

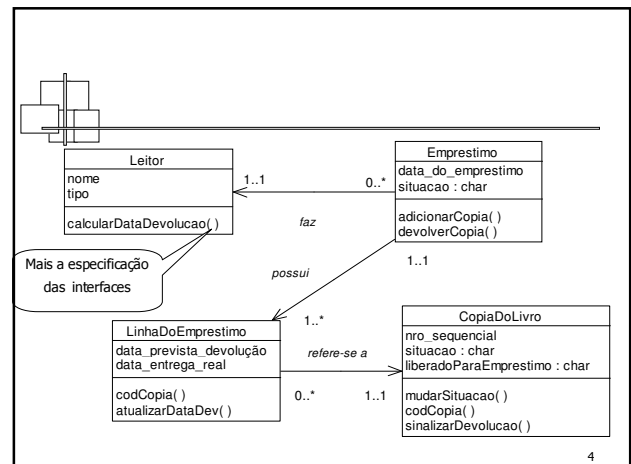
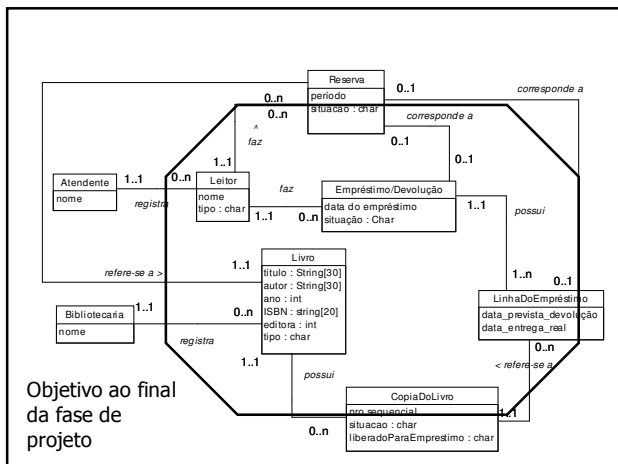
Diagramas de Comunicação e Padrões GRASP

1

O que vimos até agora

- Diagramas de Caso de Uso
- Casos de uso no formato completo (abstrato ou concreto)
- Modelo Conceitual
- Diagramas de Sequência do Sistema (Usados em Cenários)
- Contratos de Operações
- Notação dos Diagramas de Colaboração/Comunicação
- Diagrama de Classes

2



Como projetar as responsabilidades de cada objeto?

- Sabemos que os objetos precisam se comunicar
- Os Diagramas de comunicação mostram escolhas de atribuição de responsabilidade a objetos
- Mas quem é o melhor candidato para realizar/implementar/materializar cada uma das operações e dos métodos do sistema?

5

Como projetar as responsabilidades de cada objeto?

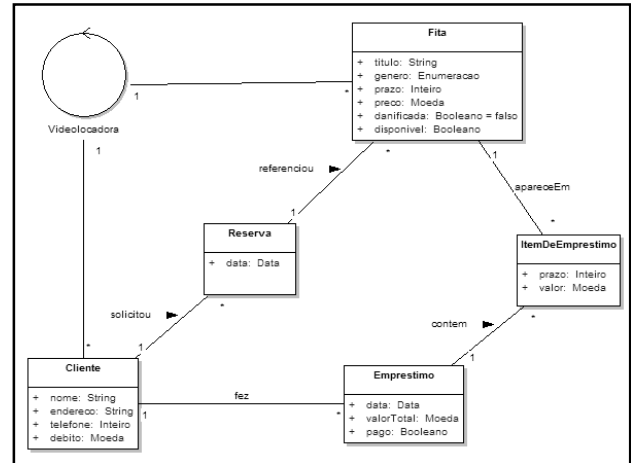
- Responsabilidade:
 - um contrato ou obrigação de um tipo ou classe
 - serviços fornecidos por um elemento (classe ou subsistema)
- Dois tipos de responsabilidades básicas:
 - **Fazer**
 - fazer algo (criar um objeto, executar uma operação,...)
 - iniciar ações em outros objetos (delegação)
 - coordenar e controlar atividades em outros objetos
 - **Saber**
 - conhecer dados privados encapsulados
 - conhecer objetos relacionados
 - conhecer dados/atributos que podem ser derivados ou calculados

6

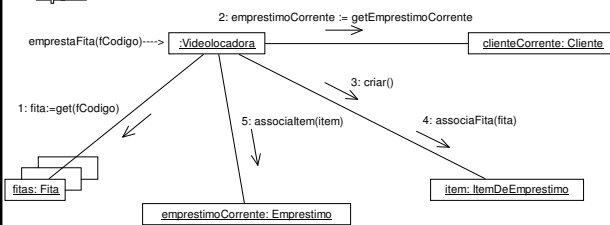
Exemplo

- Implementação inchada ou concentradora,
- Versus
- Implementação leve, distribuída

7



Comunicação entre os objetos



9

Pseudocódigo concentrador

Classe VideoLocadora

fitas : Conjunto;
clienteCorrente : Cliente;

Método emprestaFita(fCodigo: String)

```

fita : Fita;
emprestimoCorrente : Emprestimo;
item : ItemDeEmprestimo;

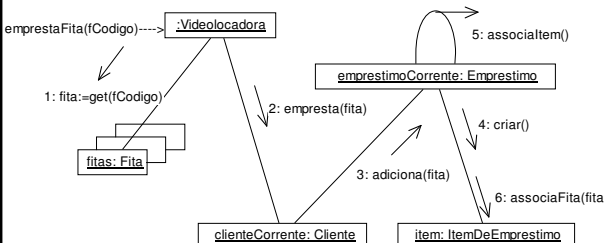
fita := fitas.get(fCodigo);
emprestimoCorrente := clienteCorrente.getEmprestimoCorrente();
item := ItemDeEmprestimo.new();
item.associaFita(fita);
emprestimoCorrente.associaItem(item);

```

Fim Método;
Fim Classe.

(by Raul Wazlawick)

Diagrama de Comunicação não concentrador



11

Código com Responsabilidades Distribuídas

Classe VideoLocadora

fitas : Conjunto ;
clienteCorrente : Cliente;

Método emprestaFita(fCodigo : String):

```

fita : Fita;

fita := fitas.get(fCodigo);
clienteCorrente.empresta(fita)

```

Fim Método;
Fim Classe.

Classe Cliente

emprestimoCorrente : Emprestimo;

Método empresta(fita : Fita):

```

emprestimoCorrente.adiciona(fita);

```

Fim Método;
Fim Classe.

Classe Emprestimo

itens : Conjunto;

Método adiciona(fita : Fita):

```

item : ItemDeEmprestimo;

item := ItemDeEmprestimo.new();
self.associaItem(item);
item.associaFita(fita);

```

Fim Método;
Fim Classe.

(de Raul Wazlawick)

Discussão

- Qual dos códigos é mais fácil de entender e manter?
- Em qual dos códigos as responsabilidades das classes parecem mais intuitivas?
- Para desenvolver um bom projeto, precisamos de princípios que nos guiem na atribuição de responsabilidades → padrões

13

Padrões

- Desenvolvedores de software experientes criaram um repertório de princípios gerais e boas soluções para guiar a construção de software
- Essas soluções foram descritas em um formato padronizado (nome, problema, solução) e podem ser usadas em outros contextos
- Surgiram com base no trabalho do arquiteto Christopher Alexander, 1977
- Ganham impulso após a publicação dos Design Patterns – Gamma e outros – GoF - 1994

14

Padrões

- Padrões usualmente não contêm novas idéias
 - organizam conhecimentos e princípios existentes, testados e consagrados
- **Padrão** é uma descrição nomeada de um problema e uma solução, que pode ser aplicado em novos contextos

15

Padrões GRASP

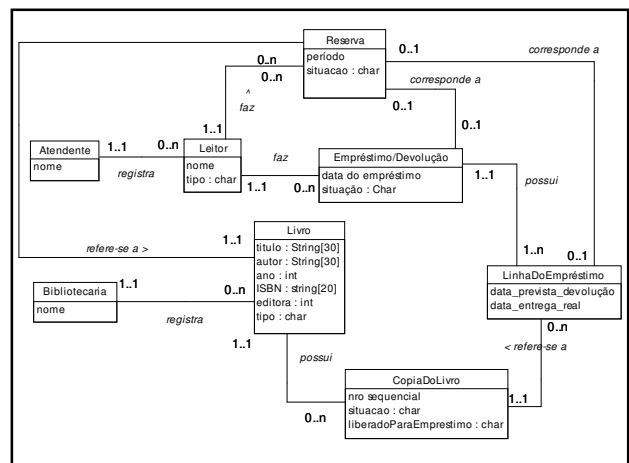
- GRASP = **General Responsibility Assignment Software Patterns**
- Descrevem princípios fundamentais de atribuição de responsabilidade a objetos
- Alguns padrões GRASP principais:
 - Especialista (*Expert*)
 - Criador (*Creator*)
 - Coesão alta (*High Cohesion*)
 - Acoplamento fraco (*Low Coupling*)
 - Controlador (*Controller*)

16

Especialista

- **Problema:** qual é o princípio mais básico de atribuição de responsabilidades a objetos ?
- **Solução:** Atribuir responsabilidade ao especialista da informação.
- Exemplo: no sistema de biblioteca, quem seria o responsável por calcular a data de devolução de um livro?

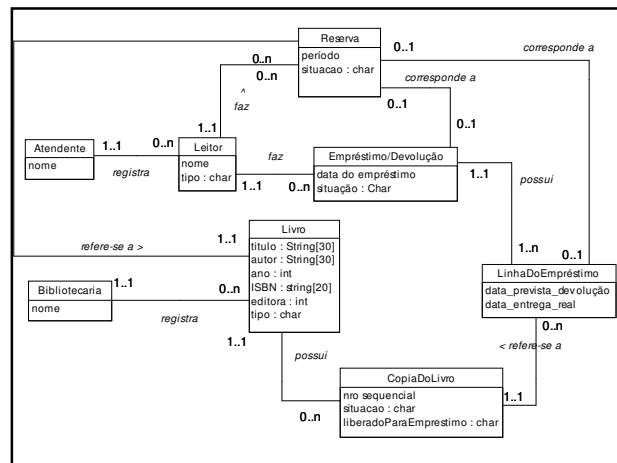
17



Especialista

- A data de devolução ficará armazenada no atributo `data_prevista_devolucao` do objeto `LinhaDoEmprestimo`
- Mas quem possui conhecimentos necessários para calculá-la?

19

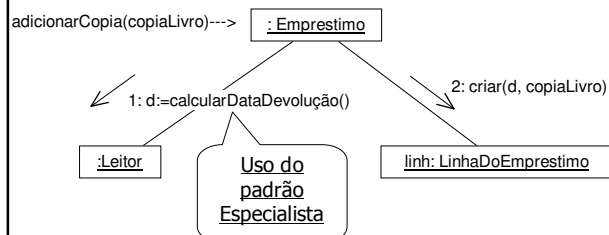


Especialista

- Pelo padrão especialista, `Leitor` deve receber esta atribuição, pois conhece o tipo de `Leitor` (por exemplo, aluno de graduação, aluno de pós-graduação, professor, etc), que é utilizado para calcular a data em que o livro deve ser devolvido

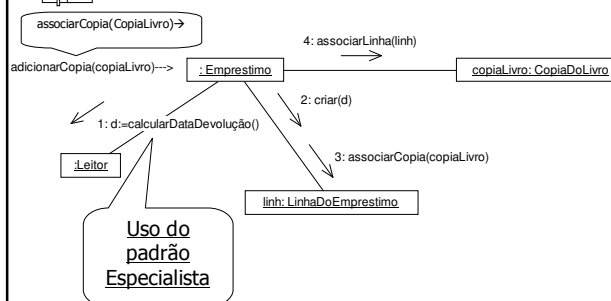
21

Especialista



22

Especialista – Alternativa com mais detalhes



23

Especialista

- Onde procurar pela classe especialista?
 - Começar pelas classes já estabelecidas durante o projeto
 - Se não encontrar, utilizar o Modelo Conceitual
- Lembrar que existem especialistas parciais que colaboram numa tarefa
 - informação espalhada → comunicação via mensagens
- Existe uma analogia com o mundo real

24

Especialista

- **Benefícios:**
 - Mantém encapsulamento → favorece o acoplamento fraco
 - O comportamento fica distribuído entre as classes que têm a informação necessária (classes "leves") → favorece alta coesão
 - Favorece o reuso
- **Contra-indicações**
 - contra indicado quando aumenta acoplamento e reduz coesão
 - Ex: quem é responsável por salvar um Empréstimo no banco de dados?

25

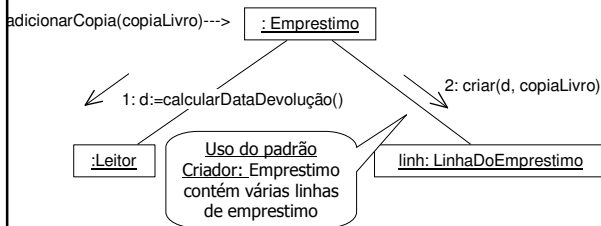
Criador

- **Problema:** Quem deveria ser responsável pela criação de uma nova instância de alguma classe ?
- **Solução:** atribua à classe B a responsabilidade de criar uma nova instância da classe A se uma das seguintes condições for verdadeira:
 - B agrega objetos de A
 - B contém objetos de A
 - B registra objetos de A
 - B usa objetos de A
 - B tem os valores iniciais que serão passados para objetos de A, quando de sua criação

26

Criador

- Exemplo: No sistema da Biblioteca, quem é responsável pela criação de um *ItemDeEmprestimo* ?



27

Criador

- **Discussão**
 - objetivo do padrão: definir como criador o objeto que precise ser conectado ao objeto criado em algum evento
 - escolha adequada favorece acoplamento fraco
 - objetos agregados, contêineres e registradores são bons candidatos à responsabilidade de criar outros objetos
 - algumas vezes o candidato a criador é o objeto que conhece os dados iniciais do objeto a ser criado

28

Acoplamento

- **Acoplamento:** dependência entre elementos (classes, subsistemas,...), normalmente resultante de colaboração para atender a uma responsabilidade
- O acoplamento mede o quanto um objeto está conectado a, tem conhecimento de, ou depende de outros objetos
 - acoplamento fraco (ou baixo) – um objeto não depende de muitos outros
 - acoplamento forte (ou alto) – um objeto depende de muitos outros

29

Acoplamento

- **Problemas do acoplamento alto:**
 - mudanças em classes interdependentes forçam mudanças locais
 - dificulta a compreensão do objetivo de cada classe
 - dificulta a reutilização

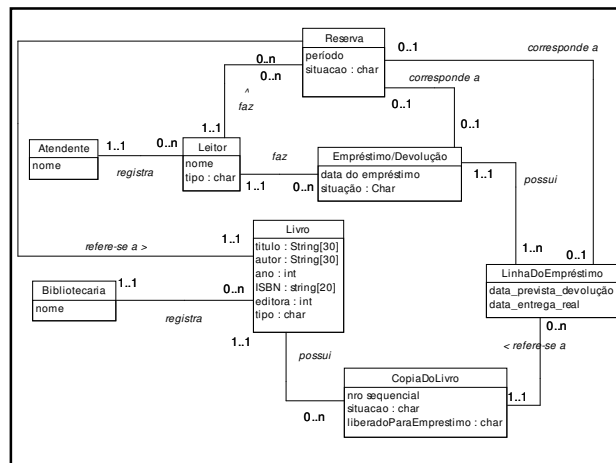
30



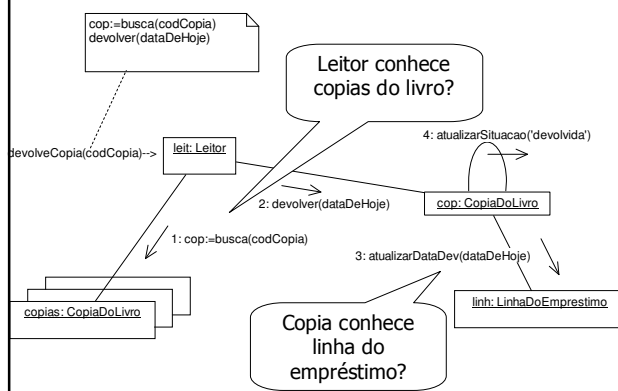
Acoplamento Fraco

- **Problema:** como favorecer a baixa dependência e aumentar a reutilização ?
- **Solução:** Atribuir responsabilidade de maneira que o acoplamento permaneça baixo.
- **Exemplo:** No sistema de biblioteca, suponha que queremos realizar a devolução da cópia do livro. Qual classe deve ser responsável por essa tarefa?

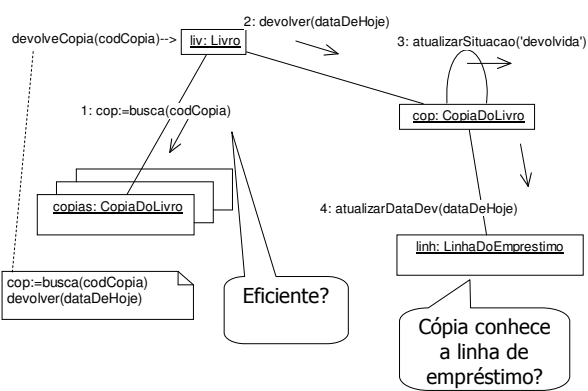
31



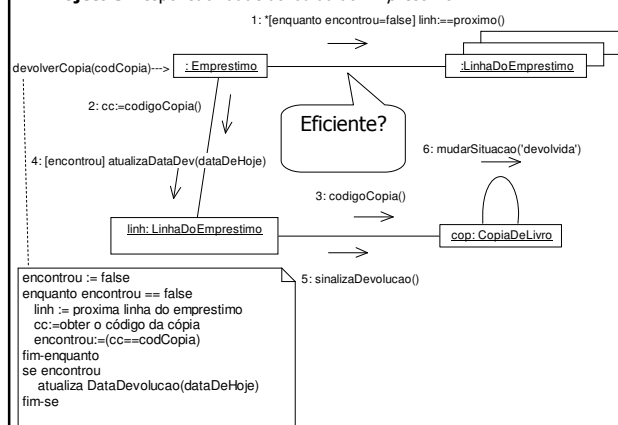
Projeto 1: responsabilidade atribuída ao *Leitor*



Projeto 2: responsabilidade atribuída ao *Livro*



Projeto 3: responsabilidade atribuída ao *Empréstimo*



Qual projeto é melhor?

- Qual dos projetos anteriores favorece o acoplamento fraco?
 - projeto 1 e 2 – acoplamento aumenta (entre cópia do livro e linha do empréstimo, acoplamento entre leitor e cópia do livro)
 - projeto 3 – não aumenta acoplamento

←
PREFERÍVEL

36



Formas de Acoplamento

- Um objeto tem um atributo que referencia um objeto de outra classe
- Um objeto tem um método que referencia um objeto de outra classe
 - parâmetro, variável local ou retorno
- Um objeto invoca os serviços de um objeto de outra classe
- Uma classe é subclasse de outra, direta ou indiretamente

37



Acoplamento Fraco

- Discussão:
 - Acoplamento fraco → classes mais independentes
 - reduz impacto de mudanças
 - favorece reuso de classes
 - Considerado em conjunto com outros padrões
 - Extremo de acoplamento fraco não é desejável
 - fere princípios da tecnologia de objetos – comunicação por mensagens
 - projeto pobre: objetos inchados e complexos, responsáveis por muito trabalho → baixa coesão

38



Acoplamento Fraco

- Discussão:
 - dica: concentre-se em reduzir o acoplamento em pontos de evolução ou de alta instabilidade do sistema
- Benefícios:
 - classes são pouco afetadas por mudanças em outras partes
 - classes são simples de entender isoladamente
 - conveniente para reutilização

39



Coesão

- **Coesão** mede o quanto as responsabilidades de um elemento (classe, objeto, subsistema,...) são fortemente focalizadas e relacionadas
- Objeto com Coesão Alta → objeto cujas responsabilidades são altamente relacionadas e que não executa um volume muito grande de trabalho
- Objeto com Coesão Baixa → objeto que faz muitas tarefas não relacionadas ou executa muitas tarefas
 - difícil de compreender, reutilizar e manter
 - constantemente afetado por mudanças

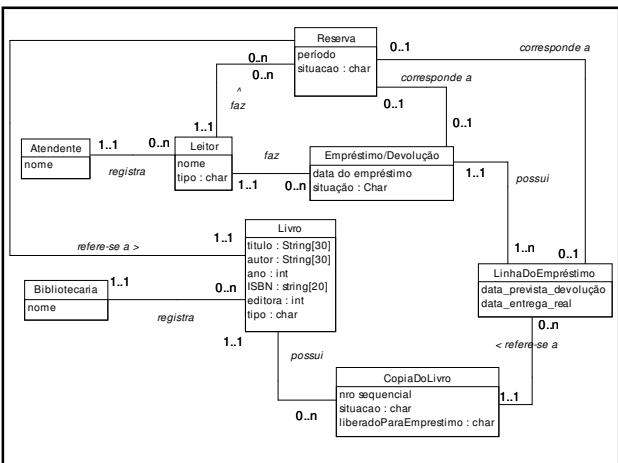
40



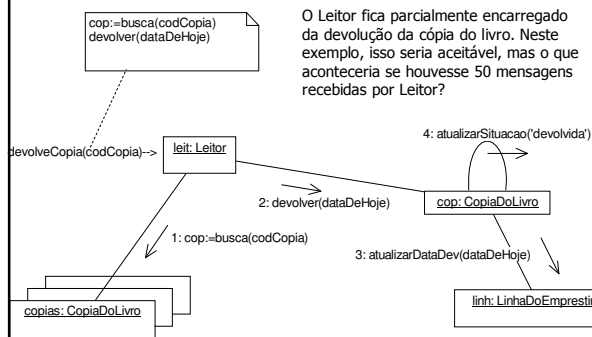
Coesão Alta

- Problema: Como manter a complexidade sob controle?
- Solução: Atribuir responsabilidade de tal forma que a coesão permaneça alta.
- Exemplo (o mesmo para o acoplamento fraco): No sistema de biblioteca, suponha que queremos realizar a devolução da cópia do livro. Qual classe deve ser responsável por essa tarefa?

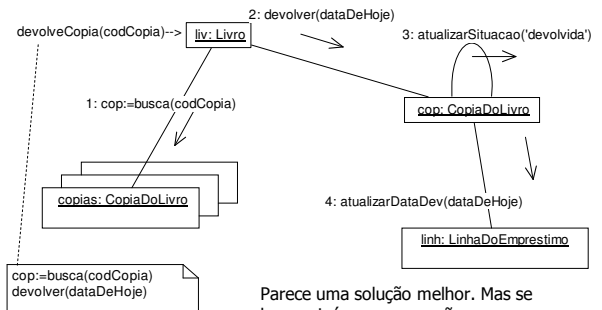
41



Projeto 1: responsabilidade atribuída ao *Leitor*

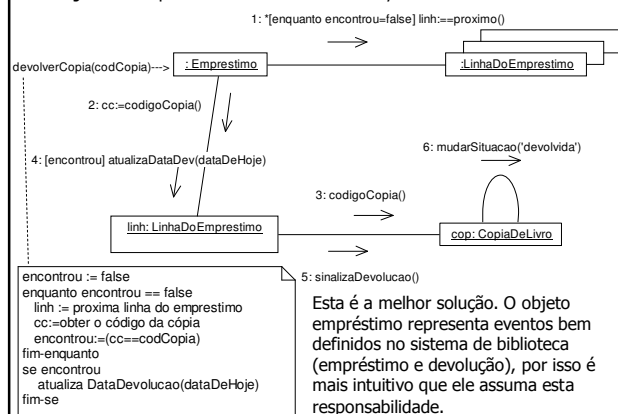


Projeto 2: responsabilidade atribuída ao *Livro*



Parece uma solução melhor. Mas se houver inúmeras operações a serem feitas com o livro, ocorre o mesmo problema de Leitor.

Projeto 3: responsabilidade atribuída ao *Empréstimo*



Coesão Alta

■ Discussão:

- Coesão alta, assim como Acoplamento Fraco, são princípios que devem ser considerados no projeto de objetos
 - má coesão traz acoplamento ruim e vice-versa
- regra prática: classe com coesão alta tem um número relativamente pequeno de métodos, com funcionalidades relacionadas, e não executa muito trabalho
- analogia com mundo real
 - ex: pessoas que assumem muitas responsabilidades não associadas podem tornar-se (e normalmente tornam-se) ineficientes

46

Coesão Alta

■ Benefícios

- mais clareza e facilidade de compreensão no projeto
- simplificação de manutenção e acréscimo de funcionalidade/melhorias
- favorecimento do acoplamento fraco
- aumento no potencial de reutilização
 - classe altamente coesa pode ser usada para uma finalidade bastante específica

47

Será que esta solução é ideal?

- Ainda temos um problema: quando ocorre o evento de devolução da cópia, ainda não é conhecido o objeto *empréstimo* ao qual a cópia emprestada se refere.
- Portanto, é preciso eleger alguma classe, que conheça os empréstimos, para receber a mensagem *devolverCopia*.
- Essa classe terá que identificar o objeto empréstimo cujo código de cópia seja igual ao parâmetro fornecido

48

Controlador

A pergunta anterior não está respondida nos slides a seguir. Voltaremos a ela no fim deste assunto.

49

Controlador

- **Problema:** Quem deve ser responsável por tratar um evento do sistema ?
- **Solução:** A responsabilidade de receber ou tratar as mensagens de eventos (operações) do sistema pode ser atribuída a uma classe que:
 - represente todo o sistema, um dispositivo ou um subsistema – chamado de controlador fachada - OU
 - represente um cenário de um caso de uso dentro do qual ocorra o evento
 - TratadorDe<NomeDoCasoDeUso>, ControladorDe<NomeDoCasoDeUso>

50

Controlador

- Exemplo: quem vai tratar os eventos do sistema de biblioteca?

Caso de uso: Emprestar Livro

51

Controlador

52

Exemplo: Opções de Controlador

- todo o sistema (controlador fachada): Biblioteca
- um tratador artificial do caso de uso: *ControladorDeEmprestarLivro*

53

Discussão: Controladores Fachada

- Um controlador fachada deve ser um objeto (do domínio) que seja o ponto principal para as chamadas provenientes da interface com o usuário ou de outros sistemas
 - pode ser uma abstração de uma entidade física – ex: *TerminalDeAtendimento*
 - pode ser um conceito que represente o sistema – ex: *Biblioteca*
- São adequados quando não há uma quantidade muito grande de eventos de sistema
- Não é possível redirecionar mensagens do sistema para controladores alternativos (ex: outros subsistemas)

54

Discussão :Controladores de Casos de Uso

- Deve existir um controlador diferente para cada caso de uso
 - Por exemplo, o `ControladorDeEmprestarLivro` será responsável pelas operações `iniciarEmprestimo`, `emprestarLivro` e `encerrarEmprestimo`
- Não é um objeto do domínio, e sim uma construção artificial para dar suporte ao sistema. Ex: `ControladorDeEmprestarLivro`, `ControladorDeDevolverLivro`
- Pode ser uma alternativa se a escolha de controladores fachada deixar a classe controladora com alto acoplamento e/ou baixa coesão (controlador inchado por excesso de responsabilidades)
- É uma boa alternativa quando existem muitos eventos envolvendo diferentes processos.

55

Controladores inchados

- Classe controladora mal projetada - inchada
 - coesão baixa – falta de foco e tratamento de muitas responsabilidades
- Sinais de inchaço:
 - uma única classe controladora tratando todos os eventos, que são muitos. Comum com controladores fachada
 - o próprio controlador executa as tarefas necessárias para atender o evento, sem delegar para outras classes (coesão alta, não especialista)
 - controlador tem muitos atributos e mantém informação significativa sobre o domínio, ou duplica informações existentes em outros lugares

56

Controlador

- Curas para controladores inchados
 - acrescentar mais controladores – misturar controladores fachada e de casos de uso
 - delegar responsabilidades
- **Corolário:** objetos de interface (como objetos “janela”) e da camada de apresentação não devem ter a responsabilidade de tratar eventos do sistema

57

Controlador

- Benefícios:
 - aumento das possibilidades de reutilização de classes e do uso de interfaces “plugáveis”.
 - conhecimento do estado do caso de uso – controlador pode armazenar estado do caso de uso, garantindo a sequência correta de execução de operações

58

