Pós-Graduação
Engenharia de Software

Arquitetura de Software e Padrões de Projeto

Aula 03

Prof. Msc Rogério Augusto Rondini
rarondini.paradygma@gmail.com

Conteúdo

- Design por contrato
- Padrões de Projeto – parte 01
  - DAO
  - Factory
  - Service Layer (Facade)
  - Transfer Object / Data Transfer Object / Value Object
  - Front Controller
  - Command
  - Interceptadores e Filtros

Design por Contrato

- Técnica de desenvolvimento de software onde define-se contratos especificados através de interfaces bem definidas
- O termo foi definido em 1986 por Bertran Meyer
  - Um de seus principais trabalhos é o livro *"Object Oriented Software Construction"*

Design por Contrato

- Esta técnica define como elementos de software comunicam entre si, baseados em obrigações mútuas
- Obedece os mesmos princípios de “contratos de negócios”
  - Fornecedor irá fornecer um determinado produto
  - Cliente irá pagar um determinado valor

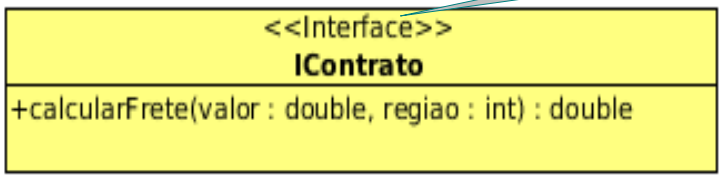
Design por Contrato

- Dois conceitos importantes para especificar contratos
  - Interface → Elemento que define apenas contrato, sem qualquer tipo de implementação
  - Classes Abstratas → Elemento que define contrato, porém, permite implementação de alguns métodos. Normalmente utiliza-se da palavra "*abstract*"

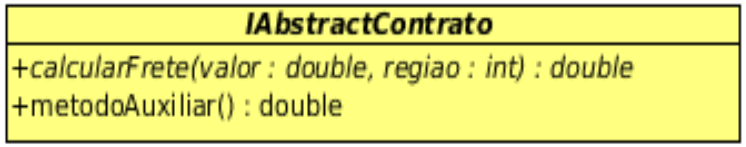
Design por Contrato

- Representação UML

UML representa interface utilizando estereótipo



UML representa classes abstratas e métodos abstratos em itálico



DAO

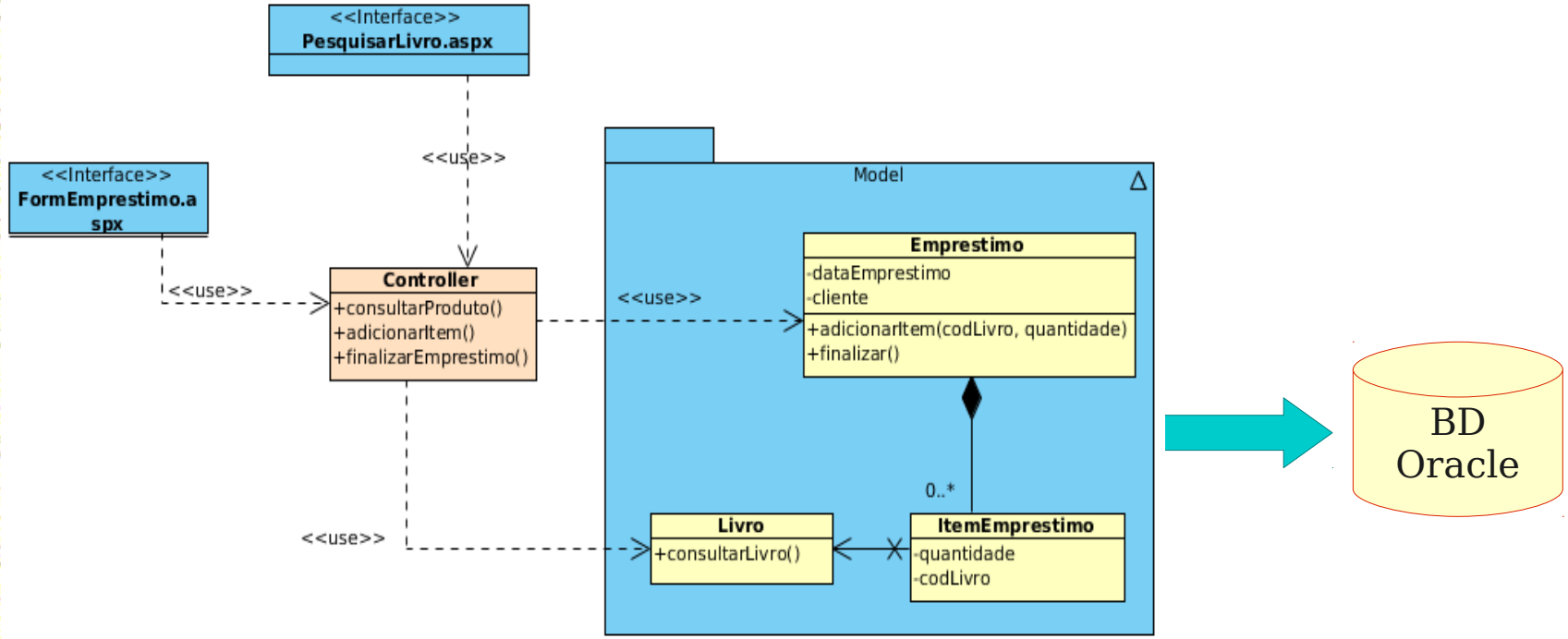
- Problema
  - A grande maioria das aplicações comerciais precisam persistir dados em bancos de dados relacionais ou outros meios de armazenamento
  - Algumas aplicações, ainda, precisam acessar dados necessitam acessar dados armazenados em outros sistemas, tais como, mainframe, bases de identidade LDAP, etc.
  - Existem ainda situações onde a aplicação deve ser independente do meio de armazenamento

DAO

- Problema
  - O acesso aos mecanismos de persistência variam de acordo com a tecnologia utilizada e o fornecedor
  - Seguindo o princípio básico da orientação a objeto – reduzir o acoplamento – torna-se necessária a adoção de estratégia que promova o desacoplamento das regras de negócio do acesso ao mecanismo de persistência

DAO

- No exemplo a seguir (já com MVC), quais objetos são responsáveis por gravar as informações do Banco de Dados ?



OBS: Independente de qual objeto tenha essa responsabilidade, ele estará de alguma forma dentro da camada *Model*.

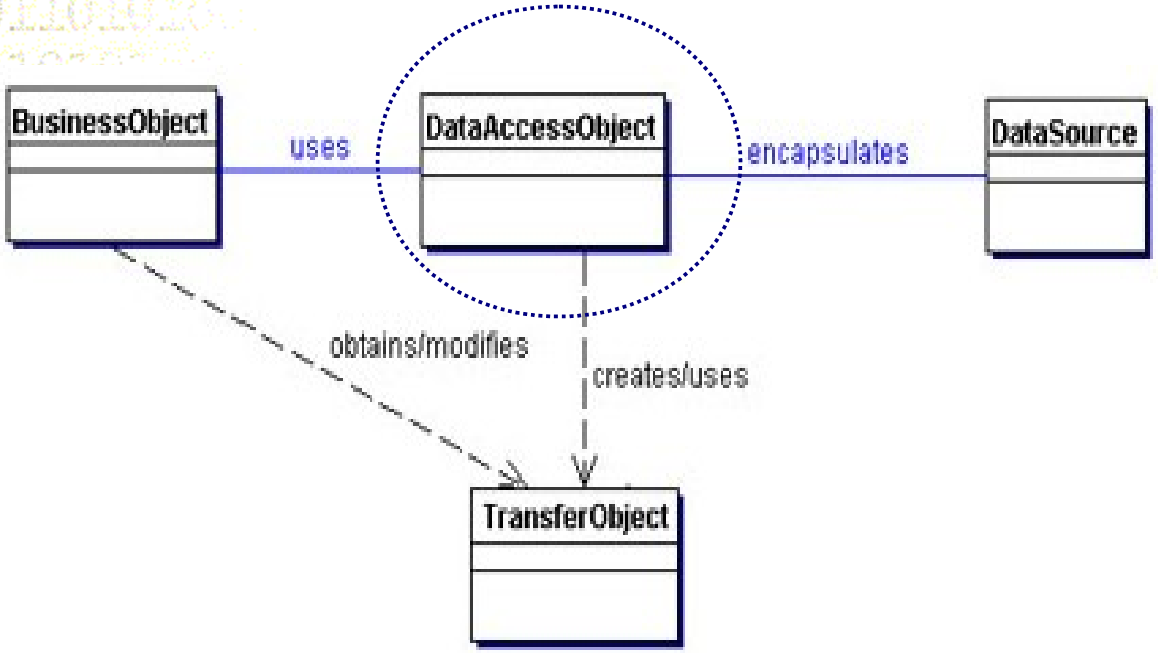
DAO

- Solução
  - Utilizar o padrão de projeto DAO (*Data Access Object*) para **encapsular** todos os acessos à fonte de dados
  - **Reduzindo**, assim, o **acoplamento** dos objetos de negócio (classe `Emprestimo` por exemplo) com fontes de dados específicas

Palavrinha mágica em Orientação a Objetos

DAO

- Visão geral do padrão DAO e seus relacionamentos

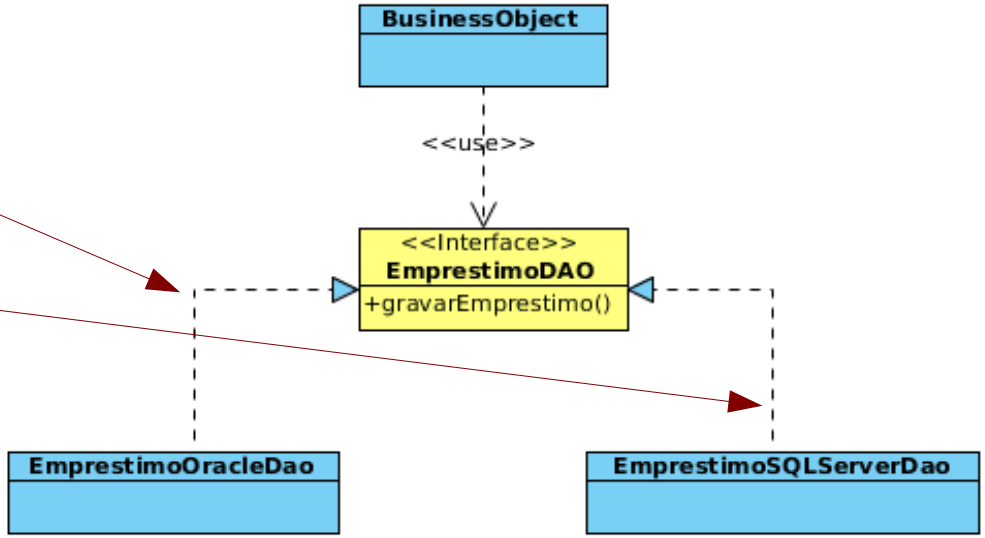


- Objeto de negócio (*Business Object*) faz acesso ao DAO ao invés de acessar a fonte de dados (*Data Source*)
- DAO encapsula o acesso à fonte de dados (*Data Source*)
- DAO pode utilizar objetos auxiliares (*Transfer Object*) para receber e retornar dados

DAO

- Desacoplando a implementação do DAO do Objeto de Negócio
  - Utilizando Design por Contrato para criar uma interface bem definida que pode ser implementada de diversas maneiras

Importante observar o estilo da seta para não confundir com dependência



DAO

```
Interface EmprestimoDAO{
  método gravarEmprestimo(...) ;
}

Classe EmprestimoOracleDAO implementa EmprestimoDAO{
  método gravarEmprestimo(...){
    // código específico para acesso ao Oracle
  }
}

Classe EmprestimoSQLServerDAO implementa
EmprestimoDAO{
  método gravarEmprestimo(...){
    // código específico para acesso ao SQLServer
  }
}
```

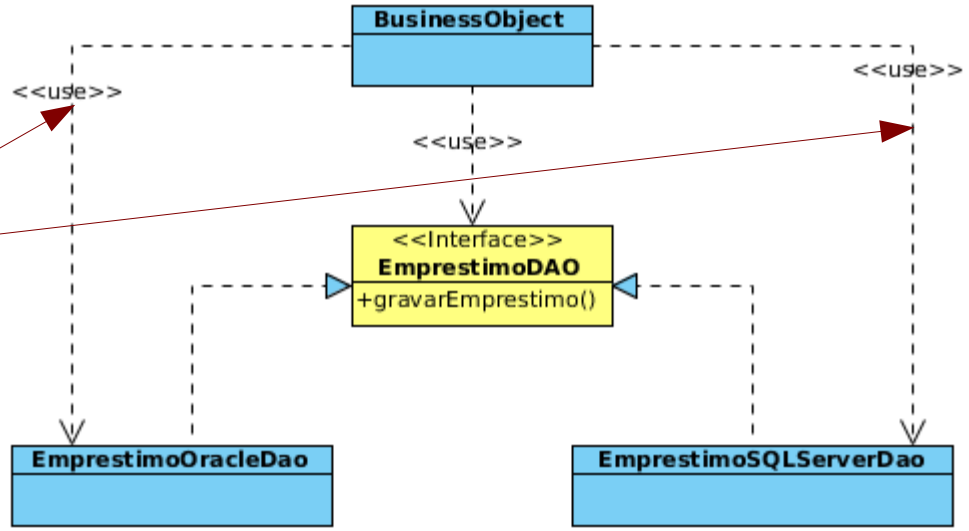
?

- Se o objetivo é desacoplar o objeto de negócio da implementação do DAO...
- Se em tempo de execução é necessário saber qual implementação de DAO será utilizada..
- Como selecionar a implementação do DAO e ainda manter o desacoplamento do objeto de negócio ? Quem deve ter essa responsabilidade ?

Factory

- O cenário apresentado no slide anterior ilustra uma situação onde a tendência é deixar para o próprio objeto de negócio a responsabilidade de seleção e de criação de instâncias das implementações do DAO

Anti-Pattern



Factory

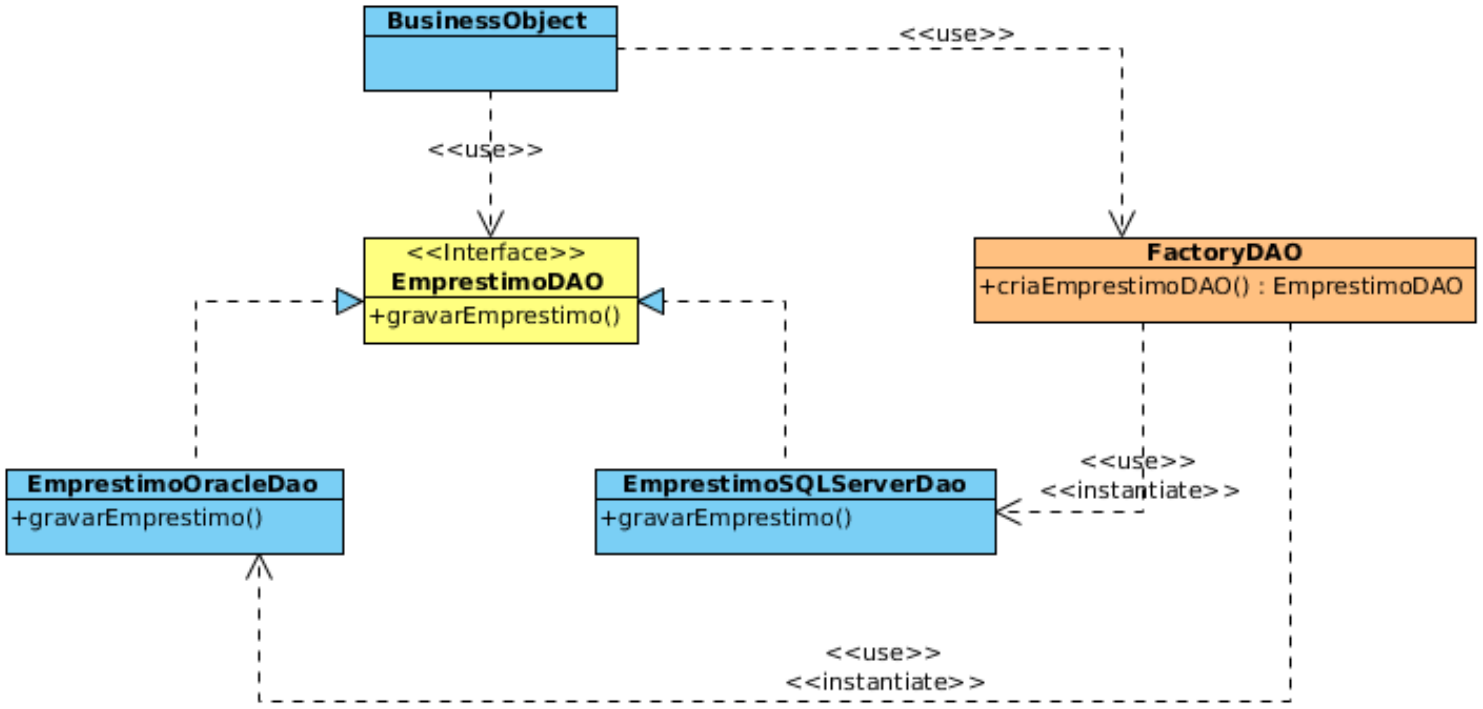
- O padrão *Factory* (e suas variantes *FactoryMethod* e *AbstractFactory*) tem por objetivo encapsular a lógica de criação de objetos
- Pode ser utilizado em situações diversas, onde a criação de objetos depende de analisar o contexto da execução ou mesmo arquivos de configuração

Factory

- Vantagens
  - Desacoplamento
  - Controle do ciclo de vida de objetos
  - Permite a utilização de cache e pooling

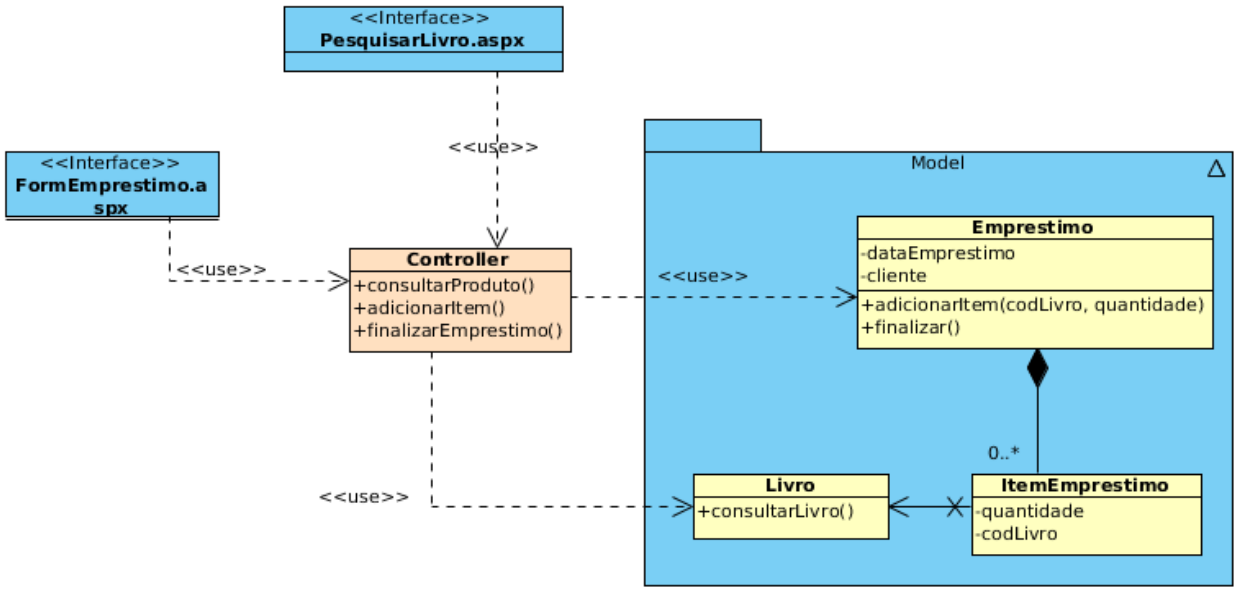
DAO e Factory

- Factory se responsabiliza pela criação de implementações específicas de DAO



Service Layer

- Em nosso exemplo de MVC, vamos observar o controlador

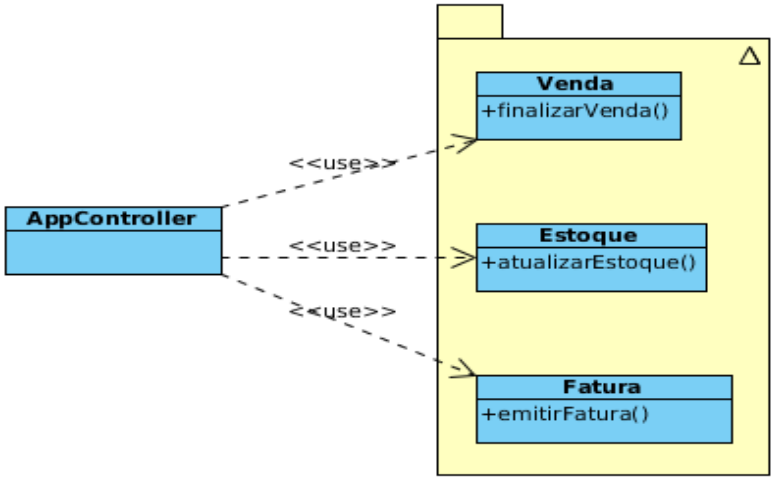


- O controlador faz acesso direto aos objetos de negócio

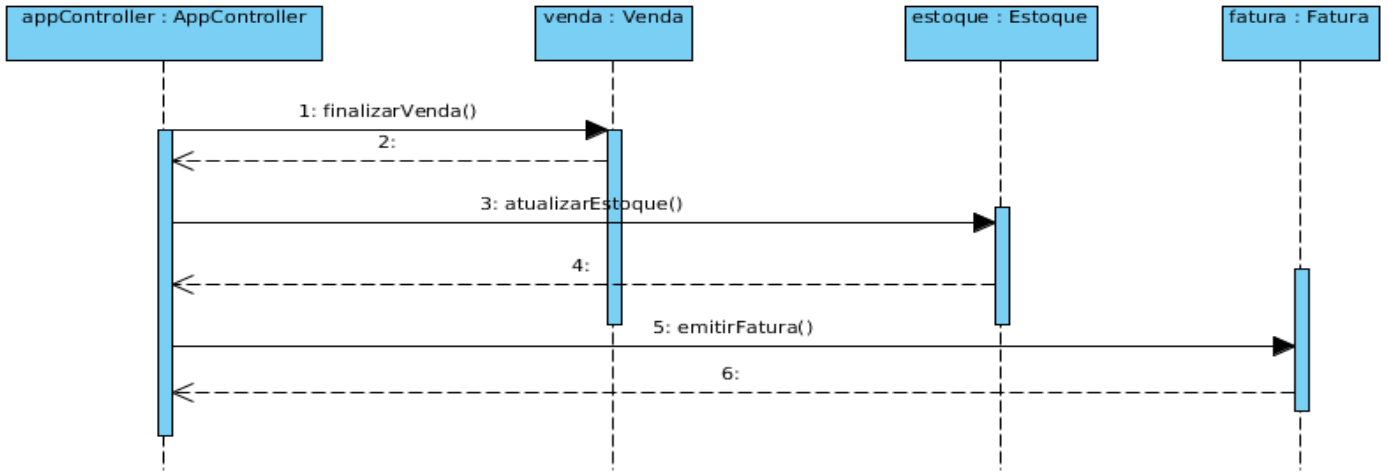
Service Layer

- Normalmente, uma operação de negócio (transação) envolve acesso a diversos objetos
  - Supondo um cenário onde uma transação de venda envolve a gravação dos dados da venda, atualização do estoque e emissão de fatura...

Service Layer



Controlador tem a responsabilidade de definir o fluxo de execução do negócio, quando deveria ter apenas a responsabilidade de definir o fluxo de navegação



Service Layer

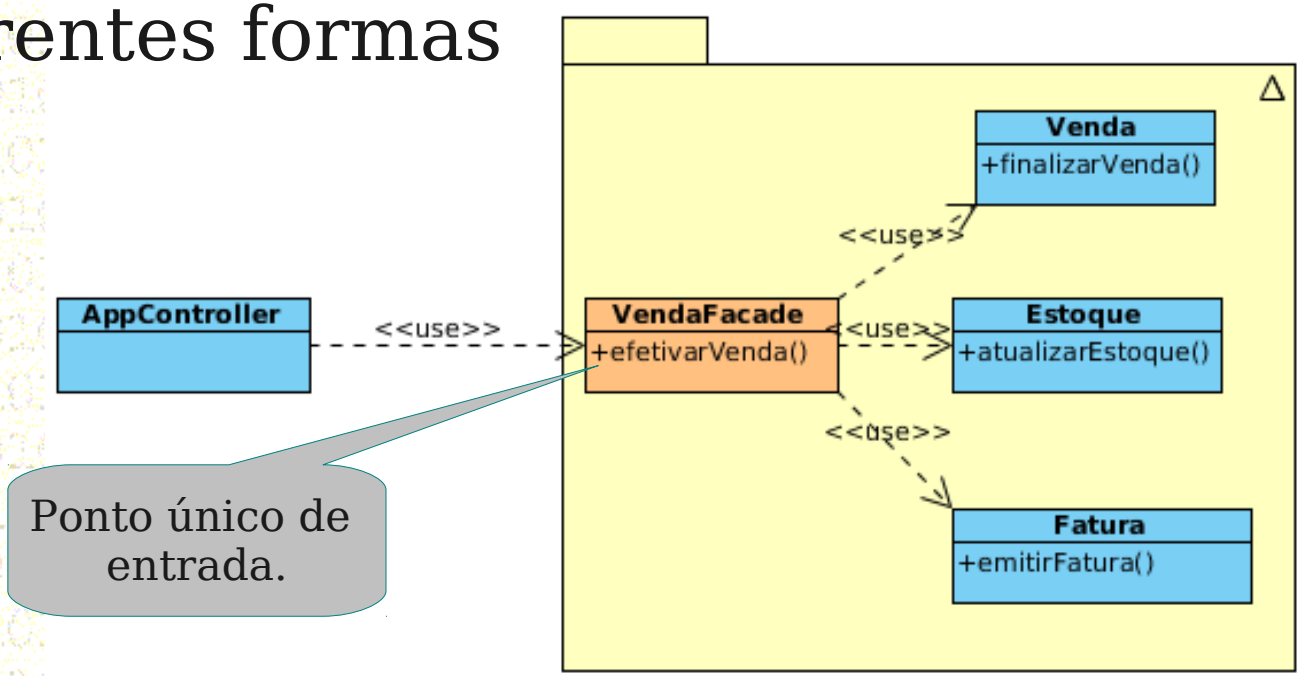
- Repetição de código
  - Dessa forma, se houver a necessidade de alteração do controlador ou mesmo o acesso através de diferentes formas de apresentação (web, desktop, mobile...), o código referente ao fluxo de execução será repetido em diferentes pontos do sistema

Service Layer

- A solução nesse caso é criar uma interface de serviço que seja responsável por encapsular a lógica de execução da transação de negócio
- O padrão ***Service Layer*** também é chamado de ***Façade***, ou fachada de serviço.

Service Layer

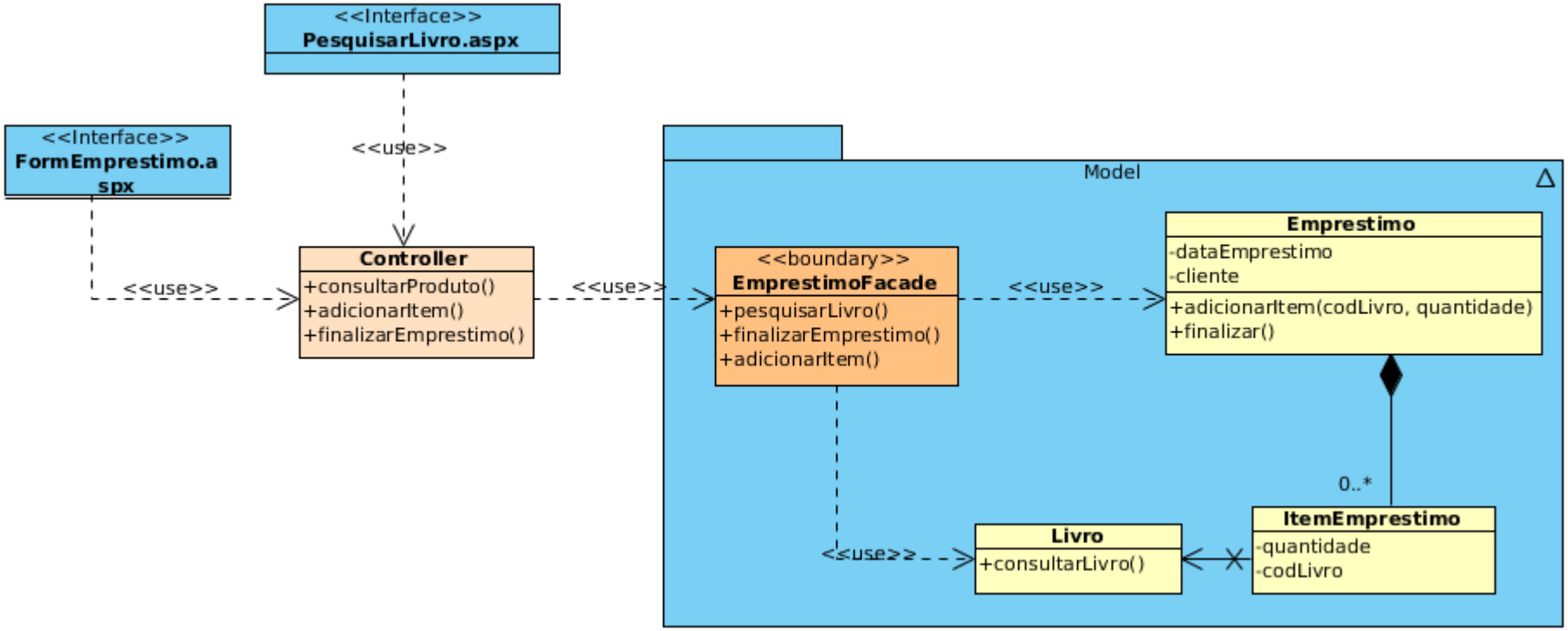
- Agora a aplicação possui um ponto único de entra, que pode ser acessado de diferentes formas



Como fica agora o fluxo de execução no diagrama de sequência ?

Service Layer

- Aplicando o *Façade* em nosso exemplo



T.O / V.O / D.T.O

- Os padrões *Transfer Object* (T.O), *Value Object* (V.O) e *Data Transfer Object* (D.T.O) , apesar das diferentes nomenclatura, apresentam a mesma utilidade: trafegar dados entre camadas
- Até o momento, não mencionamos como os dados saem da View e chegam até o DAO, e vice-versa.

T.O / V.O / D.T.O

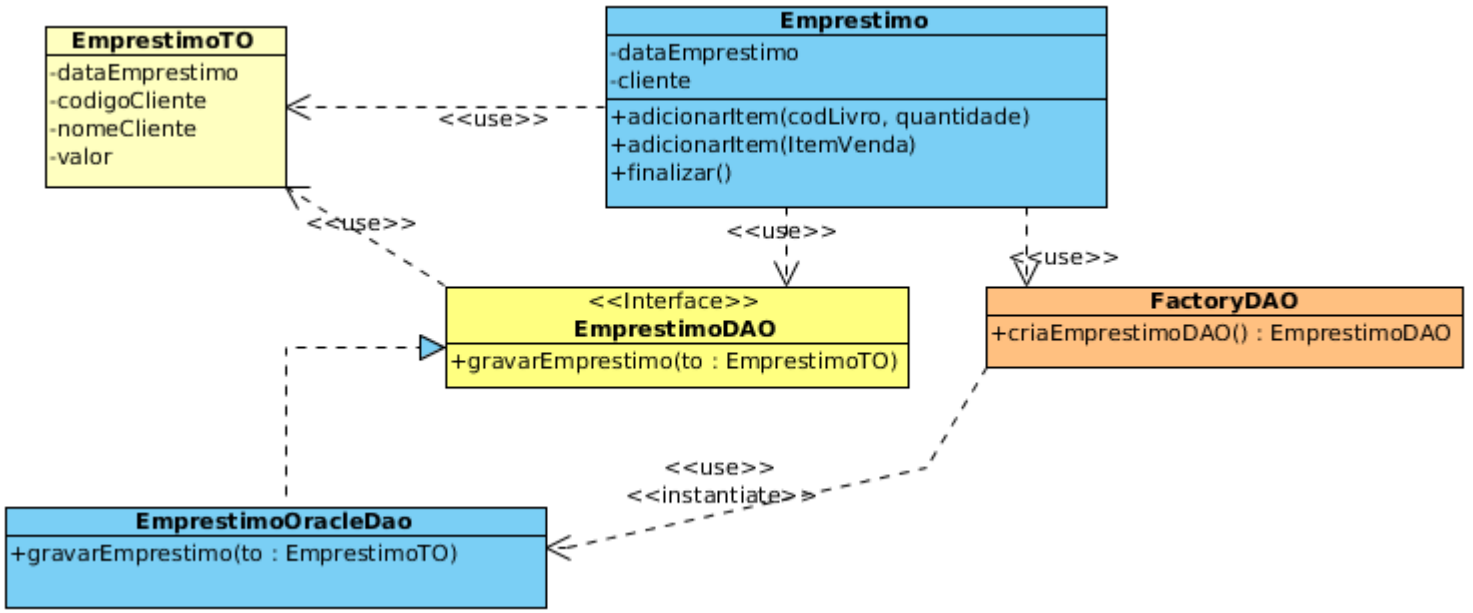
- Algumas considerações
  - Como os dados de um empréstimo, por exemplo, serão apresentados ao usuário ? O que a View precisa fazer ?
  - Como reduzir a utilização de conexões a bancos de dados ?

T.O / V.O / D.T.O

- O *Transfer Object* é uma representação de uma ou várias entidades de negócio
  - Não possui implementação de regras de negócio
  - Pode representar diversas entidades
  - Pode conter informações que facilitem a apresentação, tais como, data formatada para apresentação na tela (tipo “ViewHelper”)

T.O / V.O / D.T.O

- Neste exemplo, o DAO recebe como parâmetro um objeto `EmprestimoTO`



Front Controller

- O *Front Controller* é uma variação do *Controller* utilizado no padrão MVC, onde apenas um controlador é o responsável por receber todas as requisições, ao contrário da definição básica de *Controller* onde se tem um controlador para cada caso de uso.

Front Controller

- Vantagens
  - Centralização de Requisições
  - Maior controle das requisições
  - Evita duplicação de código
    - Exemplo: verificação se usuário está autenticado na aplicação
- Desvantagem/Problema
  - Centralização pode acarretar em código complexo para manutenção

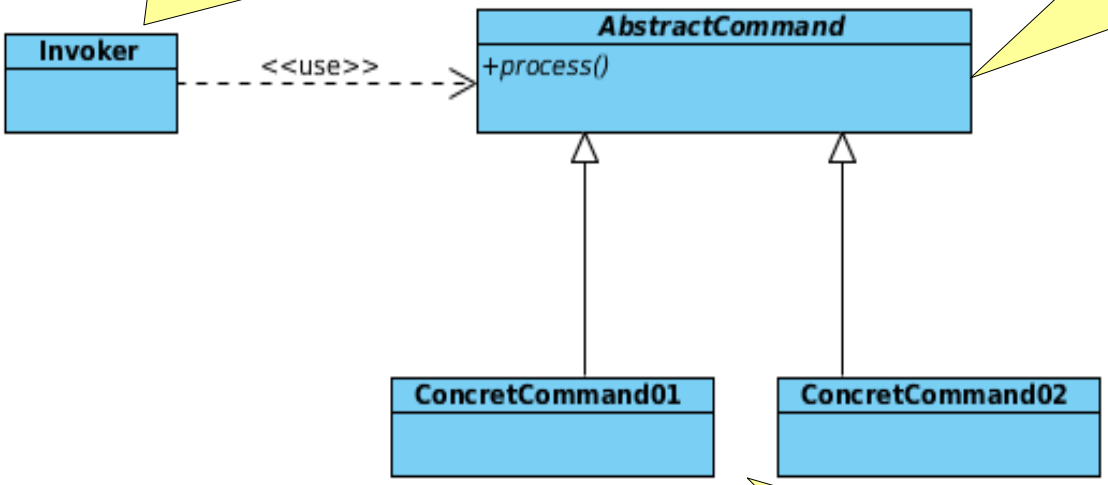
Command

- Command é um padrão de projeto utilizado quando se tem ações que precisam ser executadas dependendo do evento que foi disparado.
- Como o próprio nome diz, são “comandos” que tem regras e fluxos de negócio específicos, exemplo:
  - Abrir arquivo
  - Salvar
  - Salvar como
  - ...

Command

Classe que executa o command
Não deve ter conhecimento das implementações específicas

Classe abstrata (ou interface)
Deverá ser implementada por cada "command" específico

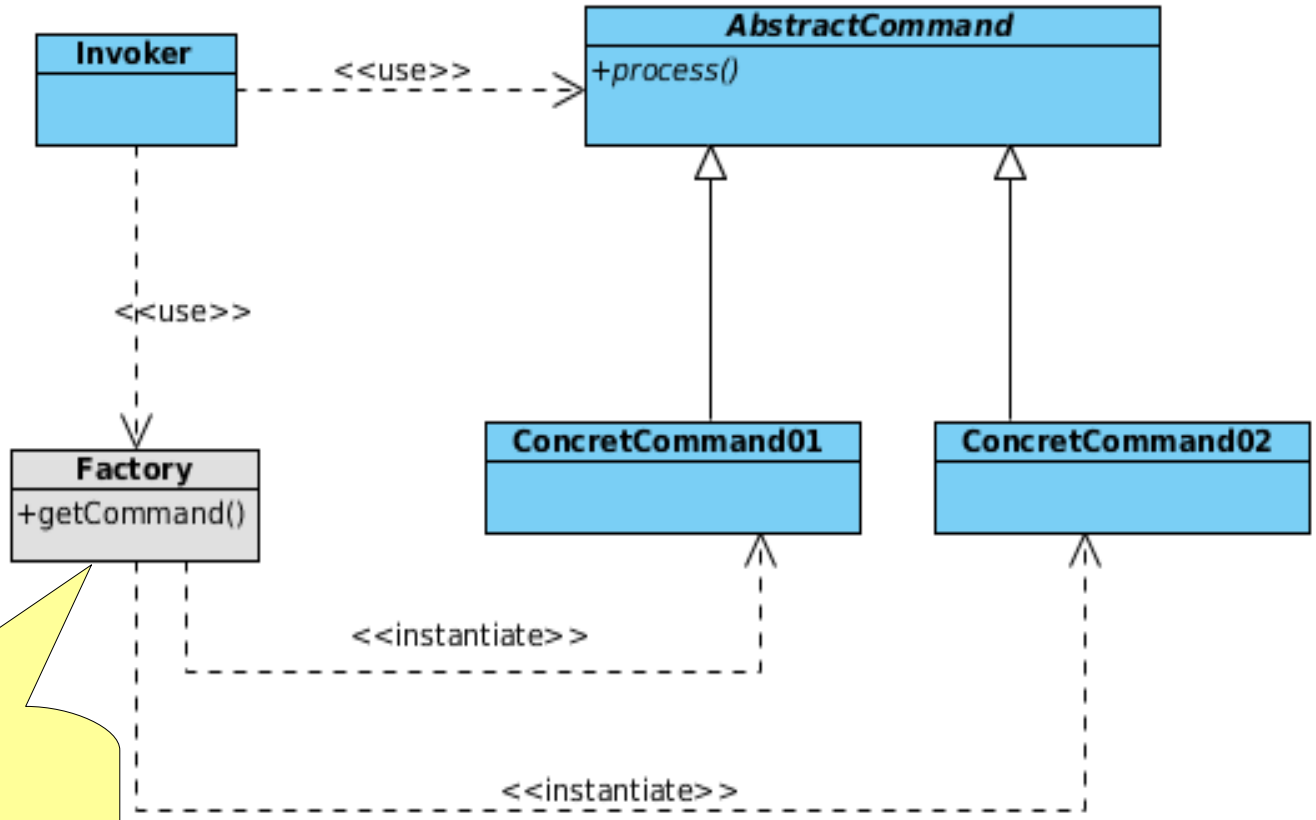


Implementações específicas de Command.
Exemplo: Salvar, Pesquisar...

Command

- Considerando que *“AbstractCommand”* é uma classe abstrata ou interface, e que ambos não podem ser instanciados...
- Considerando que o *“Invoker”* não deve ter conhecimento das implementações de *command*...
- **PERGUNTA: Como desacoplar o Invoker das Implementações de *command* ??**

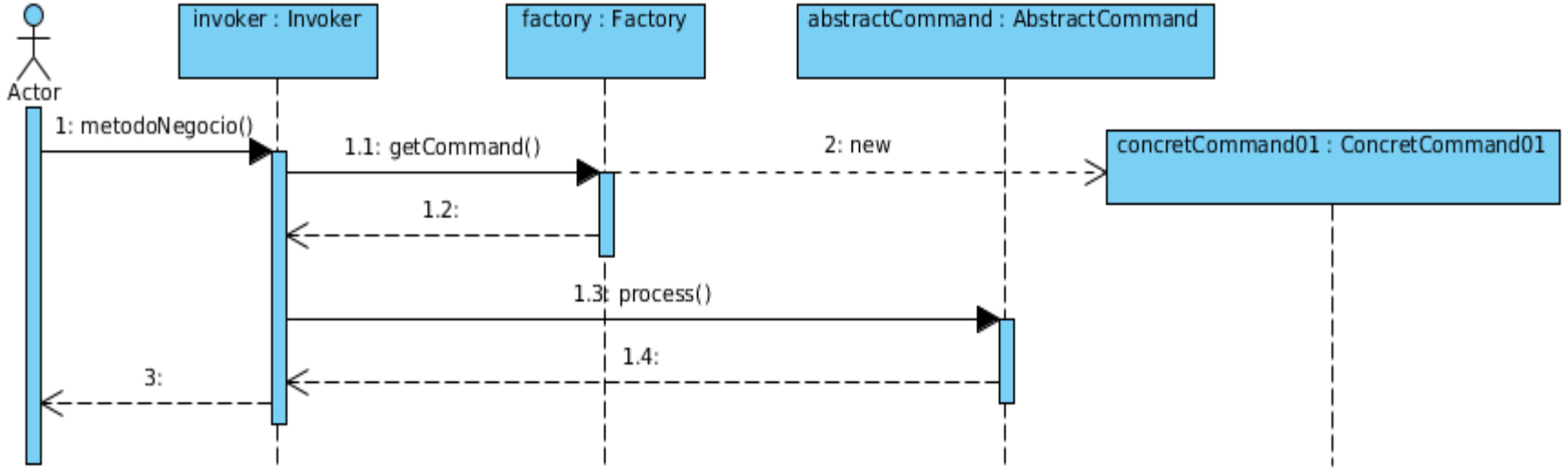
Command


Invoker
<<use>>
AbstractCommand
+process()
<<use>>
Factory
+getCommand()
ConcretCommand01
ConcretCommand02
<<instantiate>>
<<instantiate>>
Factory fica responsável pela instanciação da implementação

Command

Note que a *factory* cria instância de ConcretCommand01,
ou seja, da implementação específica
de um *command*.
Porém, o Invoker não tem conhecimento disso,
basta ele ter conhecimento de AbstractCommand.

PERGUNTA: Por que isso funciona ?



Command

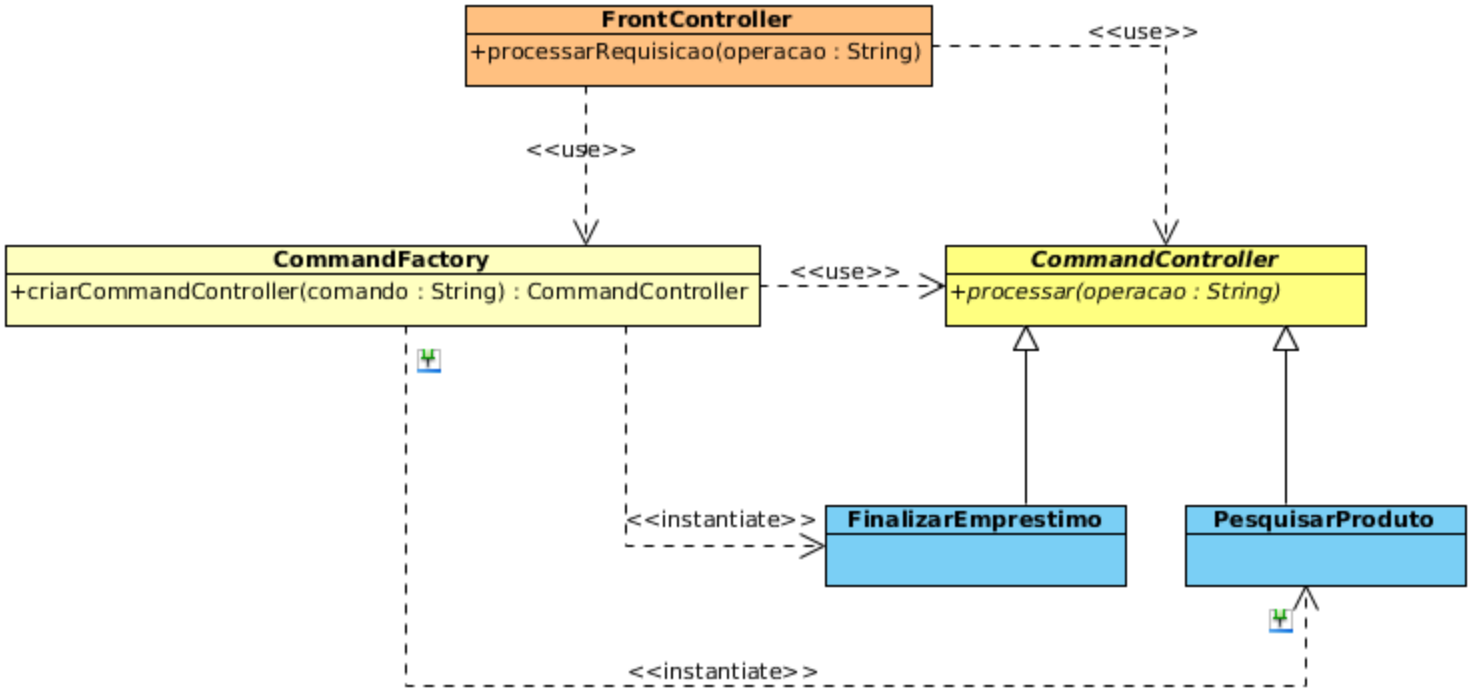
```
Classe abstrata AbstractCommand {
  método abstrato processar(...) ;
}

Classe ConcretCommand01 estende AbstractCommand {
  método processar(...) {
    // implementação do método
  }
}

    Classe Factory {
      AbstractCommand obterCommand(...) {
        Retorna novo ConcretCommand01()
      }
    }
```

Front Controller + Command

- Conforme comentamos anteriormente, a centralização do *controller* (*Front Controller*) pode gerar baixa coesão, aumentando a complexidade de implementação e manutenção do *Controller*
- Em conjunto com o *Command* e o *Factory*, podemos distribuir as responsabilidades, e deixar o *Controller* apenas tratando requisições

Front Controller + Command

- Agora, o *Front Controller* somente recebe requisições e delega para o *Command* apropriado
- O *Factory* é responsável por criar a implementação de *Command* e devolver para o *Front Controller*

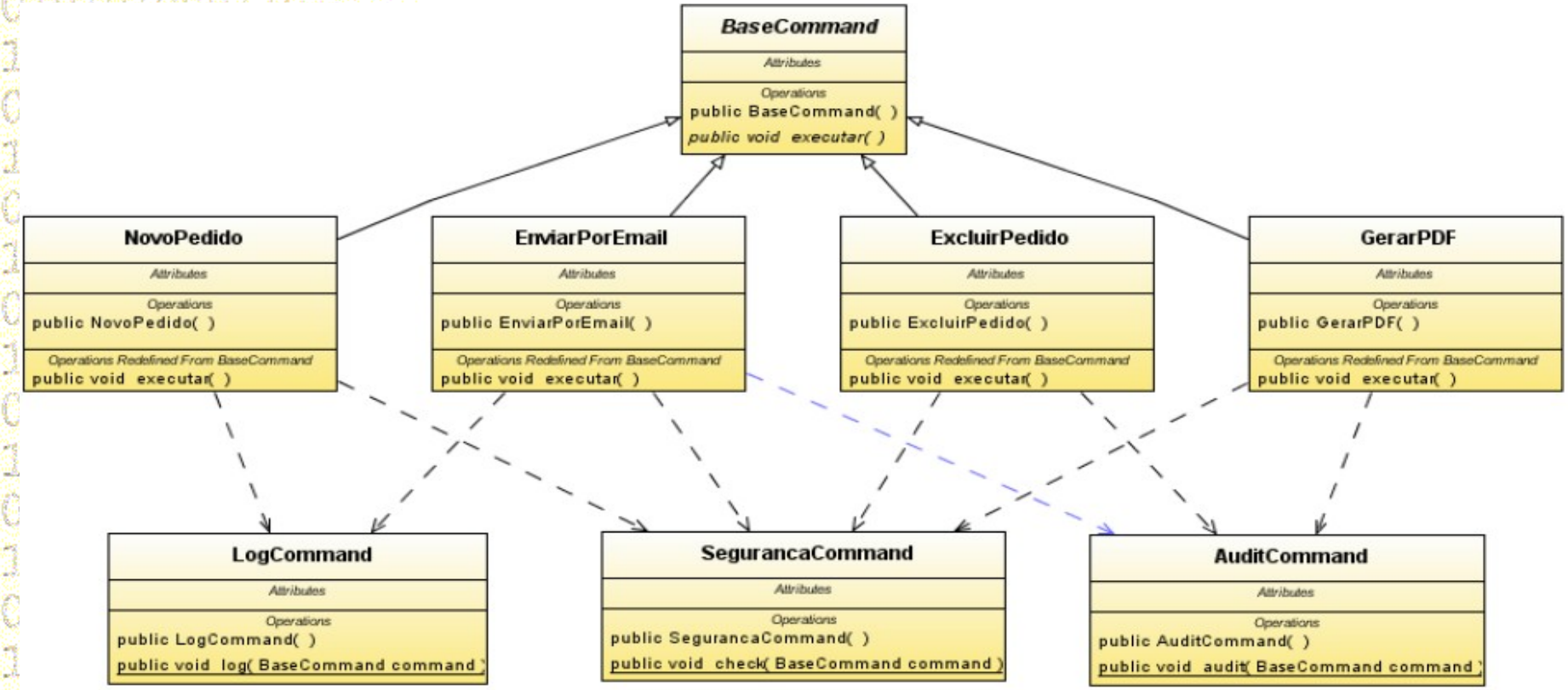


Interceptadores e Filtros

- Normalmente, aplicações precisam executar operações denominadas de operações de infra estrutura, tais como auditoria, log, autenticação, autorização, entre outras
- Esse tipo de operação pode ocorrem em diversas camadas da aplicação, podendo acarretar em redundância de código

Interceptadores e Filtros

- Exemplo
  - Alto grau de acoplamento



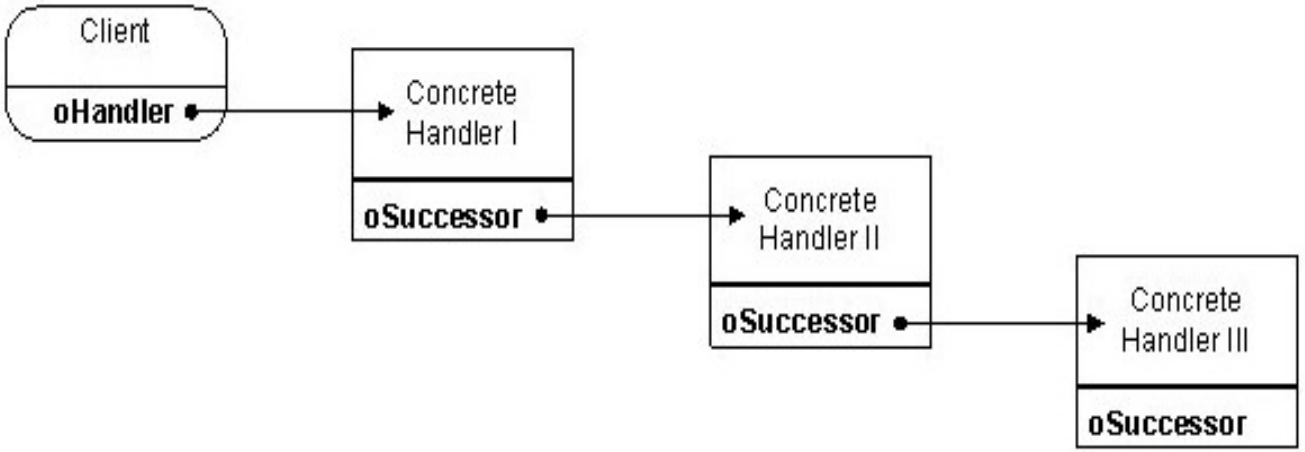
Ref: 33 design-patterns aplicados com Java. Vinicius Senger e Kleber Xavier

Interceptadores e Filtros

- Existem dois padrões que podem simplificar esse trabalho
  - Chain of Responsibility
    - Cadeia de responsabilidade
  - Intercepting Filter
    - Baseado no chain of responsibility

Interceptadores e Filtros

- Permite o encadeamento de requisições, cada uma representando uma operação
- A execução do comando subsequente depende do sucesso da execução an



Atividade 02

- Prática de laboratório com UML (alteração da descrição do material (aula 02)
  - Criar um diagrama de pacotes representando uma arquitetura de referência para o sistema PDV
    - *Chamaremos de Versão 01 da Arq. de Ref.*
  - Baseado na arquitetura de referência versão 01
    - Criar um diagrama de classe para o caso de uso selecionados na Atividade 01 (“capturar venda”)
    - Criar um diagrama de sequências representando um dos fluxos do Caso de Uso
  - Preencher a Visão Lógica e Visão de Caso de Uso do documento de arquitetura (seguir modelo)