INF011 – Padrões de Projeto

17 – *Command*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

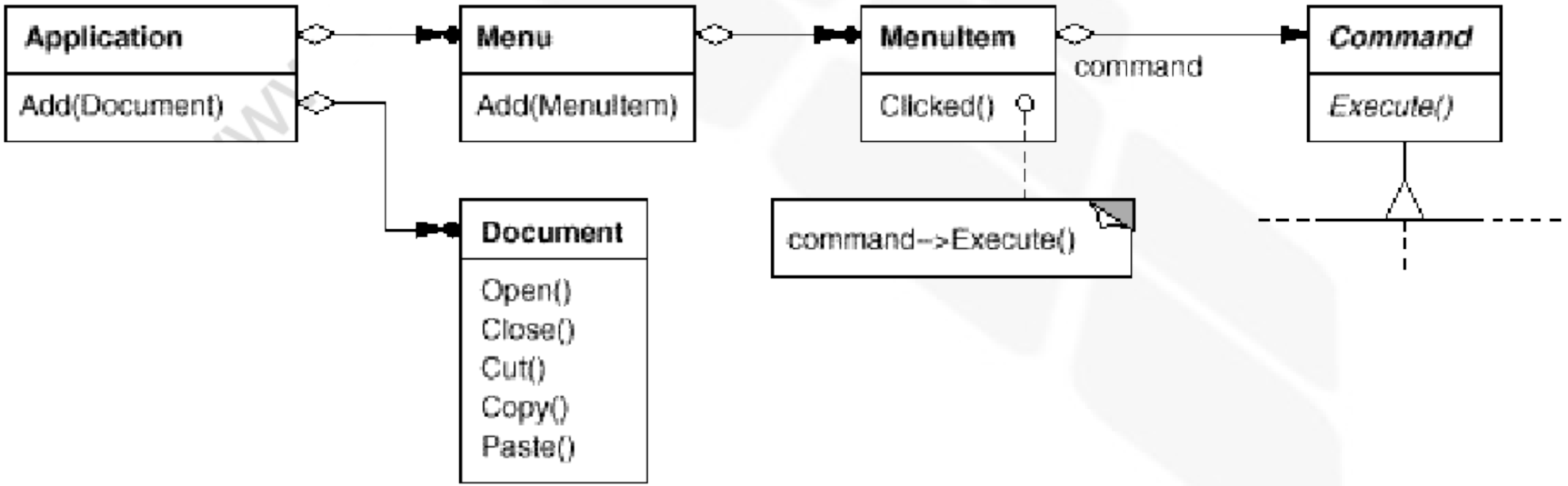
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Command

- Propósito:
  - Encapsular uma requisição sob a forma de um objeto, permitindo parametrizar clientes com requisições, filas e *logs* diferentes, além de suportar *undo*
- Também conhecido como: *Action*, *Transaction*
- Motivação:
  - Necessidade de emissão de requisições a objetos sem conhecer nada a respeito da **operação** sendo requerida
  - Exemplo: botões e menus em um *toolkit* gráfico
  - Solução: transformar a própria requisição em um objeto

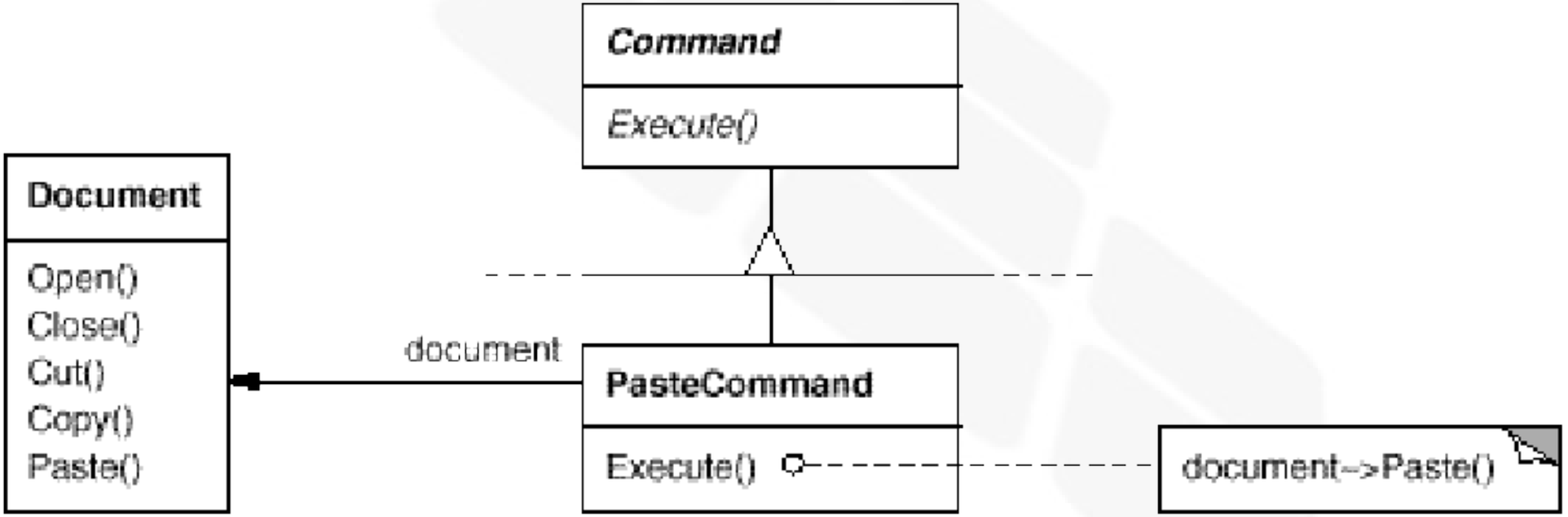
Command

- Motivação:
  - Sub-classes concretas de *Command* armazenam o receptor da requisição e invocam operações neste receptor



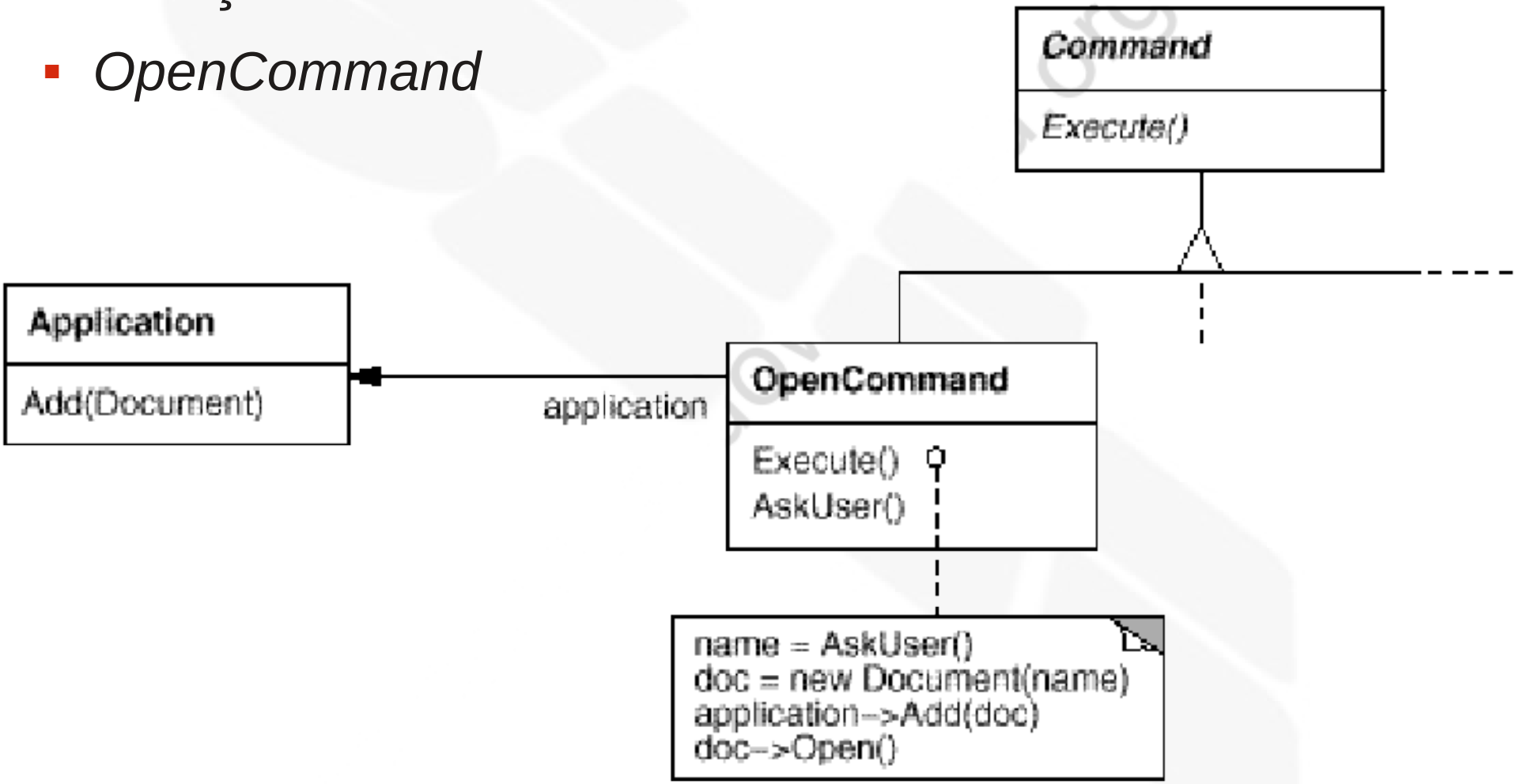
Command

- Motivação:
  - *PasteCommand*



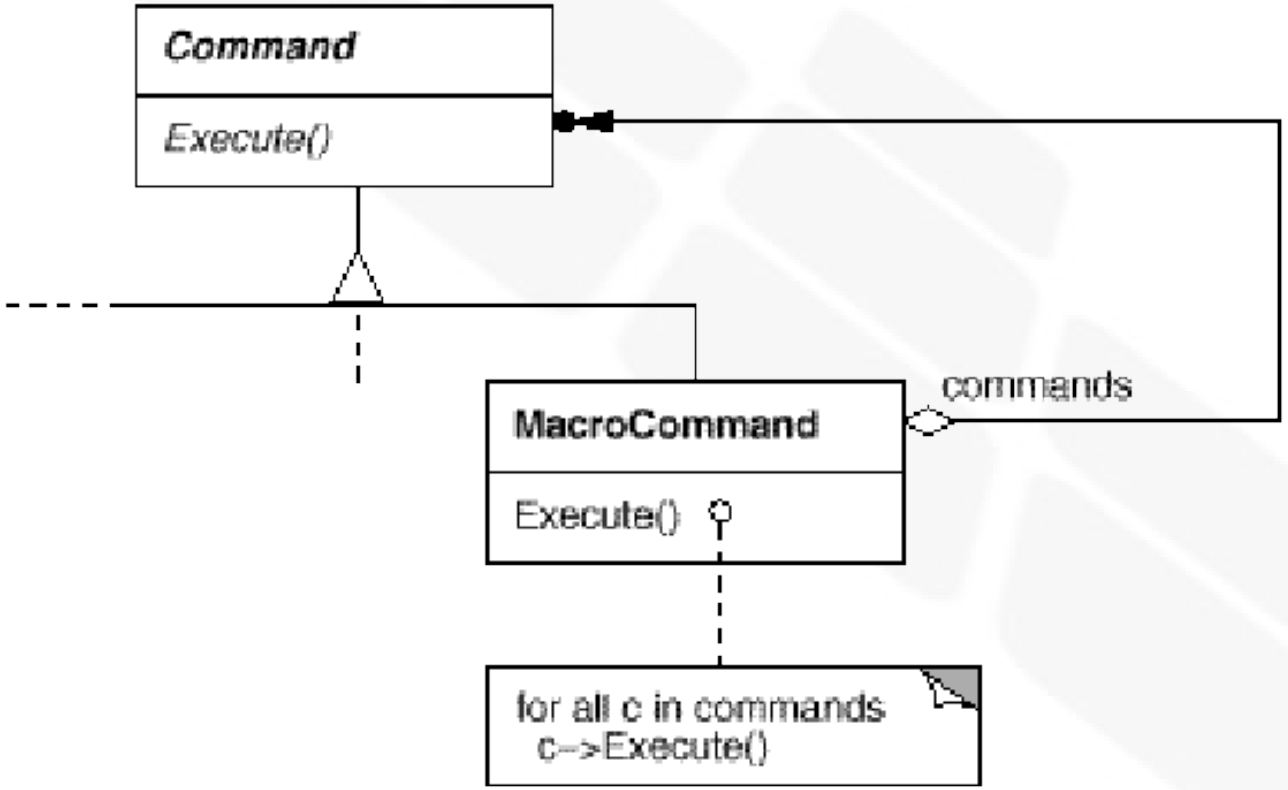
Command

- Motivação:
  - *OpenCommand*



Command

- Motivação:
  - *MacroCommand*



Command

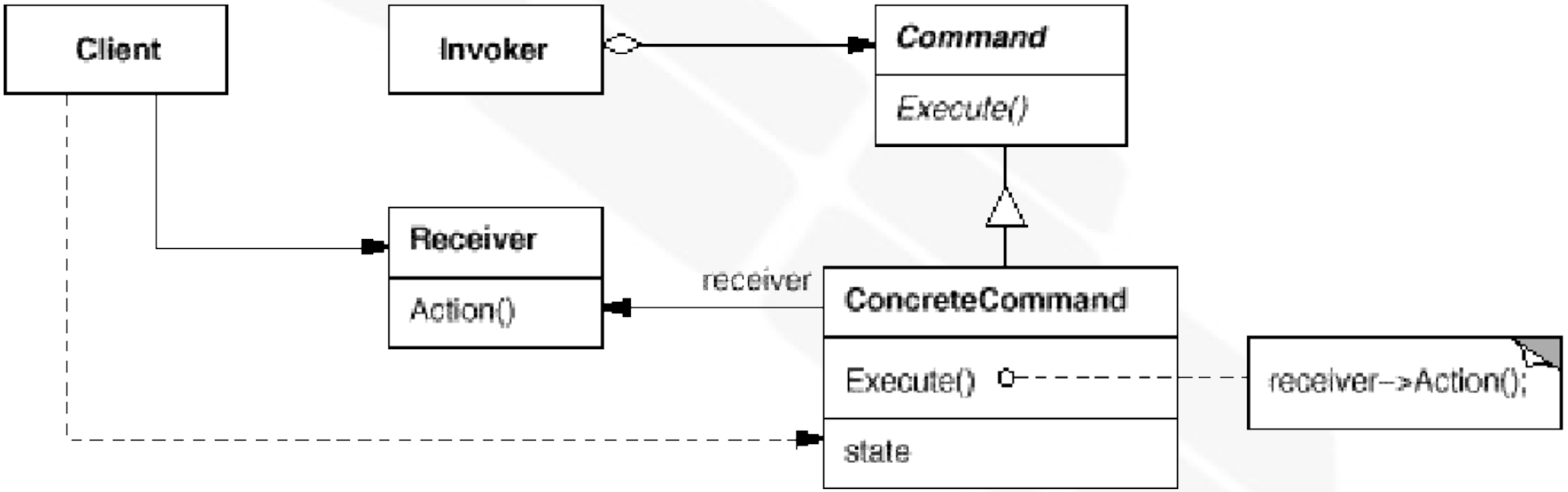
- Aplicabilidade:
  - Deseja-se parametrizar objetos com uma ação a ser realizada:
    - O *Command* é o substituto orientado a objetos para os *callbacks* procedurais
  - Para especificar, enfileirar e executar requisições em momentos diferentes:
    - O *lifetime* dos objetos *Command* é independente da requisição original
    - Se o receptor puder ser representado de uma forma independente de espaço de endereçamento, o *Command* pode ser transferido para ser executado em um diferente processo ou *host*

Command

- Aplicabilidade:
  - Deseja-se suporte a *undo*:
    - A operação *execute()* do *Command* pode armazenar estado necessário para reverter os efeitos do comando
    - A interface do *Command* é estendida com uma operação *unexecute()*
    - *Commands* já executados são armazenados em uma lista de histórico
    - *Undo* e *redo* ilimitado através da navegação para trás e para frente no histórico, executando as operações *unexecute()* e *execute()*, respectivamente
  - Deseja-se estruturar o sistema com operações de alto nível baseadas em operações primitivas - transações

Command

- Estrutura:



Command

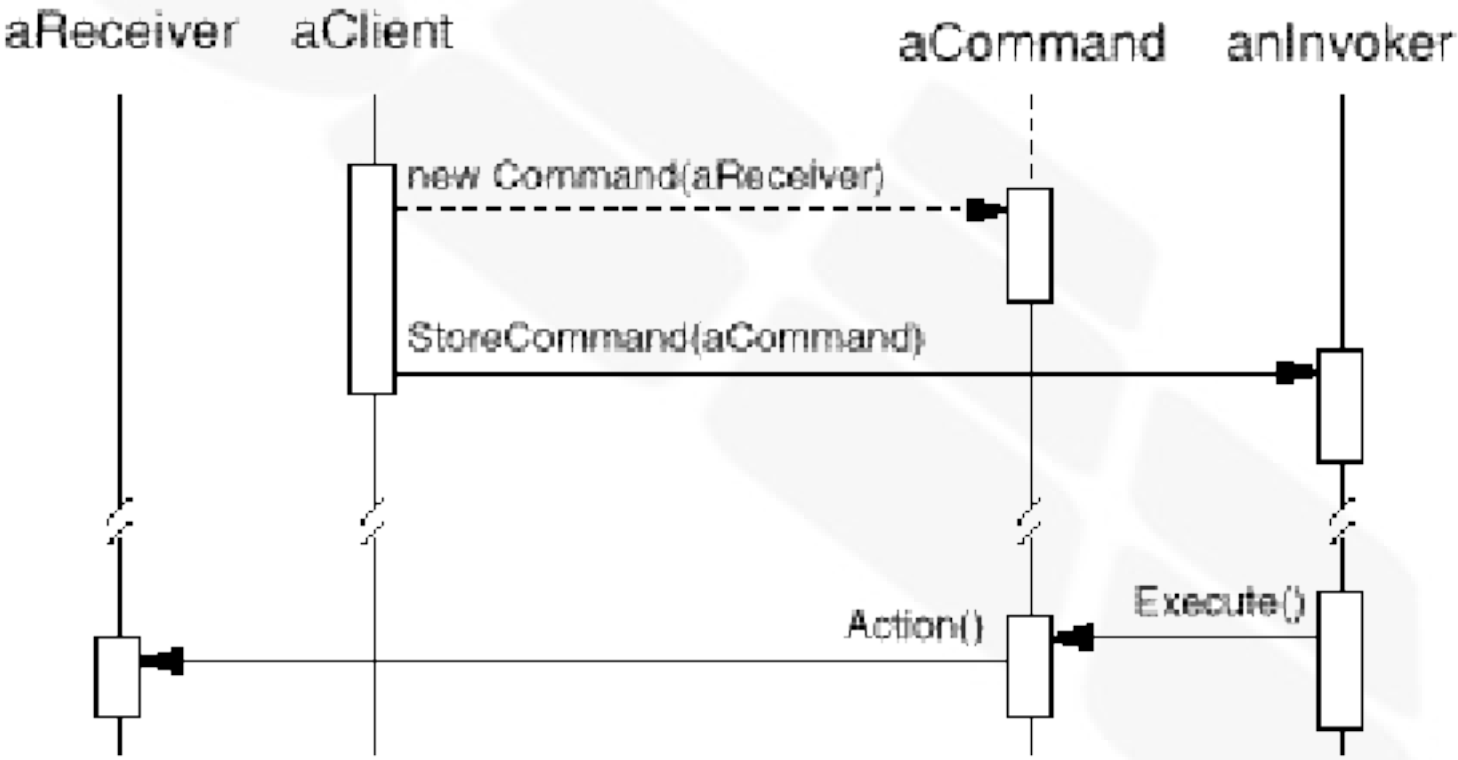
- Participantes:
  - *Command* (HelpHandler):
    - Declara a interface para execução da operação
  - *ConcreteCommand* (PasteCommand, OpenCommand):
    - Define a ligação entre o objeto receptor e a ação
    - Implementa *execute()* através de invocações no receptor
  - *Client* (Application):
    - Cria um objeto *ConcreteCommand* e ajusta seu receptor
  - *Invoker* (MenuItem):
    - Solicita ao *Command* a execução da requisição
  - *Receiver* (Document, Application):
    - Sabe como realizar as operações necessárias ao atendimento da requisição
    - Qualquer classe pode funcionar como um *Receiver*

Command

- Colaborações:
  - O cliente cria um *ConcreteCommand* e especifica seu receptor
  - Um *Invoker* armazena um objeto *ConcreteCommand*
  - O *Invoker* emite uma requisição invocando o método *execute()* no *ConcreteCommand*. Se o comando for *undoable* armazena-se o estado para desfazer o comando antes de propriamente executá-lo
  - O *ConcreteCommand* invoca operações no seu receptor para atender à requisição

Command

- Colaborações:



Command

- Consequências:
  - Desacopla o objeto que invoca a operação daquele que sabe como realizá-la
  - *Commands* são objetos de primeira-classe, podem ser manipulados e derivados como qualquer outro objeto
  - Pode-se combinar comandos em um *Composite Command*
  - Pode-se facilmente adicionar novos comandos sem modificar as classes já existentes

Command

- Implementação:
  - Quão inteligente um *Command* deve ser ?
    - Simples: realiza somente a ligação entre a ação e o receptor e armazena as operações a serem realizadas
    - Intermediário: descoberta dinâmica do receptor
    - Complexos: implementa tudo o que for necessário sem recorrer a um receptor
  - Suportando *undo* e *redo*:
    - Pode ser necessário armazenar estado adicional, por exemplo:
      - O objeto *Receiver*
      - Os argumentos da operação realizada pelo *Receiver*
      - Quaisquer valores do *Receiver* modificados pela requisição
    - Um nível x vários níveis (ambos podem requerer cópia do *Command*)

Command

- Código exemplo:

```
class Command {
public:
    virtual ~Command();

    virtual void Execute() = 0;
protected:
    Command();
};
```

```
class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}
```

Command

- Código exemplo:

```
class PasteCommand : public Command {
public:
    PasteCommand(Document*);

    virtual void Execute();
private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}
```

Command

- Código exemplo:

```
template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }

    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};
```

```
template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->*_action)();
}
```

```
MyClass* receiver = new MyClass;
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```

Command

- Código exemplo:

```
class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();
    virtual void Add(Command*);
    virtual void Remove(Command*);

    virtual void Execute();
private:
    List<Command*>* _cmds;
};
```

```
void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}
```

```
void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}
```

Command

- Usos conhecidos:
  - MacApp, ET++, KDE (*Plasma Services*), Struts/XWork
  - THINK

Command

- Padrões relacionados:
  - O *Composite* pode ser utilizado para *MacroCommands*
  - O *Memento* pode armazenar o estado que o comando necessita para desfazer a sua operação

INF011 – Padrões de Projeto

17 – *Command*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas