Command Pattern

Propósito

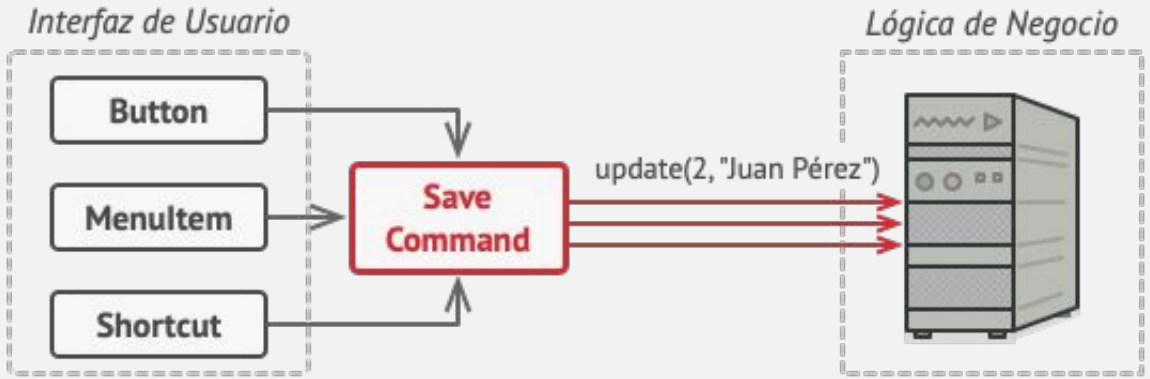
- Este patrón convierte una solicitud en un objeto independiente que convierte toda la información de la solicitud. Esto permite retrasar o poner en cola la ejecución de la solicitud.

Analogía en el mundo real

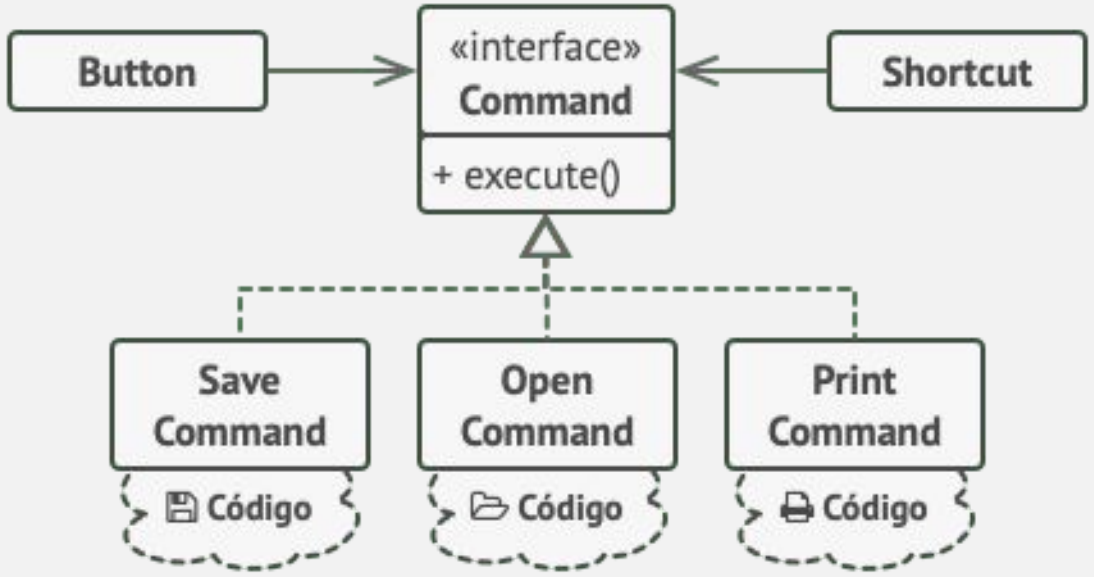
Problema ejemplo

- Tenemos un programa con multitud de botones asignados a la clase padre de BOTÓN. La solución más simple es crear una subclase para cada botón.
- El problema con esto es que el código terminará siendo muy largo y podrías descomponer el código de las subclases cada vez que modifiques la clase BOTÓN.

Solución ejemplo



- El patrón Command sugiere que los objetos de la GUI no envíen las solicitudes directamente. En lugar de esto, extraes todos los detalles de la solicitud, así como el objeto que estás llamando, el nombre del método y la lista de argumentos; y ponerlos en una clase comando separada con un único método que activa la solicitud.



- Lo siguiente que tienes que hacer es que tus comandos implementen la misma interfaz.

commands/Command.java

```java
package refactoring_guru.command.example.commands;

import refactoring_guru.command.example.editor.Editor;

public abstract class Command {
    public Editor editor;
    private String backup;

    Command(Editor editor) {
        this.editor = editor;
    }

    void backup() {
        backup = editor.textField.getText();
    }

    public void undo() {
        editor.textField.setText(backup);
    }

    public abstract boolean execute();
}
```

commands/CopyCommand.java

```java
package refactoring_guru.command.example.commands;

import refactoring_guru.command.example.editor.Editor;

public class CopyCommand extends Command {

    public CopyCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        editor.clipboard = editor.textField.getSelectedText();
        return false;
    }
}
```

commands/PasteCommand.java

```java
package refactoring_guru.command.example.commands;

import refactoring_guru.command.example.editor.Editor;

public class PasteCommand extends Command {

    public PasteCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        if (editor.clipboard == null || editor.clipboard.isEmpty()) return false;

        backup();
        editor.textField.insert(editor.clipboard, editor.textField.getCaretPosition());
        return true;
    }
}
```

commands/CutCommand.java

```
package refactoring_guru.command.example.commands;

import refactoring_guru.command.example.editor.Editor;

public class CutCommand extends Command {

    public CutCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        if (editor.textField.getSelectedText().isEmpty()) return false;

        backup();
        String source = editor.textField.getText();
        editor.clipboard = editor.textField.getSelectedText();
        editor.textField.setText(cutString(source));
        return true;
    }

    private String cutString(String source) {
        String start = source.substring(0, editor.textField.getSelectionStart());
        String end = source.substring(editor.textField.getSelectionEnd());
        return start + end;
    }
}
```

editor/Editor.java

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Editor {
    public JTextArea textField;
    public String clipboard;
    private CommandHistory history = new CommandHistory();

    public void init() {
        JFrame frame = new JFrame("Text editor (type & use buttons, Luke!)");
        JPanel content = new JPanel();
        frame.setContentPane(content);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
        textField = new JTextArea();
        textField.setLineWrap(true);
        content.add(textField);
        JPanel buttons = new JPanel(new FlowLayout(FlowLayout.CENTER));
        JButton ctrlC = new JButton("Ctrl+C");
        JButton ctrlX = new JButton("Ctrl+X");
        JButton ctrlV = new JButton("Ctrl+V");
        JButton ctrlZ = new JButton("Ctrl+Z");
        Editor editor = this;
        ctrlC.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                executeCommand(new CopyCommand(editor));
            }
        });
        ctrlX.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                executeCommand(new CutCommand(editor));
            }
        });
        ctrlV.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                executeCommand(new PasteCommand(editor));
            }
        });
        ctrlZ.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                undo();
            }
        });
        buttons.add(ctrlC);
        buttons.add(ctrlX);
        buttons.add(ctrlV);
        buttons.add(ctrlZ);
        content.add(buttons);
        frame.setSize(450, 200);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private void executeCommand(Command command) {
        if (command.execute()) {
            history.push(command);
        }
    }
```

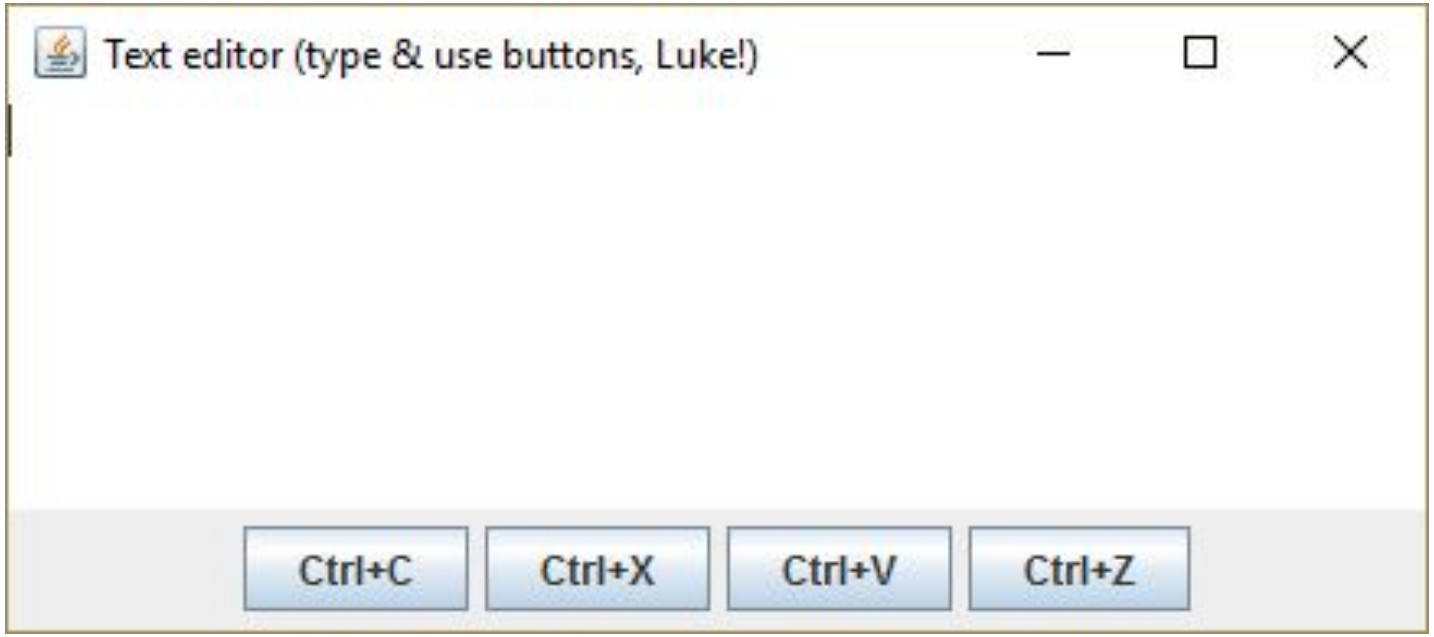
Demo.java

```java
package refactoring_guru.command.example;

import refactoring_guru.command.example.editor.Editor;

public class Demo {
    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.init();
    }
}
```

Output



Actividad

- Implementar diagrama de clases del editor de texto mostrado.
- Command en Java / Patrones de diseño