Command

Department of Computer Science
University of Pretoria

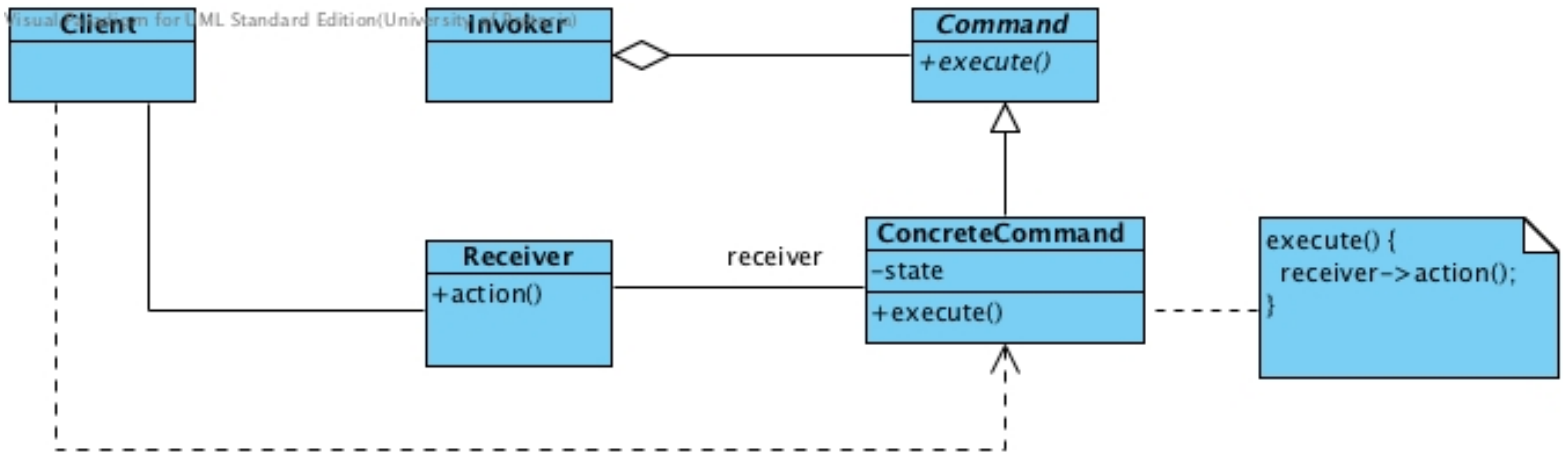
07 September 2023

Name and Classification:

Command (Object Behavioural)

Intent:

"Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations." GoF(263)


Client
Invoker
Command
+execute()
Receiver
+action()
receiver
ConcreteCommand
-state
+execute()
execute() {
receiver->action();
}

Command

- declares an interface for executing an operation.

ConcreteCommand

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

Client (Application)

Invoker

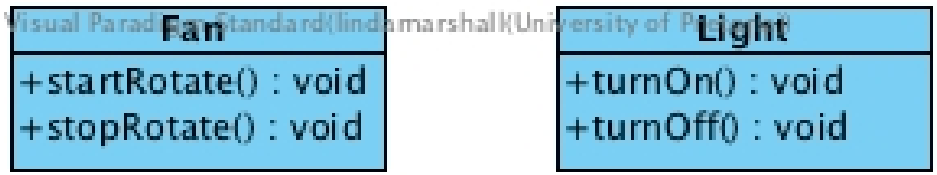
- asks the command to carry out the request.

Receiver

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

- **Chain of Responsibility** (GoF 251) can use Command to represent requests as objects.
- **Composite** (GoF 183), can be used to implement MacroCommands.
- **Memento** (GoF 316), can keep the state of the command required to undo its effect.
- A command that must be copied before being placed on the history list acts as a **Prototype** (GoF 133).

What do we have?




```
Light*  testLight = new Light( );
Fan* testFan = new Fan();
testLight -> turnOn();
testLight -> turnOff();
testFan -> startRotate();
testFan -> stopRotate();
```

We would like to use an instruction to...

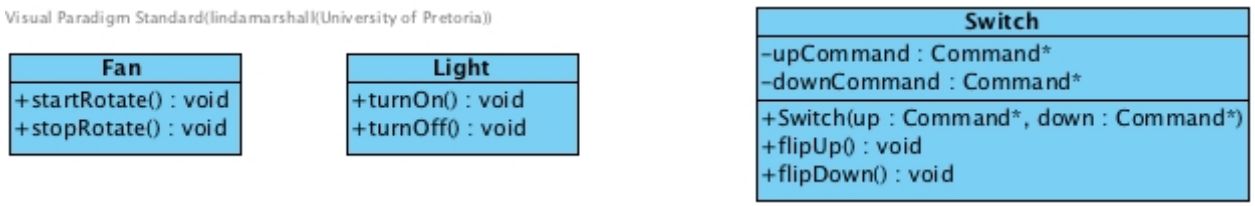
- ... turn the light on or start the fan rotating
- ... turn the light off or stop the fan from rotating

Assume these instructions are `flipup` and `flipdown` respectively and are members of the `Switch` class.

We have 3 classes – not related:

- `Fan` – the class which controls the fan operations
- `Light` – the class controlling the light operations
- `Switch` – the class that will make the fan or light do their thing ;-)

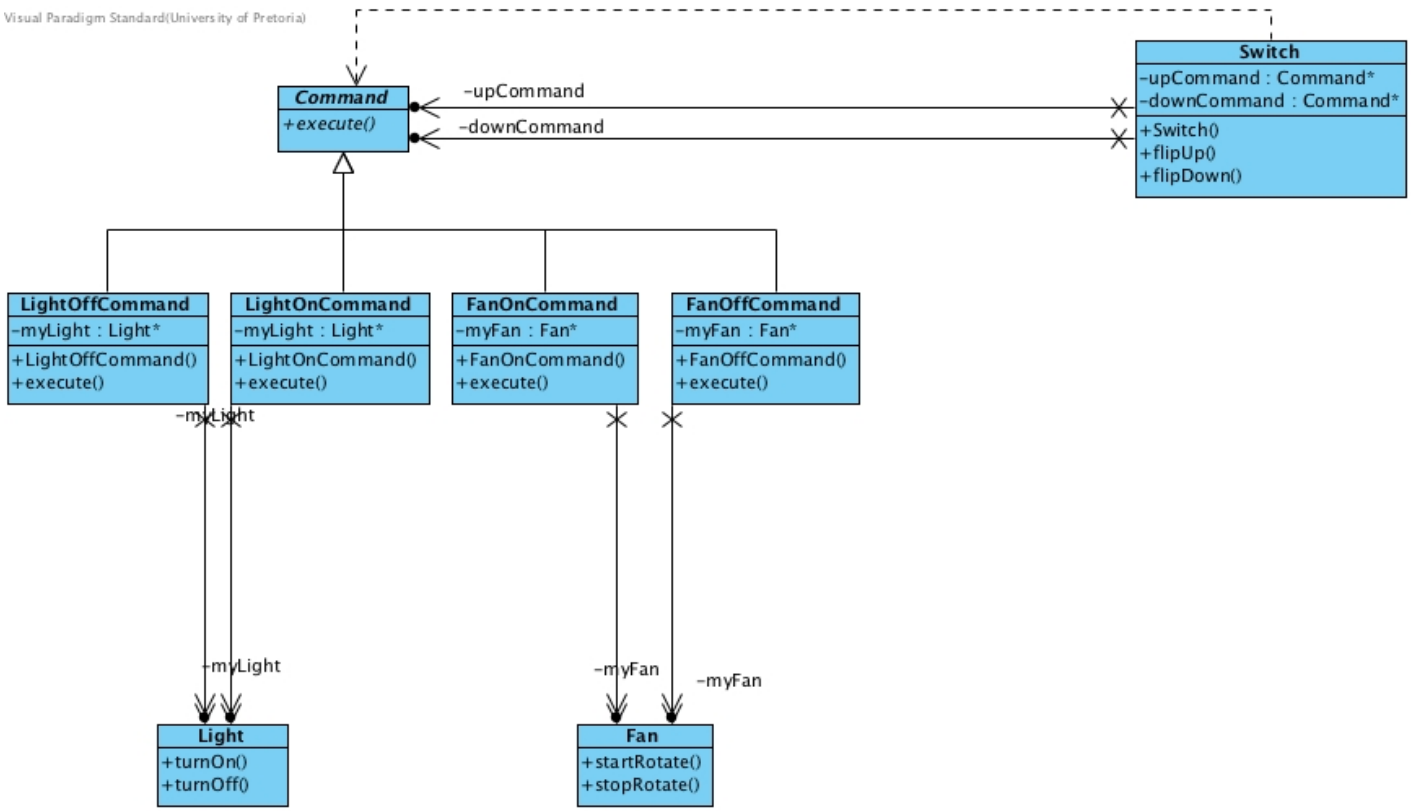
We have the *Invoker* participant (`Switch`) and two *Receiver* participants (`Fan` and `Light`).



```
class Fan {
public:
    void startRotate() { cout << "Fan is rotating" << endl;}
    void stopRotate() { cout << "Fan is not rotating" << endl;}
};

class Light {
public:
    void turnOn( ) { cout << "Light is on " << endl; }
    void turnOff( ) { cout << "Light is off" << endl; }
};
```

```
class Switch {
public:
    Switch(Command* up, Command* down) {
        upCommand = up;
        downCommand = down;
    }

    void flipUp() { upCommand->execute( );};
    void flipDown() { downCommand->execute ( );};

private:
    Command* upCommand;
    Command* downCommand;
};
```

Now, draw the `Command` hierarchy and link it into what you already have ...


Visual Paradigm Standard(University of Pretoria)
Command
+execute()
-upCommand
-downCommand
Switch
-upCommand : Command*
-downCommand : Command*
+Switch()
+flipUp()
+flipDown()
LightOffCommand
-myLight : Light*
+LightOffCommand()
+execute()
LightOnCommand
-myLight : Light*
+LightOnCommand()
+execute()
FanOnCommand
-myFan : Fan*
+FanOnCommand()
+execute()
FanOffCommand
-myFan : Fan*
+FanOffCommand()
+execute()
-myLight
-myLight
Light
+turnOn()
+turnOff()
-myFan
-myFan
Fan
+startRotate()
+stopRotate()

```
class Command {
public:
    virtual void execute () = 0;
};

class LightOnCommand : public Command {
public:
    LightOnCommand (Light* L) { myLight = L;}
    void execute() { myLight->turnOn(); }
private:
    Light* myLight;
};

class LightOffCommand : public Command {
public:
    LightOffCommand (Light* L) { myLight = L;}
    void execute() { myLight->turnOff(); }
private:
    Light* myLight;
};
```

```
class FanOnCommand : public Command {
public:
    FanOnCommand (Fan* f) { myFan = f;}
    void execute() { myFan->startRotate();}
private:
    Fan* myFan;
};

class FanOffCommand : public Command {
public:
    FanOffCommand (Fan* f) { myFan = f;}
    void execute() {myFan->stopRotate();}
private:
    Fan* myFan;
};
```

```
Light*  testLight = new Light( );
Fan* testFan = new Fan();

LightOnCommand* testLiOnCmnd = new LightOnCommand(testLight);
LightOffCommand* testLiOffCmnd = new LightOffCommand(testLight);
FanOnCommand* testFaOnCmnd = new FanOnCommand(testFan);
FanOffCommand* testFaOffCmnd = new FanOffCommand(testFan);

Switch* lightSwitch = new Switch(testLiOnCmnd,testLiOffCmnd);
Switch* fanSwitch = new Switch(testFaOnCmnd,testFaOffCmnd);

lightSwitch -> flipUp();
lightSwitch -> flipDown();
fanSwitch -> flipUp();
fanSwitch -> flipDown();
```

The code as written requires two switches. A switch to operate the light and a switch to operate the fan. Modify the `Switch` class so that it will turn on the light and the fan when flipped up and turn off the light and fan when flipped down.