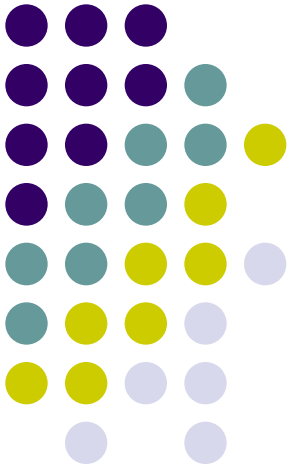
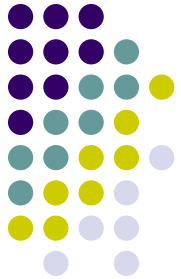
Padrões de Projeto

Prof. Esp. Wagner Mendes Voltz
wagnerfusca@gmail.com

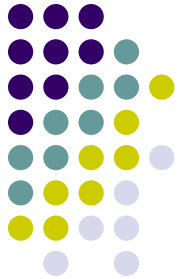


Orientação a Objetos (OO)



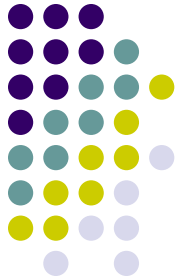
- O que é OO?
- Quando programo em Java/PHP, sempre uso OO?
- OO depende de linguagem?
- OO é difícil de entender?
- Sem diagramas UML não consigo fazer OO?
- OO resolve tudo?

Orientação a Objetos (OO)



- Paradigma que reflete problemas reais
- Utilizam a abstração de objetos para representar entidades do mundo real
- Organizar e escrever menos
- Concentrar as responsabilidades nos pontos certos
- Fexibilizando sua aplicação, encapsulando a lógica de negócios.

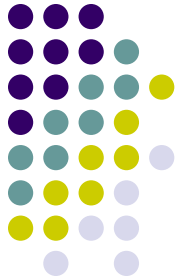
Orientação a Objetos (OO)

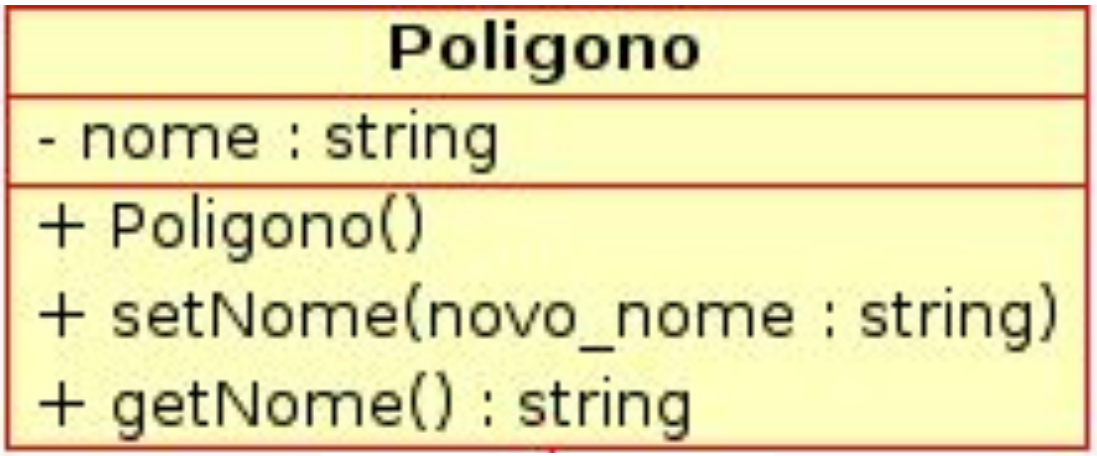


- Classe
  - conjunto de objetos com características afins
- Atributo
  - Características do objeto
- Métodos
  - comportamento/função de um objeto na classe
- Objeto
  - Instância da classe

http://bimg2.mlstatic.com/coleco-completa-carros-classicos-nacionais-serie-2-12-uni_MLB-F-237368984_856.jpg

Classe – notação UML





http://www.metropoledigital.ufrn.br/aulas/disciplinas/poo/imagens/a01_figura4.jpg

http://www.desenvolvimentosoftware.com/wordpress/wp-content/uploads/2013/01/orientacao-a-objetos.png

Abstração

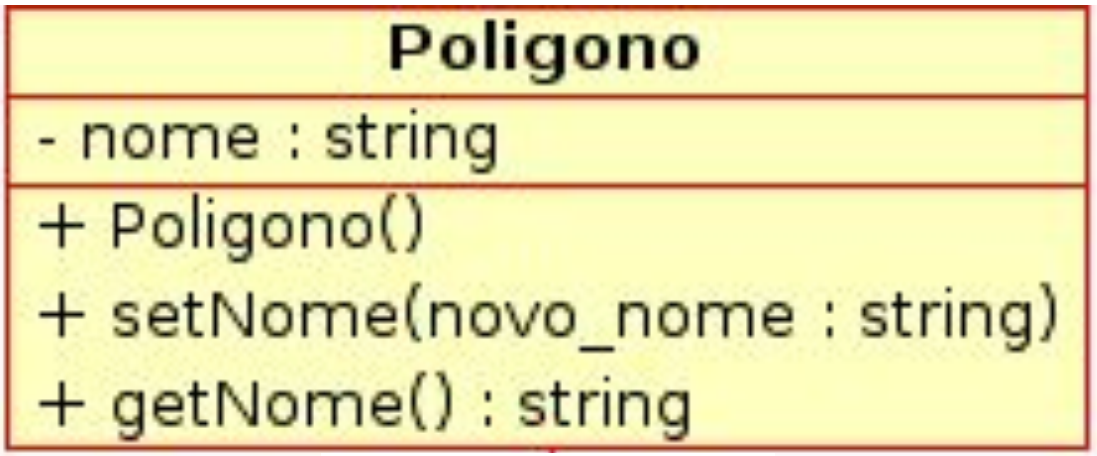
- O que um objeto faz
- Não importa como ele faz

Encapsulamento

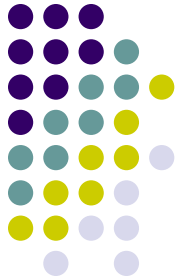
- Esconder informações desnecessárias
- Separa detalhes internos dos externos
- Permite que um objeto possa ser modificado sem afetar a aplicação

Encapsulamento

- #protected
- -private
- +public

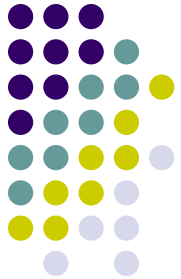


Composição e Associação



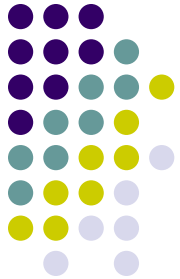
Descrevem uma relação entre classes.

Composição e Associação



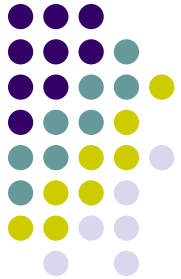
- Formada entre dois objetos quando um deles contém uma referência ao outro.
- A referência é geralmente armazenada como uma variável de instância
- Pode ser uni ou bidirecional

Composição e Associação



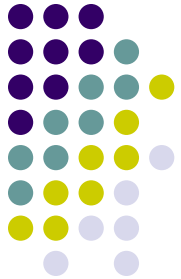
- Associação direta – TEM UM
- Associação por composição – composto por
- Associação por agregação – uma parte
- Associação temporária - dependência
- 1 – 1
- 1 – N
- N - N

Herança



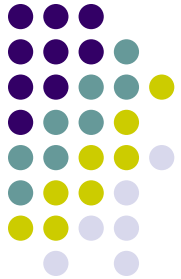
- Superclasse x subclasse
- (herança = é um)

Classe Abstrata



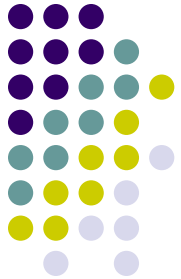
- objetos mais genéricos e que sejam base para as demais classes.
- Uma base para outras classes do que uma classe com instâncias específicas que você quer utilizar.

Método Abstrato



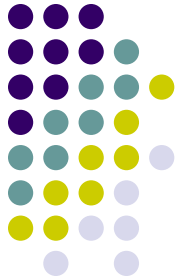
- Funcionam como marcadores de lugar para métodos que são implementados nas subclasses
- Definem a estrutura do método na superclasse e passam a responsabilidade de implementação e personalização do método para as subclasses.

Interface



- Se a classe não tiver nenhum método não abstrato, podemos criá-la como uma interface
- Segue um modelo de declaração diferente do usado para classes mas tem funcionalidade similar à de classes abstratas.

Interface e método abstrato



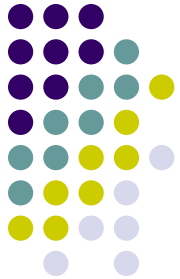
A diferença essencial entre classes abstratas e interfaces é que uma classe que usa herança somente pode herdar uma única classe (abstrata ou não), enquanto qualquer classe pode implementar várias interfaces simultaneamente.


C O
D E

Polimorfismo

- Implementado pelo uso de herança, classes abstratas, métodos abstratos e interfaces

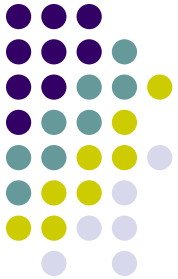
Acoplamento



- é o nível de inter-dependência entre os módulos de um software.
- está diretamente relacionado com a coesão
- quanto maior for o acoplamento menor será o nível de coesão.
- Lei de Demeter (LoD) (encapsulamento)
  - Não devemos encadear várias chamadas de métodos para chegar a classe que queremos “conversar”
- Tell, Don’t Ask

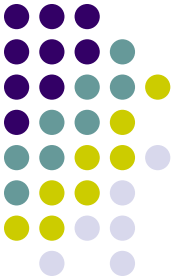
http://wpjr2.wordpress.com/2008/04/22/coesao-e-acoplamento/

Coesão



- A separação de responsabilidades em classes
- considerado um princípio do paradigma, conhecido como *Single Responsibility Principle*.
- uma classe deve ter somente uma responsabilidade e que uma responsabilidade deve estar encapsulada inteiramente em uma classe.
- deixa nosso código mais simples,
- fica mais fácil saber qual parte do código executa cada lógica e onde precisamos mexer para alterar um determinado comportamento.

Princípios comuns de design



- **Keep It Simple Stupid (KISS)**
  Mantenha Isto Estupidamente Simples
- **Don’t Repeat Yourself (DRY)**
  Não Repita Você Mesmo
- **Tell, Don’t Ask**
  Fale, não pergunte
- **You Ain’t Gonna Need It (YAGNI)**
  Você Não Vai precisar Disso
- **Separation Of Concerns (SoC)**
  Separação de Responsabilidades

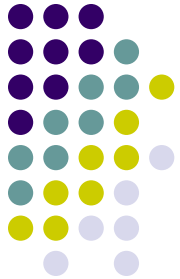
Keep It Simple Stupid (KISS)

- Mantenha Isto Estupidamente Simples
- manter o código simples, mas não simplista, assim evitando complexidade desnecessária.
- complicar demais a solução = problema.

KISS

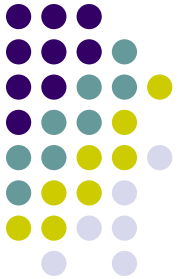
Keep. It. Simple. Stupid.

Don’t Repeat Yourself (DRY)



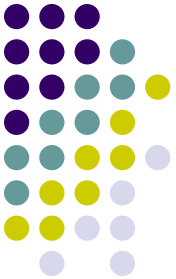
- **Não Repita Você Mesmo**
- evitar a repetição de qualquer parte do sistema abstraindo as coisas que são comuns entre si e colocá-las em um lugar único.
- Esse princípio não se preocupa somente com o código, mas qualquer lógica que está duplicada no sistema.

Tell, Don’t Ask



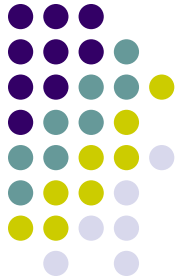
- **Fale, não pergunte**
- Encapsulamento e responsabilidade
- você deve dizer aos objetos quais ações você quer que eles realizem, ao invés de fazer perguntas sobre o estado do objeto e então tomar uma decisão por si próprio em cima da ação que você quer realizar.
- BdClass.Connection.close();
- BdClass.FecharConexao();

You Ain’t Gonna Need It (YAGNI)

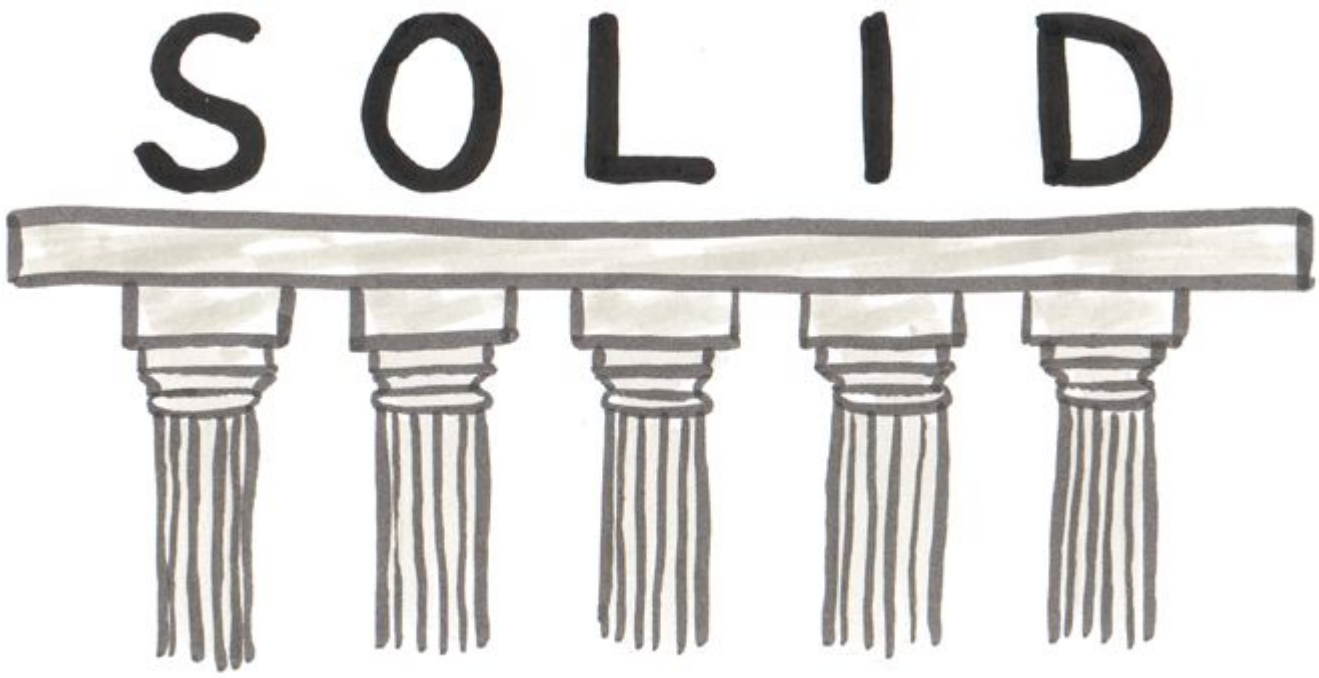


- **Você Não Vai precisar Disso**
- adicionar somente as funcionalidades que são necessárias para a aplicação
- deixar de lado qualquer tentação de adicionar outras funcionalidades que você acha que precisa.
- test-driven development (TDD) – desenvolvimento orientado a testes.

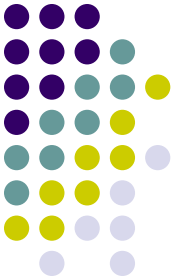
Separation Of Concerns (SoC)



- **Separação de Responsabilidades**
- dissecação de uma parte de software em distintas características que encapsulam um único comportamento e dados que podem ser utilizados por outras classes.
- aumenta significativamente a reutilização de código, manutenção e testabilidade.

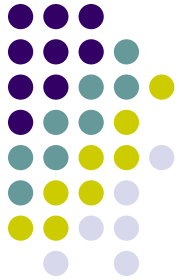

SOLID

SOLID



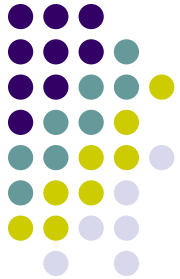
- **[S]**ingle Responsability Principle/Princípio da Responsabilidade Única (SRP)
- **[O]**pen/Closed Principle/Princípio do Aberto/Fechado (OCP)
- **[L]**iskov Substitution Principle/Princípio da Substituição de Liskov (LSP)
- **[I]**nterface Segregation Principle/Princípio da Segregação de Interface (ISP)
- **[D]**ependency Inversion Principle/Princípio da Inversão de Dependência (DIP)

Princípio da Responsabilidade Única



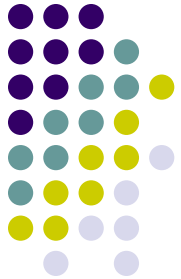
- **Uma classe deve ter somente uma razão para mudar**
- **Coesão**
- **Sintomas de quebra do principio:**
  - Dificuldade de compreensão e, portanto, dificuldade de manutenção.
  - Dificuldade de reuso.
  - Alto acoplamento (classe tem um número excessivo de dependências)

Princípio do Aberto/Fechado



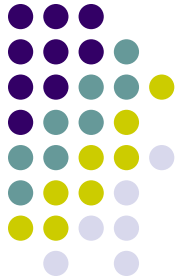
- Entidades de software (classes, módulos, funções, etc.) devem ser **abertas** para extensão mas **fechadas** para modificação
- quando eu precisar estender o comportamento de um código, eu crio código novo ao invés de alterar o código existente
- Abstração
- Exemplo: Classe ArquivoTxt, ArquivoPDF, ArquivoXLS + ifs

Princípio da Substituição de Liskov



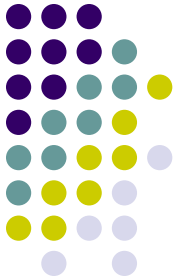
- Esse princípio diz que precisamos ter cuidado para usar herança
- Exemplo: uma herança com sub classes com métodos independentes, ferindo o polimorfismo

Princípio da Segregação de Interface



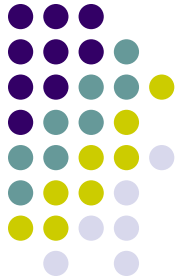
- clientes não devem ser forçados a depender de métodos que não usam.
- nossos módulos devem ser enxutos, ou seja, devem ter poucos comportamentos.
- Interfaces que tem muitos comportamentos geralmente acabam se espalhando por todo o sistema, dificultando manutenção.

Princípio da Inversão de Dependência

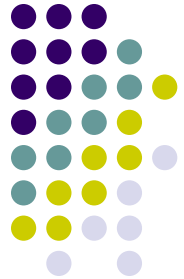


- Esse princípio diz que devemos sempre depender de abstrações, afinal abstrações mudam menos e facilitam a mudança de comportamento e as futuras evoluções do código.

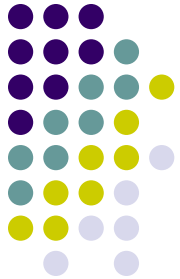
Referencias SOLID



- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf



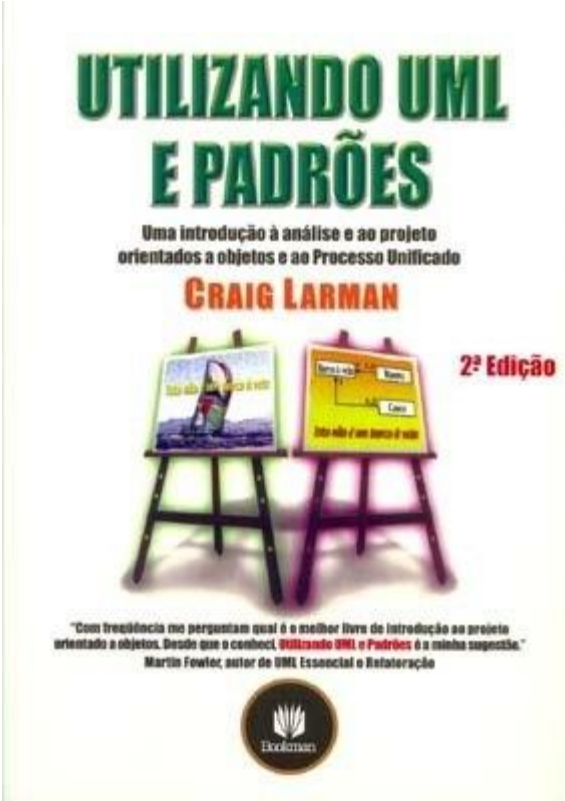
O que é um padrão de projeto?

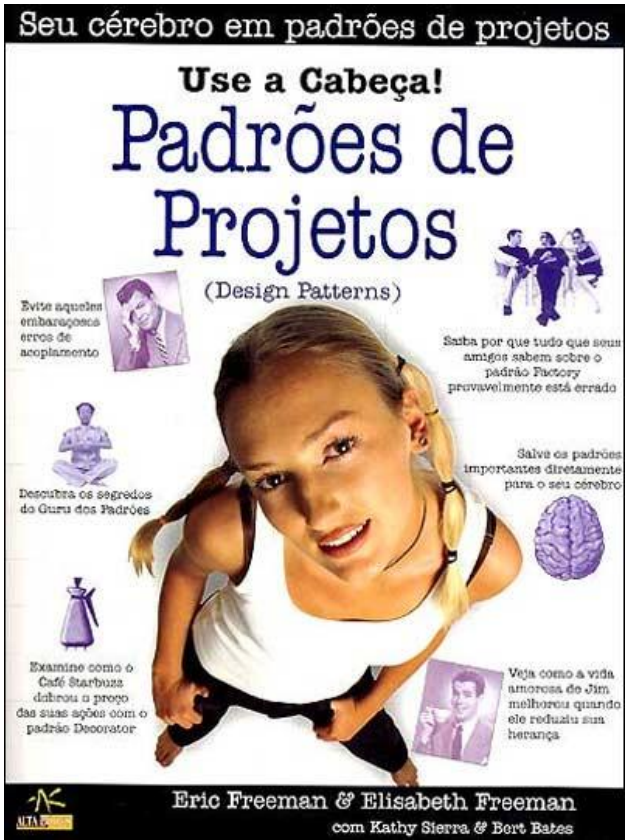


- Maneira testada ou documentada de alcançar um objetivo qualquer
- Chamados de design pattern
- *"Os Design Patterns **não exigem nenhum recurso incomum da linguagem**, **nem truques de programação** surpreendentes para impressionar seus amigos e gerentes." (GoF, 1995, prefácio vii)*

Padrões

- 23 padrões clássicos (GoF)
- 9 padrões de atribuições de responsabilidade (GRASP)





Design Patterns GRASP	
Padrões Fundamentais	Controller
	Creator
	Information Expert
	Low Coupling
	High Cohesion
Padrões Avançados	Indirection
	Protected Variations
	Polymorphism
	Pure Fabrication

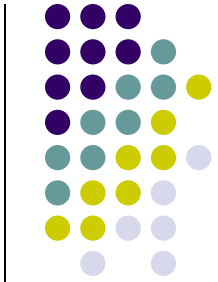
- http://ferhenriquef.files.wordpress.com/2012/08/grasp1.png

Classificação dos 23 padrões segundo GoF

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

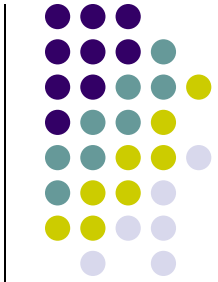
http://www.devmedia.com.br/imagens/articles/226729/Classificacao%20gof.jpg

Outros design pattern



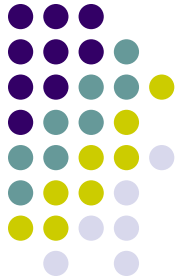
- Antipadrões
- Padrões SOA
- Padrões CoreJava
- Padrões security
- Padrão de arquitetura (MVC)

O padrão deve ter:



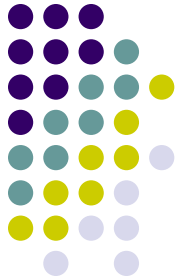
- Nome
- Problema que soluciona
- Solução do problema

Exemplo

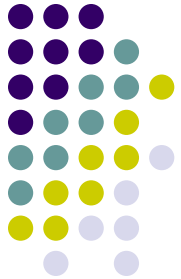


- Nome: moedor de bataba
- problema: como é o processo de moer batatas?
- Solução: Lavar as batatas é importante e descasca-las também. Somente pressionando-as, após lavagem e descacagem é que se chega ao ponto ideal.

Responsabilidade



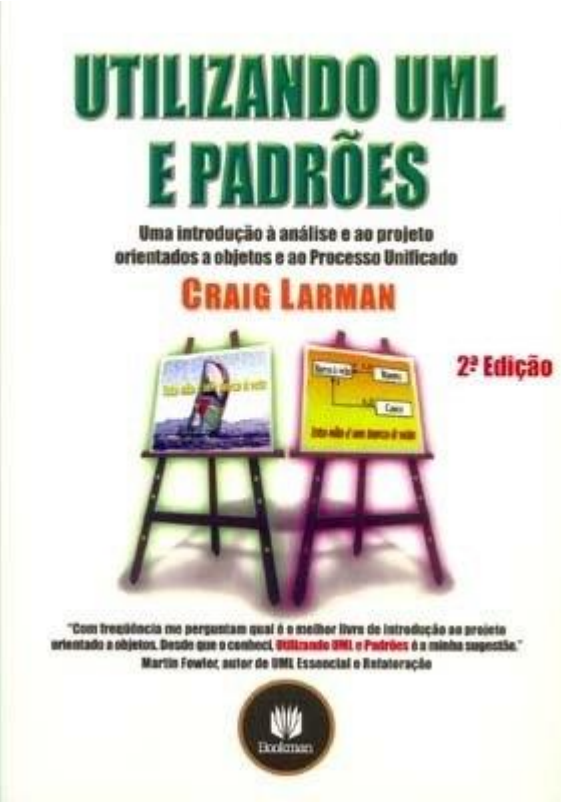
- “é um contrato ou obrigação de um classificador” [Booch e Rumbaugh]
- Estão relacionadas com as obrigações de um objeto em **termos de comportamento**
- Tipos:
  - Conhecer
  - Fazer



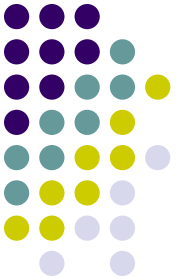
- Responsabilidade não é a mesma coisa que um método.
- Métodos são implementados para satisfazer as responsabilidades
- Exemplos
  - Um objeto Venda tem a responsabilidade de criar um item de venda (fazer)
  - Um objeto Venda tem a responsabilidade de saber valor total (conhecer)

Padrão GRASP

- Padrões de Princípios Gerais para Atribuição de Responsabilidade
- Introduzidos por Craig Larman em seu livro “Applying UML and Patterns”
- ***Descrevem os princípios fundamentais de projeto baseado em objetos e atribuição de responsabilidade aos mesmos***



Padrão GRASP



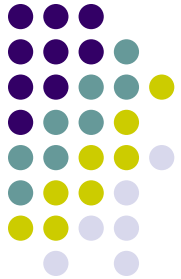
fundamentais

- Criador (creator)
- Especialista na Informação (Information Expert)
- Acoplamento Baixo (Low coupling)
- Controlador (Controller)
- Coesão Alta (high cohesin)

Design Patterns GRASP	
Padrões Fundamentais	Controller
	Creator
	Information Expert
	Low Coupling
	High Cohesion
Padrões Avançados	Indirection
	Protected Variations
	Polymorphism
	Pure Fabrication

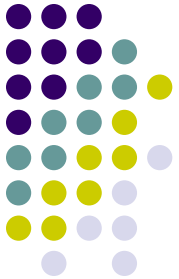
- http://ferhenriquef.files.wordpress.com/2012/08/grasp1.png

Especialista na Informação



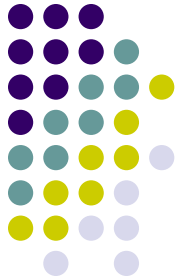
- Atribuir uma responsabilidade ao especialista na informação: a classe que tem a informação necessária para satisfazer a responsabilidade

Especialista na Informação



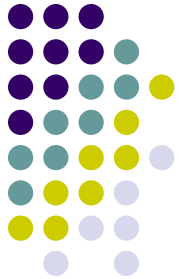
- Exemplo:
  - Num sistema ERP, alguma classe precisa conhecer o total geral de uma venda.
  - Quem deve ser o responsável por conhecer o total geral de uma venda?

Desafio



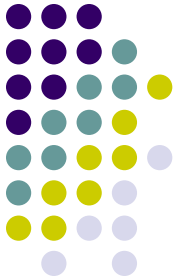
- usando um software para modelagem UML, faça (usando diagrama de classes):
- Crie as seguintes classes: Venda, ItensVenda, Produto, Tabela Preco
  - ***obs: no nosso caso, cada item pode ter uma tabela de preço***
- Quem deve ser o responsável por conhecer o subtotal de uma venda?
- Quem deve ser o responsável pelo preço do produto?
- responda com uma anotação

Desafio



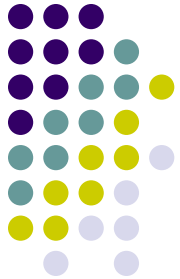
- Imagine as seguintes classes: Venda, ItensVenda, Produto, TabelaPreco
  - ***obs: no nosso caso, cada item pode ter uma tabela de preço***
- Quem deve ser o responsável por conhecer o subtotal de uma venda?
- Quem deve ser o responsável pelo preço do produto?

Especialista na Informação



- Benefícios:
  - Encapsulamento é mantido
  - Acoplamento fraco
  - Comportamento distribuído entre as classes
  - Coesão Alta

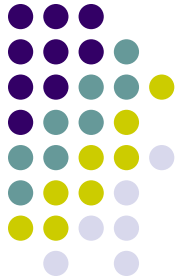
Criador (creator)



Como descobrir qual classe é responsável por criar objetos? Quando classe B deve crirar instância da classe A?

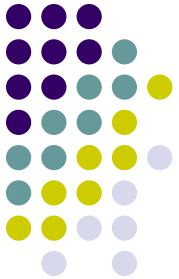
- Instâncias de B contêm ou agregam instâncias de A;
- Instâncias de B gravam instâncias de A;
- Instâncias de B utilizam de perto instâncias de A;
- Instâncias de B têm as informações de iniciação das instâncias de A e passam isso na criação.

Criador (creator)

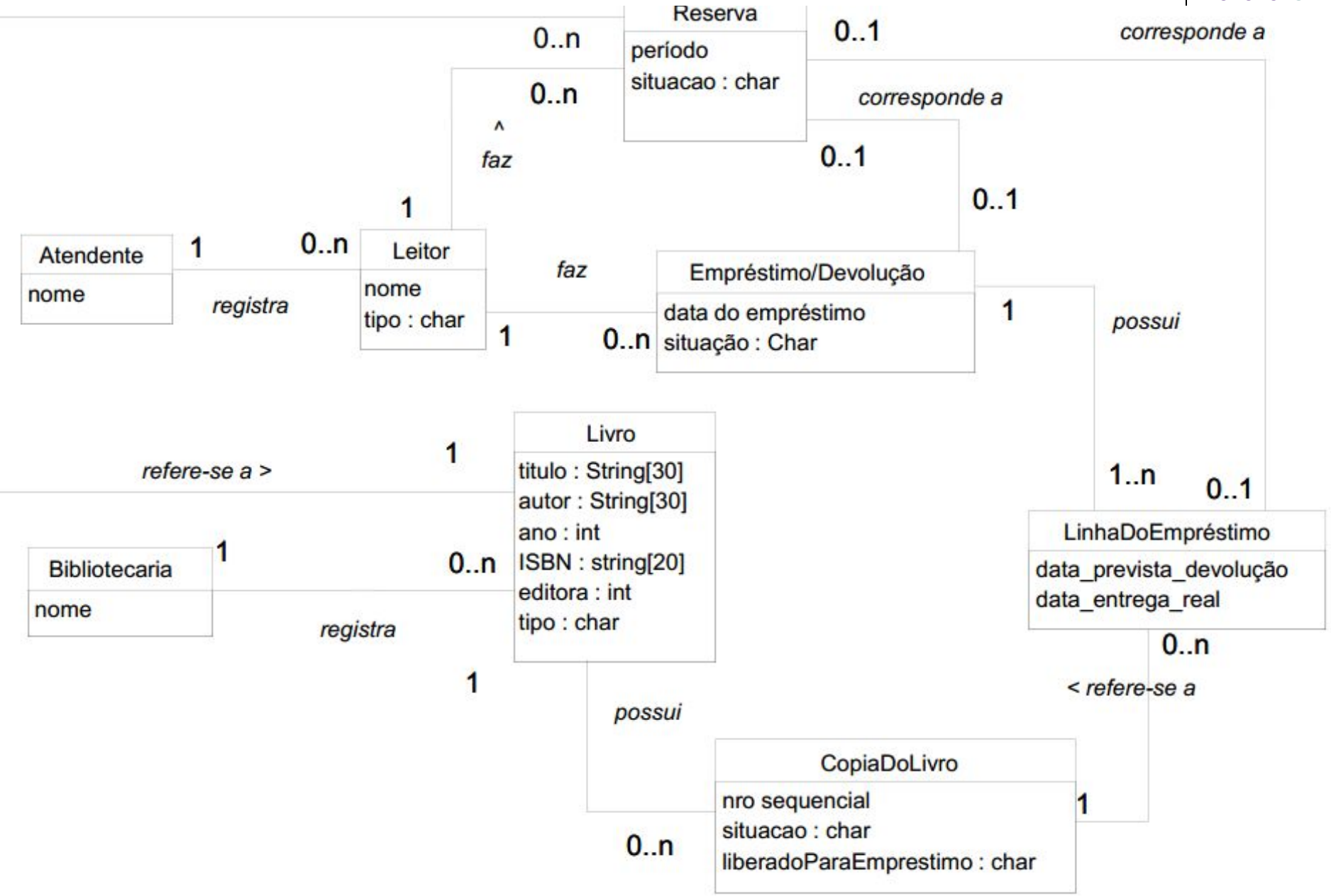


- Imagine as seguintes classes: Venda, ItensVenda, Produto, TabelaPreco:
  - Quem deve ser o responsável por criar uma instância de ItensVenda?
  - Quem deve ser o responsável por criar uma instância de Venda?
  - Quem deve ser o responsável por criar uma instância de Produto?
  - Quem deve ser o responsável por criar uma instância de Tabela Preco?

Criador (creator)



- Benefícios:
  - Acoplamento fraco
  - Maior clareza
  - encapsulamento
  - reutilização


Reserva
período
situacao : char
0..n
0..1
corresponde a
0..n
^
faz
corresponde a
0..1
0..1
1
Atendente
nome
1
registra
0..n
Leitor
nome
tipo : char
faz
1
0..n
Empréstimo/Devolução
data do empréstimo
situação : Char
1
possui
Livro
titulo : String[30]
autor : String[30]
ano : int
ISBN : string[20]
editora : int
tipo : char
1
refere-se a >
1..n
0..1
LinhaDoEmpréstimo
data_prevista_devolução
data_entrega_real
Bibliotecaria
nome
1
0..n
registra
0..n
< refere-se a
1
possui
CopiaDoLivro
nro sequencial
situacao : char
liberadoParaEmprestimo : char
0..n
1

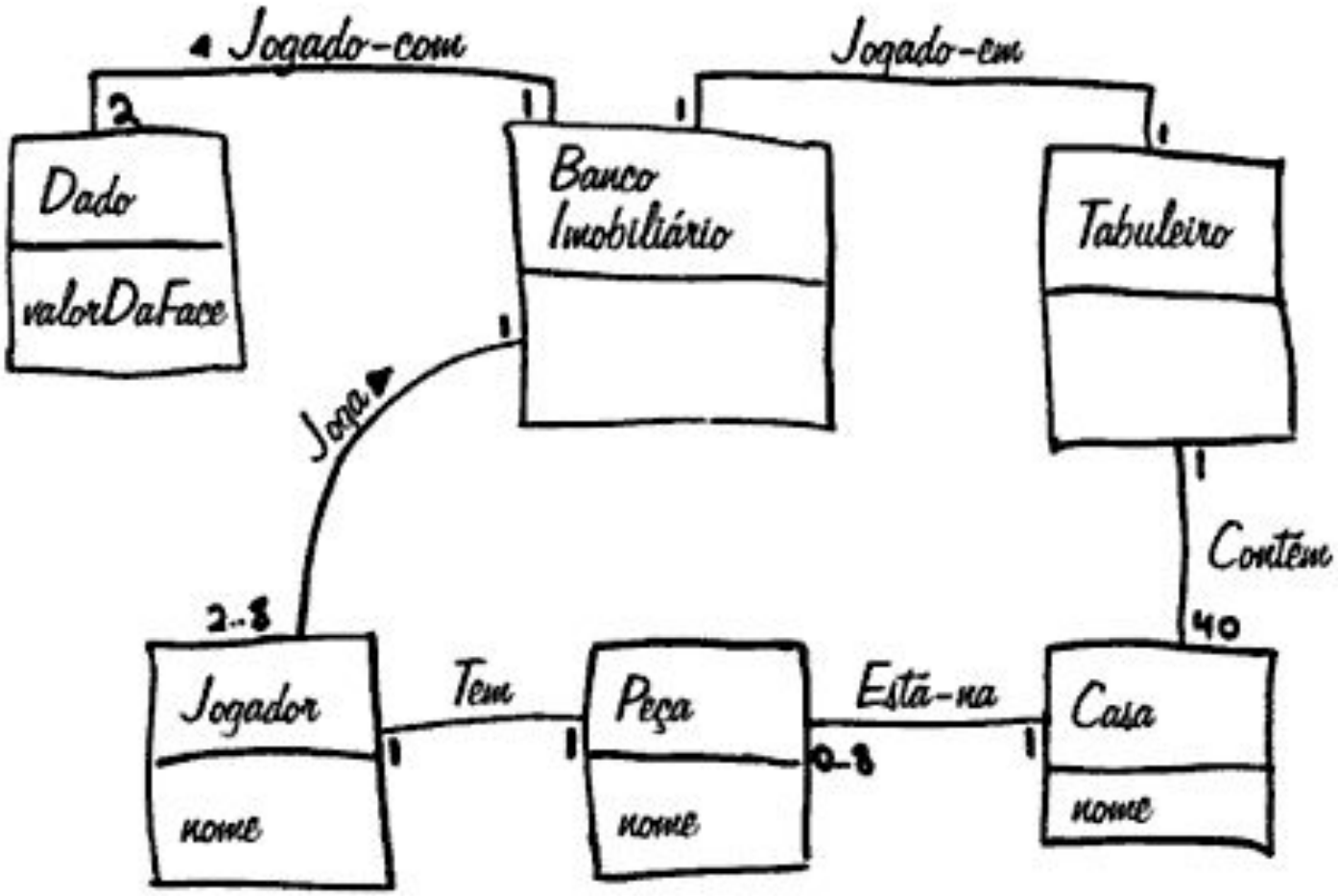
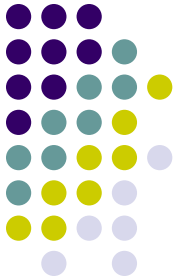


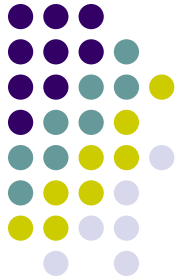
Figura 17.3 Modelo de domínio para a iteração 1 do Banco Imobiliário.

Acoplamento Fraco



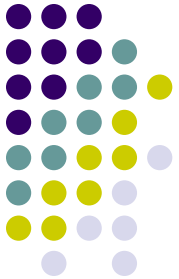
- Atribuir uma responsabilidade de maneira que o acoplamento permaneça fraco
  - menor dependência entre as classes,
  - mudança em uma classe com menor impacto em outras,
  - maior potencial de reutilização

Acoplamento Fraco



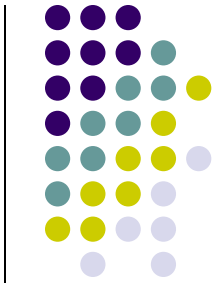
- Usando um software para modelagem UML, faça (usando diagrama de interação):
- Crie as classes Registro, Pagamento e Venda
- Ao chegar a solicitação de fazer o pagamento, o registro deve criar o pagamento e adicionar o pagamento a venda.
- Crie a sequência com acoplamento fraco

Acoplamento Fraco



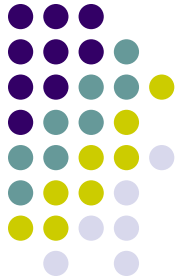
- Benefícios
  - Não é afetado por mudanças em outros componentes
  - É simples de entender isoladamente
  - É conveniente para reutilização

Alta Coesão



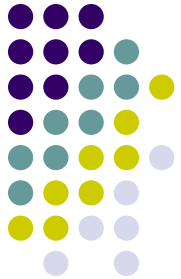
- Problema: Como manter a complexidade sob controle?
- Solução: Atribuir uma responsabilidade de forma que a coesão permaneça alta

Alta Coesão



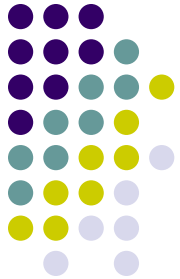
- Uma classe com coesão baixa faz muitas coisas não relacionadas ou executa muitas tarefas.
  - difíceis de compreender
  - difíceis de reutilizar
  - difíceis de manter
  - delicadas, constantemente afetadas pelas mudanças

Alta Coesão

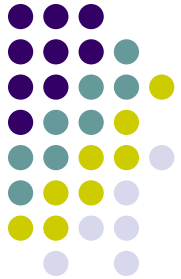


- Benefícios
  - mais clareza e facilidade de compreensão no projeto
  - simplificação da manutenção e do acréscimo de melhorias
  - favorecimento do acomplamento fraco

Coesão



- Coesão muito baixa
  - uma única classe é responsável por muitas coisas em áreas funcionais muito diferentes.
- Coesão baixa
  - Uma classe é a única responsável por uma tarefa complexa em uma área funcional
- Coesão Alta
  - uma classe tem responsabilidades moderadas em uma área funcional e colabora com outras classes para a realização das tarefas

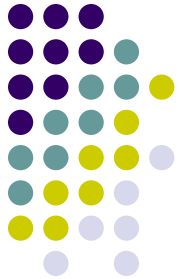


mal coesão

=

mal acoplamento

Controlador (controller)



- Problema: Quem deve ser o responsável por tratar um evento do sistema?

- Solução: Atribuir a responsabilidade de receber ou tratar uma mensagem de um evento do sistema .
- Um objeto controlador é um objeto de interface não-usuário, responsável por receber ou manipular um evento do sistema

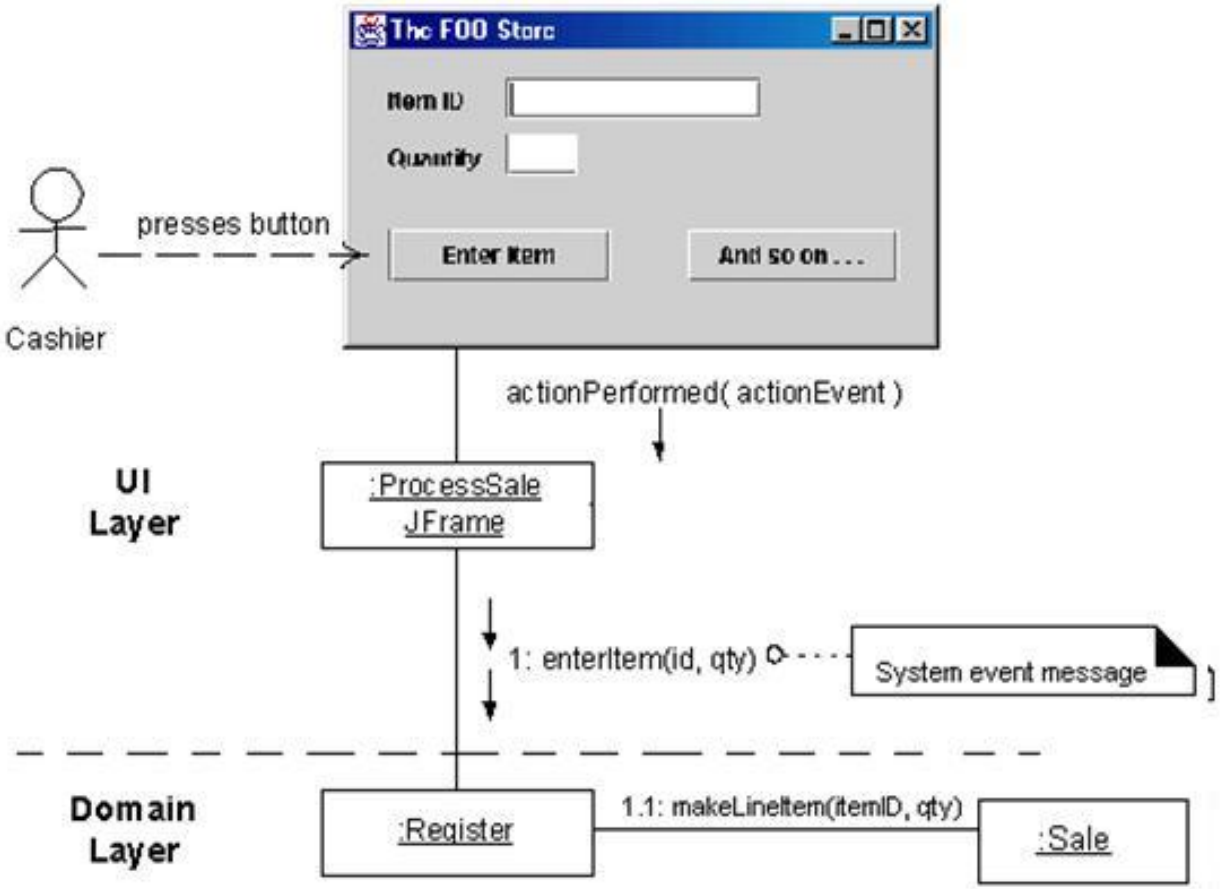
Which class of object should be responsible for receiving the enterItem() system event message?

Register? POSSystem?

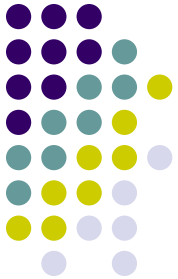
These represent overall "system", "root object", device, or subsystem.

ProcessSaleHandler? ProcessSaleSession? (these would be new)

These represent a receiver or handler of all system events of a use case scenario.



GRASP



fundamentais

- Criador (creator)
- Especialista na Informação (Information Expert)
- Acoplamento Baixo (Low coupling)
- Coesão Alta (high cohesin)
- Controlador (Controller)

Avançados

- Polimorfismo
- Invenção Pura (services, classe artificial)
- Indireção (controller no MVC, mediação entre o M e o V)
- Variações Protegidas

Classificação dos 23 padrões segundo GoF

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

http://www.devmedia.com.br/imagens/articles/226729/Classificacao%20gof.jpg