# *Programação Orientada a Objetos II*

## Padrões de Projeto - c

Prof. Dr. Fábio Fagundes Silveira
fsilveira@unifesp.br
http://fabiosilveira.net

# Créditos

- Grande parte destes slides foram baseados:
  - no curso de Padrões de Projeto, preparado e ministrado por Helder Rocha, da empresa Argonavis; e
  - no livro: Design Patterns: Elements of Reusable Object-oriented Software - Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

# Introdução: Responsabilidades

- A decomposição em classes e objetos distribui responsabilidades em cada objeto e método do sistema
- Responsabilidades podem ser controladas em Java usando encapsulamento e controle de acesso, definindo
  - interfaces globais estáticas (public static)
  - interfaces para clientes de objetos (public)
  - interfaces para clientes de classes (protected)
  - interfaces internas ao desenvolvimento (package-private)
  - dados e operações internas à classe (private)
- Design patterns utilizam esses recursos para ir além e oferecer controle mais preciso
  - Controle distribuído em objetos específicos
  - Controle centralizado em um um único objeto
  - Notificação e intermediação de requisições

# Além das Responsabilidades comuns

- *Padrões oriendados a responsabilidades lidam com situações em que é preciso fugir da regra comum que a responsabilidade deve ser distribuída ao máximo*
  - *Singleton: centraliza a responsabilidade em uma única instância de uma classe*
  - *Observer: desacopla um objeto do conhecimento de que outros objetos dependem dele*
  - *Mediator: centraliza a responsabilidade em uma classe que determina como outros objetos interagem*
  - *Chain of Responsibility: permite que uma requisição passe por uma corrente de objetos até encontrar um que a processe*
  - *Flyweight: centraliza a responsabilidade em objetos compartilhados de alta granularidade*

5

# Singleton

"*Garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela.*" *[GoF]*

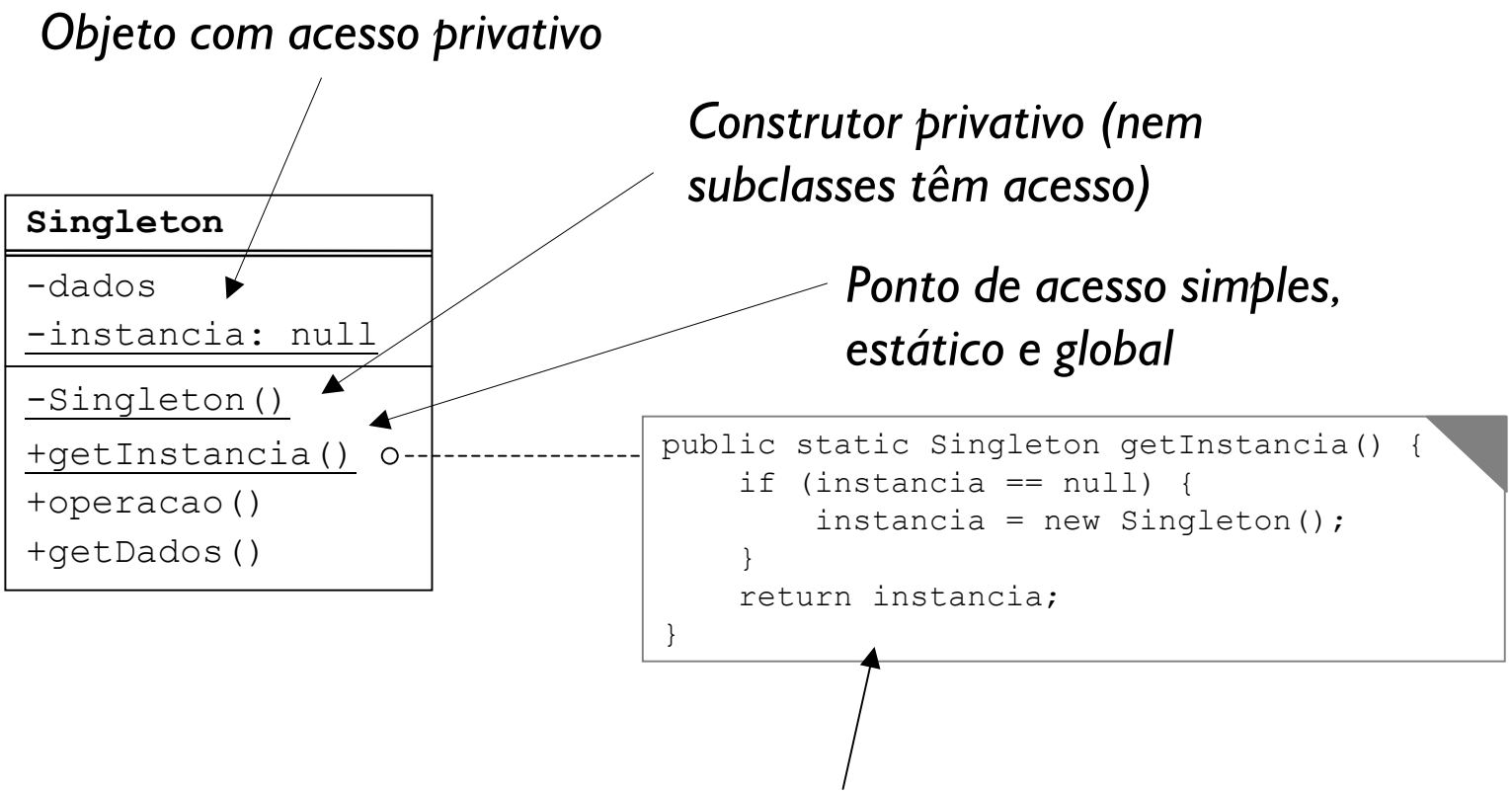
# Problema

- Garantir que apenas um objeto exista, independente do número de requisições que receber para criá-lo
- Aplicações
  - Um único banco de dados
  - Um único acesso ao arquivo de log
  - Um único objeto que representa um vídeo
  - Uma única fachada (Façade pattern)
- Poderia-se usar um membro estático ...
  - ... e perder o encapsulamento
  - ... e perder a flexibilidade de usar objetos

# *Problema (2)*

- *Objetivo: garantir que uma classe só tenha uma instância. Questões:*
  - *Como controlar (contar) o número de instâncias da classe?*
  - *Como armazenar a(s) instância(s)?*
  - *Como controlar ou impedir a construção normal? Se for possível usar new e um construtor para criar o objeto, há como limitar instâncias?*
  - *Como definir o acesso à um número limitado de instâncias (no caso, uma apenas)?*
  - *Como garantir que o sistema continuará funcionando se a classe participar de uma hierarquia de classes?*

# Estrutura de Singleton



# Singleton em Java

```
public class Highlander {
  private Highlander() {}
  private static Highlander instancia;
  public static Highlander obterInstancia() {
    if (instancia == null)
      instancia = new Highlander();
    }
    return instancia;
  }
}
```

*Esta classe implementa o design pattern Singleton*

```
public class Fabrica {
  public static void main(String[] args) {
    Highlander h1, h2, h3;
    //h1 = new Highlander(); // nao compila!
    h2 = Highlander.obterInstancia();
    h3 = Highlander.obterInstancia();
    if (h2 == h3) {
       System.out.println("h2 e h3 são mesmo objeto!");
    }
  }
}
```

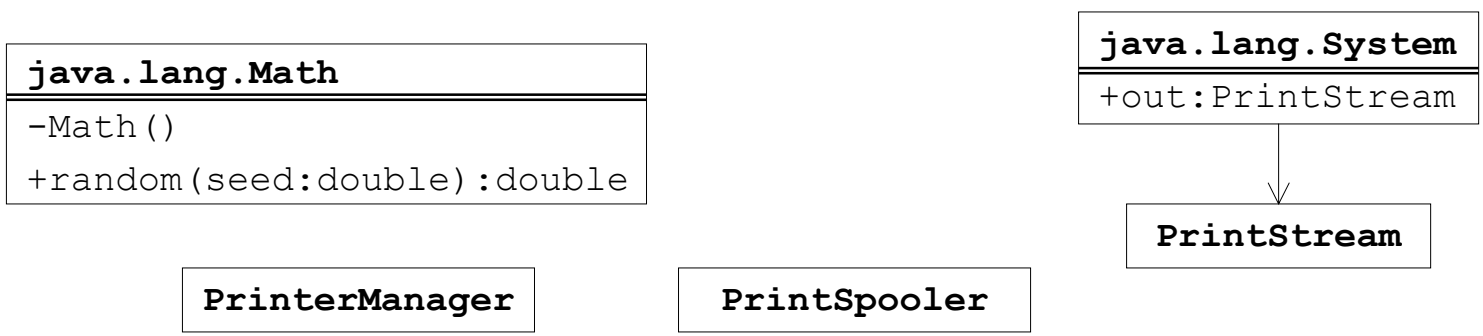
*Esta classe cria apenas um objeto Highlander*

# Prós e contras

- Vantagens
  - Acesso central e extensível a recursos e objetos
  - Pode ter subclasses (o que seria impossível se fosse apenas usada uma classe com métodos estáticos)
- Desvantagens
  - Difícil de testar (simulações dependem de instância extra)
  - Uso (abuso) como substituto para variáveis globais
  - Criação "preguiçosa" é complicada em ambiente multithreaded
  - Difícil ou impossível de implementar em ambiente distribuído (é preciso garantir que cópias serializadas refiram-se ao mesmo objeto)

# Exercícios

- *5.1 Transforme a Fachada que você criou no capítulo sobre Façade em um Singleton*
- *5.2 Escreva uma classe executável (com main) que obtenha três referências do objeto que você escreveu no exercício anterior e verifique que realmente se tratam da mesma referência.*
- *5.3 [Challenge 8.4] Para cada classe abaixo, diga se aparenta ser um Singleton e por que.*



# Resumo

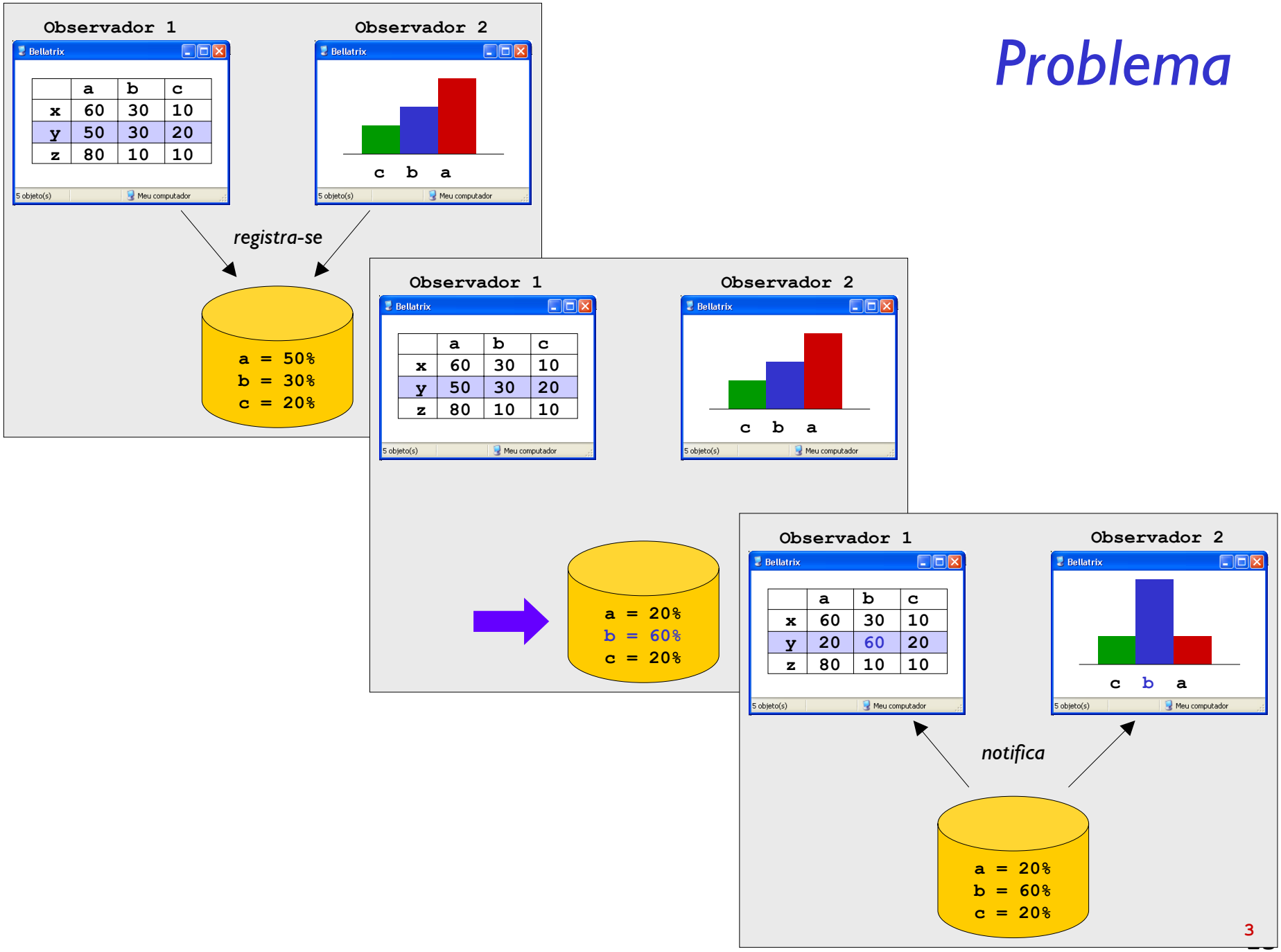
- *Singletons são uma forma de implementar uma responsabilidade centralizada*
  - *Garante que uma classe só tenha uma instância*
  - *Oferece um ponto de acesso global*
- *O instanciamento do objeto pode ser feito quando a classe for carregada ou quando o método de criação for chamado pela primeira vez*
  - *Neste caso, é preciso garantir que outros objetos não tentarão criar outro Singleton declarando o bloco crítico com synchronized.*

6

# Observer

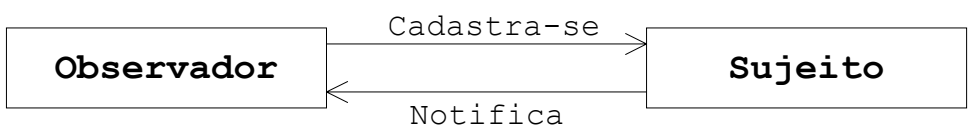
*"Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente." [GoF]*

# Problema

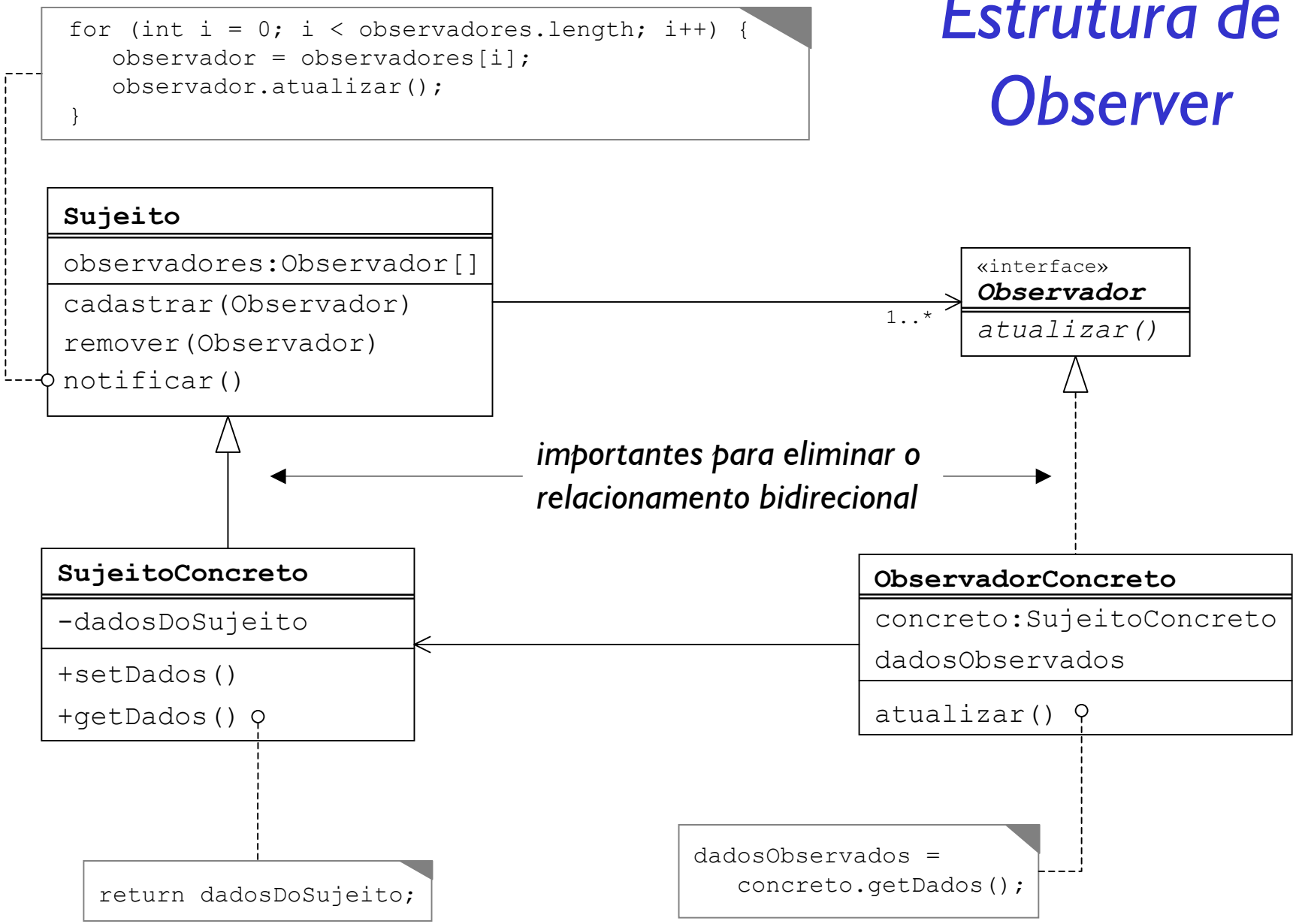

Observador 1
Observador 2
Bellatrix
a b c
x 60 30 10
y 50 30 20
z 80 10 10
c b a
5 objeto(s)
Meu computador
registra-se
a = 50%
b = 30%
c = 20%
a = 20%
b = 60%
c = 20%
notifica

# *Problema (2)*

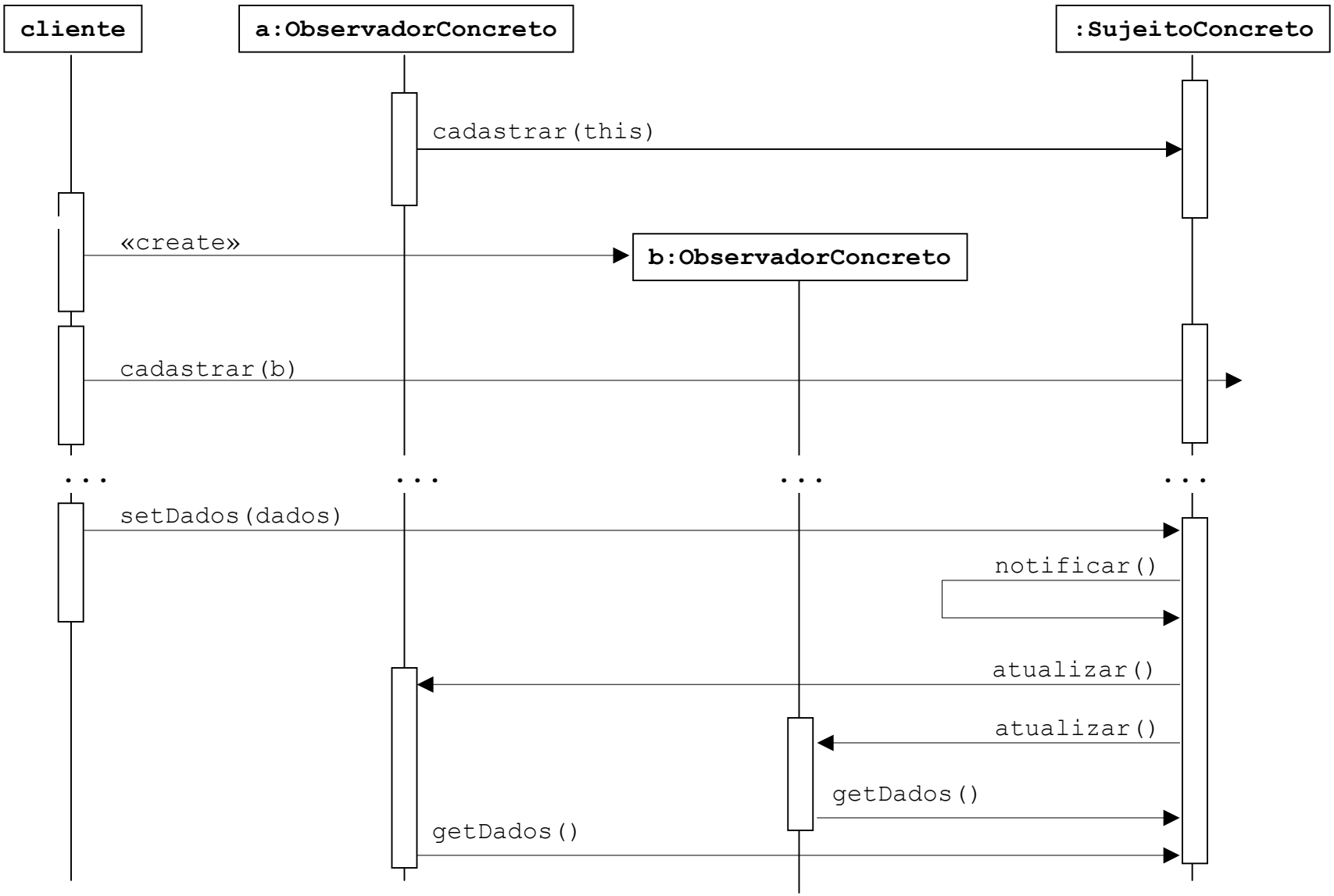
- *Como garantir que objetos que dependem de outro objeto fiquem em dia com mudanças naquele objeto?*
  - *Como fazer com que os observadores tomem conhecimento do objeto de interesse?*
  - *Como fazer com que o objeto de interesse atualize os observadores quando seu estado mudar?*
- *Possíveis riscos*
  - *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 {
  Observer 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);
}
```

# *Prós e contras*

- *Vantagens*
  - *Tanto observadores quando sujeitos observados podem ser reutilizados e ter sua interface e implementação alteradas sem afetar o sistema*
  - *O acoplamento forte implicado pelo relacionamento bidirecional é reduzido com o uso de interfaces e classes abstratas*
- *Desvantagens*
  - *O abuso pode causar sério impacto na performance. Sistemas onde todos notificam todos a cada mudança ficam inundados de requisições*

# *Exercícios*

- *6.1 Implemente uma aplicação simples usando Swing que abra duas janelas, cada uma contendo um botão e duas caixas de texto. Faça com que qualquer texto digitado em uma das janelas seja copiado para a outra ao apertar o botão.*
- *6.2 [Challenge 9.4]*

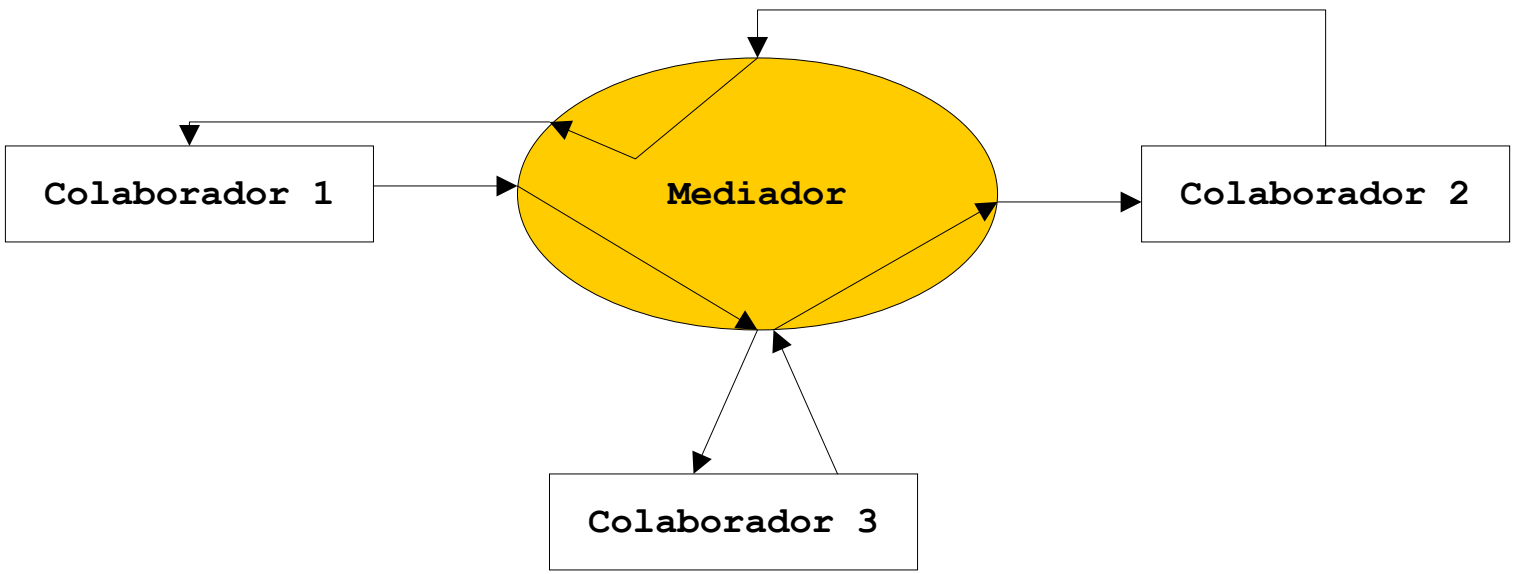
7

# Mediator

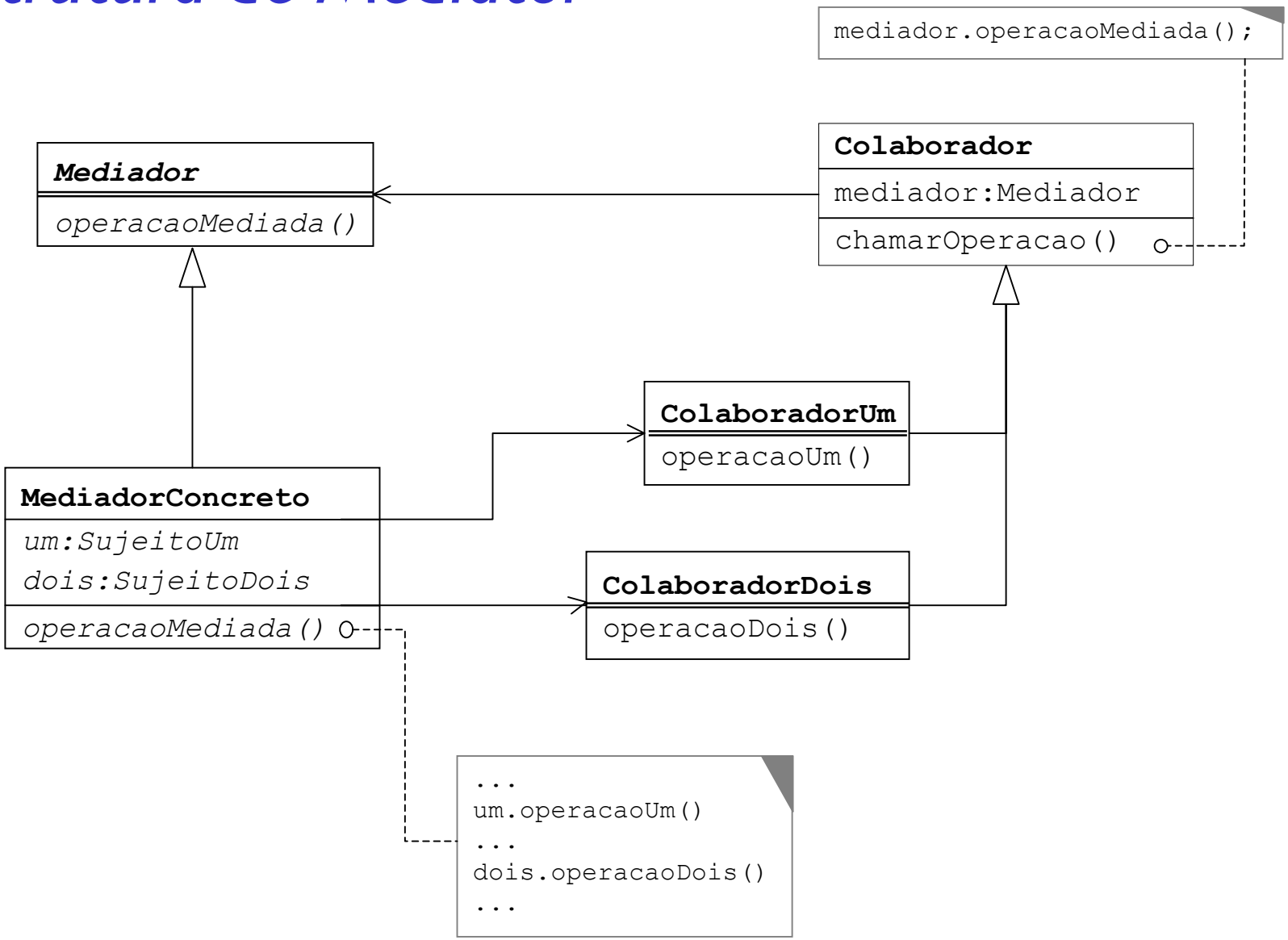
*"Definir um objeto que encapsula como um conjunto de objetos interagem. Mediator promove acoplamento fraco ao manter objetos que não se referem um ao outro explicitamente, permitindo variar sua interação independentemente." [GoF]*

# Problema

- *Como permitir que um grupo de objetos se comunique entre si sem que haja acoplamento entre eles?*



# Estrutura de Mediator



# Descrição da solução

- *Um objeto Mediador deve encapsular toda a comunicação entre um grupo de objetos*
  - *Cada objeto participante conhece o mediador mas ignora a existência dos outros objetos*
  - *O mediador conhece cada um dos objetos participantes*
- *A interface do Mediador é usada pelos colaboradores para iniciar a comunicação e receber notificações*
  - *O mediador recebe requisições dos remetentes*
  - *O mediador repassa as requisições aos destinatários*
  - *Toda a política de comunicação é determinada pelo mediador (geralmente através de uma implementação concreta do mediador)*

# Mediator em Java

```
public interface Mediator {
  public void chaseOperation();
  public void escapeOperation();
}
```

```
public class ConcreteMediator
             implements Mediator {

  Colleague dog = new Dog();
  Colleague hare = new Hare();

  public ConcreteMediator() {
    dog.setMediator(this);
    hare.setMediator(this);
  }
  public void chaseOperation() {
    ...
    hare.escapeDog();
  }
  public void escapeOperation() {
    ...
    dog.chaseHare();
  }
}
```

```
public abstract class Colleague {
  private Mediator mediator;

  public void setMediator(Mediator m) {
    mediator = m;
  }

}
```

```
public class Hare extends Colleague {

  public void escapeDog() {
    mediator.escapeOperation();
  }

}
```

```
public class Dog extends Colleague {

  public void chaseHare() {
    mediator.chaseOperation();
  }

}
```

# *Prós e contras*

- *Vantagens*
  - *Desacoplamento entre os diversos participantes da rede de comunicação*
  - *Eliminação de relacionamentos muitos para muitos (são todos substituídos por relacionamentos um para muitos)*
  - *A política de comunicações está centralizada no mediador e pode ser alterada sem mexer nos colaboradores.*
- *Desvantagens*
  - *A centralização pode ser uma fonte de gargalos de performance e de risco para o sistema em caso de falha*

# *Exercícios*

- *7.1 Implemente o exercício 6.1 usando um mediador*
- *7.2 [Challenge 10.4]*

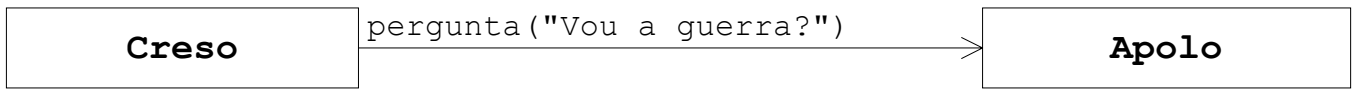
8

# Proxy

"*Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro.*" *[GoF]*

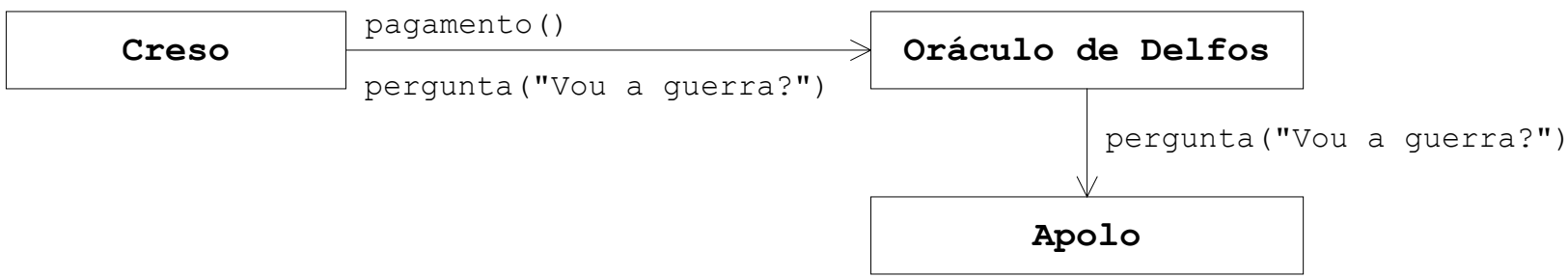
# Problema

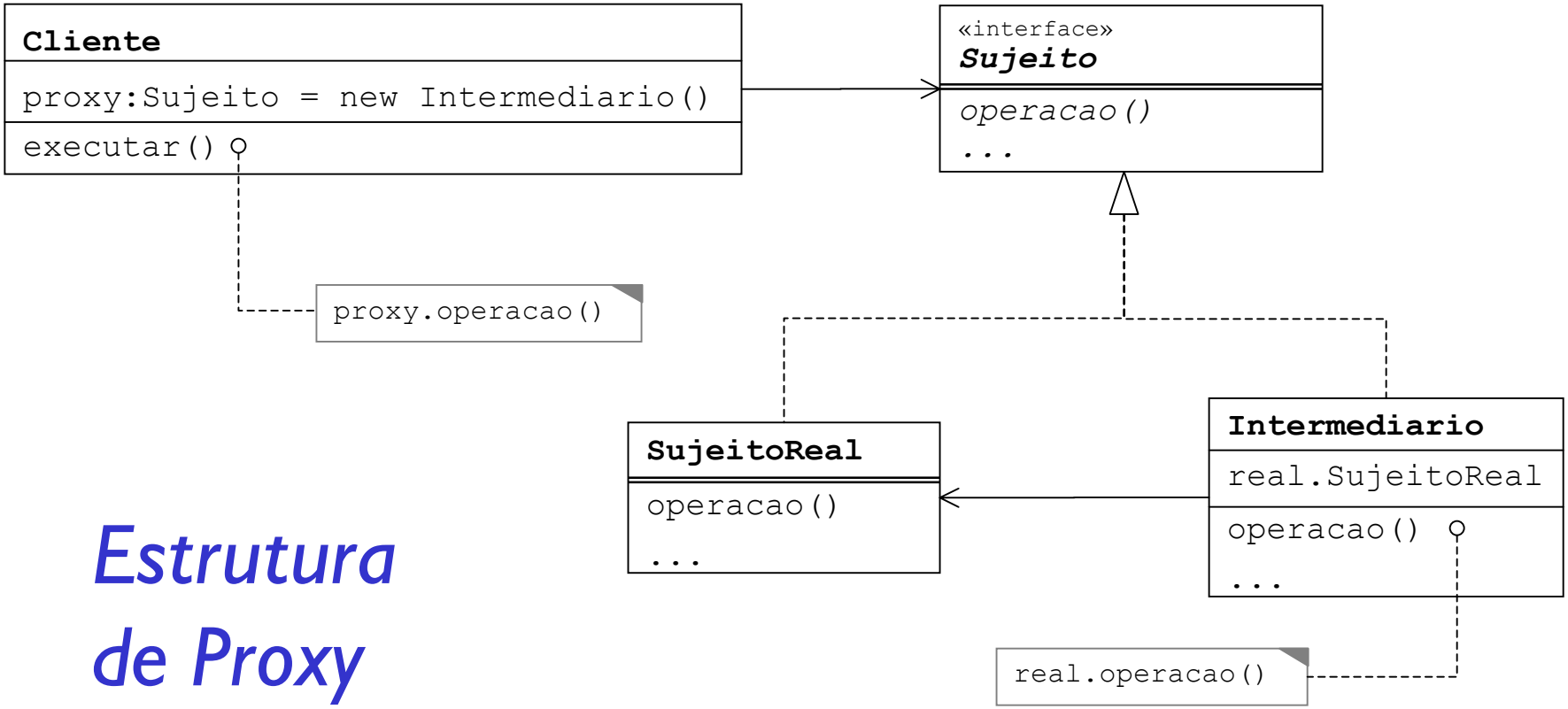
- *Sistema quer utilizar objeto real...*




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


- *Solução: arranjar um substituto 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 Cliente {
   ...
         Sujeito sujeito = Fabrica.getSujeito();
         sujeito.operacao();
   ...
}
```

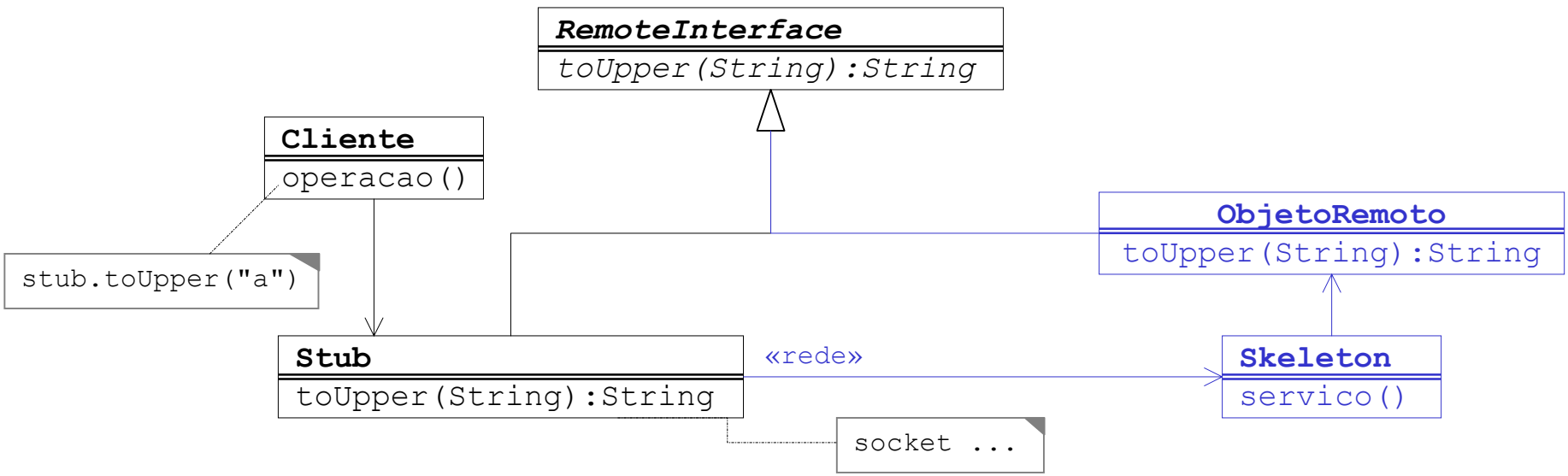
```
public class SujeitoReal implements Sujeito {
  public Object operacao() {
    return thing;
  }
}
```

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

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

# Quando usar?

- *A aplicação mais comum é em objetos distribuídos*
- *Exemplo: RMI*
  - *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*

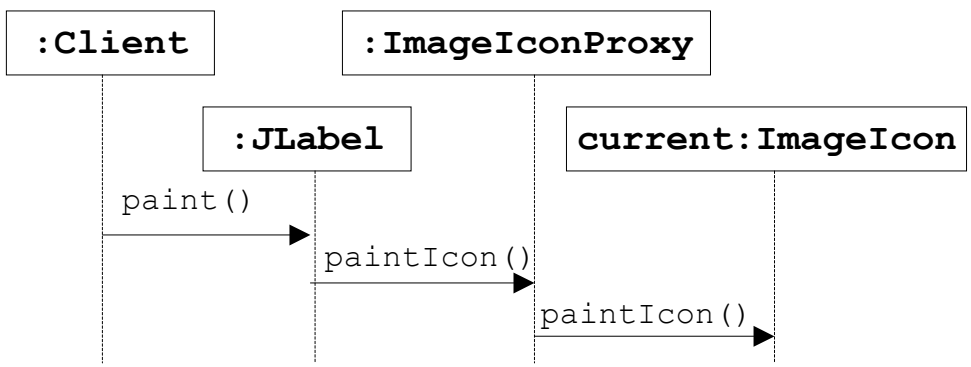
# *Prós e contras*

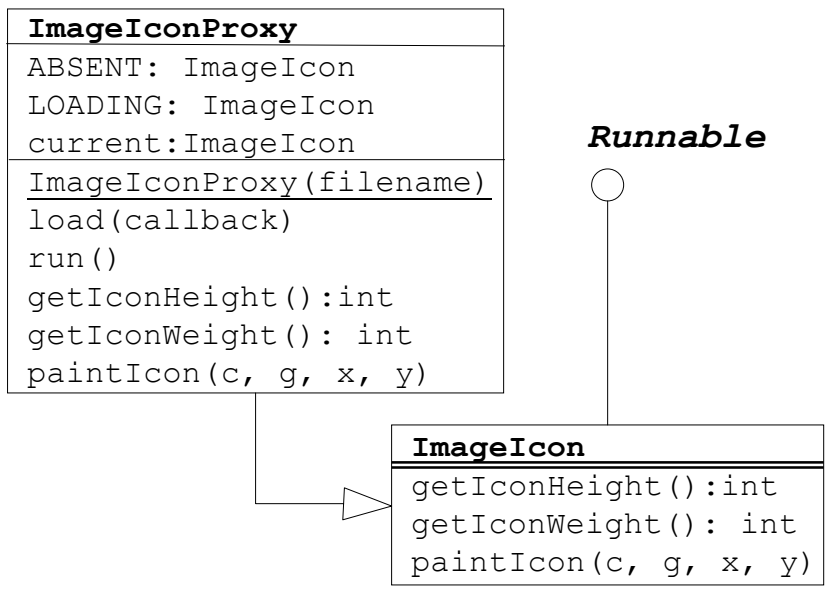
- *Vantagens*
  - *Transparência: mesma sintaxe usada na comunicação entre o cliente e sujeito real é usada no proxy*
  - *Permite o tratamento inteligente dos dados no cliente*
  - *Permite maior eficiência com caching no cliente*
- *Desvantagens*
  - *Possível impacto na performance*
  - *Transparência nem sempre é 100% (fatores externos como queda da rede podem tornar o proxy inoperante ou desatualizado)*

# Exercícios

```
public class ImageIconProxy extends ImageIcon implements Runnable {
  final static ImageIcon ABSENT = new ImageIcon("absent.jpg");
  final static ImageIcon LOADING = new ImageIcon("loading.jpg");
  ImageIcon current = ABSENT;
  protected String filename;
  protected JFrame callbackFrame;
  public ImageIconProxy(String filename) {
    super(ABSENT.getImage());
    this.filename = filename;
  }
  public int getIconHeight() {
    // Challenge!
  }
  public int getIconWidth() {
    // Challenge!
  }
  public void load(JFrame callbackFrame) {
    ...
  }
  public synchronized void paintIcon(
    Component c, Graphics g, int x, int y) {
    // Challenge!
  }
}
```

- *8.1 [Challenge 11.1] Um objeto ImageIconProxy aceita três chamadas de display que precisa passar à imagem atual. Escreva o código para os 3 métodos incompletos ao lado*



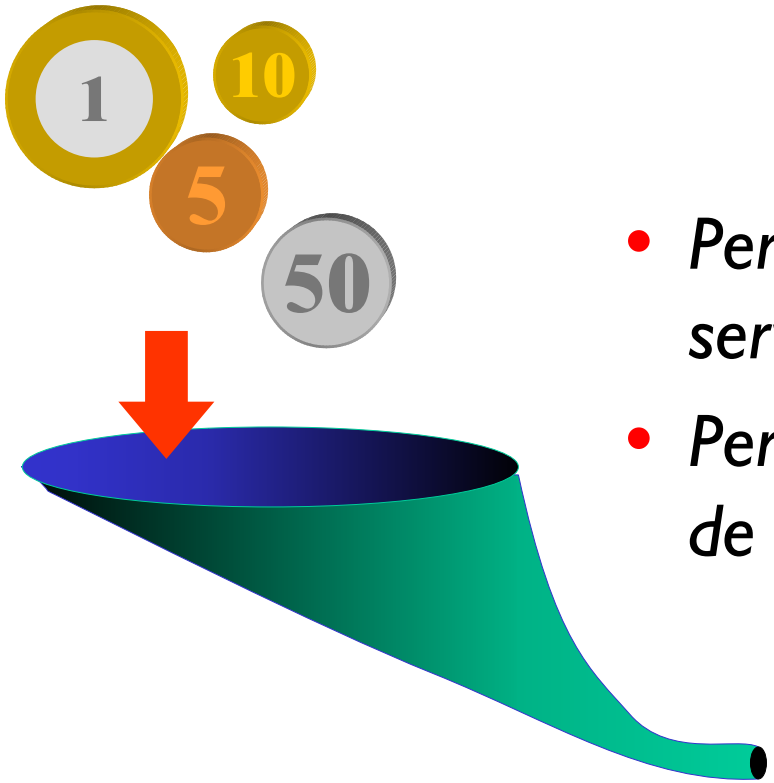



9

# Chain of Responsibility

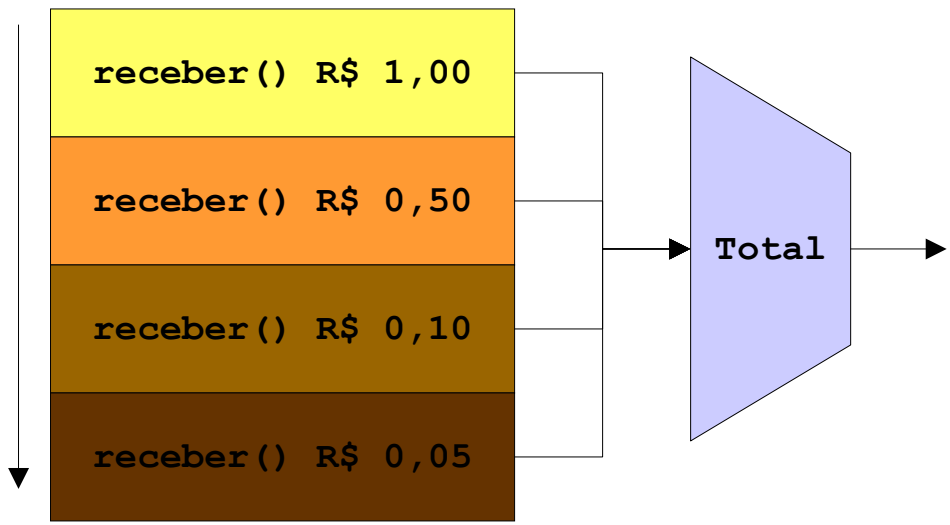
*"Evita acoplar o remetente de uma requisição ao seu destinatário ao dar a mais de um objeto a chance de servir a requisição. Compõe os objetos em cascata e passa a requisição pela corrente até que um objeto a sirva." [GoF]*

# Problema

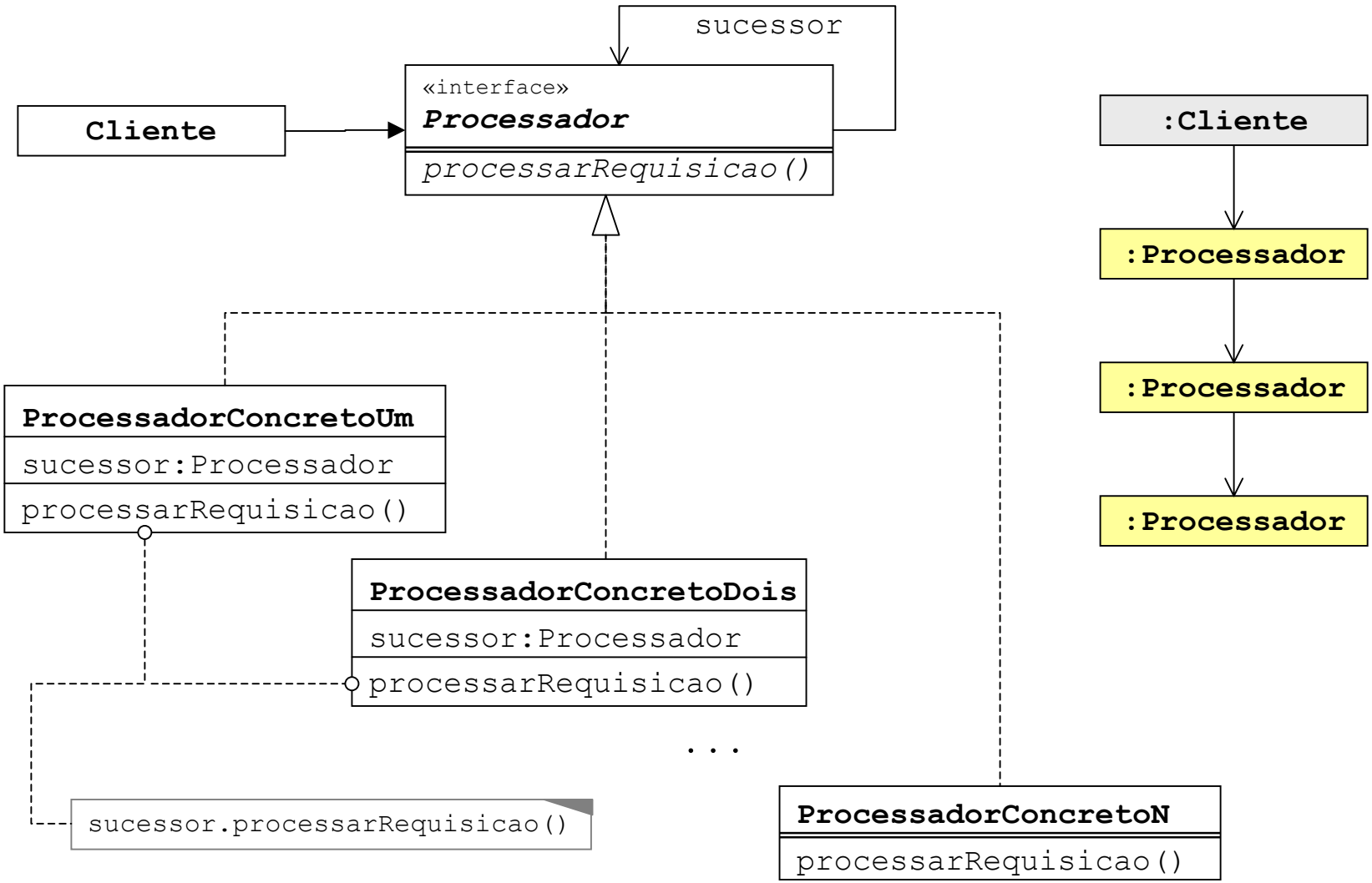



- *Permitir que vários objetos possam servir a uma requisição ou repassá-la*
- *Permitir divisão de responsabilidades de forma transparente*

*Um objeto pode ser uma folha ou uma composição de outros objetos*



# Estrutura de Chain of Responsibility



# Chain of Responsibility em Java

```
public class cliente {
    ...
          Processador p1 = ...
          Object resultado = p1.processarRequisicao();
    ...
}
```

```
public class ProcessadorUm implements Processador {
  public Object processarRequisicao() {
    ... // codigo um
    return sucessor.processarRequisicao();
  }
}
```

```
public class ProcessadorFinal implements Processador {
  public Object processarRequisicao() {
    return objeto;
  }
}
```

```
public interface Processador {
  public Object processarRequisicao();
}
```

# Exercícios

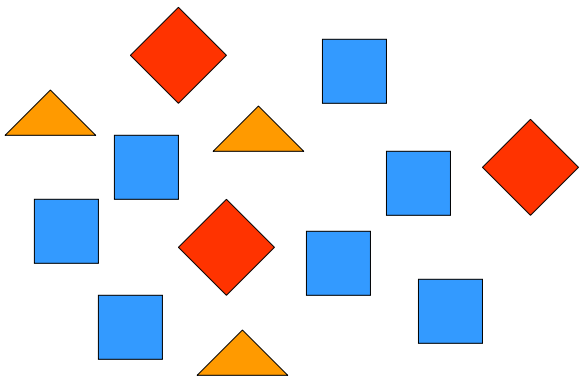
- 9.1 Escreva uma aplicação em Java que receba um texto ou arquivo de texto da linha de comando
  - O texto deve ser lido e estatísticas devem ser impressas sobre: a) o número de espaços encontrados, b) o número de letras 'a' e c) o número de pontos.
  - Use Chain of Responsibility e faça com que cada tipo de caractere seja tratado por um elo diferente da corrente.

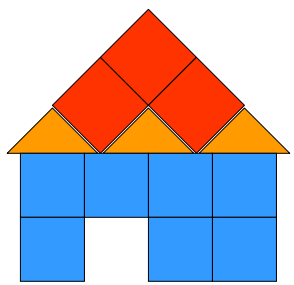
# 10

# Flyweight

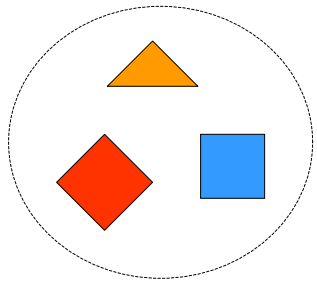
"*Usar compartilhamento para suportar grandes quantidades de objetos refinados eficientemente.*" *[GoF]*

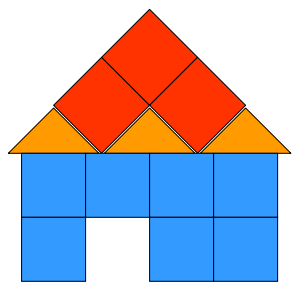
# Problema





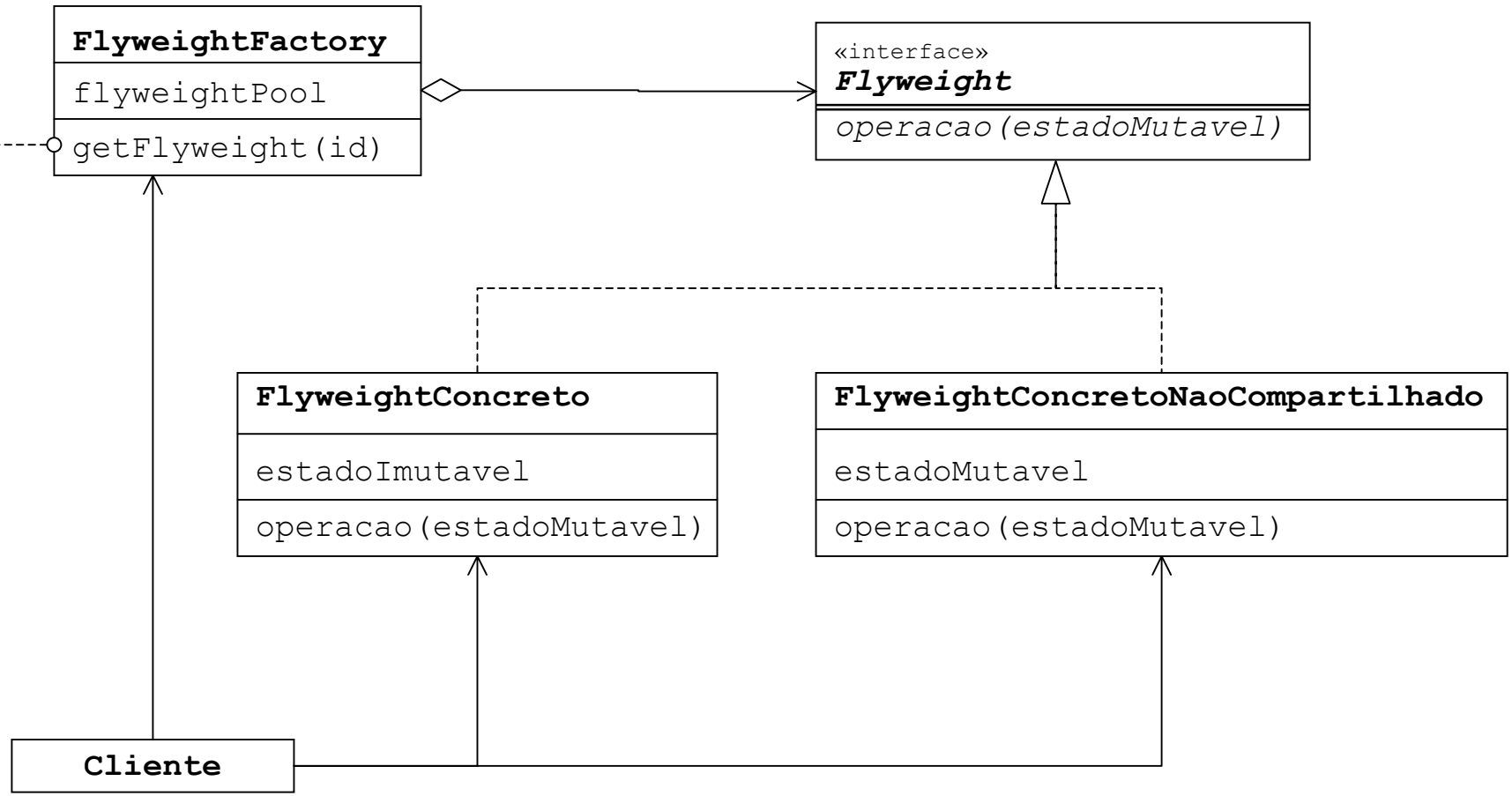
*Pool de objetos imutáveis compartilhados*





# Estrutura de Flyweight

```
if(flyweightPool.containsKey(id)) {
    return (Flyweight)flyweightMap.get(id);
} else {
    Flyweight fly = new FlyweightConcreto( genKey() );
    flyweightPool.put(fly.getKey(), fly);
    return fly;
}
```



# *Prós e Contras*

- *Quando usar Flyweight*
  - *Quando o tamanho do conjunto de objetos for significativamente menor que a quantidade de vezes em que eles são usados na aplicação*
  - *Quando objetos podem ser usados em diferentes contextos ao mesmo tempo (agindo sempre como um objeto indepentente)*
- *Quando não usar*
  - *Quando o estado dos objetos não for imutável (é preciso passar o estado mutável como parâmetro e isto pode ser impraticável se o estado consistir de vários objetos)*
  - *Quando for necessário elaborar um algoritmo ou algo complicado para separar objetos mutáveis de imutáveis*

# *Exercícios*

- *10.1 Implemente uma aplicação que imprima aleatoriamente 10 números de 10 algarismos.*
  - *Cada algarismo deve ser uma instancia do objeto Algarismo que possui o numero 1, 2, 3, etc. como membro imutável.*
  - *Use Flyweight para que objetos que representam o mesmo algarismo sejam reutilizados.*

# Resumo: Quando usar?

- *Singleton*
  - *Quando apenas uma instância for permitida*
- *Observer*
  - *Quando houver necessidade de notificação automática*
- *Mediator*
  - *Para controlar a interação entre dois objetos independentes*
- *Proxy*
  - *Quando for preciso um intermediário para o objeto real*
- *Chain of Responsibility*
  - *Quando uma requisição puder ou precisar ser tratada por um ou mais entre vários objetos*
- *Flyweight*
  - *Quando for necessário reutilizar objetos visando performance*

# *Testes*

*1. Descreva a diferença entre*

- *Adapter e Proxy*
- *Observer e Mediator*
- *Flyweight e Composite*
- *Singleton e Façade*
- *Mediator e Proxy*
- *Chain of Responsibility e Adapter*

# Fontes


[1] Steven John Metsker, *Design Patterns Java Workbook*. Addison-Wesley, 2002, Caps. 7 a 12. *Exemplos em Java, diagramas em UML e exercícios sobre Singleton, Proxy, Observer, Mediator, Chain of Responsibility e Flyweight.*

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995. *Singleton, Proxy, Observer, Mediator, Chain of Responsibility e Flyweight. Referência com exemplos em C++ e Smalltalk.*

[3] James W. Cooper. *The Design Patterns Java Companion. http://www.patterndepot.com/put/8/JavaPatterns.htm*