Engenharia de Software III - Professor Ugo Henrique

Nome: Bruno Gondim dos Santos SP 3039064

Padrões Gof

A origem dos Design Patterns vem do trabalho de um arquiteto chamado Christopher Alexander, no final da década de 70.

Ele escreveu dois livros: A Pattern Language,

A Timeless Way of Building

Nos quais ele exemplificava o uso e descrevia seu raciocínio para documentar os padrões,neles ele estabelece que um padrão deve ter as seguintes características:

Encapsulamento,

Generalidade,

Equilíbrio,

Abstração,

Abertura e Combinatoriedade

O catálogo de padrões do GoF basicamente está dividido em três seções:

Padrão de Criação

Relacionados à criação de classes e objetos. Ligados ao processo de instanciação.

Padrão Estrutural

Tratam da alteração da estrutura de um programa, e das associações entre classes e objetos.

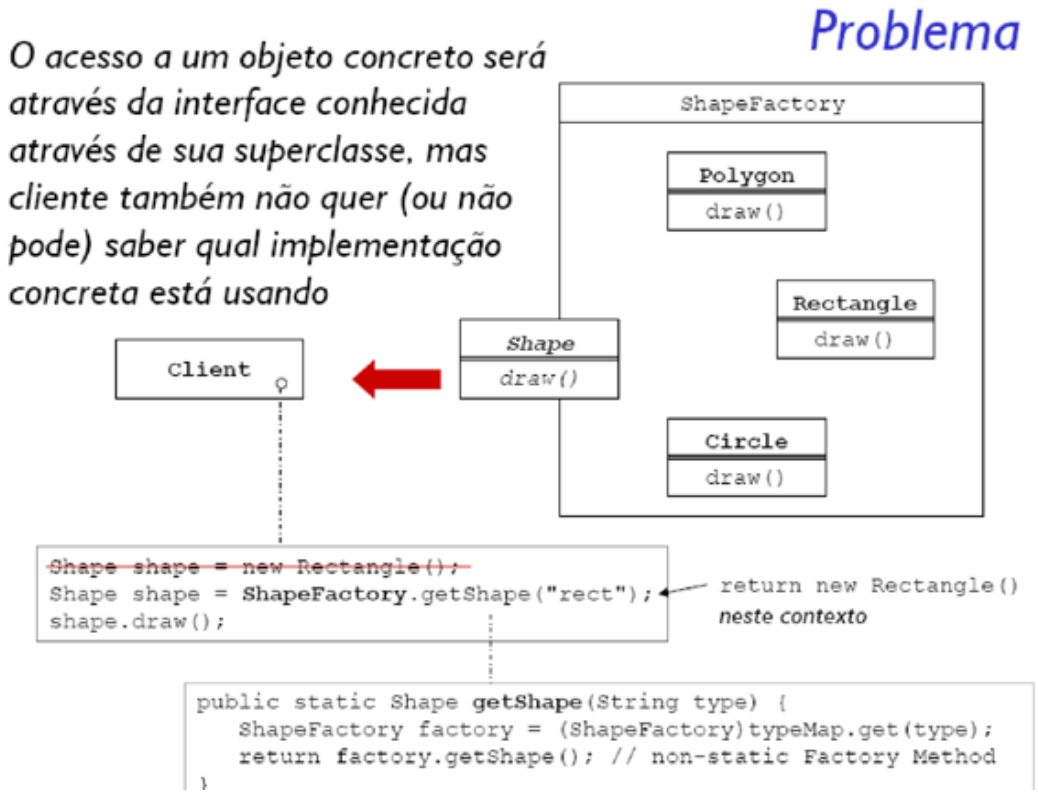
Padrão Comportamental

Observam a maneira com que classes e objetos podem interagir.

padrões de criação,

Factory: Permite a criação de famílias de objetos relacionados ou dependentes, através de uma única interface e sem que a classe concreta seja especificada.

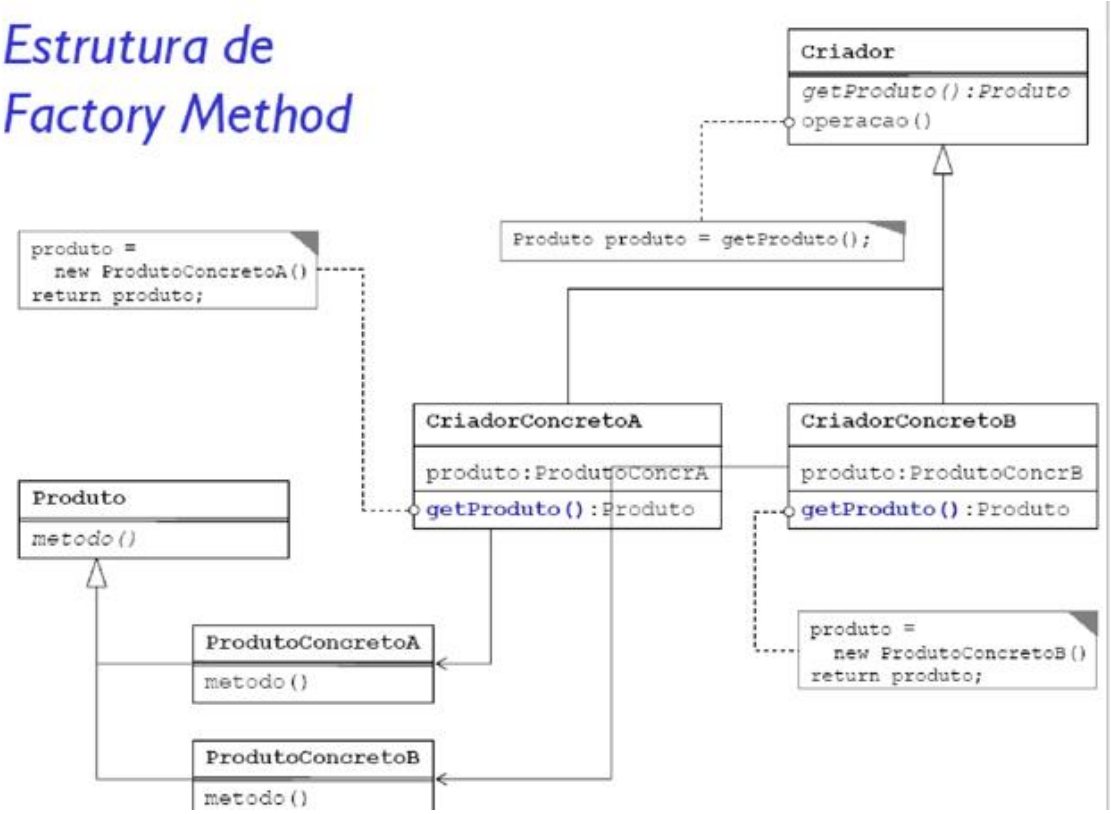
Permite que uma classe delegue a responsabilidade de instanciamento às subclasses.



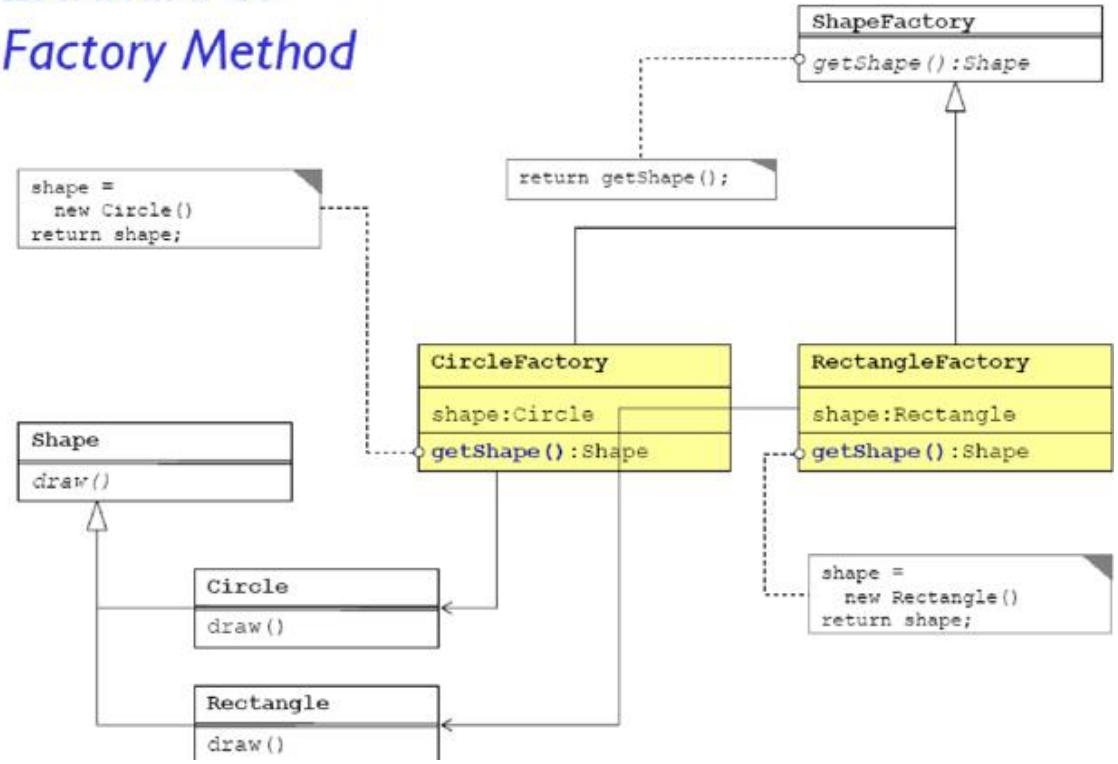
Como implementar?

- É possível criar um objeto sem ter conhecimento algum de sua classe concreta?
  - Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente
  - FactoryMethod define uma interface comum para criar objetos
  - O objeto específico é determinado nas diferentes implementações dessa interface
  - O cliente do FactoryMethod precisa saber sobre implementações concretas do objeto criador do produto desejado

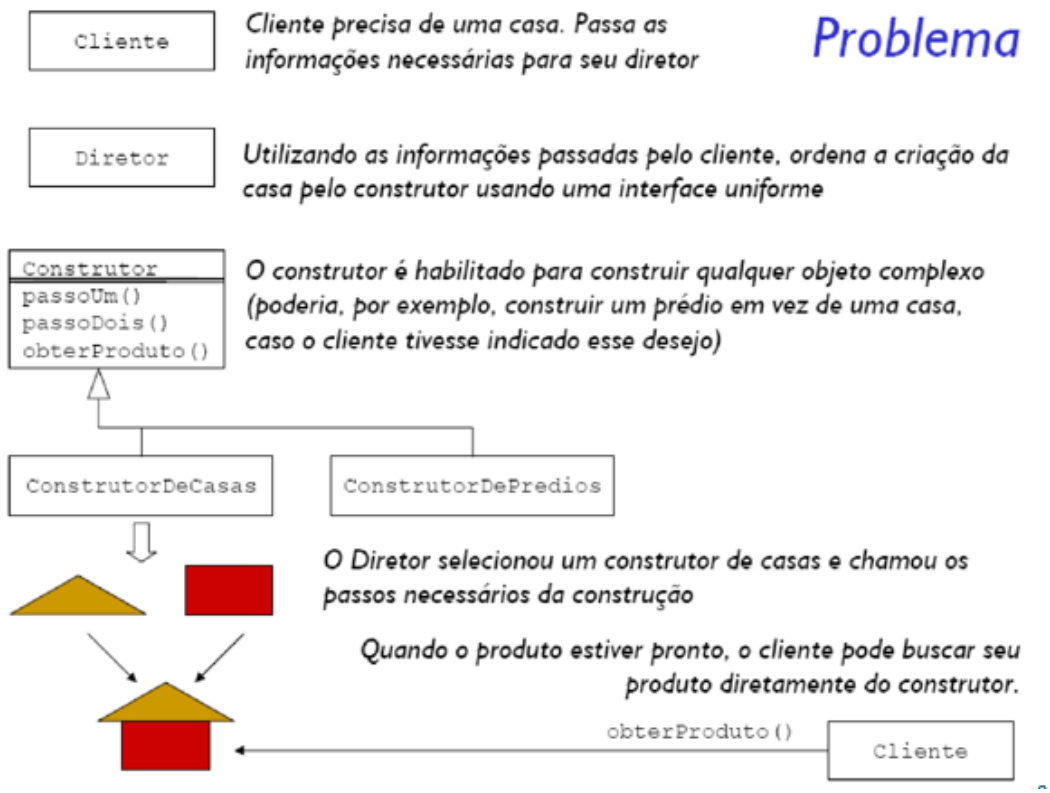
Estrutura de Factory Method

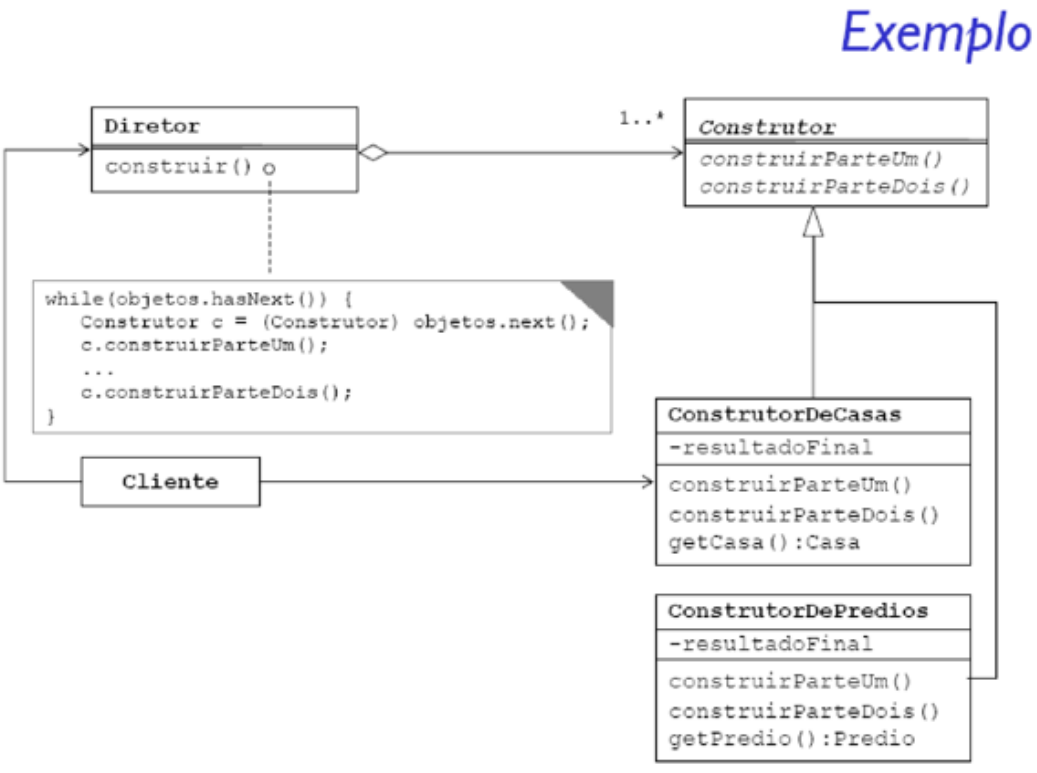


Estrutura de Factory Method

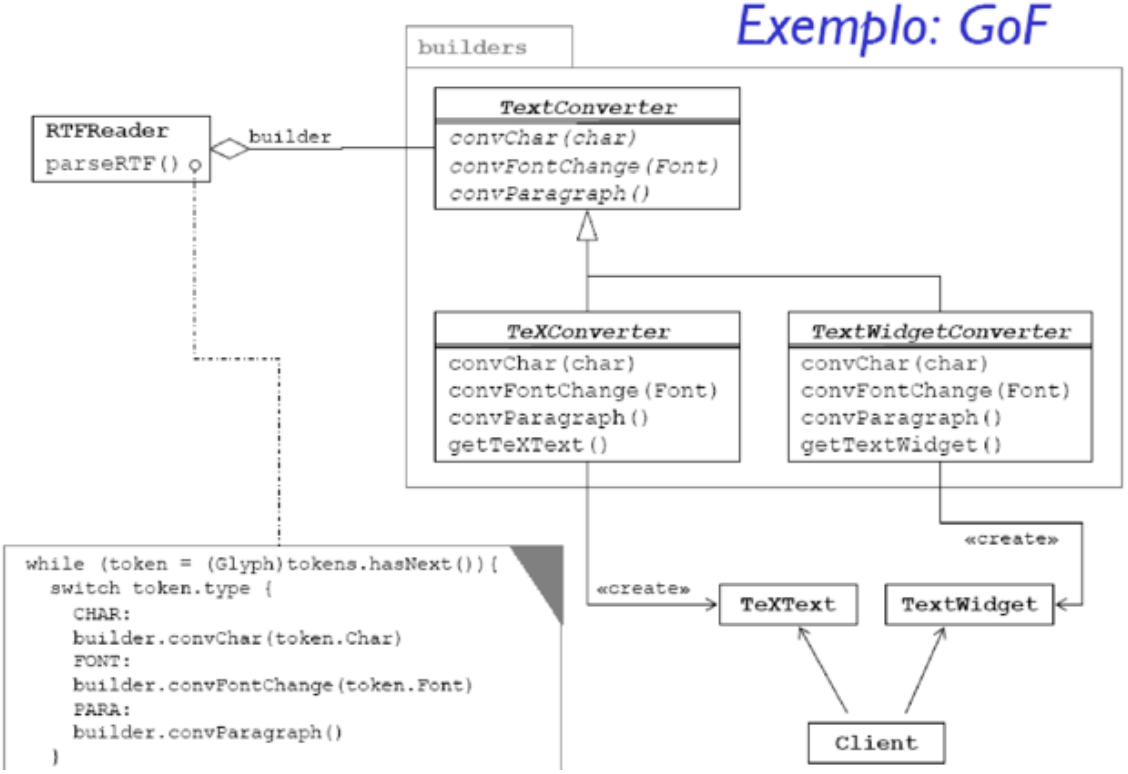


Builder: Permite a separação da construção de um objeto complexo da sua representação.

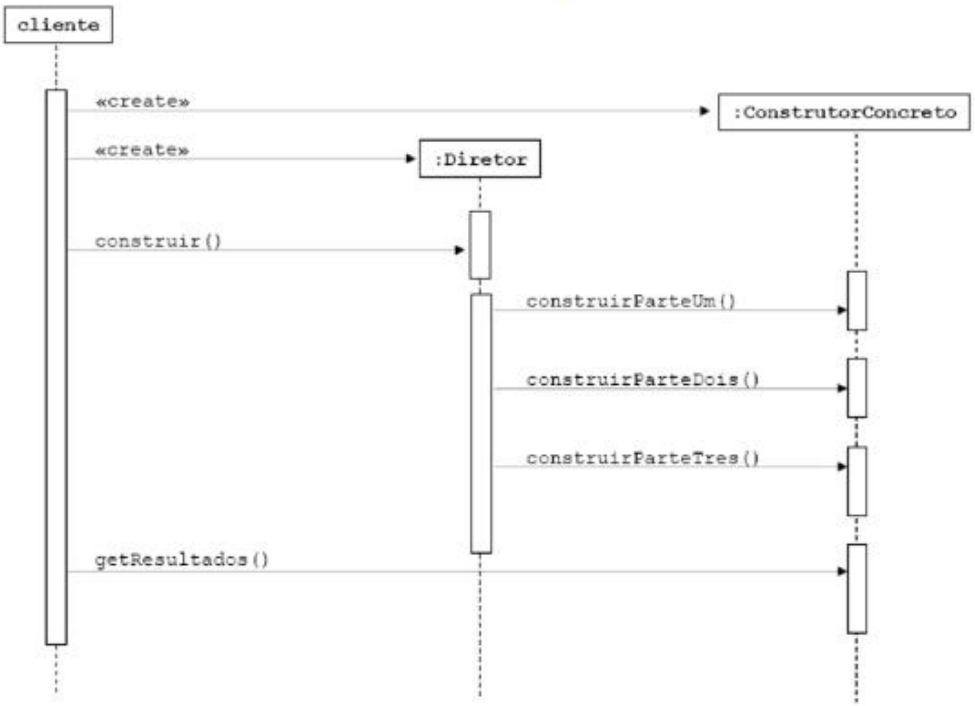




Exemplo: GoF



Seqüência de Builder



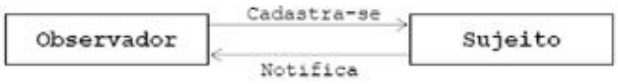
Quando usar?

- Builder permite que uma classe se preocupe com apenas uma parte da construção de um objeto. É útil em algoritmos de construção complexos
  - Use-o quando o algoritmo para criar um objeto complexo precisar ser independente das partes que compõem o objeto e da forma como o objeto é construído
- Builder também suporta substituição dos construtores, permitindo que a mesma interface seja usada para construir representações diferentes dos mesmos dados
  - Use quando o processo de construção precisar suportar representações diferentes do objeto que está sendo construído

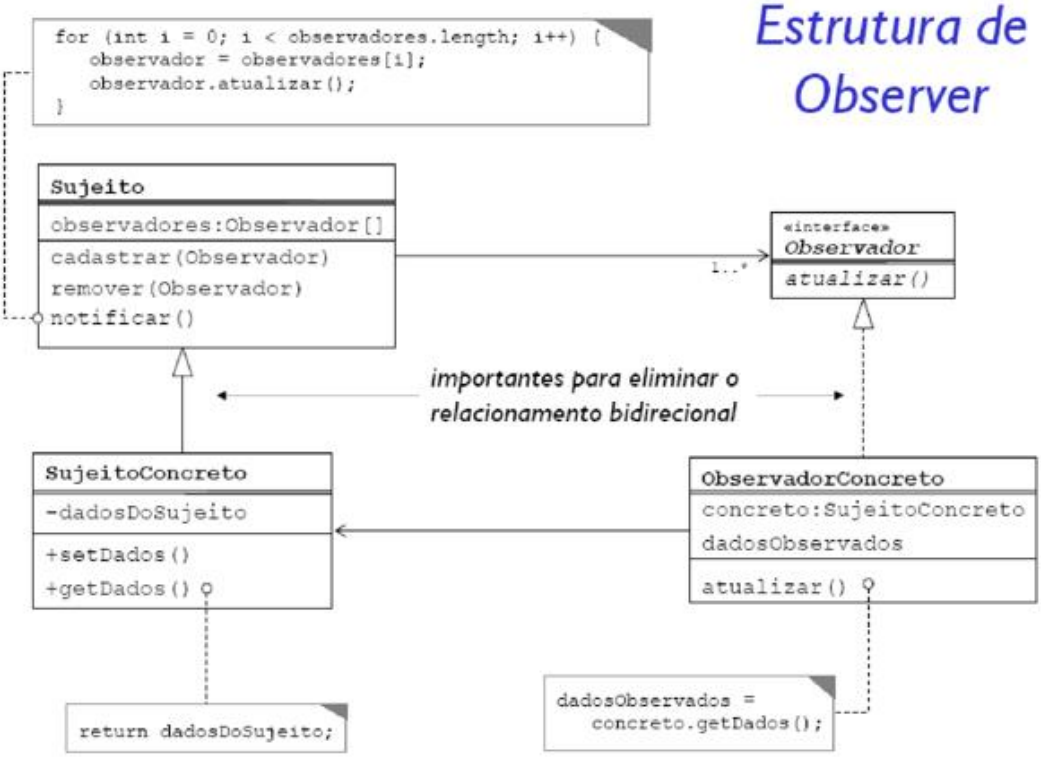
padrões comportamentais;

Observer : Define uma relação de dependência 1:N de forma que quando um certo objeto (assunto) tem seu estado modificado os demais (observadores) são notificados.

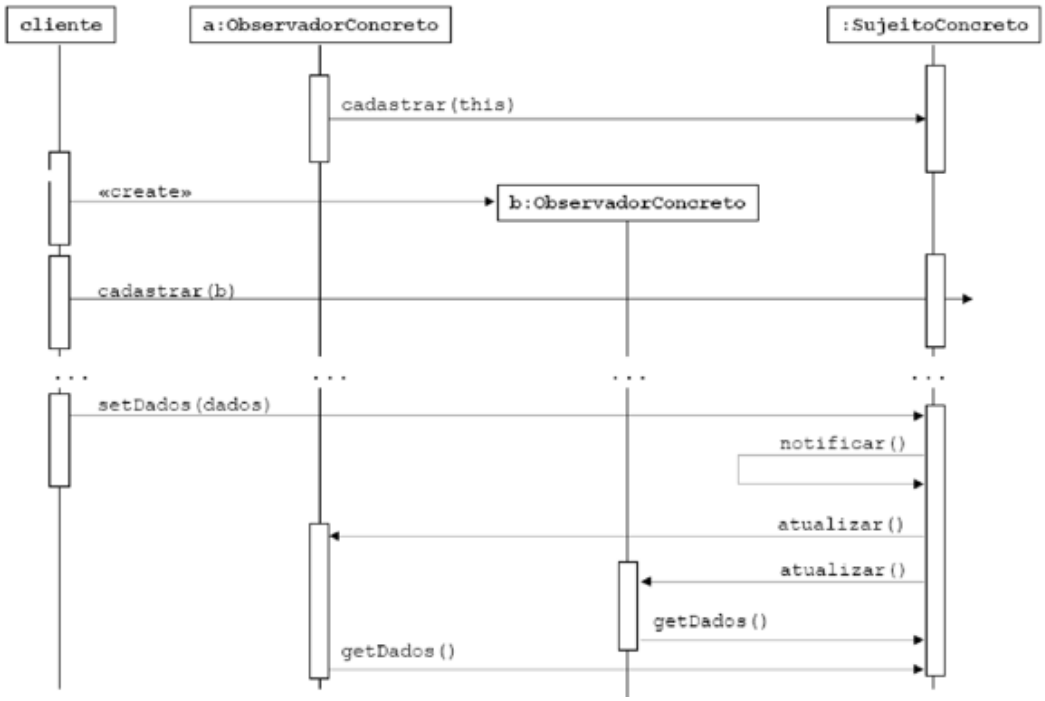
- *Como garantir que objetos que dependem de outro objeto fiquem em dia com mudanças naquele objeto?*
  - *Como fazer com que os observadores tomem conhecimento do objeto de interesse?*
  - *Como fazer com que o objeto de interesse atualize os observadores quando seu estado mudar?*
- *Possíveis riscos*
  - *Relacionamento (bidirecional) implica alto acoplamento. Como podemos eliminar o relacionamento bidirecional?*



Estrutura de Observer



Seqüência de Observer



Observer em Java

```
public class ConcreteObserver
             implements Observer {

  public void update(Observable o) {
    ObservableData data = (ObservableData) o;
    data.getData();
  }
}
```

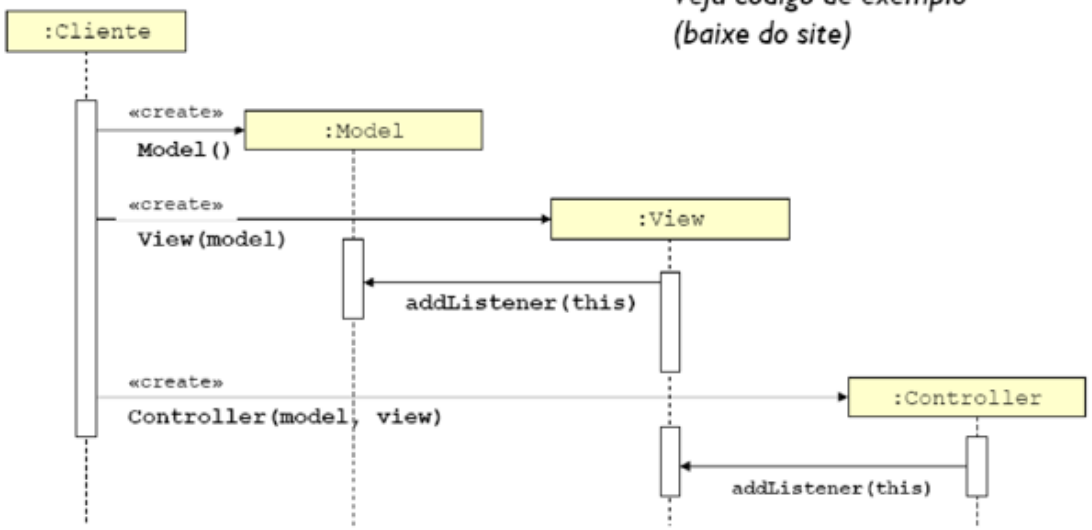
```
public class Observable {
  List observers = new ArrayList();

  public void add(Observer o) {
    observers.add(o);
  }

  public void remove(Observer o) {
    observers.remove(o);
  }

  public void notify() {
    Iterator it = observers.iterator();
    while(it.hasNext()) {
      Observer o = (Observer)it.next();
      o.update(this);
    }
  }
}
```

```
public class ObservableData
             extends Observable {
  private Object myData;

  public void setData(Object myData) {
    this.myData = myData;
    notify();
  }

  public Object getData() {
    return myData();
  }
}
```

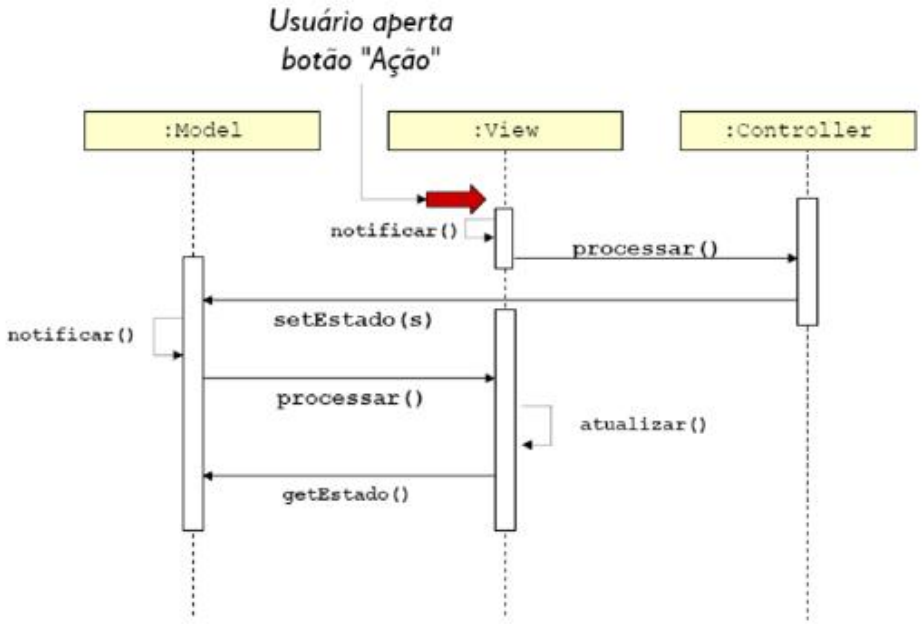
```
public interface Observer {
  public void update(Observable o);
}
```

Seqüência de registro das ligações

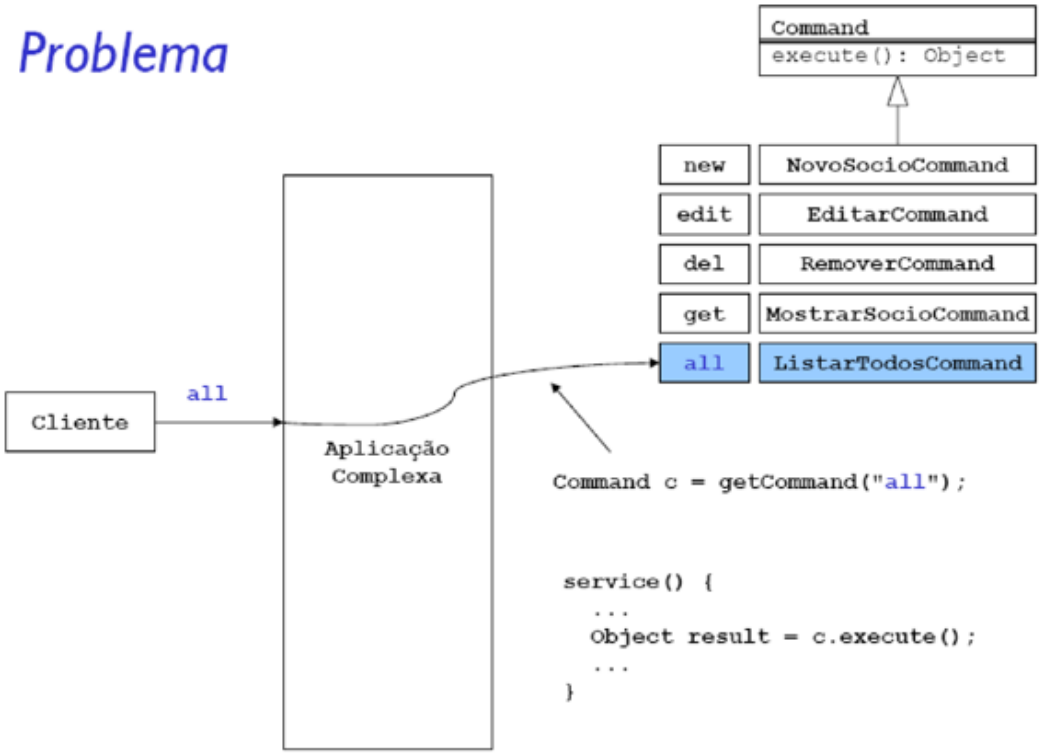
Veja código de exemplo (baixe do site)

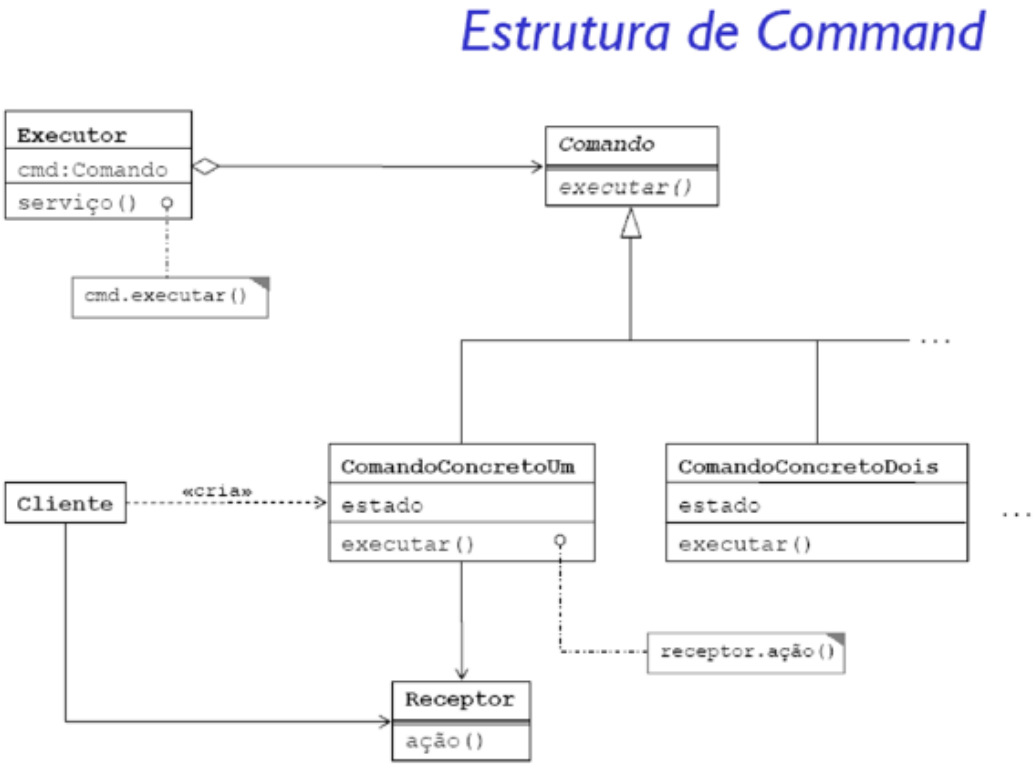


Seqüência de operação



Command : Associa uma ação a diferentes objetos através de uma interface conhecida.





Command em Java

```
public interface Command {
   public Object execute(Object arg);
}
```

```
public class Server {
  private Database db = ...;
  private HashMap cmds = new HashMap();

  public Server() {
    initCommands();
  }

  private void initCommands() {
    cmds.put("new", new NewCommand(db));
    cmds.put("del",
                  new DeleteCommand(db));
    ...
  }

  public void service(String cmd,
                      Object data) {
    ...
    Command c = (Command)cmds.get(cmd);
    ...
    Object result = c.execute(data);
    ...
  }
}
```

```
public interface NewCommand implements Command {

  public NewCommand(Database db) {
    this.db = db;
  }

  public Object execute(Object arg) {
    Data d = (Data)arg;
    int id = d.getArg(0);
    String nome = d.getArg(1);
    db.insert(new Member(id, nome));
  }
}
```

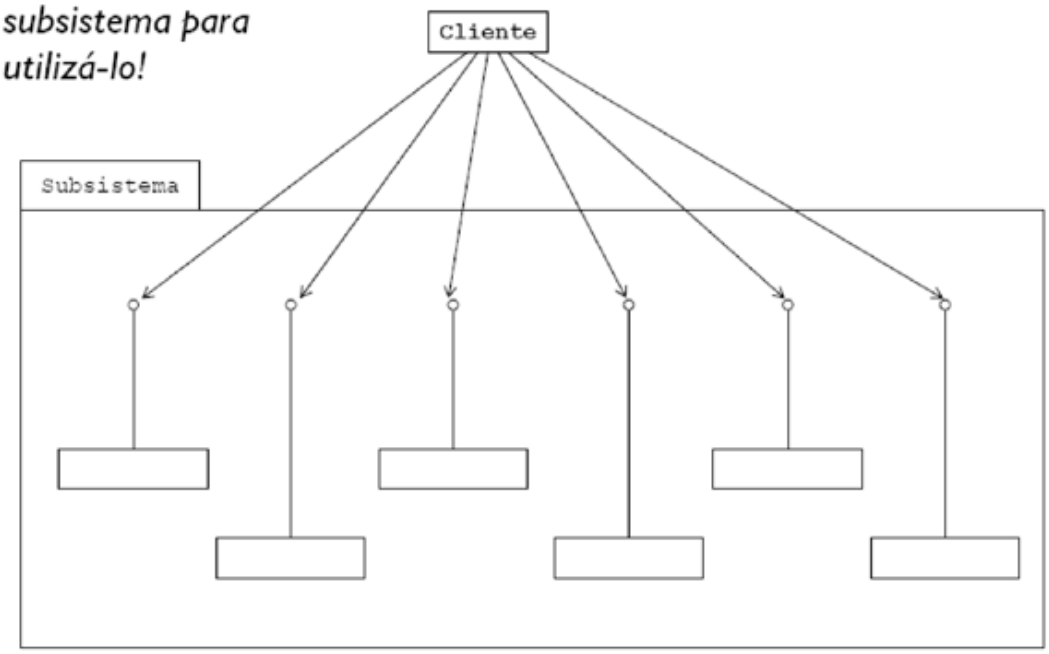
```
public class DeleteCommand implements Command {

  public DeleteCommand(Database db) {
    this.db = db;
  }

  public Object execute(Object arg) {
    Data d = (Data)arg;
    int id = d.getArg(0);
    db.delete(id);
  }
}
```

Padrões Estruturais

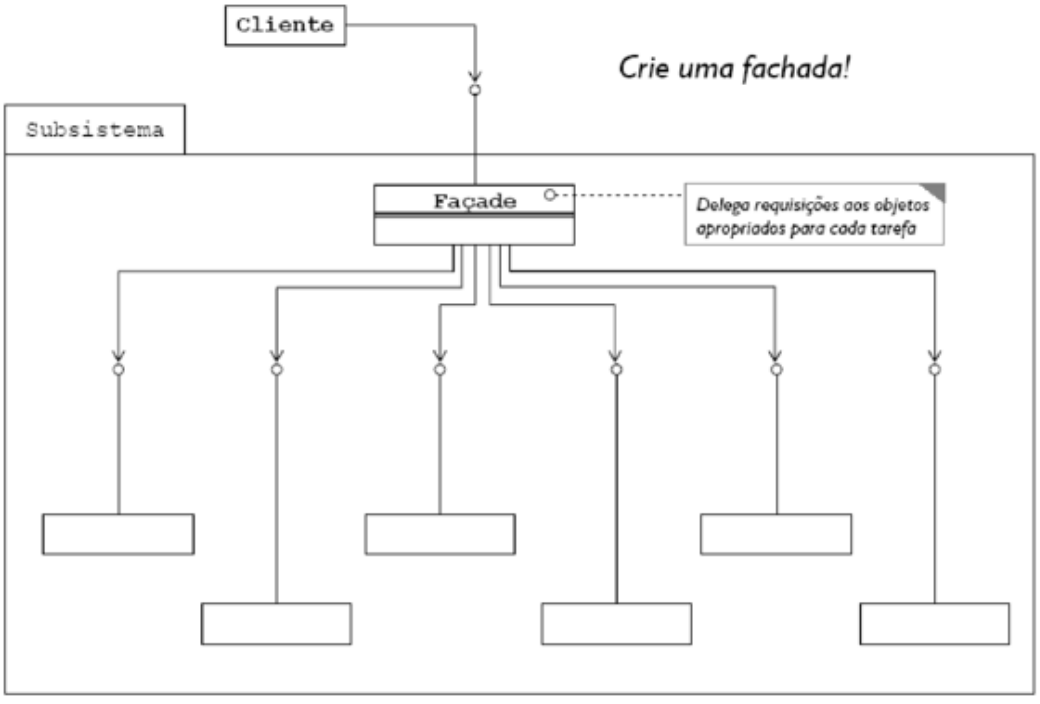
Facade : Interface unificada para um subsistema. Torna o subsistema mais fácil de usar.

Cliente precisa saber muitos detalhes do subsistema para utilizá-lo!

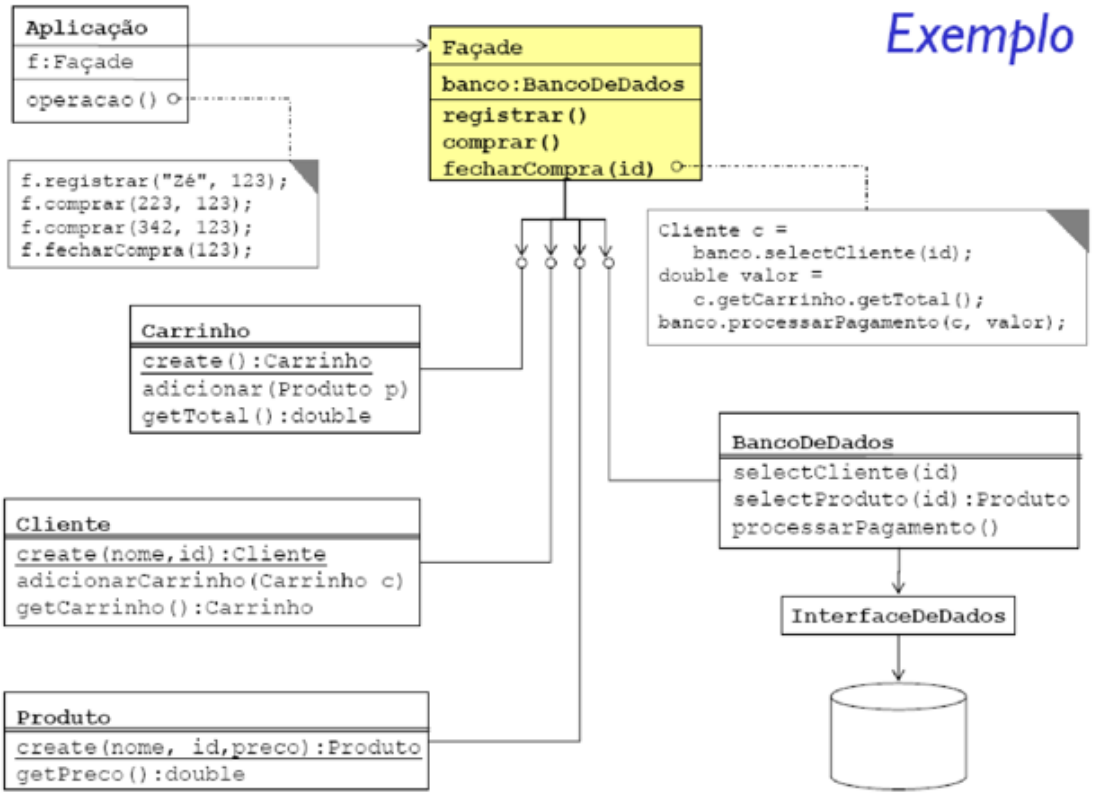
Problema



Estrutura de Façade



Exemplo



Façade em Java

```
class Aplicação {
    ...
    Facade f;
    // Obtem instancia f
    f.registrar("Zé", 123);
    f.comprar(223, 123);
    f.comprar(342, 123);
    f.fecharCompra(123);
    ...
}
```

```
public class Facade {
  BancoDeDados banco = Sistema.obterBanco();
  public void registrar(String nome, int id) {
     Cliente c = Cliente.create(nome, id);
     Carrinho c = Carrinho.create();
     c.adicionarCarrinho();
  }
  public void comprar(int prodID, int clienteID) {
     Cliente c = banco.selectCliente(cliente ID);
     Produto p = banco.selectProduto(prodID) {
     c.getCarrinho().adicionar(p);
   }
   public void fecharCompra(int clienteID) {
      Cliente c = banco.selectCliente(clienteID);
      double valor = c.getCarrinho.getTotal();
      banco.processarPagamento(c, valor);
   }
}
```

```
public class Carrinho {
  static Carrinho create() {...}
  void adicionar(Produto p) {...}
  double getTotal() {...}
}
```

```
public class Produto {
  static Produto create(String nome,
                        int id,double preco) {...}
  double getPreco() {...}
}
```

```
public class Cliente {
 static Cliente create(String nome,
                       int id) {...}
 void adicionarCarrinho(Carrinho c) {...}
 Carrinho getCarrinho() {...}
}
```

```
public class BancoDeDados {
 Cliente selectCliente(int id) {...}
 Produto selectProduto(int id) {...}
 void processarPagamento() {...}
}
```

Quando usar?

- Sempre que for desejável criar uma interface para um conjunto de objetos com o objetivo de facilitar o uso da aplicação
  - Permite que objetos individuais cuidem de uma única tarefa, deixando que a fachada se encarregue de divulgar as suas operações
- Fachadas viabilizam a separação em camadas com alto grau de desacoplamento
- Existem em várias partes da aplicação
  - Fachada da aplicação para interface do usuário
  - Fachada para sistema de persistência: Data Access Object

Nível de acoplamento

- Fachadas podem oferecer maior ou menor isolamento entre aplicação cliente e objetos
  - Nível ideal deve ser determinado pelo nível de acoplamento desejado entre os sistemas
- A fachada mostrada como exemplo isola totalmente o cliente dos objetos

```
Facade f; // Obtem instancia f
f.registrar("Zé", 123);
```

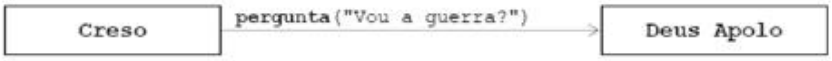
- Outra versão com menor isolamento (requer que aplicação-cliente conheça objeto Cliente)

```
Cliente joao = Cliente.create("João", 15);
f.registrar(joao); // método registrar(Cliente c)
```

Proxy : Fornece objeto representante de outro objeto para controlar o acesso ao mesmo.

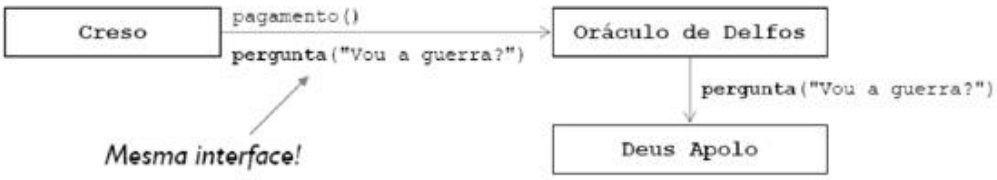
Problema

- *Sistema quer utilizar objeto real...*

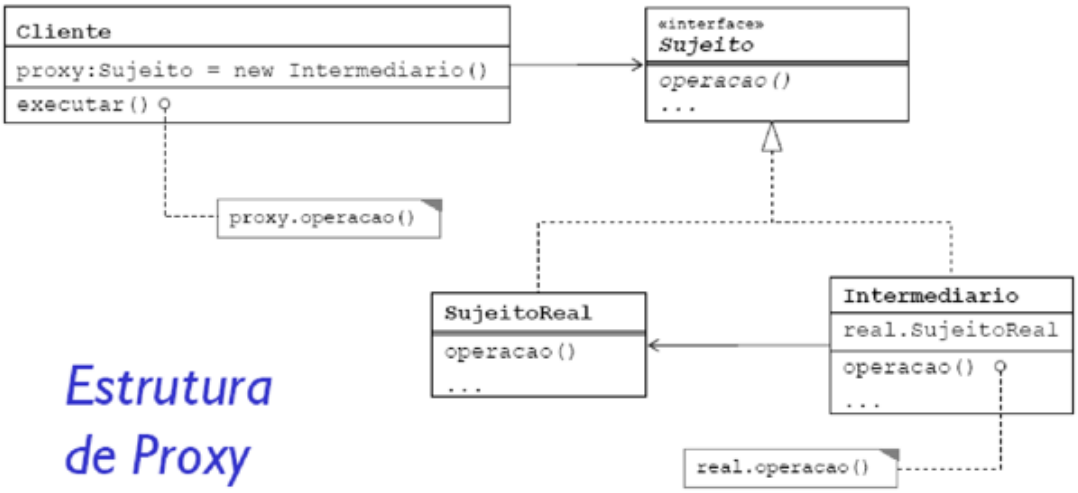


- *Mas ele não está disponível (remoto, inaccessível, ...)*

- *Solução: arranjar um **intermediário** que saiba se comunicar com ele eficientemente*



Estrutura de Proxy



- *Cliente usa intermediário em vez de sujeito real*
- *Intermediário suporta a mesma interface que sujeito real*
- *Intermediário contém uma referência para o sujeito real e repassa chamadas, possivelmente, acrescentando informações ou filtrando dados no processo*

Proxy em Java

```
public class Creso {
  ...
        Sujeito apolo = Fabrica.getSujeito();
        apolo.operacao();
  ...
}
```

```
public class SujeitoReal implements Sujeito {
  public Object operacao() {
    return coisaUtil;
  }
}
```

inaccessível pelo cliente

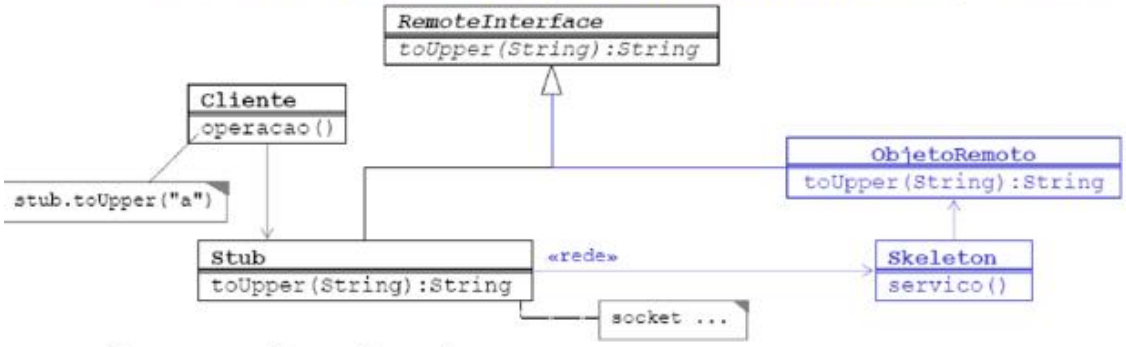
```
public class Intermediario implements Sujeito {
  private SujeitoReal real;
  public Object operacao() {
    cobraTaxa();
    return real.operacao();
  }
}
```

cliente comunica-se com este objeto

```
public interface Sujeito {
  public Object operacao();
}
```

Quando usar?

- *A aplicação mais comum é em objetos distribuídos*
- *Exemplo: RMI (e EJB)*
  - *O Stub é proxy do cliente para o objeto remoto*
  - *O Skeleton é parte do proxy: cliente remoto chamado pelo Stub*



- *Outras aplicações típicas*
  - *Image proxy: guarda o lugar de imagem sendo carregada*