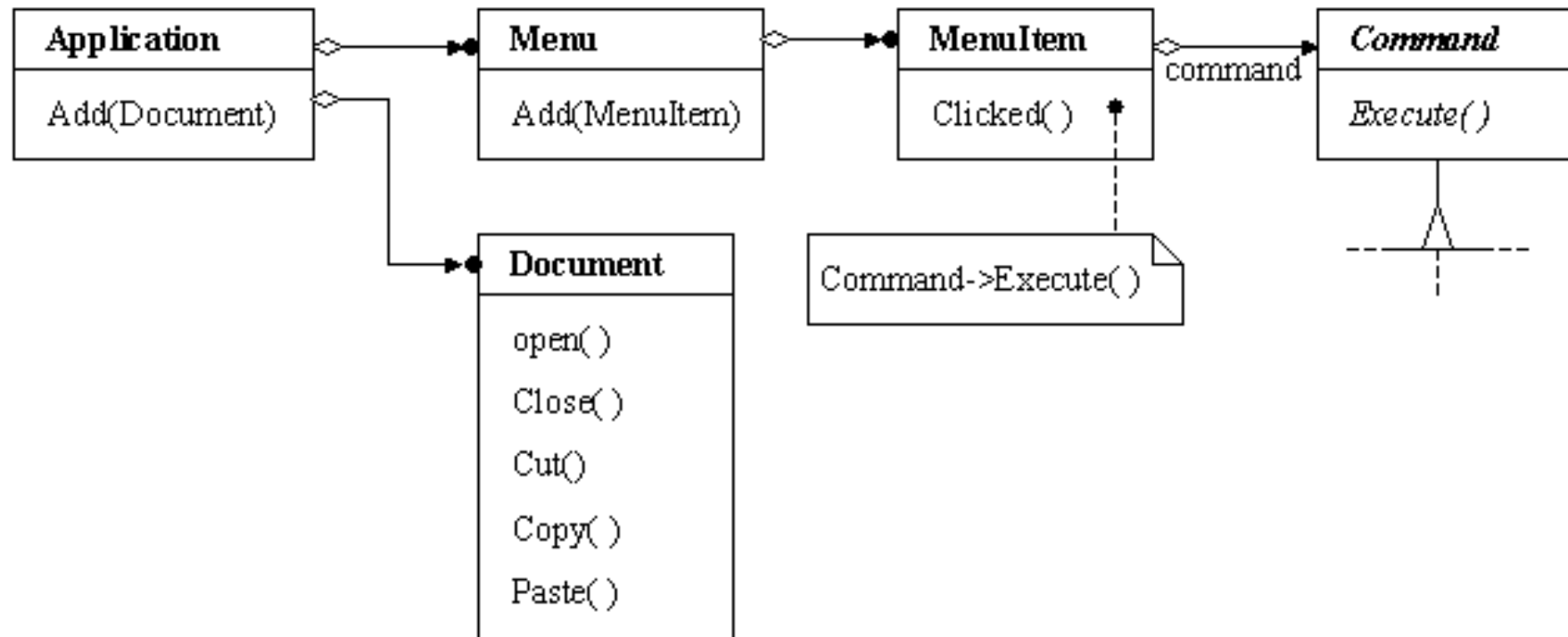
- ▶ One object can send messages to other objects without knowing anything about the actual operation or the type of object
- ▶ Polymorphism lets us encapsulate a request for services as an object
 - ▶ Establish a method signature name as an interface
 - ▶ Vary the algorithms in the called methods



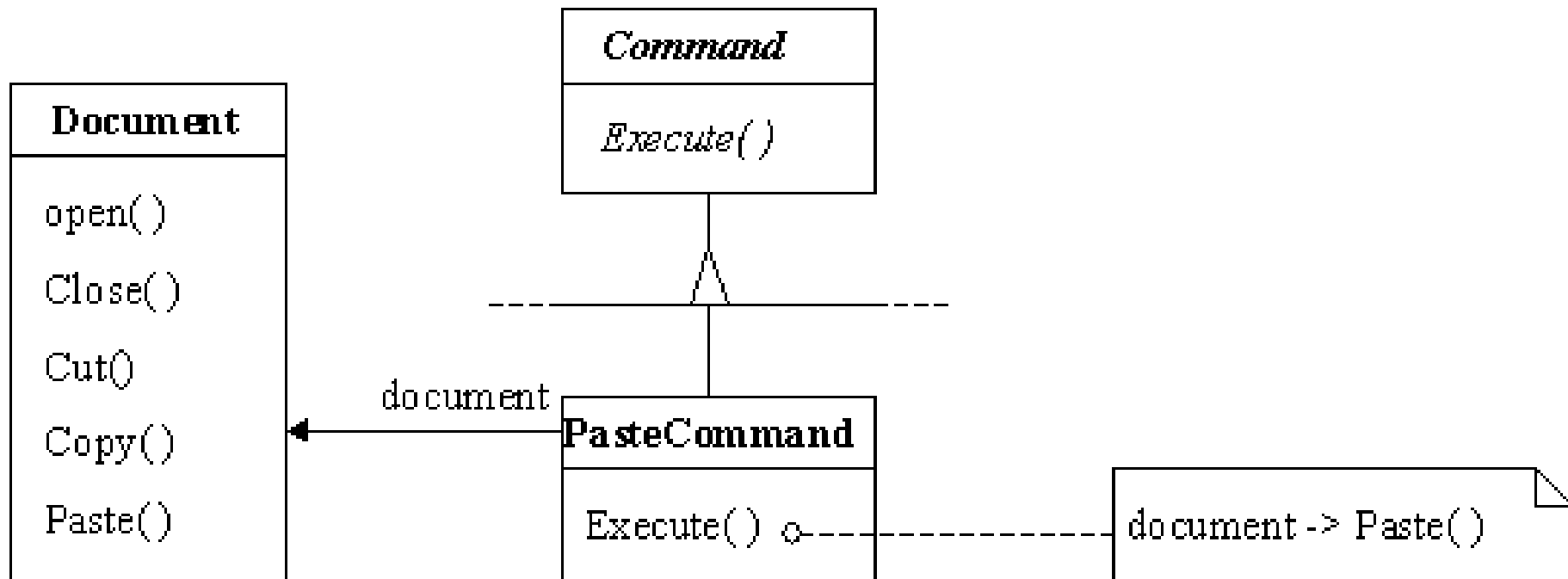
Uses

- ▶ The Command object can also be used when you need to tell the program to execute the command later.
 - ▶ In such cases, you are saving commands as objects to be executed later

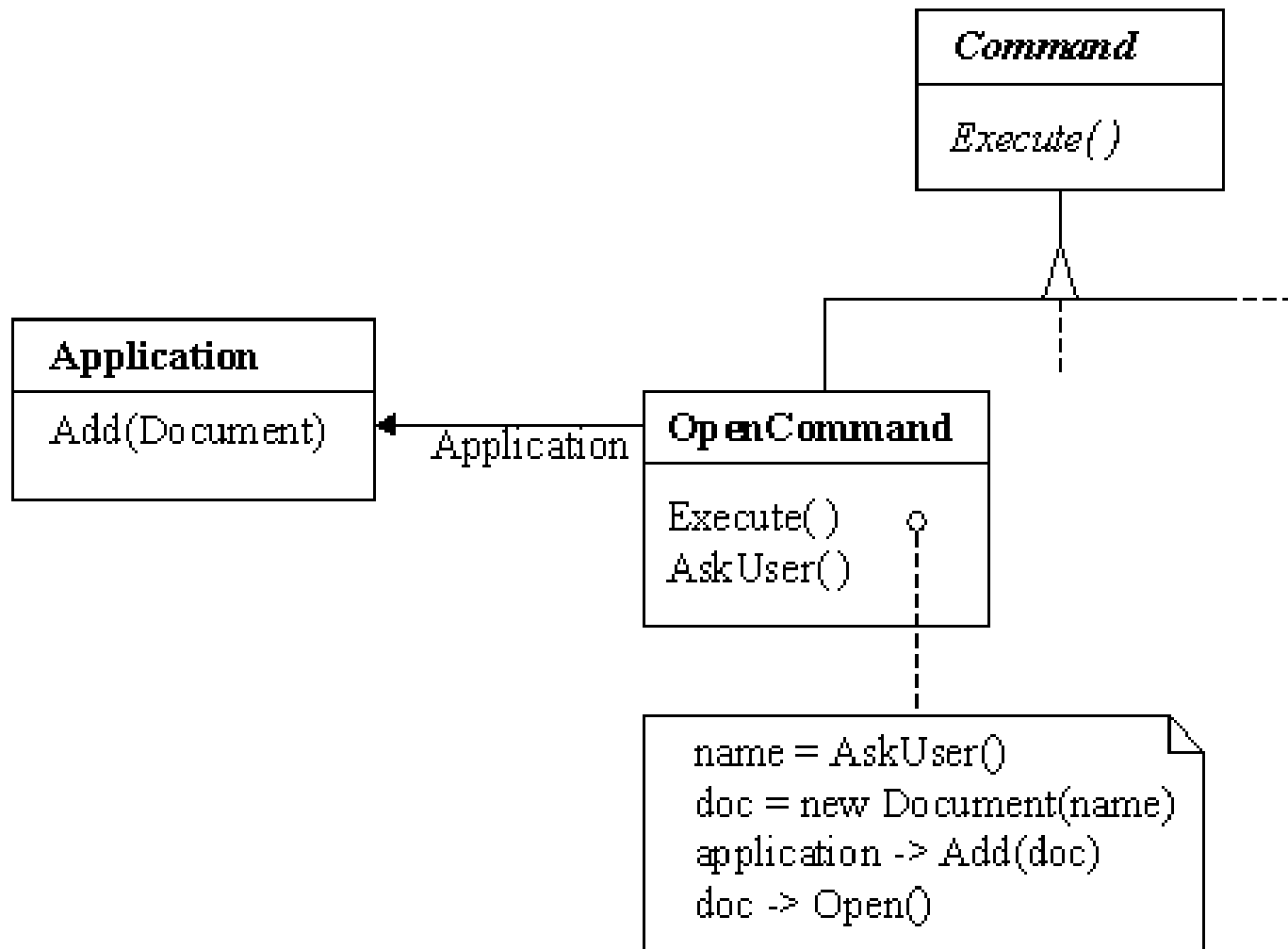
GoF example



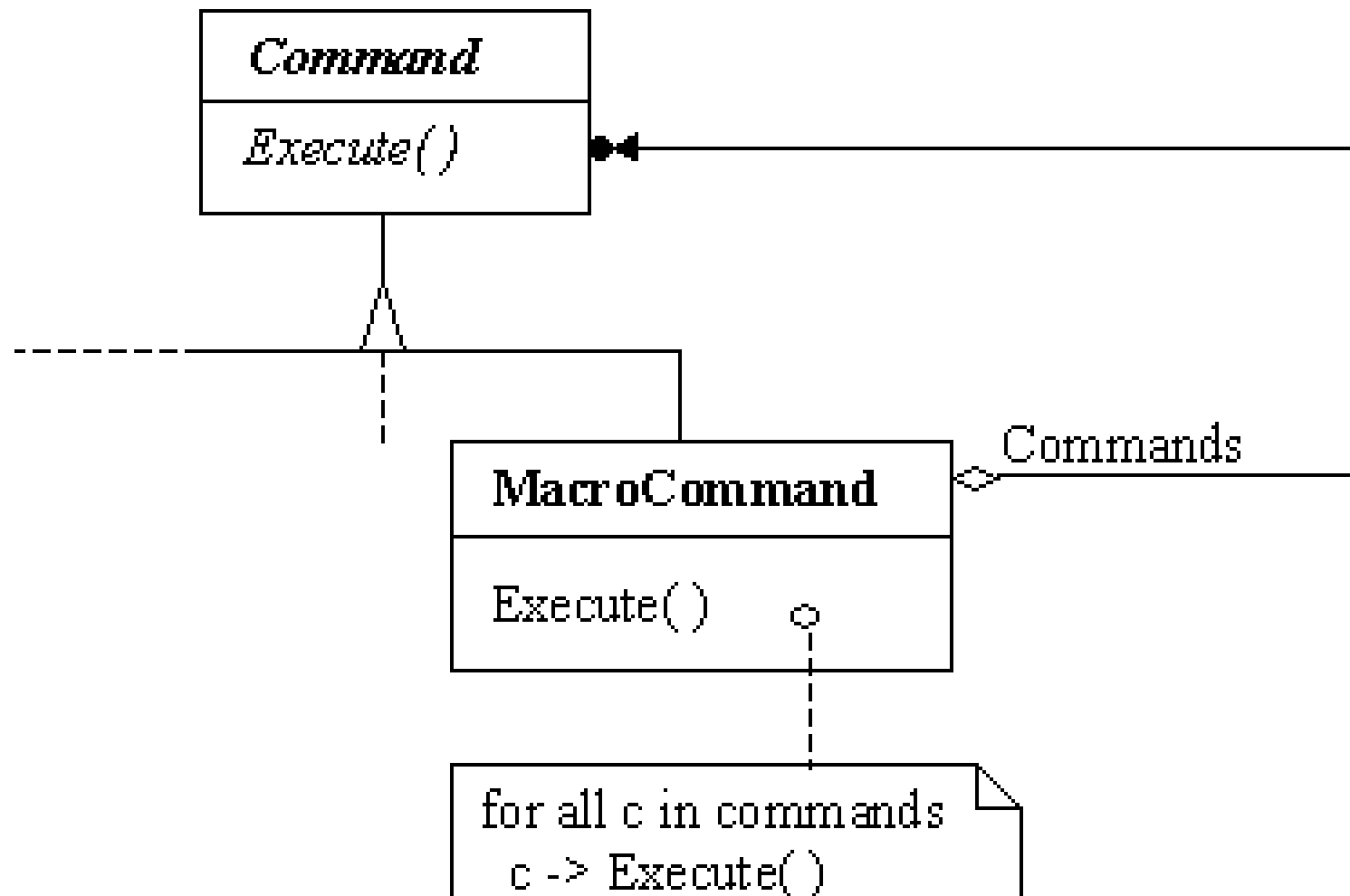
PasteCommand is a concrete Command that implements paste function.



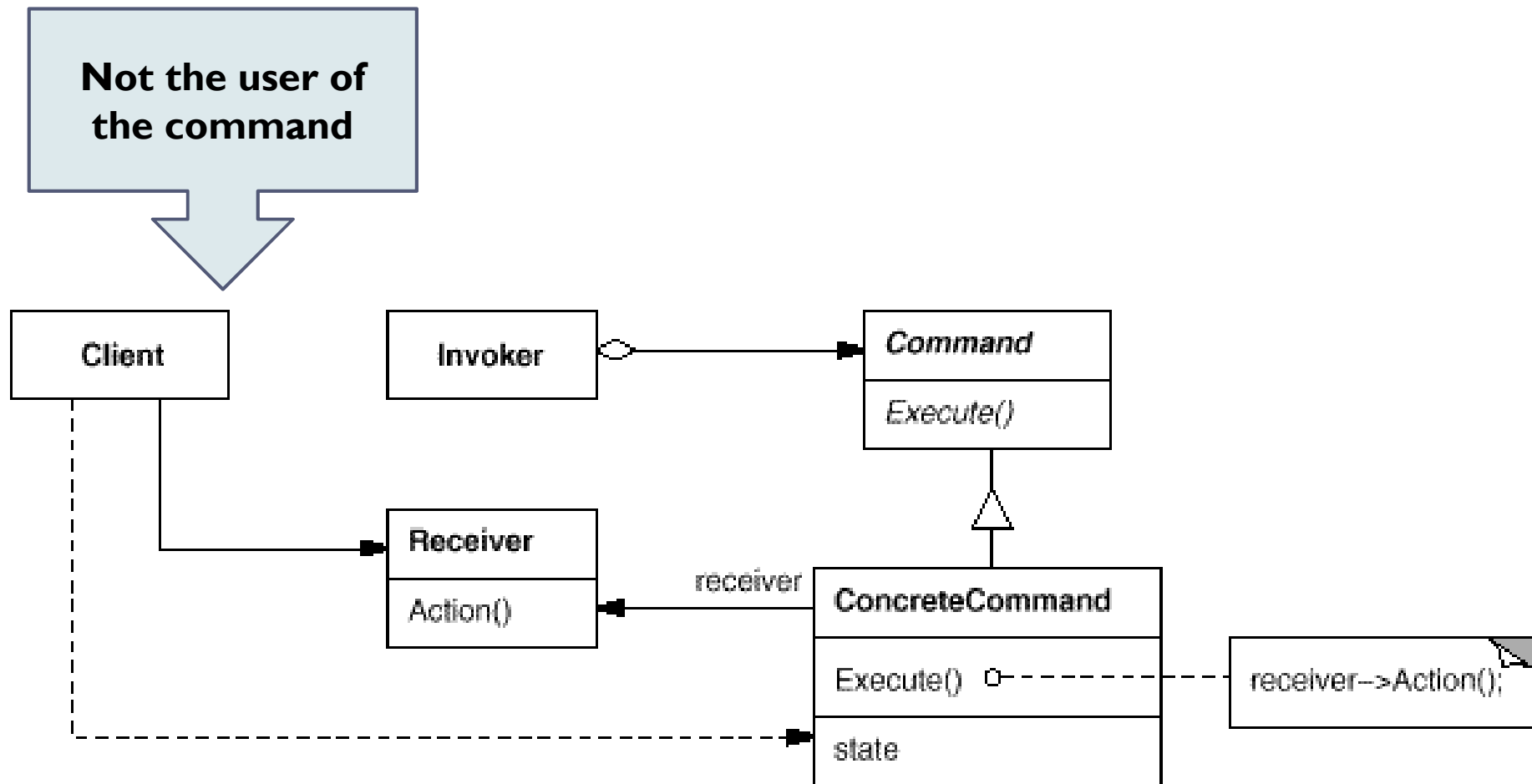
OpenCommand is a concrete Command that implements open function.



MacroCommand is a concrete Command that executes a sequence of commands.



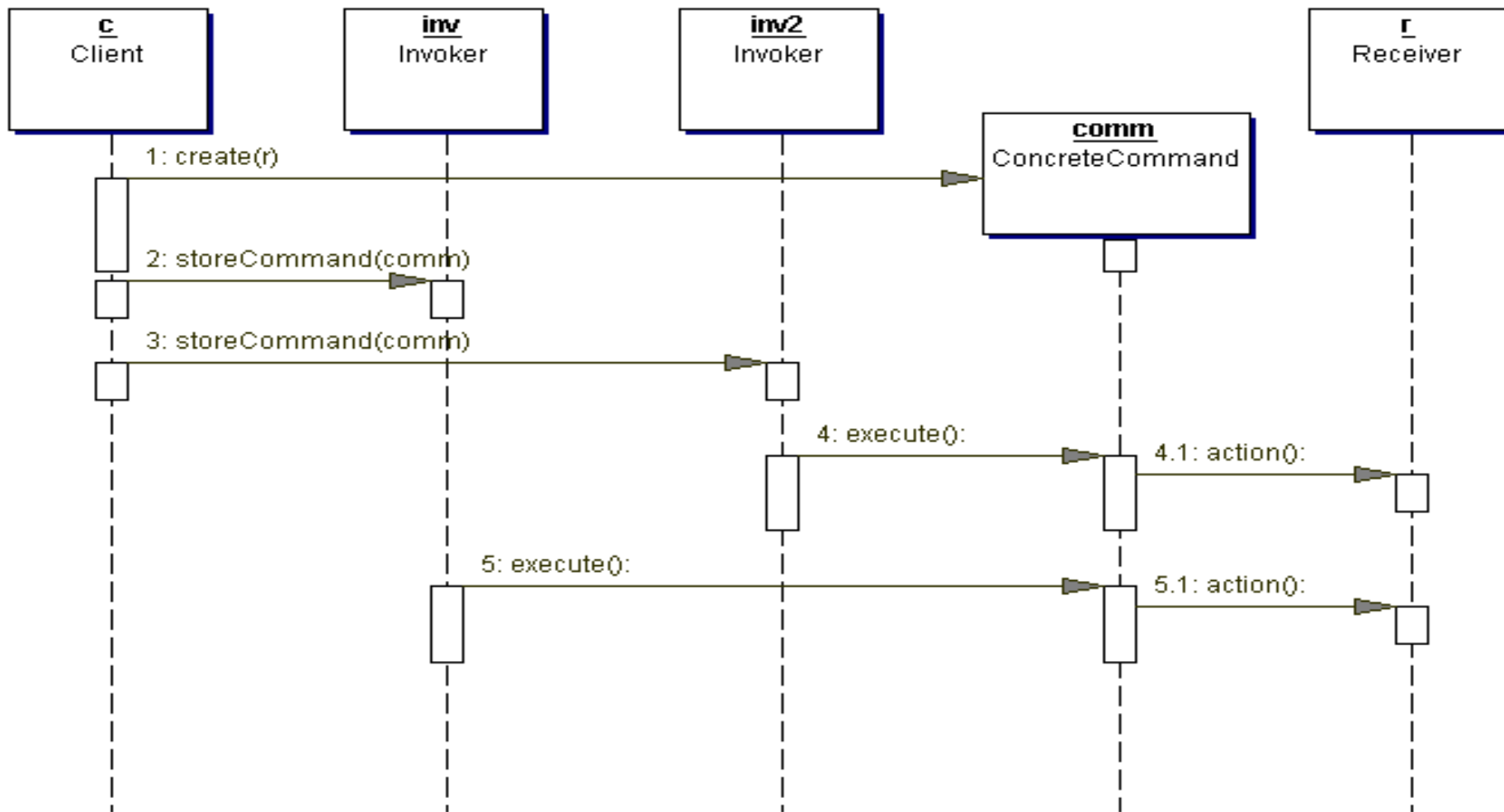
The Command Pattern structure



Command: Participants

- ▶ **Command:** declares an interface for executing an operation.
- ▶ **ConcreteCommand:** defines a binding between a Receiver object and an action, and implements Execute.
- ▶ **Client:** creates a ConcreteCommand object and sets its receiver.
- ▶ **Invoker:** asks the command to carry out the request.
- ▶ **Receiver:** knows how to perform the operations.

Command: collaboration (with two invokers for a command)



Implementation issues

- ▶ **How intelligent should a command be?**
 - ▶ one extreme: A command only defines a binding between a receiver and the actions that carry out the request.
 - ▶ the other extreme: A command implements everything itself without delegating to a receiver at all.
- ▶ **Supporting undo and redo. A ConcreteCommand class might need to store some additional states:**
 - ▶ the Receiver object
 - ▶ the arguments to the operation performed on the receiver
 - ▶ any original values in the receiver that may change as a result of handling the request

Command pattern: Consequences

- ▶ You can undo/redo any Command
 - ▶ Each Command stores what it needs to restore state
- ▶ You can store Commands in a stack or queue
 - ▶ Command processor pattern maintains a history
- ▶ It is easy to add new Commands, because you do not have to change existing classes
 - ▶ Command is an abstract class, from which you derive new classes
 - ▶ `execute()`, `undo()` and `redo()` are polymorphic functions



Asynchronous Method Invocation

- ▶ Another usage for Command is to run commands asynchronously in background of an application.
 - ▶ In this case the invoker is running in the main thread and sends the requests to the receiver which is running in a separate thread.
 - ▶ The invoker will keep a queue of commands to be run and will send them to the receiver while it finishes running them.
- ▶ Instead of using one thread in which the receiver is running more threads can be created for this. The invoker will use a pool of receiver threads to run command asynchronously.

Summary

- ▶ The Command design pattern encapsulates the concept of a command into an object.
- ▶ A command object could be sent across a network to be executed elsewhere or it could be saved as a log of operations.

Example for Undo

- ▶ For assume we have a class `Account` representing a bank account.
- ▶ Operations
 - ▶ `withdraw()`
 - ▶ `deposit()`
- ▶ We want to undo them
 - ▶ In inverse chronological order

```
public class Account {
    private double balance; // Saldo del conto

    public Account(double initialBalance) {
        balance=initialBalance;
    }
    // Restituisce il saldo
    public double getBalance() {
        return balance;
    }
    // Esegue un versamento
    public void deposit(double amount) {
        balance += amount;
    }
    // Esegue un prelievo
    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

```
}
```

```
public abstract class Command {
    protected Account account;
    protected Command(Account account) {
        this.account = account;
    }

    public abstract void perform();
    public abstract void undo();
}
```

```
public class DepositCommand extends Command {
    private double amount;
    public DepositCommand(Account account, double amount) {
        super(account);
        this.amount=amount;
    }

    public void perform() {
        account.deposit(amount);
    }

    public void undo() {
        account.withdraw(amount);
    }
}
```

```
public class WithdrawCommand extends Command {
    private double amount;
    public WithdrawCommand(Account account, double amount) {
        super(account);
        this.amount=amount;
    }

    public void perform() {
        account.withdraw(amount);
    }

    public void undo() {
        account.deposit(amount);
    }
}
```

```
import java.util.Stack;

public class AccountManager {
    private Account account;
    private Stack<Command> commandHistory;

    public AccountManager(Account account) {
        this.account=account;
        commandHistory=new Stack<Command>();
    }

    public double getBalance() {
        return account.getBalance();
    }

    // continua ...
}
```



```
// ... continua
public void deposit(double amount) {
    Command cmd=new DepositCommand(account, amount);
    commandHistory.push(cmd);
    cmd.perform();
}
public void withdraw(double amount) {
    Command cmd=new WithdrawCommand(account, amount);
    commandHistory.push(cmd);
    cmd.perform();
}
public void undo() {
    Command last=commandHistory.pop();
    last.undo();
}
}
```

Homework

- ▶ A company has set up a network crisis center to quickly respond to problems that may arise on any of its dozens of Web servers. E.g., if a major problem arises at its Denver server, someone might need to shut down that server by
 - ▶ connecting to it, issuing multiple shutdown commands (backing up files, rerouting network connectivity around the server, performing system diagnostics, etc.), and ultimately disconnecting.
- ▶ If a less severe problem arises at the Miami server, crisis center personnel might need to reboot that server by
 - ▶ connecting to it, issuing multiple reboot commands (transmitting warnings to connected servers, loading appropriate boot code, confirming proper restart, etc.), and then disconnecting.

Homework (cont'd)

- ▶ Currently, personnel must issue long sequences of commands to one of many servers whenever a problem arises.
- ▶ This could easily lead to commands being issued in the wrong sequence or to the wrong destination.
- ▶ By encapsulating the individual commands needed to handle a particular problem into a single Command object and then binding that object to a specific receiver, the potential for miscommunication is reduced.

- ▶ Permit undo.