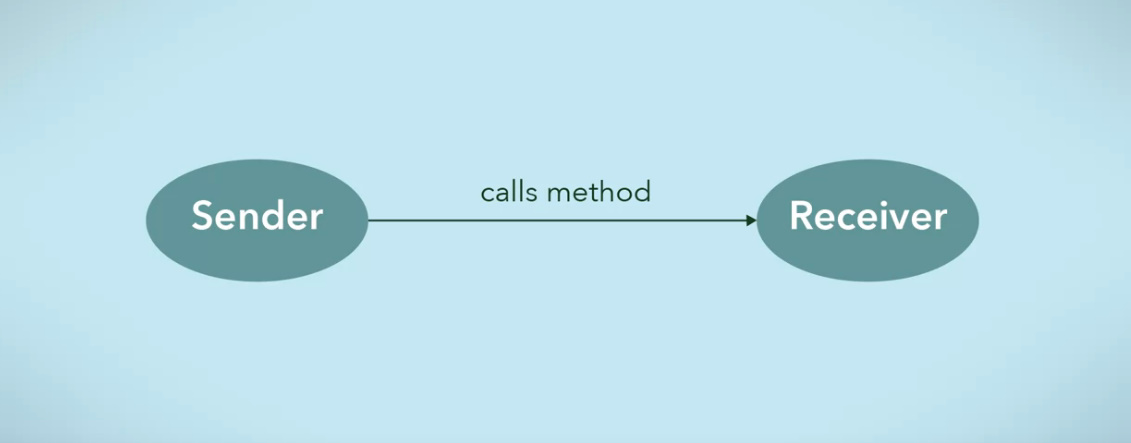
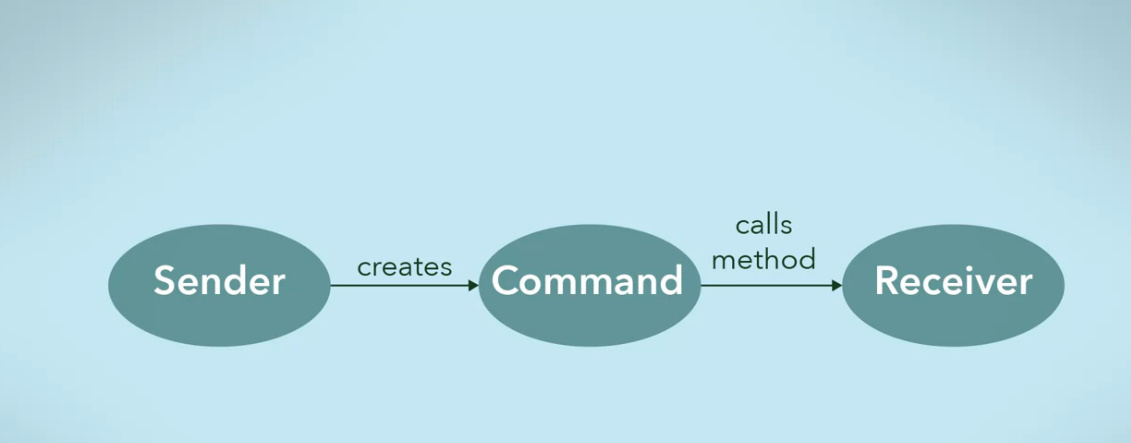
* **Command Pattern** – Incapsuleaza un request intr-un obiect.Se ocupa de trimiterea si interceptarea la requests, unde fiecare request este un obiect separat, numit Command Object.
* In mod traditonal, un sender, care trimite requests, comunica direct cu receiver, si-i spune ce metoda sa execute pentru request a lui,adica comunica direct cu el, **dar asta e rau de tot**



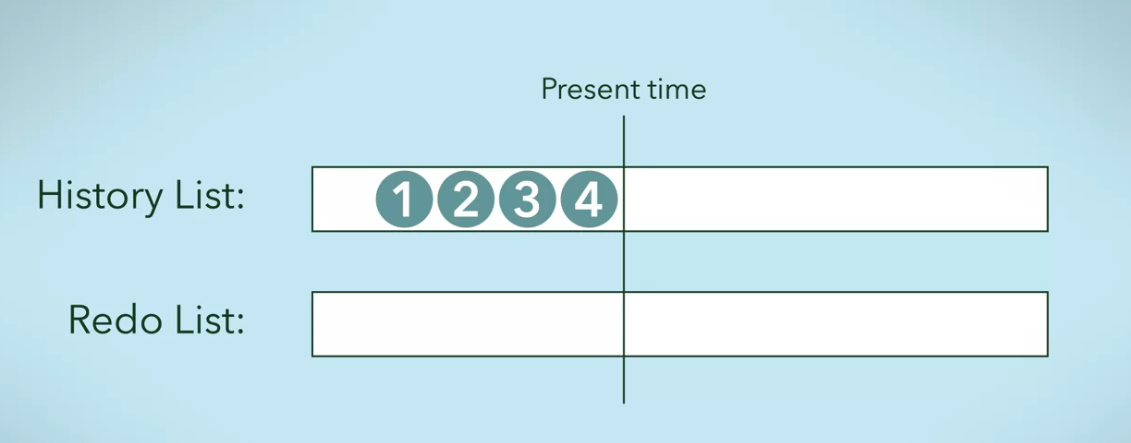
* Command Pattern se bazeaza pe aceea ca cream un Command Object ce sa fie intre Sender si Receiver si sa asigure comunicarea intre ele, si asa Sender nu mai comunica direct cu Receiver, ci Command Object va contine deja ce actiune sau request trebuie facut. **Fiecare request e un command object separat**



* De exemplu, un boss trebuie sa ofere niste taskuri la angajati, dar e ocupat. El va scrie pe foi cine si ce trebuie sa faca, deci el e Sender,foile cu instructiuni sunt Command Ojects, si apoi i va da aceste foi la secretara, ea e Invoker, si ea le va duce la angajati, adica Receivers
* Exact asa cum seful creaza foi cu taskuri, Sender creaza Command Objects
* Sender Objects creaza Command Objects pentru requests
* **Invoker** este obiectul care asigura executia la command object si deci are grija sa execute metoda de la Receiver. Asta e secretara. Deci, Invoker e doar puntea dintre Command si Receiver, el executa Command, ce are un receiver.
* **Command Manager** – Poate fi folosit si el, si anume el pastreaza ce commands au fost create si trecute prin Broker, ca un history, le manipuleaza si executa. **El da undo si redo si executa command object**
* **Rolurile la command Pattern**:

1. De a stoca request objects(command objects) intr-o lista
2. De a manipula requests(command objects) inainte de a fi executate
3. De a pune command objects intr-o queue ca sa le execute intr-o perioada de timp rand pe rand
4. De a da redo si undo

* De ex, avem o aplicatie calendar si adaugam ceva evenimente, gen la vreo data. Cand adaugam evenimentul la o data, vrem ca ceasul sa sune. Cand adaugam event la data, un Command Object e creat pentru a porni alarma cand va trebui. El e pus in queue ca sa fie executat mai tarziu
* **Command Pattern mai si permtie sa dam undo si redo, ca in documente word**
* De ex, in word:



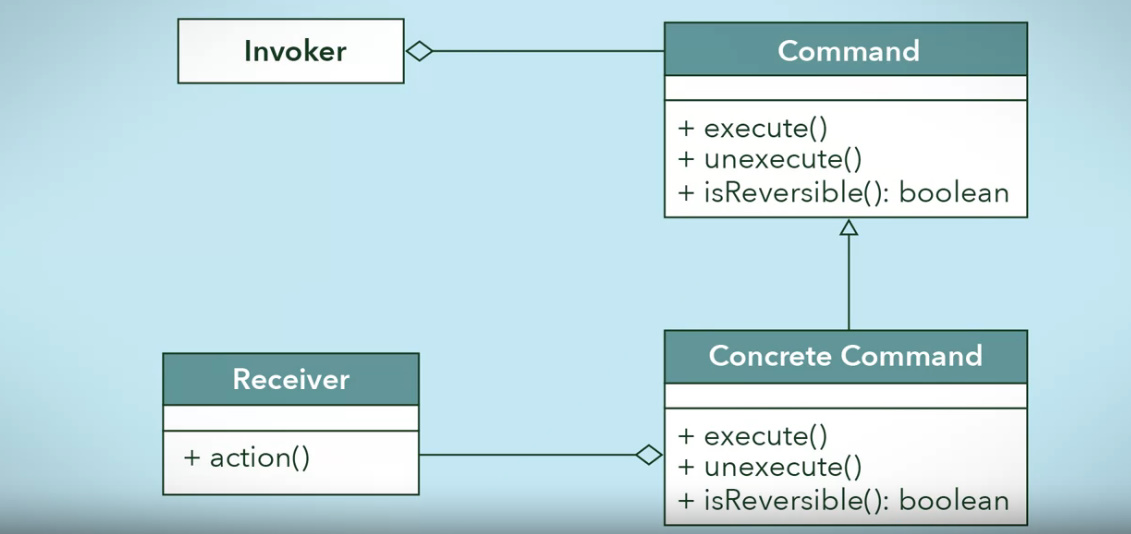
Fiecare events creaza un Command object ce e pus in history list

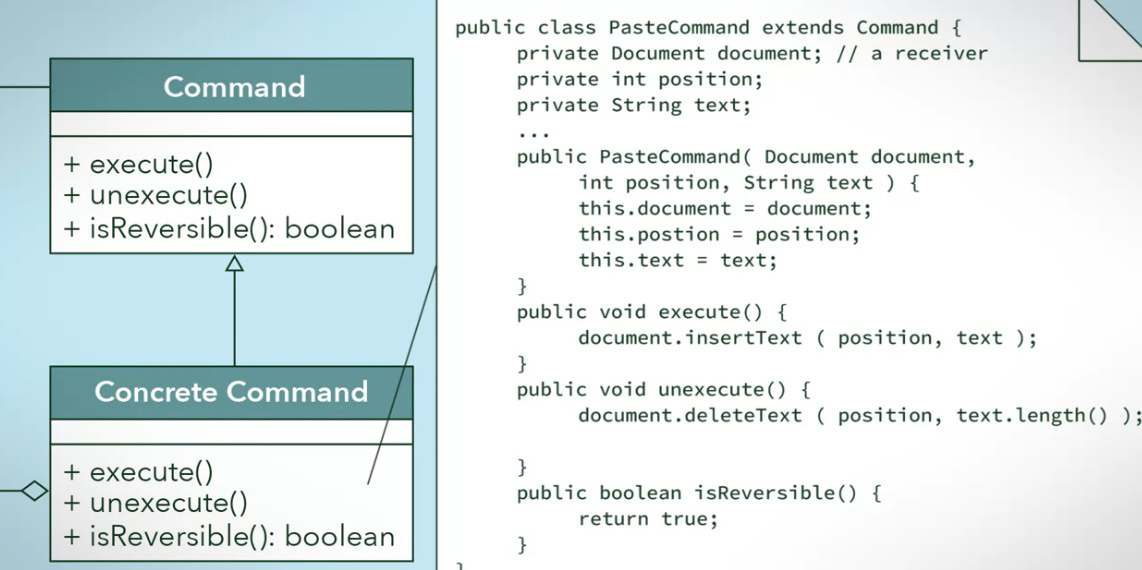
Redo pastreaza Command Object ce au fost undo, ca cand vom face redo, sa revina la ele.

Cand se va face redo, se vor lua deja din redo list si se vor pune inapoi in history list

* Partea buna e ca putem salva acste objects in memory, si daca programul da crash, tot putem sa folosim undo si redo, asa cum obiectele au fost salvate. Apelarea directa la metode nu permite nici decum asa ceva

**UML**

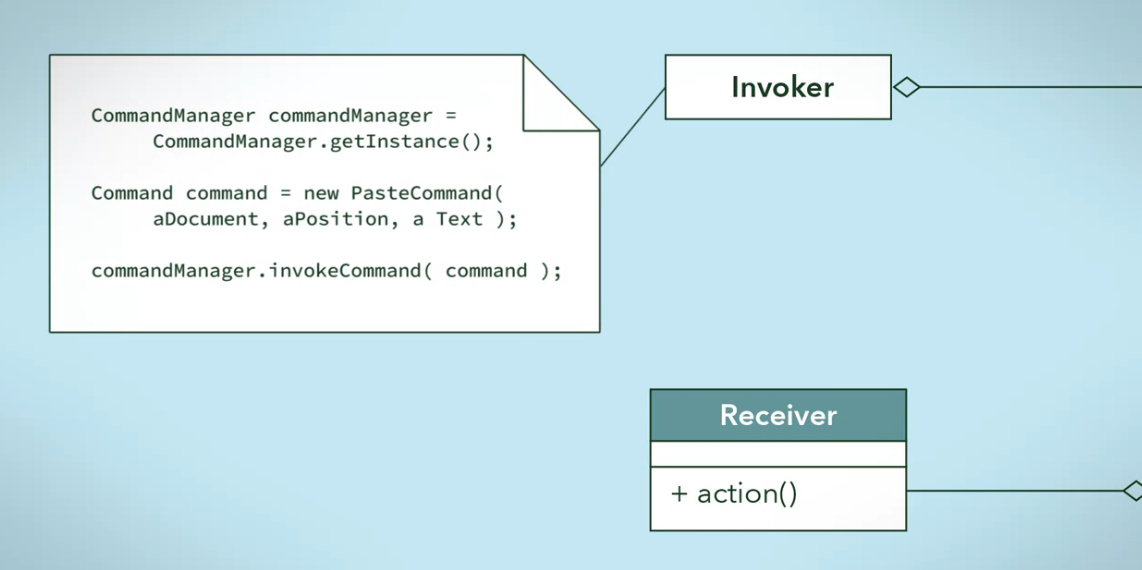




PasteCommand pastreaza informatii despre ce text va fi inserted si unde, ca sa poata da undo si redo. Command objects salveaza multe date ca sa poata permite undo si redo.

**Invoker are ca field un Command si Command are ca field un Receiver**

* Anume Command Object are ca field un Receiver si el il si foloseste pentru a indeplini request de la sender, trimis prin Invoker



**EXEMPLU 2**

Cream Receivers

public interface FileSystemReceiver {

void openFile();

void writeFile();

void closeFile();

}

Clasele receiver executa actiunile pentru events

public class UnixFileSystemReceiver implements FileSystemReceiver {

@Override

public void openFile() {

System.out.println("Opening file in unix OS");

}

@Override

public void writeFile() {

System.out.println("Writing file in unix OS");

}

@Override

public void closeFile() {

System.out.println("Closing file in unix OS");

}

}

public class WindowsFileSystemReceiver implements FileSystemReceiver {

@Override

public void openFile() {

System.out.println("Opening file in Windows OS");

}

@Override

public void writeFile() {

System.out.println("Writing file in Windows OS");

}

@Override

public void closeFile() {

System.out.println("Closing file in Windows OS");

}

}

Acum cream Commands:

public interface Command {

void execute();

}

Command are ca field un receiver

public class OpenFileCommand implements Command {

private FileSystemReceiver fileSystem;

public OpenFileCommand(FileSystemReceiver fs){

this.fileSystem=fs;

}

@Override

public void execute() {

//open command is forwarding request to openFile method

this.fileSystem.openFile();

}

}

public class CloseFileCommand implements Command {

private FileSystemReceiver fileSystem;

public CloseFileCommand(FileSystemReceiver fs){

this.fileSystem=fs;

}

@Override

public void execute() {

this.fileSystem.closeFile();

}

}

public class WriteFileCommand implements Command {

private FileSystemReceiver fileSystem;

public WriteFileCommand(FileSystemReceiver fs){

this.fileSystem=fs;

}

@Override

public void execute() {

this.fileSystem.writeFile();

}

}

Acum clasa Invoker

public class FileInvoker {

public Command command;

public FileInvoker(Command c){

this.command=c;

}

public void execute(){

this.command.execute();

}

}

Asta e o clasa de utilities, nu are nimic cu command pattern

public class FileSystemReceiverUtil {

public static FileSystemReceiver getUnderlyingFileSystem(){

String osName = System.getProperty("os.name");

System.out.println("Underlying OS is:"+osName);

if(osName.contains("Windows")){

return new WindowsFileSystemReceiver();

}else{

return new UnixFileSystemReceiver();

}

}

}

Si asta va fi un Sender

public class FileSystemClient {

public static void main(String[] args) {

//Creating the receiver object

FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();

//creating command and associating with receiver

OpenFileCommand openFileCommand = new OpenFileCommand(fs);

//Creating invoker and associating with Command

FileInvoker file = new FileInvoker(openFileCommand);

//perform action on invoker object

file.execute();

WriteFileCommand writeFileCommand = new WriteFileCommand(fs);

file = new FileInvoker(writeFileCommand);

file.execute();

CloseFileCommand closeFileCommand = new CloseFileCommand(fs);

file = new FileInvoker(closeFileCommand);

file.execute();

}

}

Client sau Sender e responsabil sa creeze instanta pentru Receiver, Command si Invoker si sa le uneasca cum trebuie.