Command

Jan Jevčák

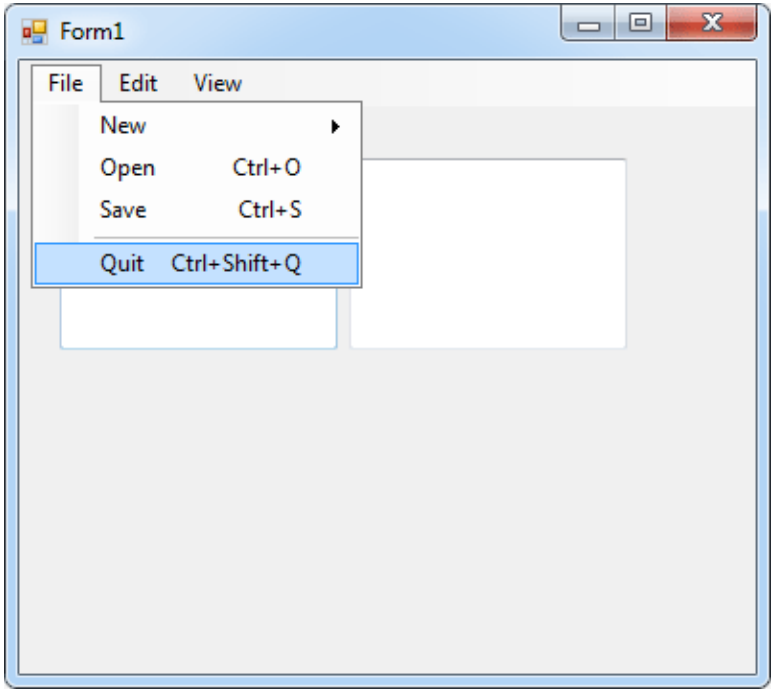
Motivace

Návrhový vzor **Command** je vzor chování, který zapouzdřuje příkazy do objektů se všemi informacemi potřebnými pro vykonání daného příkazu.

Zapouzdření umožňuje předat příkaz jako parametr, vytvořit frontu příkazů nebo jejich historii.

- **Potřebujeme reprezentovat žádost o vykonání akce**
  - standardní přístup je zavolat metodu na objektu
    - metoda se provede okamžitě
  - potřebujeme speciální sémantiku
    - zpožděné vykonání
    - vzdálené vykonání
    - explicitní kontrola akcí
  - potřeba abstrakce volání
  - oddělení volajícího (požadavku) a vykonávajího objektu

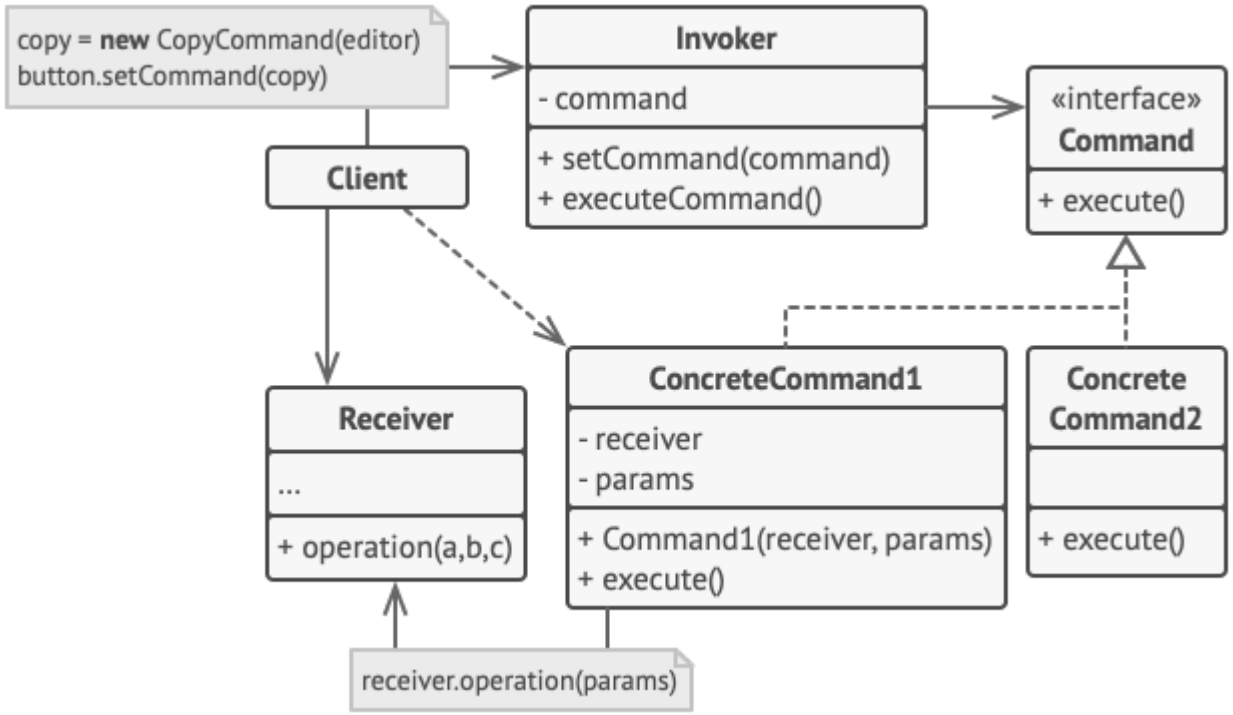
Motivace / Naivní implementace




```
public void CheckAction(string action) {
      if (action == "New")    {
            // handle action for new command
      }
      else if (action == "Open")    {
            // handle action for open command
      }
      else if (action == "Save")    {
            // handle action for save command
      }
      else (action == "Quit")    {
            // handle action for quit command
      }
}
```

Porušení principu otevřenosti a uzavřenosti OOP

Obecná struktura



Účastníci

Command

- Interface se signaturou metody *execute()*

ConcreteCommand

- Konkrétní implementace
- Drží recievera a parametry, se kterými má být vykonán
- Implementuje *execute()* zavoláním příslušných operací na Reciever

Client

- Vytváří Command objekty a nastavuje Reciver

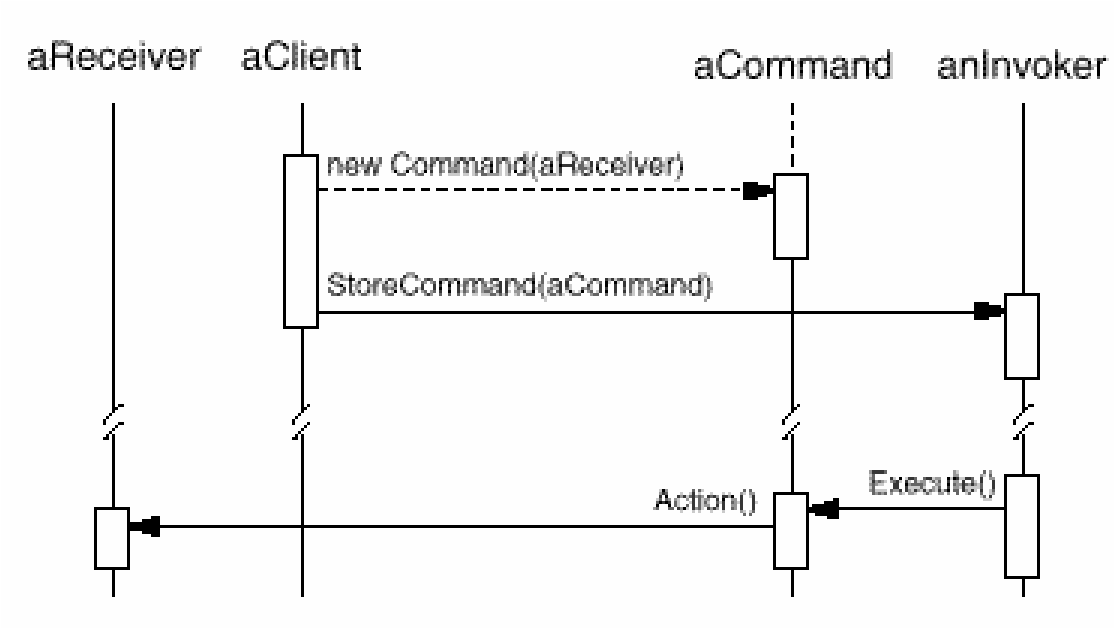
Invoker

- Zajistí provedení daného commandu
- Možnost několika zcela odlišných Invokerů

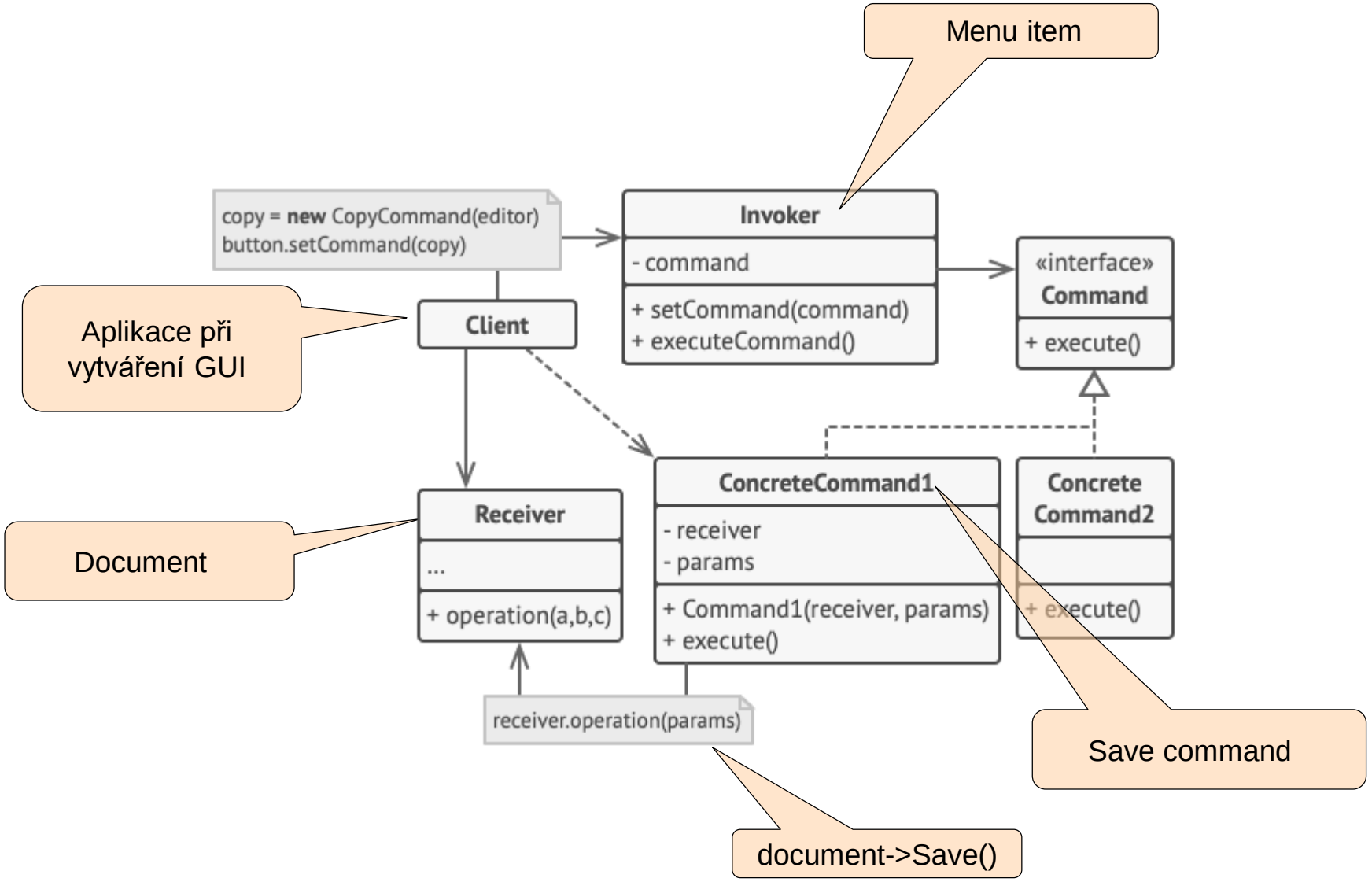
Reciever

- Třída, která ví, jak příkaz provést, nebo na které je příkaz prováděn

Konkrétní použití, runtime



Struktura příkladu



Příklad

```
class PDFDocument
{
    public void Save()
    {
        Console.WriteLine("PDF Document has
              been saved.");
    }
}
```

Receiver

```
class MenuItem
{
    private ICommand _command;

    public MenuItem(ICommand command)
    {
        _command = command;
    }

    public void Click()
    {
        if (_command != null)
        {
            _command.Execute();
        }
    }
}
```

Invoker

```
public interface ICommand
{
    void Execute();
}
```

Interface

```
class SaveCommand : ICommand
{
    private PDFDocument _pdfDocument;

    public SaveCommand(PDFDocument pdfDocument)
    {
        _pdfDocument = pdfDocument;
    }

    public void Execute()
    {
        _pdfDocument.Save();
    }
}
```

Command třída

Příklad

Client

```
internal class Program
{
    static void Main(string[] args)
    {
        // Creating receiver
        PDFDocument pdfDocument = new PDFDocument();

        // Creating command and associating with receiver
        ICommand saveCommand = new SaveCommand(pdfDocument);

        // Creating invoker
        MenuItem menuItem = new MenuItem(saveCommand);

        // Simulating user click
        menuItem.Click();
    }
}
```

Příklad

Příklad

```
class Light
{
    void TurnOn() {
        WriteLine("Light on");
    }

    void TurnOff() {
        WriteLine("Light off");
    }
}
```

Receiver

```
class Switch
{
    private ICommand upCommand, downCommand;

    public Switch(ICommand up, ICommand down)
    {
        upCommand = up;
        downCommand = down;
    }

    public void FlipUp() {
        upCommand.Execute();
    }

    public void FlipDown() {
        downCommand.Execute();
    }
}
```

Invoker

```
public interface ICommand
{
    void Execute();
}
```

Interface

```
class LightOnCommand : ICommand
{
    private Light myLight;

    LightOnCommand(Light light) {
        myLight = light;
    }

    void Execute() {
        myLight.TurnOn();
    }
}

class LightOffCommand : ICommand
{
    private Light myLight;

    LightOffCommand(Light light) {
        myLight = light;
    }

    public void Execute() {
        myLight.TurnOff();
    }
}
```

Command třídy

Příklad - rozšíření

```
class Fan
{
    void StartRotate() {
        WriteLine("Fan rotating");
    }

    void StopRotate() {
        WriteLine("Fan not rotating");
    }
}
```

Jiný receiver

```
class Switch
{
    private ICommand upCommand, downCommand;

    public Switch(ICommand up, ICommand down)
    {
        upCommand = up;
        downCommand = down;
    }

    public void FlipUp() {
        upCommand.Execute();
    }

    public void FlipDown() {
        downCommand.Execute();
    }
}
```

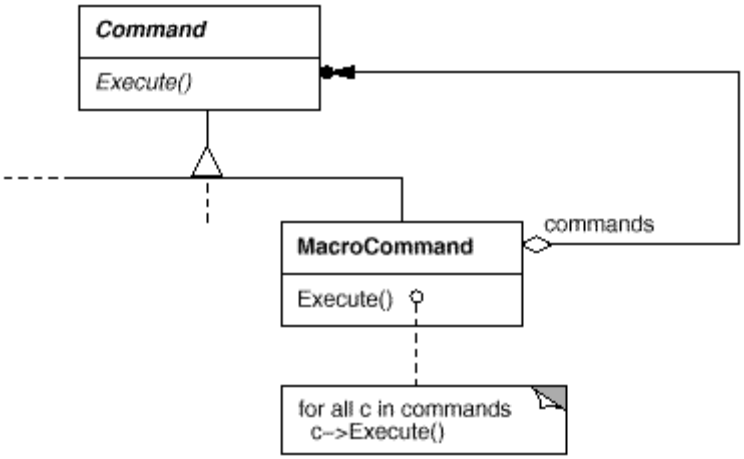
```
public interface ICommand
{
    void Execute();
}
```

```
class FanOnCommand : ICommand
{
    private Fan myFan;

    FanOnCommand(Fan fan) {
        myFan = fan;
    }

    void Execute() {
        myFan.StartRotate();
    }
}

class FanOffCommand : ICommand
{
    private Fan myFan;

    FanOffCommand(Fan fan) {
        myFan = fan;
    }

    void Execute() {
        myFan.StopRotate();
    }
}
```

Jiné command objekty

MacroCommand

- **Kompozice**
  - posloupnosti volání (makro)
  - založeno na Composite pattern
  - chování makra stejné jako jedné akce




```
class MacroCommand : ICommand
{
    private List<ICommand> commands;

    void Execute()
    {
        foreach (ICommand c in commands)
            c.Execute();

        // ...
    }
}
```

Undo, redo

- **Je třeba ukládat stav receiveru**
  - reference na objekt receiveru
  - argumenty volání Command objektu
  - původní hodnoty receiveru
- Command objekty občas musí být kopírovány
  - pokud se dále může měnit jejich stav
  - prototype pattern
- Pozor na chyby při undo/redo
  - zajistit konzistentní stav uložených command objektů (Memento)
- Alternativa: kompenzující opreace
  - např. vypni - zapni

Příklad - Undo

```
interface ICommand
{
    void Execute();
    void Undo();
}

class PasteCommand : ICommand
{
    private Document document;
    private string oldText;

    PasteCommand(Document doc)
    {
        document = doc;
    }

    void Execute()
    {
        oldText = document.GetText();
        document.Paste();
    }

    void Undo()
    {
        document.SetText(oldText);
    }
}
```

Nová metoda

Uloží stav

Obnoví stav z uložených hodnot

Známé použití

- **Multi-level Undo**
  - execute/undo, uchování/vytvoření předchozího stavu
- **Macro recording**
- **GUI toolkity**
  - Java Swing – interface Action, metoda actionPerformed
  - WPF .NET – ICommand
- **ThreadPool**
  - odkládání požadavků do fronty - zpracování, až na ně přijde řada
- **Parallel / cluster / distributed computing**
  - hi-perf scheduler, job queues
- **Předávání požadavků po síti, callbacky**
- **Progress bar**
  - getEstimatedDuration()
- **Wizards**
  - naklikat commandy, pak je najednou provést
- **“transakční” chování**
  - rollback, roll-forward

Související vzory

- **Composite**
  - vytváření maker (MacroCommand)
- **Memento**
  - uchovávání stavu objektů pro undo
- **Prototype**
  - při undo ukládání kopií akcí
- **Chain of Responsibility**
  - grafické toolkity – Command objekt může zpracovávat víc příjemců
  - propagace změny stavu, řetězení

Shrnutí

- **Výhody**
  - oddělení volajícího(požadavku) a vykonávajícího objektu
  - rozšiřitelnost - jednoduché přidání nových Commands
- **Nevýhody**
  - rychle rostoucí počet tříd - každý Command je nová třída
- **Použitelnost**
  - parametrizace objektů pomocí akcí k provedení
  - podpora undo operace
  - podpora logování změn jednotlivých operací
  - řazení požadavků ve frontě a jejich provádění v různých časech