

**J930**


# Padrões

de

# Projeto

# 3

**Padrões de  
Responsabilidade**

Helder da Rocha (helder@acm.org) argonavis.com.br

### *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 único objeto*
  - *Notificação e intermediação de requisições*

2

### *Além das Responsabilidades comuns*

- *Padrões orientados 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*
  - *Proxy: assume a responsabilidade de ...*
  - *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*

3

5

## Singleton

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

4

## Problema

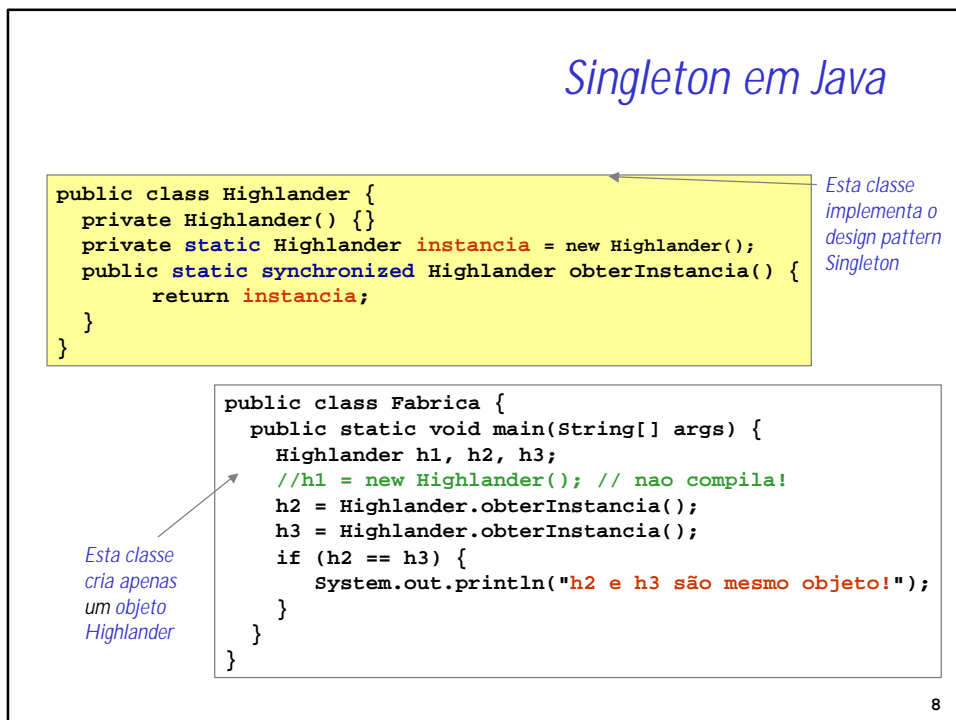
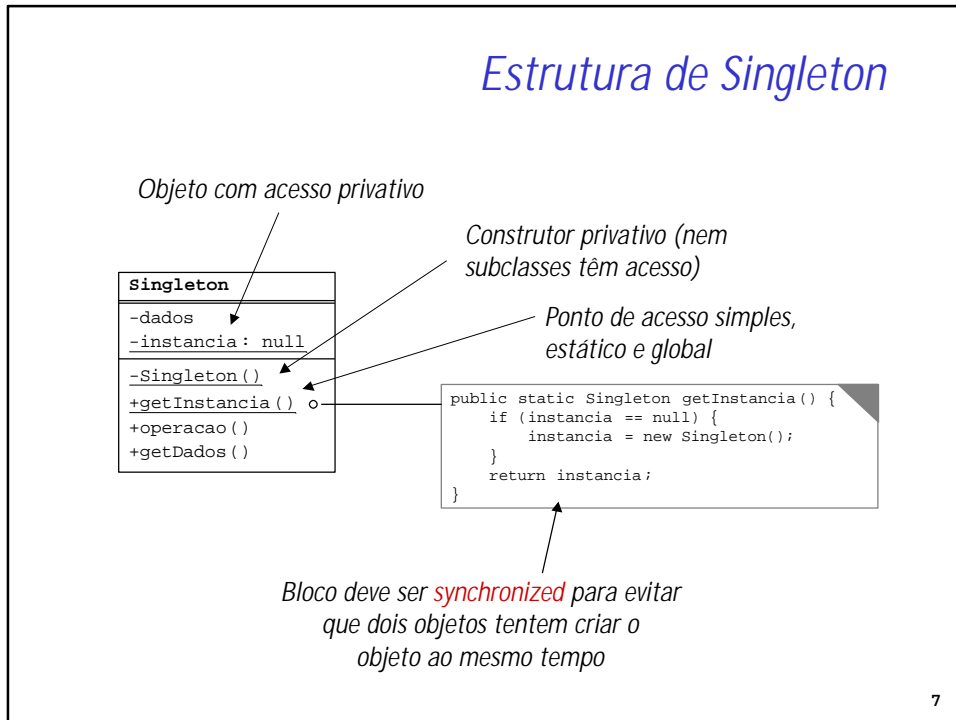
- *Garantir que apenas um objeto exista, independente do número de requisições que receber para criá-lo*
- *Aplicações*
  - *Um único banco de dados*
  - *Um único acesso ao arquivo de log*
  - *Um único objeto que representa um vídeo*
  - *Uma única fachada (Façade pattern)*
- *Poderia-se usar um membro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

5

## 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?*

6



## 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*
  - *Qualidade da implementação depende da linguagem*
  - *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)*

\* Mas é complicado: requer controle em todas as subclasses para garantir instância única (pelo menos um construtor precisa ser acessível às subclasses em Java) - use encapsulamento de pacote.

9

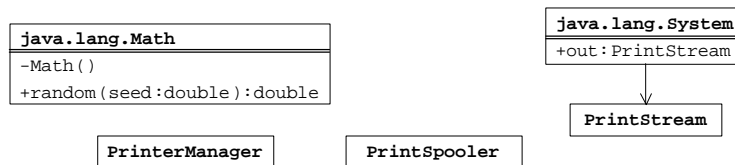
## Resumo

- *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.*

10

## 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.



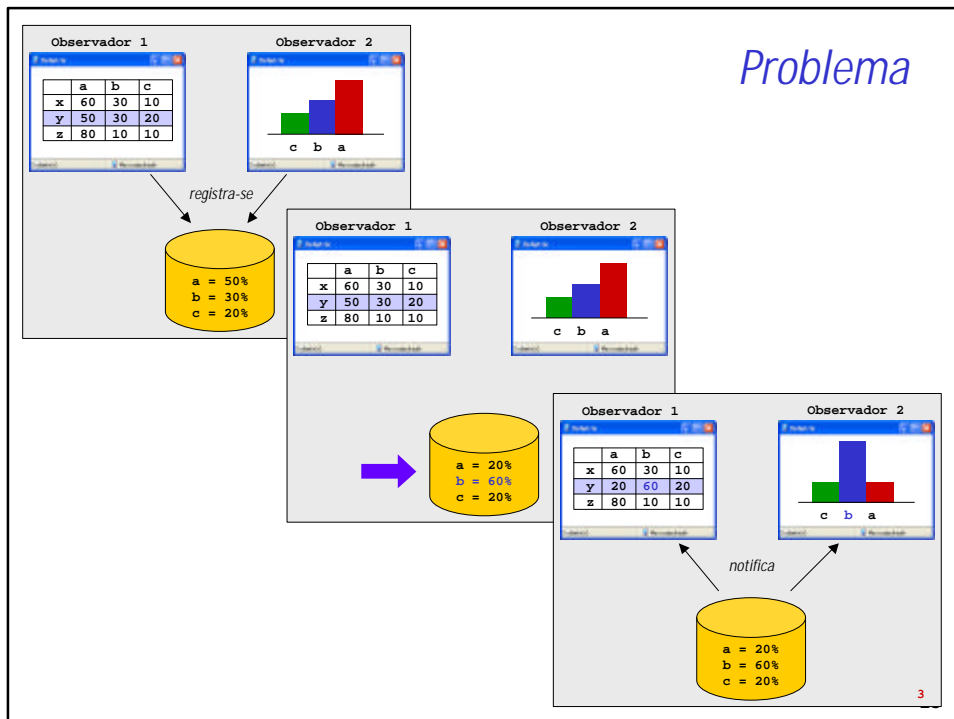
11

# 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]*

12



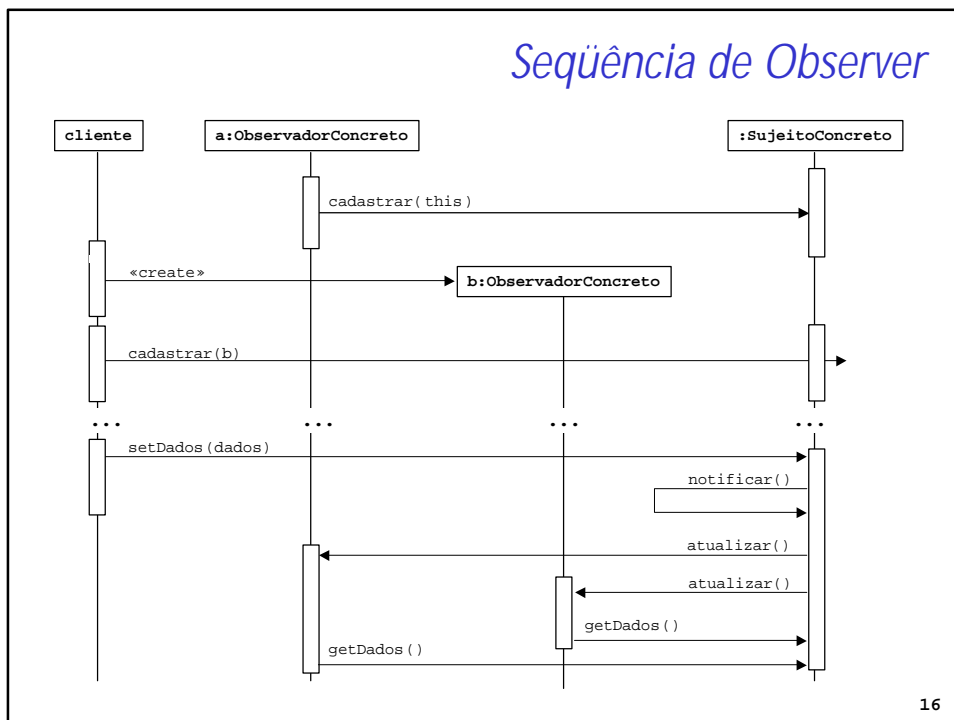
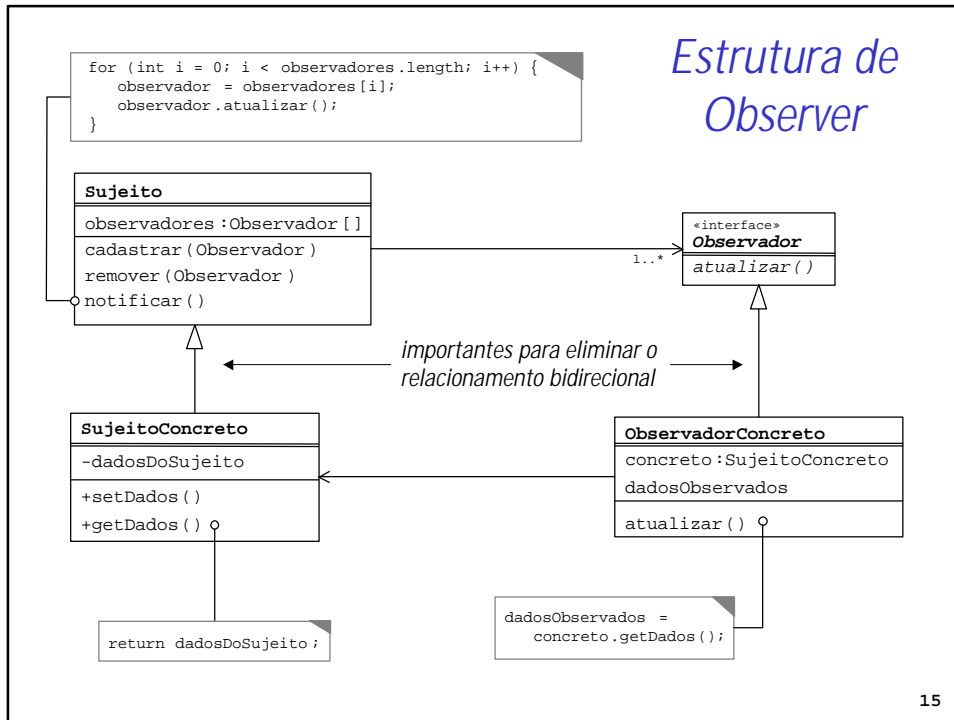
*Problema (2)*

- 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?

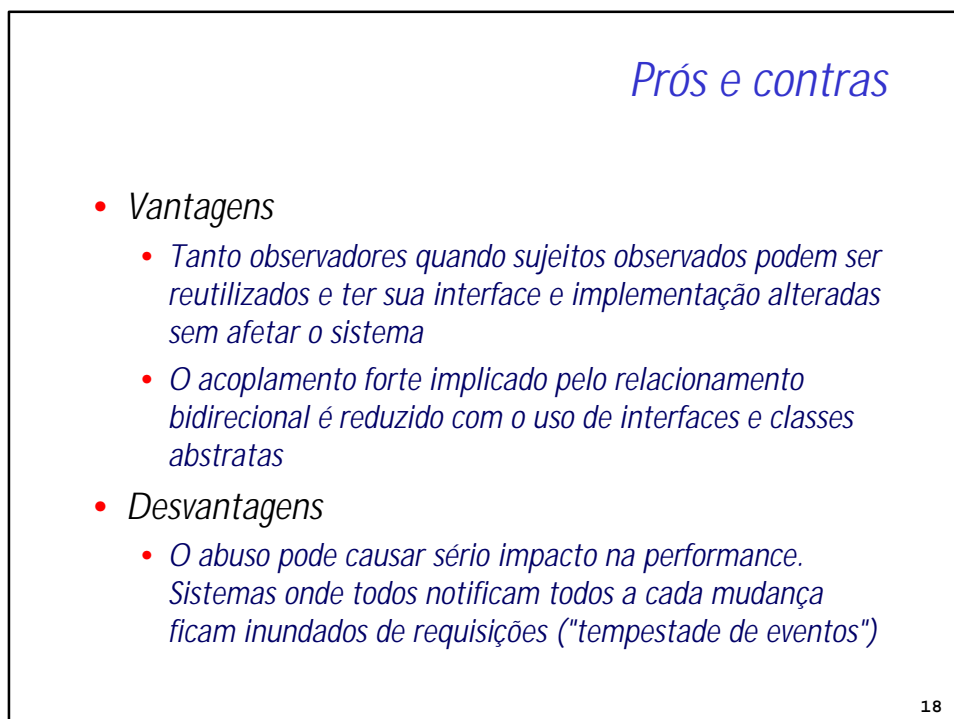
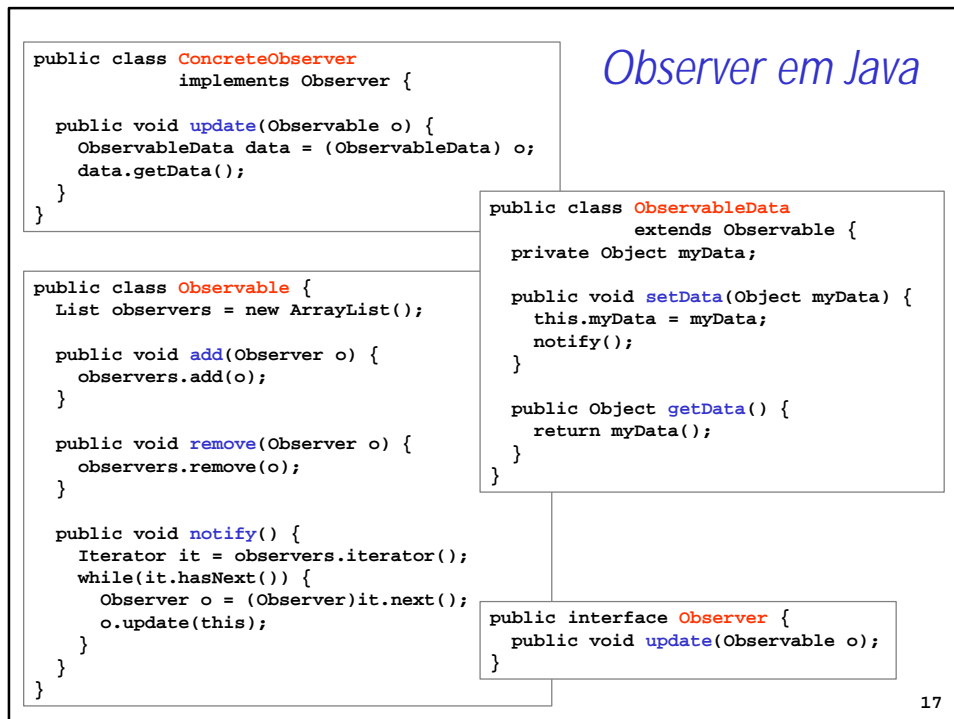
```

classDiagram
    Observador --> Sujeito : Cadastra-se
    Sujeito --> Observador : Notifica
    
```

14

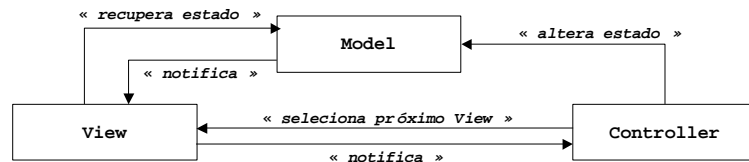






## Model-View-Controller

- O padrão de arquitetura MVC é uma combinação de padrões centrada no padrão Observer
- Consiste de três participantes
  - **Model**: representa os dados da aplicação e regras de negócio associadas com os dados. Notifica o View sobre alterações.
  - **View**: é um Observer para o Model. Notifica o Controller sobre eventos iniciados pelo usuário e lê dados do Model.
  - **Controller**: é um Observer para o View. Encapsula lógica de controle que afeta o Model e seleciona View.

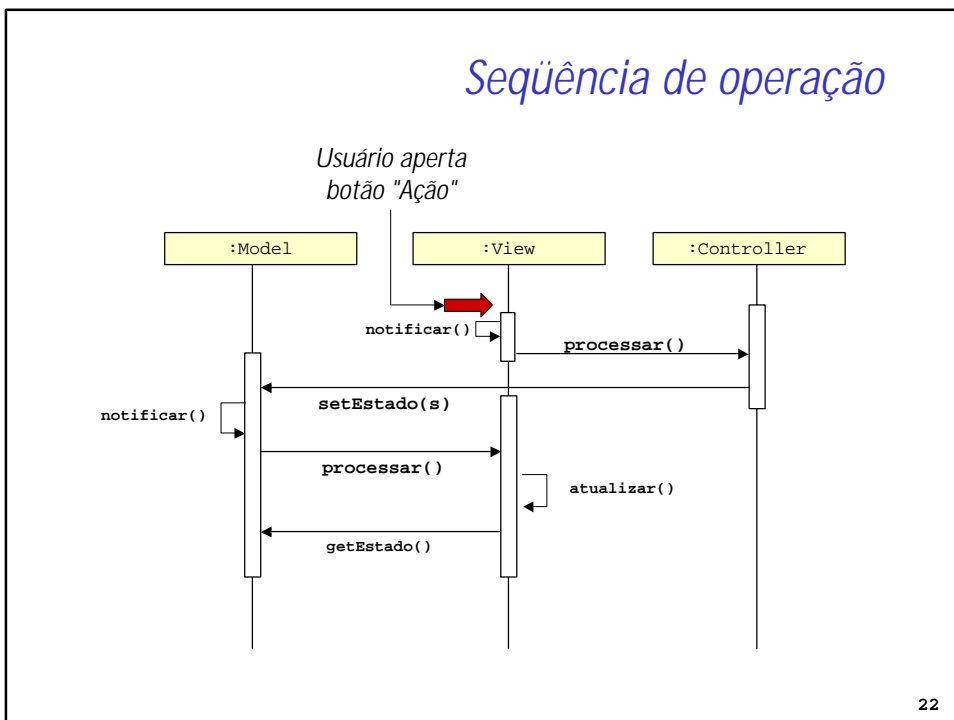
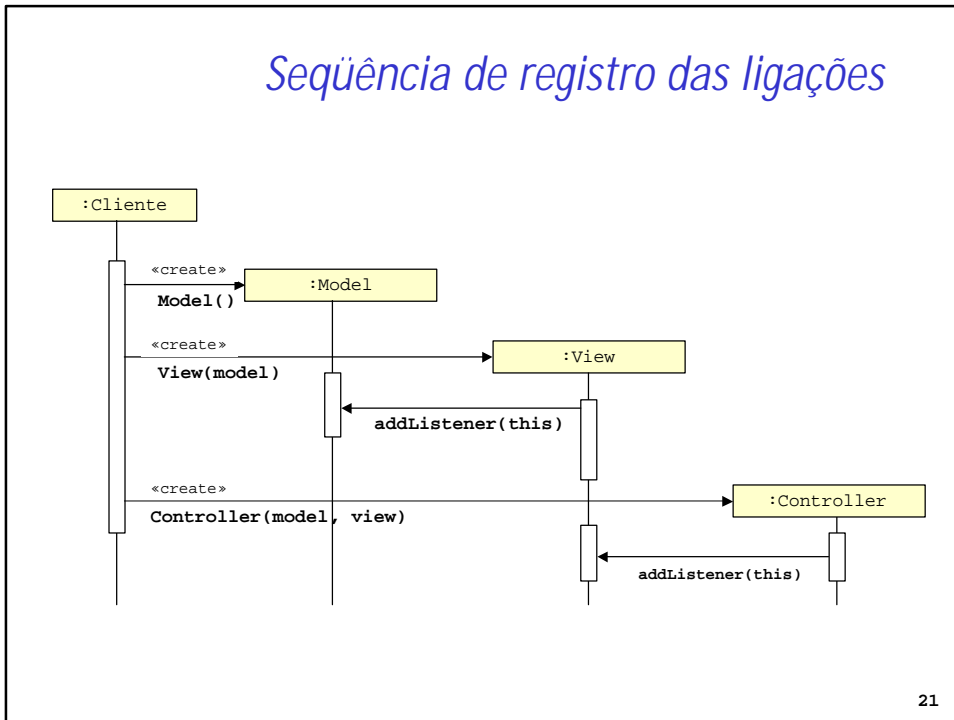


19

## Exemplo de implementação em Java

- Veja o exemplo *mvc*.
- Para executar, use o Ant: `ant run`
- Veja o código em `src/`
  - `Cliente.java`
  - `Model.java`
  - `View.java`
  - `Controller.java`

20



## Exercícios

- 6.1 Complete o código disponível nas classes *Model*, *View* e *Controller*
- 6.2 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.
  - Use o código semi-pronto fornecido
  - Identifique as partes do código que representam *Model*, *View* e *Controller*

23

# 7

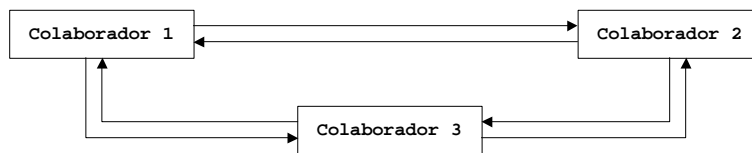
## Mediator

*"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]*

24

## Problema

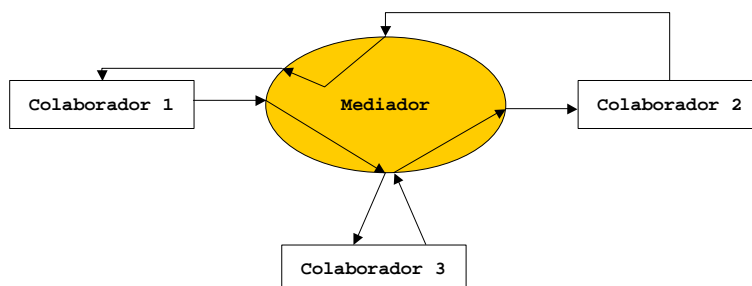
- Como permitir que um grupo de objetos se comunique entre si sem que haja acoplamento entre eles?
- Como remover o forte acoplamento presente em relacionamentos muitos para muitos?
- Como permitir que novos participantes sejam ligados ao grupo facilmente?



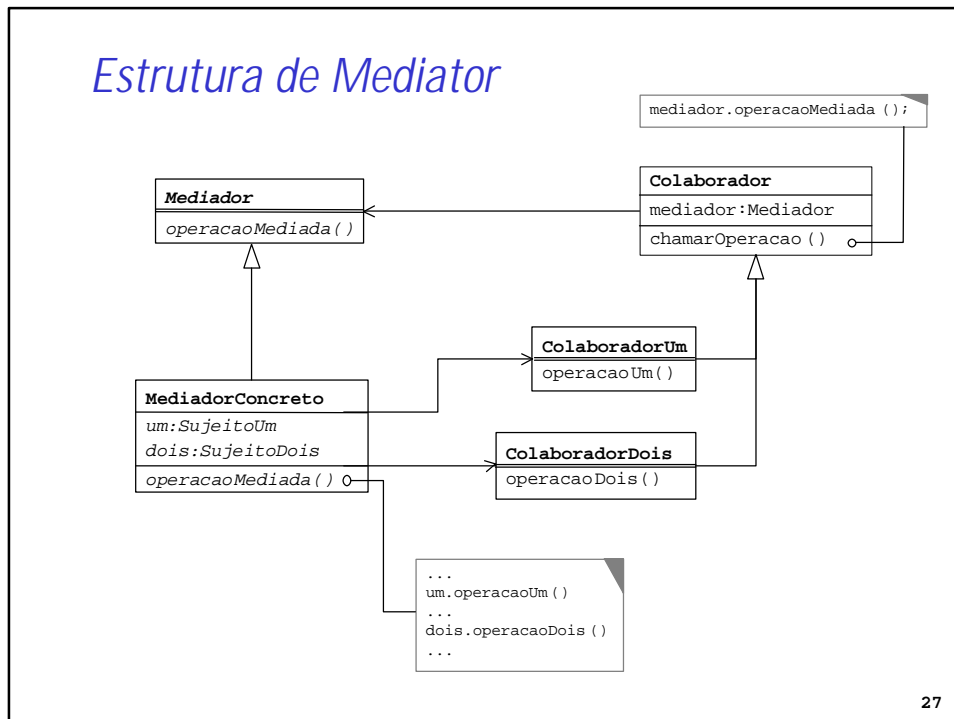
25

## Solução

- Introduzir um mediador
  - *Objetos podem se comunicar sem se conhecer*



26



### Descrição da solução

- *Um objeto Mediator 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 Mediator 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)*

28

## Mediator em Java

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

```
public class ConcreteMediator
    implements Mediator {

    Colleague felix = new Gato();
    Colleague mickey = new Rato();

    public ConcreteMediator() {
        felix.setMediator(this);
        mickey.setMediator(this);
    }
    public void chaseOperation() {
        ... //if certa condição ...
        mickey.escapeCat();
        ...
    }
    public void escapeOperation() {
        ... //if certa condição ...
        felix.chaseMouse();
        ...
    }
}
```

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

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

```
public class Rato extends Colleague {

    public void escapeCat() {
        mediator.escapeOperation();
    }
}
```

```
public class Cat extends Colleague {

    public void chaseMouse() {
        mediator.chaseOperation();
    }
}
```

29

## Prós e contras

- *Vantagens*
  - *Desacoplamento entre os diversos participantes da rede de comunicação: participantes não se conhecem.*
  - *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*

30

## Exercícios

- 1. Implemente o código que falta nas classes fornecidas para fazer o mediador funcionar
- 2. Introduza um mediador para eliminar a dependência entre as classes abaixo
  - Faça com que cada objeto só conheça o mediador
  - Implemente um mecanismo que permita que novos objetos possam se cadastrar no mediador

31

# 8

## Proxy

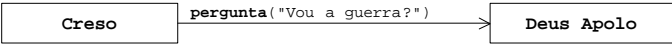
*"Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro." [GoF]*

32



### Problema


- Sistema quer utilizar objeto real...



```

    graph LR
      Creso[Creso] -- pergunta("Vou a guerra?") --> DeusApolo[Deus Apolo]
  
```

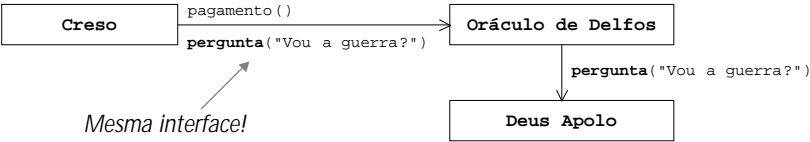
- Mas ele não está disponível (remoto, inacessível, ...)



```

    graph LR
      Creso[Creso] --> Q[?]
  
```

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



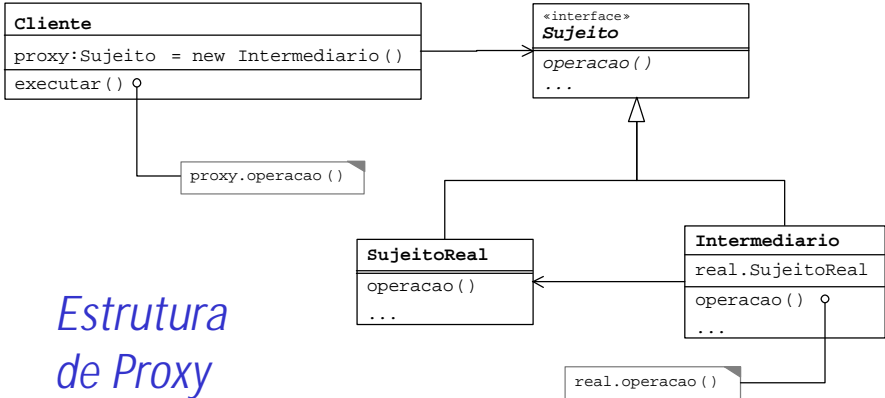
```

    graph TD
      Creso[Creso] -- pagamento() --> Oraculo[Oráculo de Delfos]
      Creso -- pergunta("Vou a guerra?") --> Oraculo
      Oraculo -- pergunta("Vou a guerra?") --> DeusApolo[Deus Apolo]
  
```

Mesma interface!

33

### Estrutura de Proxy



```

    classDiagram
      class Cliente {
        proxy: Sujeito = new Intermediario()
        executar()
      }
      class Sujeito {
        <<interface>>
        operacao()
        ...
      }
      class SujeitoReal {
        operacao()
        ...
      }
      class Intermediario {
        real: SujeitoReal
        operacao()
        ...
      }
      Cliente --> Sujeito
      SujeitoReal --|> Sujeito
      Intermediario --|> Sujeito
      Intermediario o-- SujeitoReal
  
```

- 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

34

### Proxy em Java

```

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

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

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

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

*inacessível pelo cliente*

*cliente comunica-se com este objeto*

35

### 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

```

classDiagram
    RemoteInterface <|-- Stub
    RemoteInterface <|-- Skeleton
    RemoteInterface <|-- ObjetoRemoto
    Cliente --> Stub
    Stub --> Skeleton : <<rede>>
    Stub -- socket ...
    
```

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

36

### Prós e contras

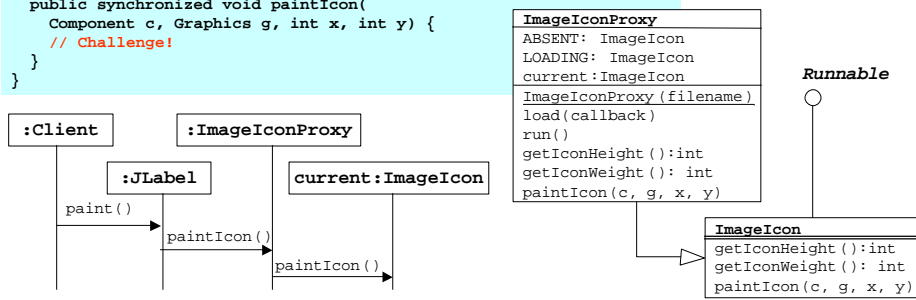
- **Vantagens**
  - *Transparência: mesma sintaxe usada na comunicação entre o cliente e sujeito real é usada no proxy*
  - *Permite o tratamento inteligente dos dados no cliente*
  - *Permite maior eficiência com caching no cliente*
- **Desvantagens**
  - *Possível impacto na performance*
  - *Transparência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

37

### 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]*

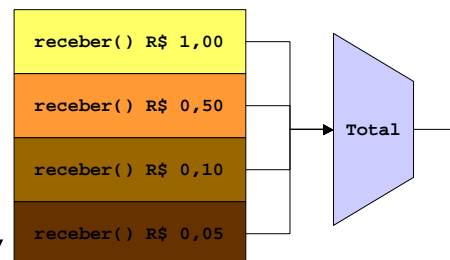
39

### Problema

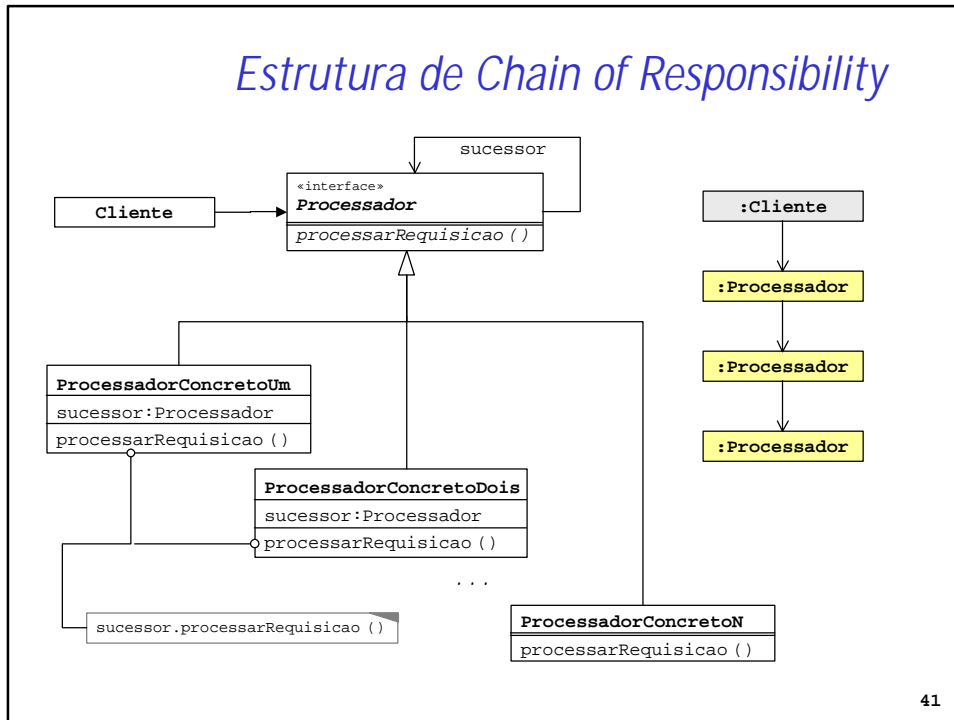


- Permitir que vários objetos possam servir a uma requisição ou repassá-la
- Permitir divisão de responsabilidades de forma transparente

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



40



### Chain of Responsibility em Java

```

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

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

...

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

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

*Nesta estratégia cada participante conhece seu sucessor*

42

### *Estratégias de Chain Of Responsibility*

- *Pode-se implementar um padrão de várias formas diferentes. Cada forma é chamada de estratégia (ou idiom\*)*
- *Chain of Responsibility pode ser implementada com estratégias que permitem maior ou menor acoplamento entre os participantes*
  - *Usando um mediador: só o mediador sabe quem é o próximo participante da cadeia*
  - *Usando delegação: cada participante conhece o seu sucessor*

\* Não são sinónimos: diferem no detalhamento e dependência de linguagem

43

### *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.*

44

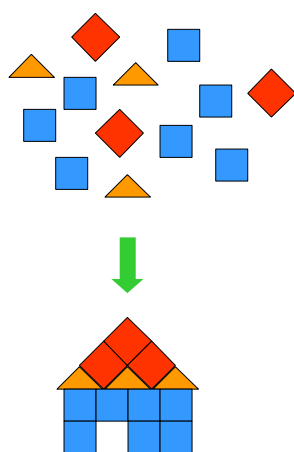
10

# Flyweight

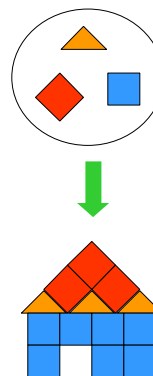
*"Usar compartilhamento para suportar grandes quantidades de objetos refinados eficientemente." [GoF]*

45

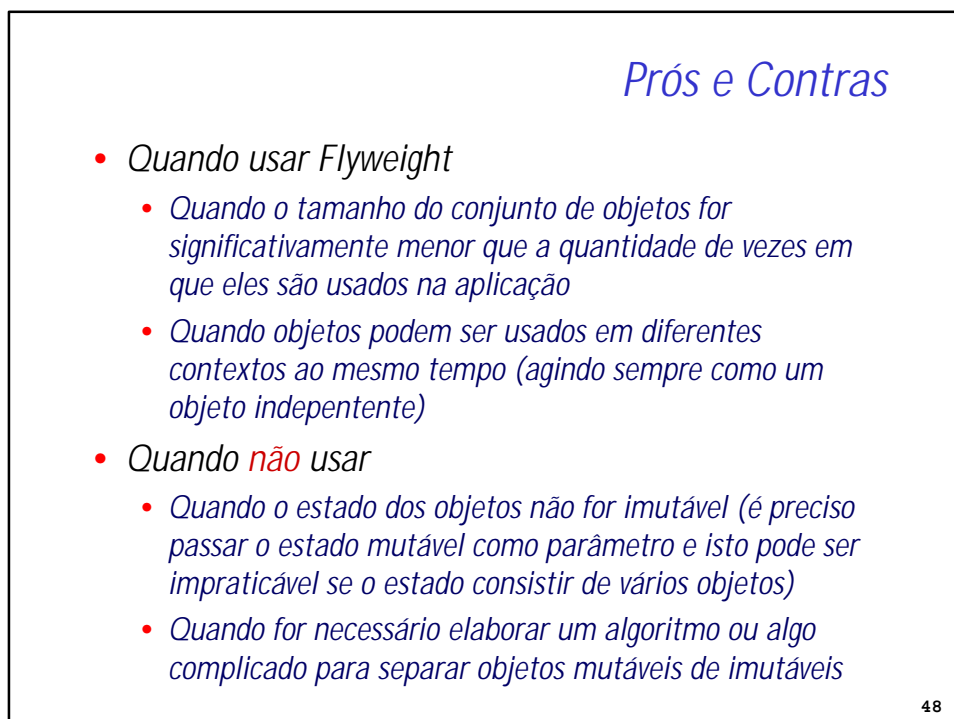
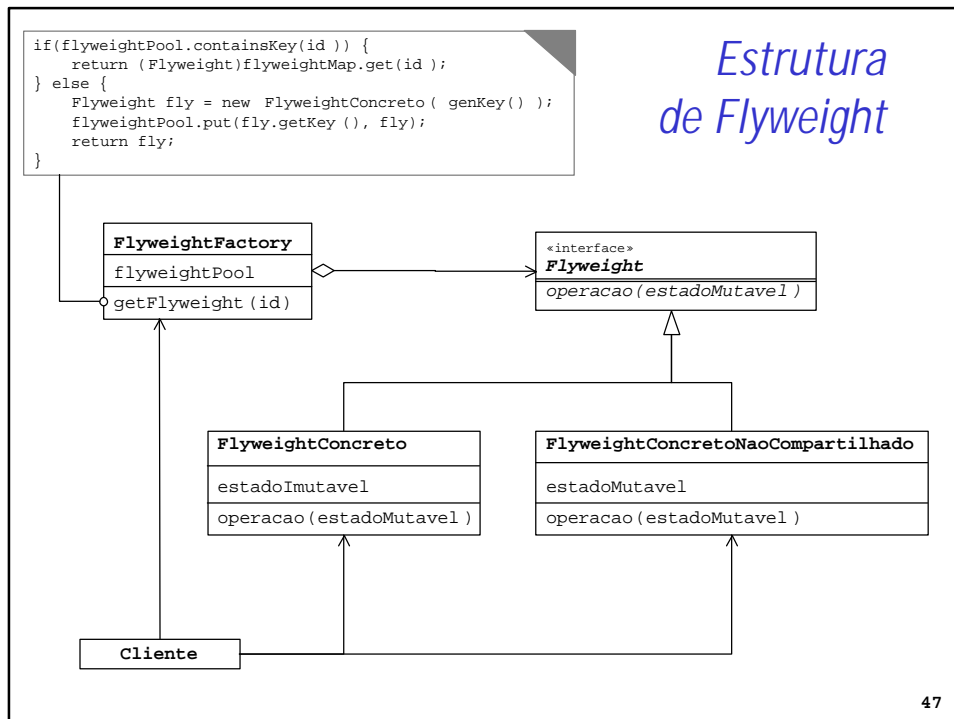
## Problema



Pool de objetos imutáveis compartilhados



46





## Exercícios

- *10.1 Implemente uma aplicação que imprima aleatoriamente 10 números de 10 Algarismos.*
  - *Cada algarismo deve ser uma instancia do objeto Algarismo que contém o numero 1, 2, 3, etc. como membro imutável.*
  - *Use Flyweight para construir um cache de objetos para que objetos que representam o mesmo algarismo sejam reutilizados.*
- *10.2 Implemente um cache de fatoriais*
  - *Cada fatorial de  $n$  equivale a  $n * \text{fatorial}(n-1)$ . Use um cache para aproveitar fatoriais já calculados*
  - *Escreva uma aplicação que calcule fatoriais de 0 a 15.*

49

## 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*

50

## 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*

51

## 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>

52

*Curso J930: Design Patterns*  
*Versão 1.1*

*[www.argonavis.com.br](http://www.argonavis.com.br)*

© 2003, Helder da Rocha  
([helder@acm.org](mailto:helder@acm.org))