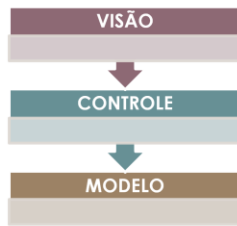


Arquitetura e Desenho de Software

AULA 10B



Profa. Milene Serrano



Agenda



Considerações Iniciais

Padrões GRASP (Continuação)

Considerações Finais

Considerações Iniciais



Considerações Iniciais

Daremos continuidade à apresentação dos padrões GRASPs...

- Polimorfismo
- Indireção
- Ivenção Pura
- Variações Protegidas

GRASP
Polimorfismo



GRASP – Polimorfismo

Problema: Como tratar alternativas com base no tipo do objeto?

- Como criar componentes de software interconectáveis? Se um projeto considerar a variabilidade de estruturas condicionais semelhantes a *if-then-else* ou *switch-case*, toda vez que surgir uma modificação no projeto implicará em uma alteração na lógica desses comandos. Isso torna difícil estender as funcionalidades do software. Além disso, o objeto determina a atividade a ser realizada. Como trocá-lo por outro sem que os demais elementos do projeto sejam afetados?

Esse padrão GRASP será muito utilizado quando forem utilizados os padrões GoFs comportamentais, tais como: Strategy e State.

GRASP – Polimorfismo

Solução: Quando os comportamentos dos objetos variarem conforme o tipo do objeto (ou da classe), atribua a responsabilidade dessas variações aos tipos usando operações polimórficas.

- Em tempo de execução, quando o tipo do objeto é conhecido, o adequado comportamento será disparado.

GRASP – Polimorfismo

Importante!

Não deve ser realizado teste para o tipo de objeto, nem mesmo condicional, visando executar alternativas de comportamento.

Vantagens:

Acoplamento baixo é apoiado, porque a classe criada provavelmente já é visível para a classe criadora em função das associações existentes entre elas;

Muito útil em projetos com variações semelhantes;

Maior facilidade em estender o projeto com novas funcionalidades, e

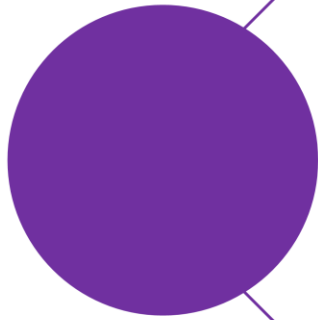
Novas funcionalidades podem ser introduzidas sem afetar os clientes.

GRASP – Polimorfismo



Debate sobre Comportamento *Default* em Classes Abstratas.
Quando usar?

GRASP – Polimorfismo



Outras Observações

- Polimorfismo implica no uso de classes abstratas ou interfaces em linguagens OO.
- Use interfaces sempre que não desejar se comprometer com uma hierarquia de classes em particular.
- Interfaces e classes abstratas serão pontos de evolução flexíveis em seu projeto de software.
- Somente utilize essa estratégia em pontos, nos quais haja uma possibilidade de variabilidade e aumento do comportamento do software.

Debate sobre Interface <<implements>> e Herança com uso de Classe Abstrata <<extends>>.

GRASP – Polimorfismo

```
// bad design  
SWITCH ON square.type
```

```
CASE GoSquare: player receives $200  
CASE IncomeTaxSquare: player pays tax
```



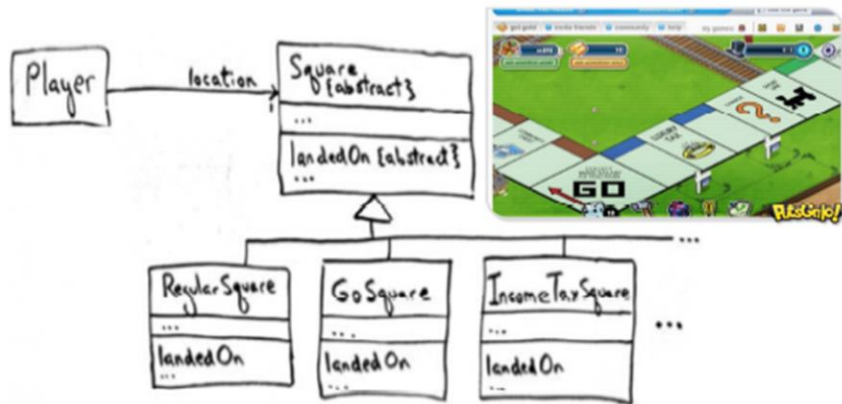
Projeto Ruim:

Comportamento sendo determinado pelo tipo usando um switch case como recurso.

Solução ruim...

Por que? Debater sobre...

GRASP – Polimorfismo



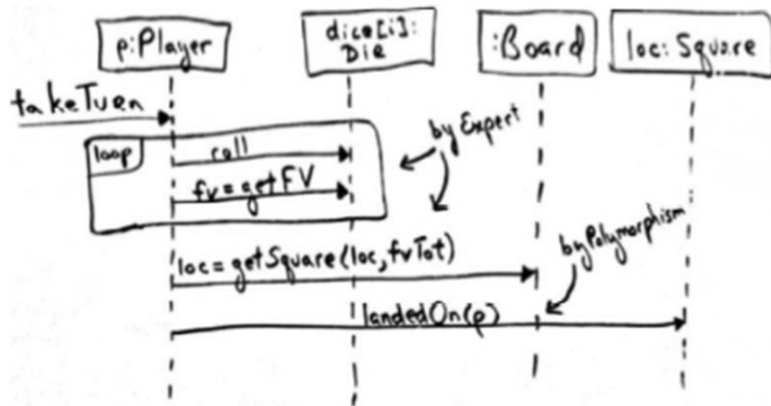
Melhor:

Polimorfismo aplicado no projeto – método *landedOn* sobrescrito

Solução mais adequada...

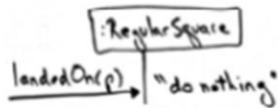
Por que? Debater sobre...

GRASP – Polimorfismo



Polimorfismo visualizado no Diagrama de Sequência

GRASP – Polimorfismo



Polimorfismo evidenciado...

mesmo método, diferentes comportamentos,
definidos conforme o tipo do objeto.

Continuação...

GRASP
Invenção Pura



GRASP – Invenção Pura

Problema: Uma responsabilidade deve ser alocada a um objeto (geralmente sugerida pelo padrão Especialista), mas irá atrapalhar a Coesão e/ou o Acoplamento daquele objeto.

- Problemas deste tipo são comuns ao atribuir responsabilidades aos objetos de software pertencentes à camada de domínio. Por exemplo: o objeto da camada de domínio representa uma entidade do mundo real, mas sua coesão diminui ao realizar atividades típicas do mundo de software.

GRASP – Invenção Pura

Solução: Atribuir um conjunto de responsabilidades altamente coeso a uma classe artificial ou de conveniência e que não represente um conceito do domínio do problema. É uma classe inventada para apoiar coesão alta, acoplamento baixo e reutilização de software.

GRASP – Invenção Pura

Importante!

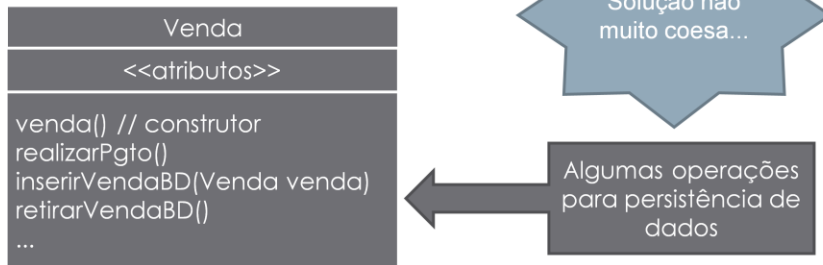
Deve-se ter cuidado no momento de agrupar comportamentos e criar as classes de "Invenção Pura", para não criar muitas classes. Assim, o comportamento estará "pulverizado" pelas classes. É necessário equilibrar número de classes e distribuição das responsabilidades.

Vantagens:

Aumento da coesão à medida em que a nova classe agrega funcionalidades altamente correlatas.

Aumento da capacidade de reutilização decorrente de classes de Invenção Pura que agrupam comportamentos muito utilizados.

GRASP – Invenção Pura



Considere que informações associadas à uma classe Venda, em um sistema, precisam ser persistidas em banco de dados. Pelo Especialista da Informação, sugere-se que ela mesmo realize esta atividade de persistência, pois possui todos os dados a serem gravados.

Mas, isso pode gera problemas...

GRASP – Invenção Pura

Principais Problemas

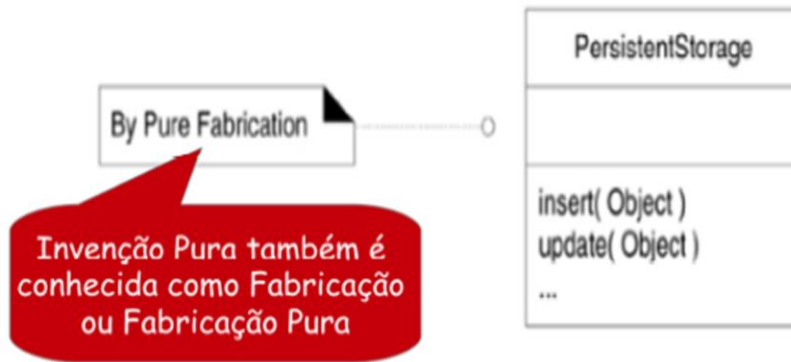
Coesão mais baixa: um grande número de operações deve ser realizado para apoio à persistência. Tais tarefas não têm relação direta com venda.

Acoplamento mais alto: classe Venda necessariamente tem que estar acoplada à uma interface especialista de banco de dados.

Menor reusabilidade: atividade de persistência é uma atividade geral que requer muitos métodos para ser implementada.

**** Solução:** Criar uma classe que seja a única responsável por realizar a persistência de dados em algum tipo de armazenamento persistente.

GRASP – Invenção Pura



Aplicação de Invenção Pura: Criou-se uma classe com atividades muito relacionadas (alta coesão), baixo acoplamento, e que poderá ser reutilizada por diversos elementos participantes do projeto.

Cuidados <<debater sobre>>:

- Usar essa estratégia em conformidade com as boas práticas de cada tecnologia.
- Dependendo da tecnologia, se a comunidade já utiliza uma estratégia – que difere dessa prática – optar por não realizar essa separação das operações de persistência.
- Estar de acordo com os padrões já reconhecidos por uma comunidade de programação – por exemplo – é mais relevante do que aplicar um padrão GRASP só para constar ou dizer que está usando-o.

GRASP
Indireção



GRASP – Indireção

Problema: Como distribuir responsabilidades entre objetos de modo a evitar acoplamento direto entre dois (ou mais) objetos?

- Como desacoplar os objetos (enfraquecer o acoplamento) e aumentar as chances de reutilização?

GRASP – Indireção

Solução: Utilizar um objeto intermediário entre dois (ou mais objetos), de modo que este objeto seja um mediador entre eles, evitando um acoplamento direto. O objeto intermediário cria uma indireção entre os outros dois.

- A ideia deste padrão é utilizar o objeto intermediário como uma proteção para o objeto cliente de futuras variações no objeto que está sendo utilizado.

GRASP – Indireção

Importante!

Muitos intermediários da Indireção são Invenções Puras.

Vantagens:

Redução do acoplamento à medida em que a associação entre dois objetos – dependente de muitas requisições (por exemplo) – é feita por um intermediário.

GRASP – Indireção

EXEMPLO

O exemplo apresentado na seção de Invenção Pura é também um exemplo de Indireção.

A medida em que você adiciona uma classe responsável por estabelecer uma comunicação entre os objetos de domínio e o banco de dados, procura-se diminuir o acoplamento entre as camadas de domínio e de serviços.

Portanto, os intermediários das indireções costumam ser invenções puras. Mas, nem toda invenção pura é indireção.

GRASP
Variações Protegidas



GRASP – Variações Protegidas

Problema: Como projetar objetos, subsistemas e sistemas de modo que as variações ou instabilidades nesses elementos não tenham impacto indesejável sobre outros elementos?

GRASP – Variações Protegidas

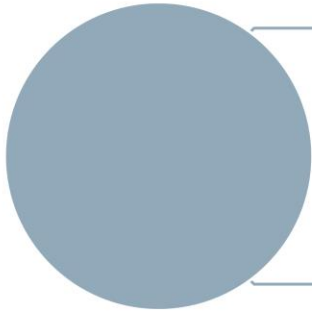
Solução: Identificar pontos de variação ou instabilidade previsíveis e atribuir responsabilidades para criar uma interface estável em torno deles.

GRASP – Variações Protegidas

Importante!

- Este padrão também é conhecido como ocultação da informação.
- É importante notar que a proteção às variações ocorre por meio da diminuição do acoplamento. Quanto menos acoplado forem dois objetos, menor será o impacto de alterações.
- A princípio, as variações protegidas são consideradas nas interfaces dos menores elementos de um projeto: classes e objetos. Contudo, elas podem evoluir para mecanismos mais avançados (os exemplos fogem do escopo de DSW, mas uma noção básica deles encontra-se na página 436 do Larman): **Projetos dirigidos por dados – Pesquisa de serviço – Projetos dirigidos por regras – Projetos reflexivos – Acesso uniforme a métodos e atributos – Linguagens normalizadas.**

GRASP – Variações Protegidas



Vantagens:

- Facilidade em adicionar extensões para novas variações;
- Novas implementações podem ser introduzidas sem afetar os clientes;
- O acoplamento fica mais baixo, e
- O impacto ou custo de modificações pode ser diminuído.

GRASP – Variações Protegidas

EXEMPLO

No sistema de venda, o cálculo de imposto pode ser realizado por diferentes entidades e outras entidades podem ser adicionadas futuramente.

Trata-se, portanto, de um ponto de instabilidade e variabilidade.

A variabilidade está nos diferentes elementos calculadores de impostos, cada um realizando os cálculos à sua maneira.

GRASP – Variações Protegidas



Por meio do padrão Indireção (ao criar uma interface entre os elementos que realizam cálculos) e Polimorfismo (ao definir uma operação polimórfica), consegue-se uma proteção contra variações. Esta proteção se dá, pois adaptadores internos contribuem para uma interface mais estável, ocultando as variações para os objetos externos ao projeto.

GRASP – Ocultação da Estrutura



É o avanço do princípio de Variação Protegida, também conhecida como Não fale com estranhos...

Resumidamente, deve-se evitar percorrer caminhos longos em estruturas de objetos enviando mensagens a objetos distantes e indiretos (estranhos).

GRASP – Ocultação da Estrutura

Boas Práticas

Preferencialmente, um método deve "interagir" com:

- O próprio objeto (this ou self);
- Um parâmetro do método;
- Um atributo de this;
- Um elemento de uma coleção que seja um atributo de this, ou
- Um objeto criado dentro do método.

GRASP – Ocultação da Estrutura

```
class Registradora
{
    private Venda venda;


    public void metodoLigeiramenteFragil()
    {
        // venda.obterPagamento() envia mensagem para um "familiar" (cumpre regra #3)
        // venda.obterPagamento().obterQuantiaEntregue() percorre um caminho longo...
        Moeda quantia = venda.obterPagamento().obterQuantiaEntregue();

        // ...
    }
    // ...
}
```

Método Ligeiramente Frágil

GRASP – Ocultação da Estrutura

```
public void metodoMaisFragil()  
{  
    TitularDaConta portador =  
        venda.obterPagamento().obterConta().obterTitularDaConta();  
  
    // ...  
}
```



Método Frágil

Extra Classe



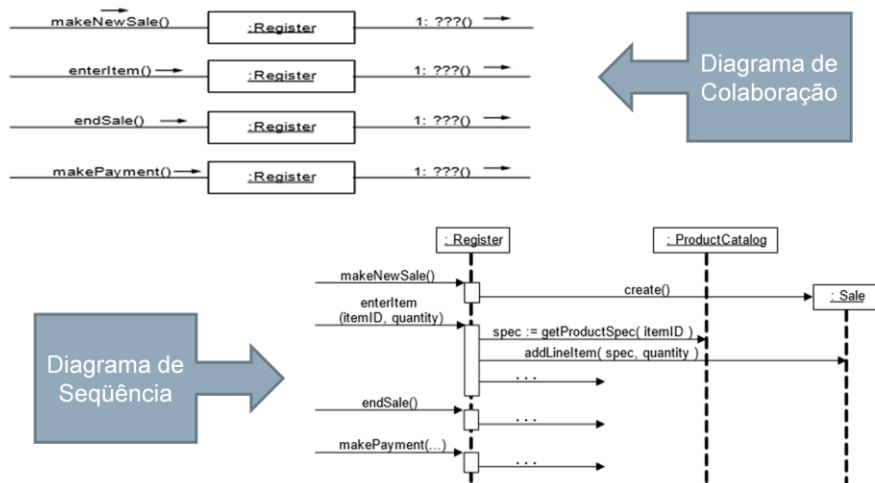
Extra Classe

Um sistema de vendas com as operações:

- *makeNewSale* – realizarNovaVenda,
- *enterItem* – registrarItem,
- *endSale* – finalizarVenda, e
- *MakePayment* – realizarPagamento.

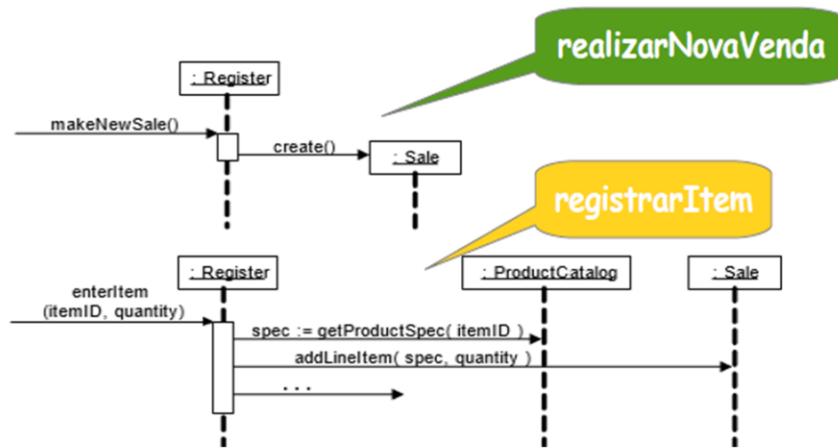
Extra Classe

Ambos os diagramas oferecem informações importantes para atribuir corretamente as responsabilidades.



Ambos os diagramas oferecem informações importantes para atribuir corretamente as responsabilidades.

Extra Classe



Utilizando múltiplas visões...

Para facilitar, dentro do possível, trabalhem com múltiplas visões...

Lembrando que uma modelagem deve ser clara.

Caso contrário, não será utilizada pelos interessados.

Extra Classe

Outros artefatos, mais detalhados como os Contratos de Operações, podem dar informações relevantes...

Operation: Cross
References:
Preconditions:
Postconditions:

enteritem(itemID : ItemID, quantity : integer) Use

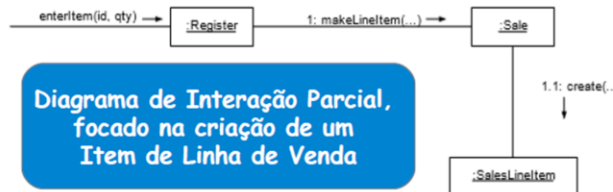
Cases: Process Sale There is a sale underway.

- A SalesLineItem instance sli was created (instance creation).

Especial atenção às pré e pós-condições...

✓ Uma instância de *SalesLineItem*, sli, foi criada...

Criação de Instância



Lembrando que *Sale* e *SalesLineItem* tem relação de TODO-PARTE, conforme visto em aulas anteriores.

Portanto, quem “dispara/autoriza” a criação de uma instância da PARTE (*SalesLineItem*) é o TODO (*Sale*).

Essa amarração pode ser implementada diretamente nos construtores de *Sale* e *SalesLineItem*.

Extra Classe

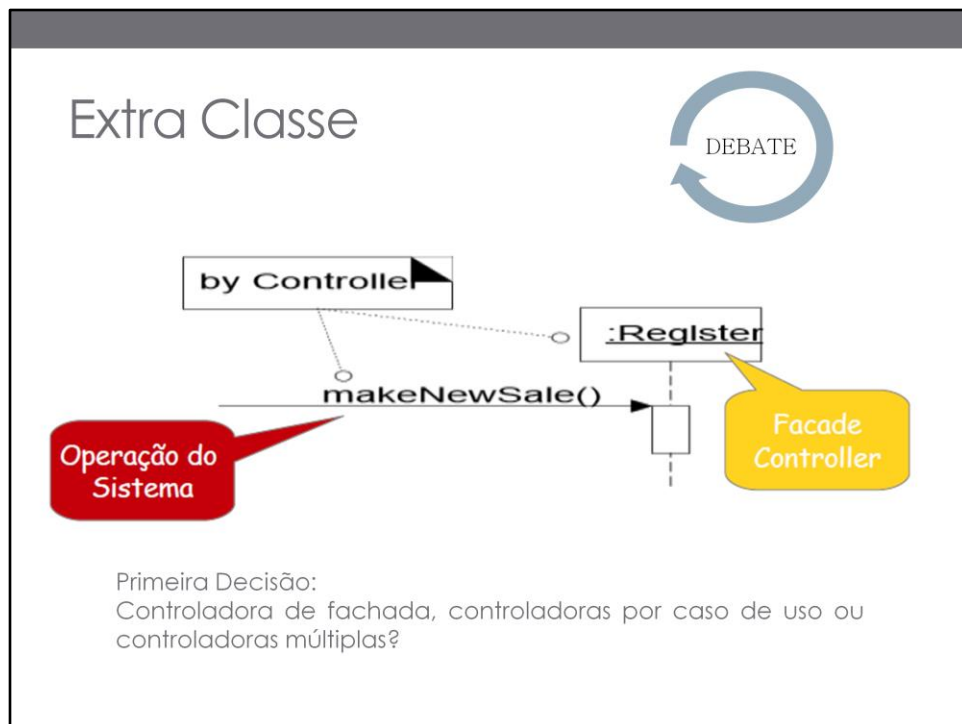
Contract CO1: makeNewSale

Operation:	Cross	makeNewSale()
References:		Use Cases: Process Sale
Preconditions:		none

Postconditions:	<ul style="list-style-type: none">- A Sale instance s was created (instance creation).- s was associated with the Register (association formed).- Attributes of s were initialized.
------------------------	---

Operação: makeNewSale()

Contrato de operação específico para a Operação do Sistema *makeNewSale()*.



Opta-se por controladora de fachada, pois o sistema de ponto de vendas é simples, com poucas operações de sistema.

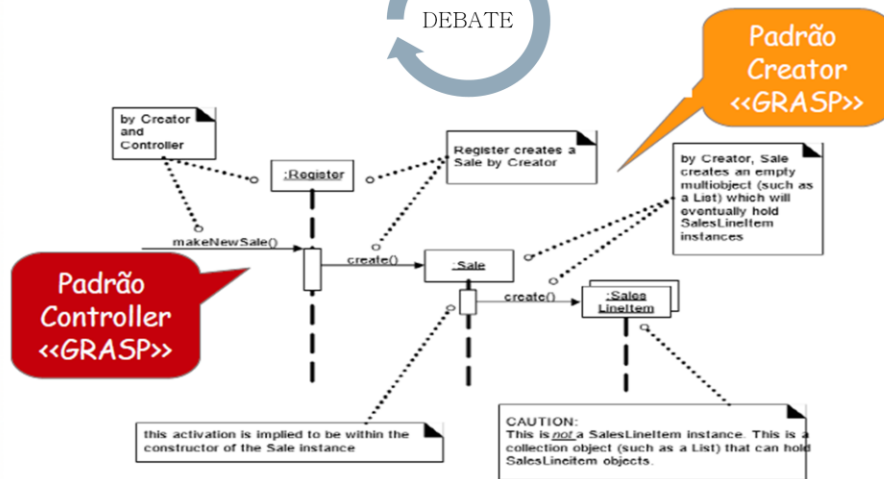
Nesse caso, a controladora não ficará sobrecarregada com as requisições demandadas pelo sistema em questão.

Mas, em uma visão mais moderna, a maioria dos sistemas utiliza controladoras múltiplas.

Visto, por exemplo, o desenvolvimento em plataformas mais emergentes (ex. Ruby On Rails e Grails).

Nesses casos, temos pelo menos uma controladora por entidade da camada Model.

Extra Classe



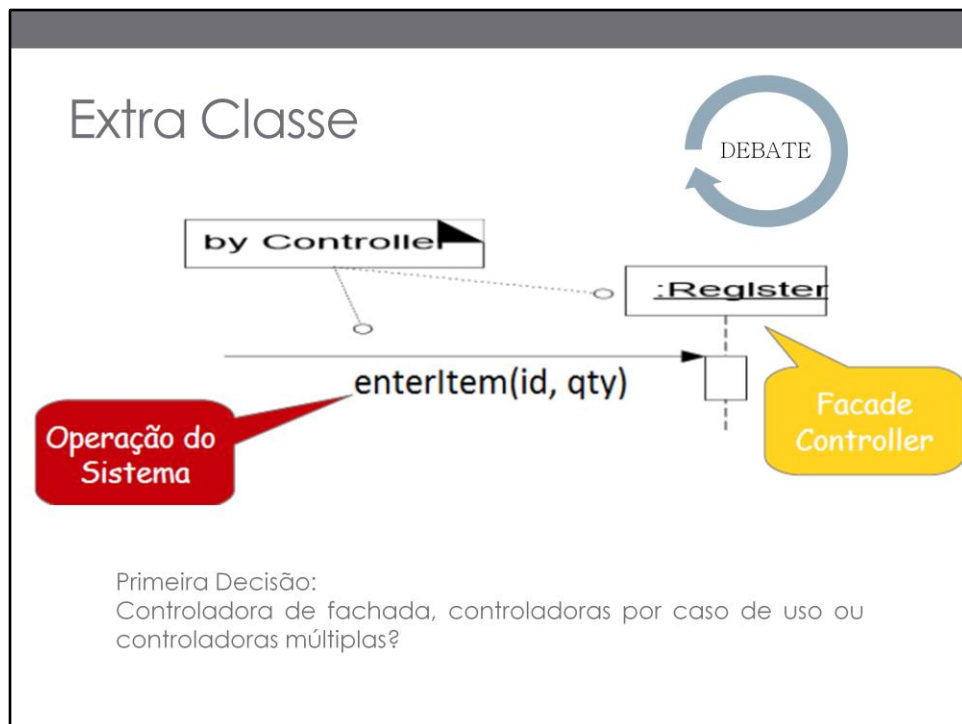
Segunda Decisão:
Como atribuir adequadamente as responsabilidades?

Extra Classe

Contract CO2: enterItem

Operation: Cross	enterItem(itemID : ItemID, quantity : integer) Use
References:	Cases: Process Sale There is an underway sale.
Preconditions:	
Postconditions:	<ul style="list-style-type: none">- A SalesLineItem instance sli was created (instance creation).- sli was associated with the current Sale (association formed).- sli.quantity became quantity (attribute modification).- sli was associated with a ProductSpecification, based on itemID match (association formed).

Operação: enterItem(...)



Opta-se por controladora de fachada, pois o sistema de ponto de vendas é simples, com poucas operações de sistema.

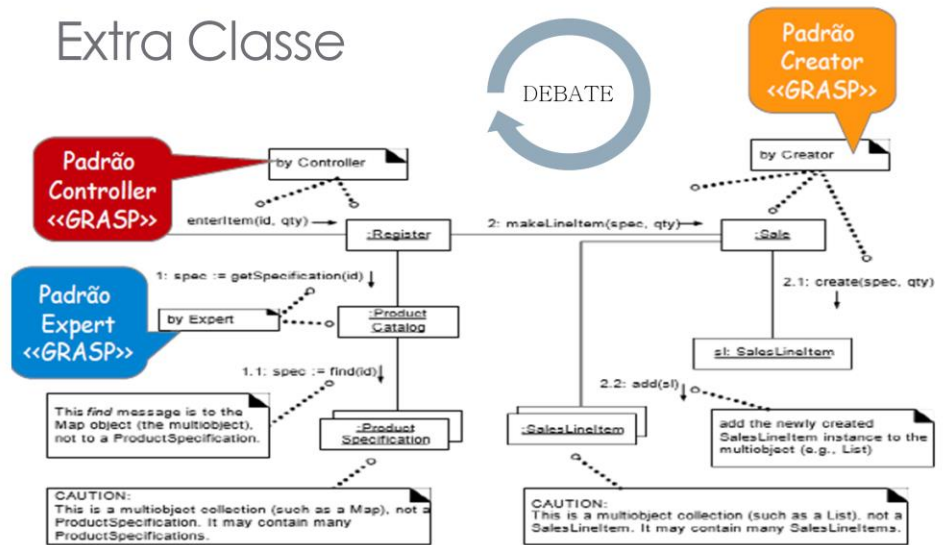
Nesse caso, a controladora não ficará sobrecarregada com as requisições demandadas pelo sistema em questão.

Mas, em uma visão mais moderna, a maioria dos sistemas utiliza controladoras múltiplas.

Visto, por exemplo, o desenvolvimento em plataformas mais emergentes (ex. Ruby On Rails e Grails).

Nesses casos, temos pelo menos uma controladora por entidade da camada Model.

Extra Classe



Segunda Decisão:
Como atribuir adequadamente as responsabilidades?

Considerações Finais



Considerações Finais

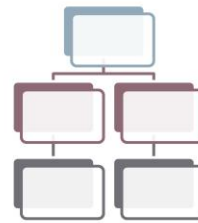
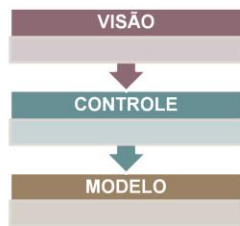
- Nessa aula, novos padrões GRASPs foram introduzidos.
- Continuem os estudos! Só se aprende praticando!
- Nas referências, têm vários materiais complementares! :)

Referências



Referências

- LARMAN, Craig. Utilizando UML e Padrões: Uma Introdução a Análise e ao Projeto
- Orientado a Objetos. 3a. edição. Bookman, 2007.
- COCKBURN, Alistair. Escrevendo Casos de Uso Eficazes. Bookman, 2005.
- SILVA, Ricardo Pereira. UML 2 em Modelagem Orientada a Objetos. Visual Books, 2007.
- PRESSMAN, Roger S. Engenharia de Software. 6a. edição . McGraw-Hill, 2006.
- IEEE. SWEBOK-Guide to the Software Engineering Body of Knowledge, 2004.
- SOMMERVILLE, Ian. Engenharia de Software. 8a. edição. Pearson, 2007.



FIM

Dúvidas?

CONTATO:
mileneserrano@unb.br
ou
mileneserrano@gmail.com

Sugestões?

