INFO1003 – Eng Sw II

Engenharia de Software

Prof. Marcelo Soares Pimenta
mpimenta@inf.ufrgs.br

Slides – arquivo 4

Reuso

Prof. Marcelo Soares Pimenta
mpimenta@inf.ufrgs.br

Reuso, patterns, fmwk, cbse

como aumentar a produtividade no desenvolvimento de software?

- trabalhe *mais rápido*
  - automação, ambientes, ferramentas
  - substitua trabalho humano
- trabalhe *mais inteligentemente*
  - melhore o(s) processo(s)
  - evite/reduza tarefas de pouco valor
- *EVITE O TRABALHO!*
  - REUSE ARTEFATOS do CICLO DE VIDA
  - *evite/reduza o desenvolvimento de artefatos específicos para cada projeto…*

resultados: boehm {dod, 1999}

- working-faster savings:
  - 8%
- working-smarter savings:
  - 17%
- work-avoidance savings:
  - 47%

Roteiro

- Reuso de Software
- Padrões (patterns)
- Frameworks
- Desenvolvimento Baseado em Componentes
- Reuso Sistemático de Software
- Casos de Sucesso e Falhas
- Mitos
- Inibidores

Reuso de Software

- "Software reuse is the use of existing software knowledge or artifacts to build new software artifacts" [Frakes, 1995]
- Vantagens (em POTENCIAL)
  - MAIS Qualidade
  - MENOS Tempo de desenvolvimento
  - MENORES custos TOTAIS no ciclo de vida... implementação, testes... integração, documentação, manutenção... evolução...

Artefatos reusáveis [D'Souza, 1999]

- Código compilado [fonte]
- Casos de testes
- Modelos e projetos: *frameworks* e padrões
- Interface de usuário
- Planos, estratégias e regras arquiteturais
- ...

A questão de reuso (1)

- Motivação para REUSO:
  - Produtividade
  - Qualidade
- Reuso de Pessoal
- Reuso nas atividades de Análise e Projeto
  - exame de outros sistemas e projetos: engenharia reversa
  - padrões de análise
  - padrões de projeto
  - frameworks
  - componentes

A questão de reuso (2)

- Reuso nas atividades de Implementação
  - Reuso de código
  - mecanismos:
    - bibliotecas de rotinas, funções, classes,componentes

- Processo de Reuso:

  Projetar visando permitir reuso (*Design for reuse*)

  versus

  Projeto Usando Reuso (*Design by reuse*)

Ciclo de Projeto Usando Reuso

- Busca em catálogo de elementos
- Seleção do melhor elemento a reusar
- Adaptação do elemento a um novo contexto

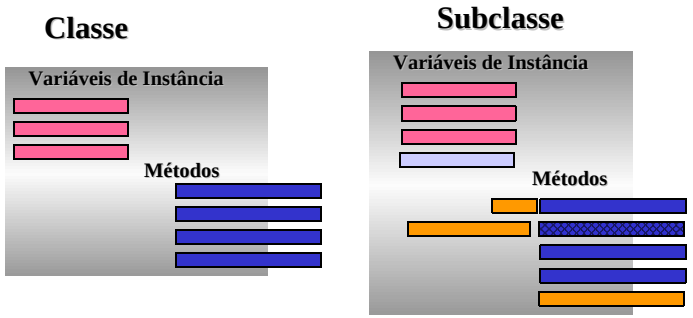
- Problema-chave:
  - CRITÉRIOS??
    - Busca, Seleção, Adaptação

Ciclo visando permitir Reuso

- Dificuldade de projeto de elementos reusáveis !!
  - Generalidade x Eficiência
  - Falta de hábito de permitir reuso
  - Falta de hábito de modelagem explícita e documentação
  - Falta de suporte metodológico (métodos, modelos)
  - Falta de ferramentas

Reuso na Orientação a Objetos

Mecanismos Básicos para a reutilização e extensão de software: Hierarquias de Generalização/Especialização (HERANÇA) e Agregação



Limitações do reuso de objetos

- O Conceito de Classe é muito limitado para grande reuso
  - Continua-se projetando a aplicação e o controle para usar classes.
  - Ganha-se na qualidade, menor manutenção
  - Maior funcionalidade requer colaboração entre vários objetos
- Bibliotecas de classe - ferramentas de localização, classificação etc.

Tentativas de aumentar reuso (1/3)

- Projetar software Reusável:
  - Achar Objetos pertinentes
  - Fatorá-los em classes no nível correto de granularidade
  - Definir interfaces das classes e as hierarquias de generalização/especialização (herança)
  - Estabelecer relações entre eles
- Dificuldade:
  - Projeto deve ser específico para o problema a resolver (eficiência) *versus*
  - Projeto deve ser genérico o suficiente para servir a futuros problemas e requisitos

Tentativas de aumentar reuso (2/3)

- Reuso da experiência de projeto:
  - Evitar Re-descobertas de soluções
  - "resolvi um problema parecido antes ...se pudesse lembrar os detalhes do problema anterior e de que forma foi resolvido, poderia reusar a experiência ao invés de descobri-la"
  - Em geral, a experiência de projeto NÃO é registrada !
  - Padrões: uma forma de registrar experiências de projeto:
    - descrição de cada padrão permite (re)uso efetivo de soluções e experiências

Tentativas de aumentar reuso (3/3)

- Padrões
  - modelo de estruturas genéricas de classes que colaboram para uma funcionalidade genérica e

    com responsabilidades pré-definidas
  - usada na fase de análise e projeto para incluir o reuso de soluções comprovadas de experiências anteriores bem sucedidas para problemas delimitados

Padrões (patterns) e frameworks

- Parte I - Padrões (Design Patterns)
  - Padrões
    - Conceitos básicos
    - Organização dos 23 padrões descritos em [GAM00]
    - Exemplos de padrões:
      - *Composite, Template Method, Singleton, Façade, Command, Adapter,Factory Method, State*

- Parte II -Frameworks: visão geral

Bibliografia Básica

E. Gamma et all: *Padrões de Projeto*, Bookman, 2000.
Principal referência sobre padrões GoF.

J. Kerievsky, Refatoração para padrões, Bookman, 2008.

Larman, C. *Aplicando UML e Padrões: uma Introdução à Análise e Projeto Orientados a Objetos*, Porto Alegre:Bookman, 2004.
Excelente cápitulo sobre frameworks.

Bibliografia adicional:
Horstmann, C. *Object Oriented Design & Patterns*, Wiley, 2004.

Links interessantes

- Design Patterns Home Page: *http://hillside.net/patterns/*
- Conceitos e Terminologia: *http://choices.cs.uiuc.edu/sane/dpatterns.html*
- The Portland Pattern Repository : *http://www.c2.com/ppr*
- Cetus Links: Patterns, hundreds of links to pattern-related pages
  http://www.objenv.com/cetus/oo_patterns.html
- Brad Appleton's "Software Patterns Links"
  http://www.enteract.com/~bradapp/links/sw-pats.html
- The OrganizationPatterns FrontPage
  http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
- Exemplos de Uso de Design Patterns (Bridge, Decorator, Mediator): *http://www.stevenblack.com/SBC%20Publications.asp*

Tentativas de aumentar reuso (3/3)

- Padrões
  - modelo de estruturas genéricas de classes que colaboram para uma funcionalidade genérica e
    com responsabilidades pré-definidas
  - usada na fase de análise e projeto para incluir o reuso de soluções comprovadas de experiências anteriores bem sucedidas para problemas delimitados

Padrões: conceitos básicos

- Definição:
  - "Um padrão é uma abstração a partir de uma forma concreta que ocorre em vários contextos específicos não arbitrários como solução de um problema."
  - "Um padrão é um "insight" (nomeado) que cobre a essência de uma solução comprovada a um problema recorrente dentro de um determinado contexto."
- Problema: ocorre em um determinado contexto
- Solução: envolve algum tipo de estrutura apropriada neste contexto de problema e que captura um "insight" essencial de tal forma que outros podem aprender e usar em situações similares.
- Alexander: "Each pattern is a three-part rule, which expresses **a relation between a certain context, a problem, and a solution**." Christopher Alexander,*The Timeless Way of Building*, Oxford University Press, 1979.

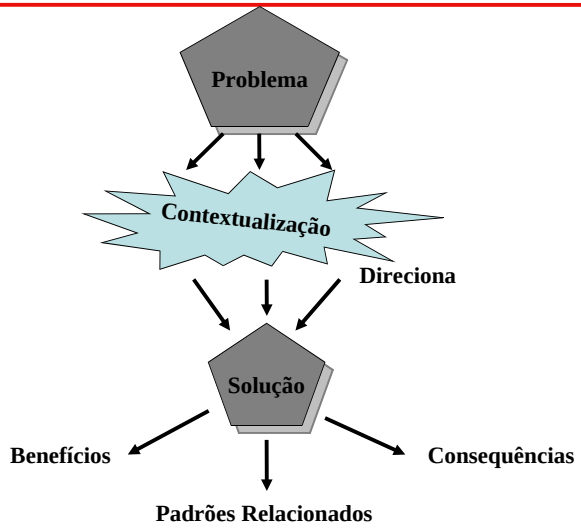
Introdução

- Design pattern é...
  - Uma forma **padrão de organizar classes e objetos;**
  - Nomes para soluções que você já modelou;
  - Uma forma de compartilhar conhecimentos sobre POO;
  - Soluções POO para problemas que incidem em diversos cenários de desenvolvimento;
  - Uma definição de conjunto finito de responsabilidades para uma classe;

Introdução

- Ao adotar design-patterns...
  - Seu código fica mais organizado;
  - Aumenta a qualidade;
  - Diminui a complexidade;
  - Facilita a comunicação dentro da equipe;
  - Facilita a ambientação de novos membros na equipe;
  - Aprende com a experiência dos outros.

Como surgem padrões?



Como documentá-los?

- Elementos de um padrão...
  - Nome
  - Problema
    - *Quando aplicar o padrão, em quais condições?*
  - Solução
    - *Como usar os recursos disponíveis (classes e objetos) para solucionar o problema contextualizado.*
  - Benefícios
  - Conseqüências
    - *Custos de utilização*
    - *Impactos na flexibilidade, portabilidade, performance, etc.*
  - Padrões relacionados

Padrões: conceitos básicos

- Descrição mínima:
  - **Nome**
  - Explicação sucinta do **problema** a que ele se aplica
  - Descrição de como os componentes (classes, relacionamentos, responsabilidades, papéis e colaborações) de um padrão são a **solução** do problema
  - **Conseqüências** através de uma discussão de vantagens e desvantagens para avaliação das alternativas de projeto e compreensão dos custos e benefícios da aplicação do padrão
- GoF = Gang of Four = Erich Gamma + Richard Helm + Ralph Johnson + John Vlissides

  ver *http://hillside.net/patterns/DPBook/GOF.html*

Padrões

- Formato de Descrição mais completo:
  - seção 1.3 Cap. 1 do livro: Erich Gamma et all: *Padrões de Projeto*, Porto Alegre: Bookman, 2000

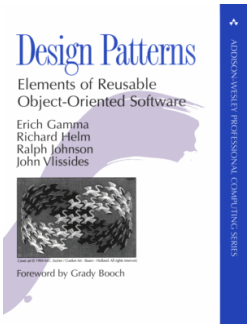
- **Intenção**: descrever sucintamente o que o padrão faz e qual o problema que ele resolve;
- **Motivação** descreve um cenário ou exemplo concreto que ilustra o problema e como a estrutura de classes e objetos do padrão resolve desse problema.
- **Aplicabilidade** são as situações nas quais o padrão deve ser aplicado.

Família de Padrões

- Existem algumas famílias conhecidas de padrões...
  - **GoF (Gang of Four)**
  - **Core J2EE Patterns**
  - GRASP
  - POSA
  - Enterprise Integration Patterns
  - SOA Patterns
  - etc.

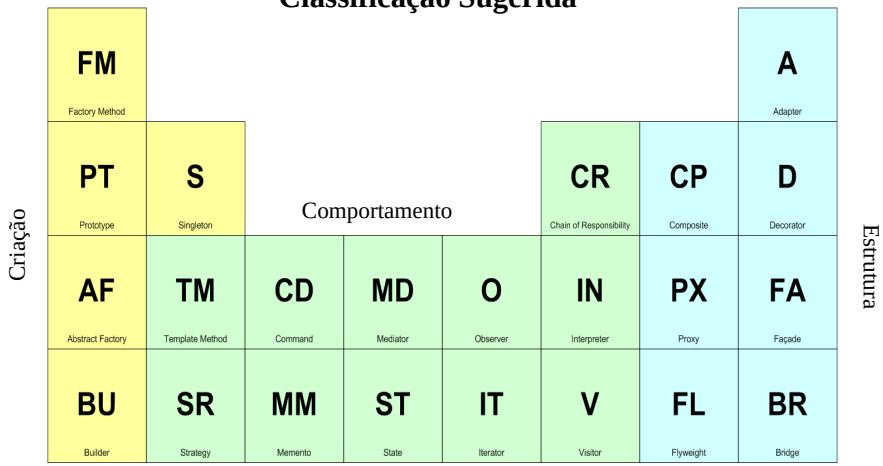
GoF Patterns

- Surgiram em 1995 com a publicação do livro “Design Patterns: Elements of Reusable Object-Oriented Software”;
- Devido ao livro possuir 4 autores, este catálogo de padrões ficou popularmente conhecimento como GoF (Gang of Four);
- Define uma lista com 23 padrões de projeto;
- A publicação deste livro é considerado um marco na evolução e utilização de padrões de projetos dentro dos processos de desenvolvimento de software.



GoF Patterns

Classificação Sugerida



Organização dos 23 padrões GoF [GAM00]

	Propósito		
Escopo	**De criação**	**Estrutura**	**Comportamento**
Classe	Factory Method	Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Padrões de Criação

- Criam objetos ao invés de permitir instanciá-los diretamente, permitindo maior flexibilidade em decidir quais objetos devem ser criados para uma dada situação.
- **Factory** é usado para escolher e retornar uma instancia de classe dentre um numero de subclasses similares de uma mesma classe básica abstrata dependendo dos dados fornecidos para a ´fábrica´;
- **Abstract Factory** provê uma interface para criar e retornar um dentre varias famílias de objetos relacionados; em alguns casos retorna um Factory;
- **Builder** separa a construção de um objeto complexo de sua representação, tal que diferentes representações podem ser criadas dependendo das necessidades do programa;
- **Prototype** copia ou clona uma classe existente ao invés de criar uma nova instância;
- **Singleton** assegura que existe uma e somente uma instância de um objeto e provê uma único ponto global de acesso a esta instância.

Padrões Estruturais

- Ajudam a compor grupos de objetos em estruturas maiores
- **Adapter** é usado para converter a interface de uma classe em outra compatível com a esperada pelos clientes;
- **Bridge** é usado para separar uma abstração de sua implementação, mantendo constante a interface da classe mesmo que a classe seja alterada;
- **Composite** é usado para construir objetos hierárquicos compostos recursivamente;
- **Decorator** é usado para adicionar novas funcionalidades a uma classe dinamicamente , passando à classe subjacente todos métodos inalterados;
- **Façade** é usado para prover uma nova interface unificada a um conjunto de objetos;
- **Flyweight** é usado para criar compartilhamento entre dados de objetos e limitar a proliferação de muitas classes similares pequenas;
- **Proxy** é usado para fornecer um objeto representante de um objeto , controlando o acesso ao mesmo.

Padrões Comportamentais (1/2)

- Ajudam a definir a comunicação entre objetos no sistema e definir como o fluxo de controle deve ser feito em um programa complexo.
- **Observer** define o modo pelo qual um conjunto de objetos pode ser notificado de uma mudança de estado de um objeto usado para prover uma nova interface unificada a um conjunto de objetos;
- **Mediator** define como a comunicação entre classes pode ser simplificada usando outra classe , ou seja, o mediador encapsula como um conjunto de objetos interage para evitar que eles tenham que conhecer uns aos outros;
- **Chain of Responsability** é usado para encadear os objetos receptores de mensagens e passar as solicitações ao longo desta cadeia até que um objeto a trate;
- **Template Method** é usado para prover uma definição abstrata (esqueleto) de um algoritmo, deixando a definição precisa de alguns passos para subclasses.
- **State** permite que um objeto altere seu comportamento quando seu estado interno muda, parecendo que o objeto mudou de classe;

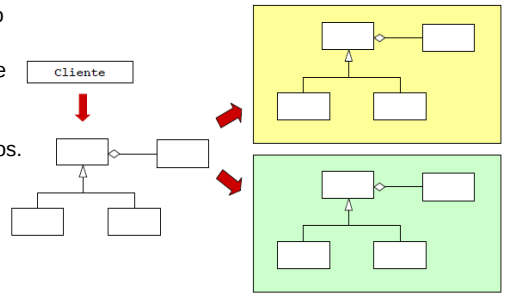
Padrões Comportamentais (2/2)

- **Interpreter** é usado para prover uma definição de como incluir elementos de uma linguagem em um programa de modo que sentenças nesta linguagem possam ser interpretadas;
- **Strategy** é usado para encapsular um algoritmo dentro de uma classe, de modo que o algoritmo possa variar independentemente dos clientes que o utilizam;
- **Visitor** é usado para representar uma operação e adicioná-la a classes sem mudar as classes dos elementos sobre os quais age;
- **Command** é usado para encapsular uma solicitação de operação como um objeto, permitindo que uma execução de um comando seja separada do ambiente que o produziu, e até mesmo enfileirada, registrada (*log*) e desfeita;
- **Iterator** é usado para fornecer um modo de acessar seqüencialmente os elementos de uma lista sem expor sua representação interna;
- **Memento** é usado para capturar o estado interno de um objeto, visando posterior restauração, sem violar o encapsulamento;

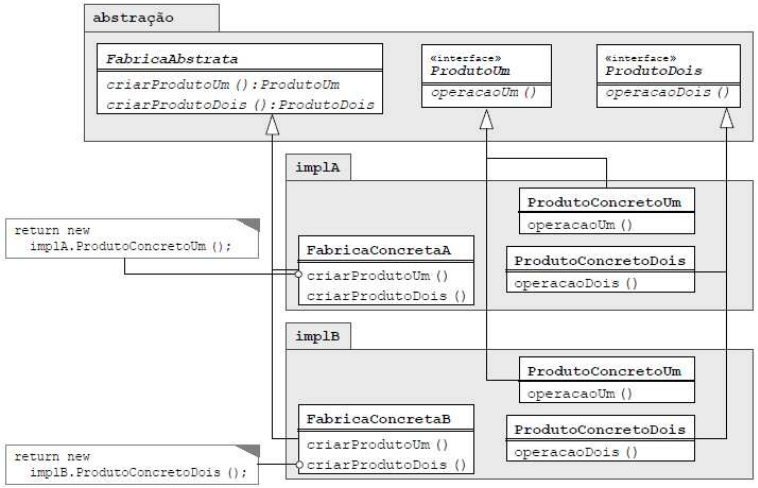
Abstract Factory

> Prover uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

> Benefícios
  > Promover o desacoplamento entre classes da aplicação;
  > Abstrair a lógica de criação e inicialização dos objetos;
  > Tornar facilitada a possível troca entre famílias de objetos.



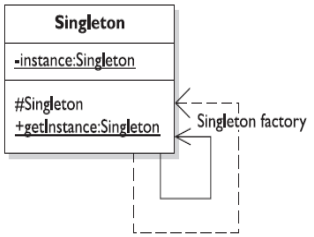
Abstract Factory



Singleton

> Garantir para que uma determinada classe do sistema terá somente um número determinado de instâncias (objeto) criadas, provendo um ponto de acesso global a mesma.

> Benefícios
  > Controlar o acesso as instâncias da classe;
  > Reduzir a utilização desnecessária de memória;
  > Fornecer mais flexibilidade que a utilização de estruturas estáticas;
  > Habilita ter subclasses.



Singleton

```
public class Highlander {
  private Highlander() {}
  private static Highlander instancia = new Highlander();
  public static synchronized Highlander obterInstancia() {
        return instancia;
  }
}
```

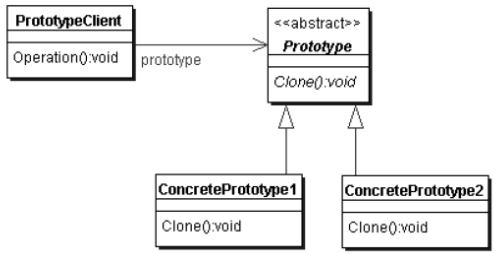
Esta classe implementa o design pattern Singleton

```
public class Fabrica {
  public static void main(String[] args) {
    Highlander h1, h2, h3;
    //h1 = new Highlander(); // nao compila!
    h2 = Highlander.obterInstancia();
    h3 = Highlander.obterInstancia();
    if (h2 == h3) {
       System.out.println("h2 e h3 são mesmo objeto!");
    }
  }
}
```

Esta classe cria apenas um objeto Highlander

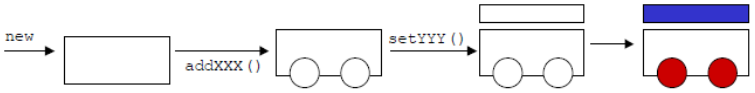
Prototype

> Criar tipo de objetos diferentes, usando como base um protótipo (instância de um objeto com estrutura semelhante).

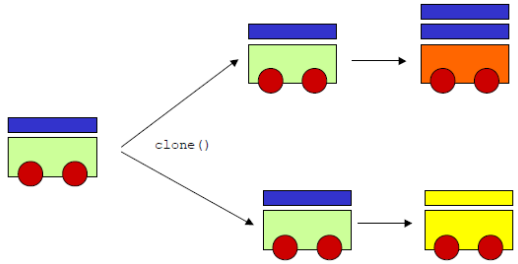


Prototype

Problema



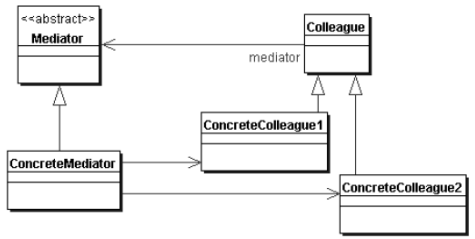
Solução



Mediator

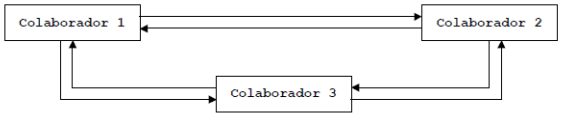
> Definir um objeto que encapsula o modelo como um conjunto de objetos interagem entre si, promovendo o fraco acoplamento.

> Benefícios
  > Desacoplar os diversos participantes;
  > Eliminar relacionamentos N-to-N;
  > Centralizar o controle;
  > Facilitar inclusão de novos participantes.

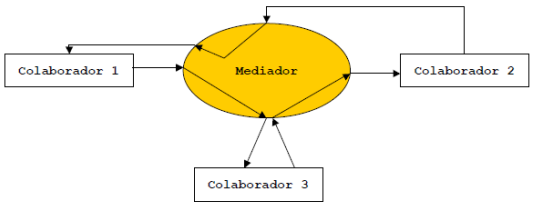


Mediator

Problema



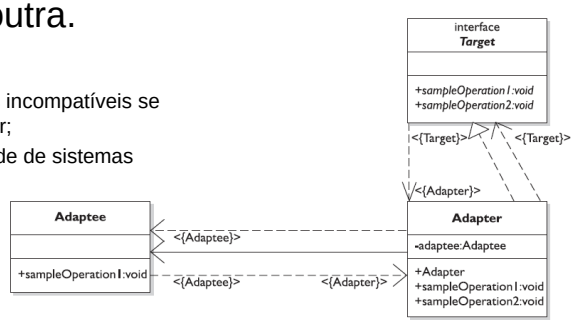
Solução



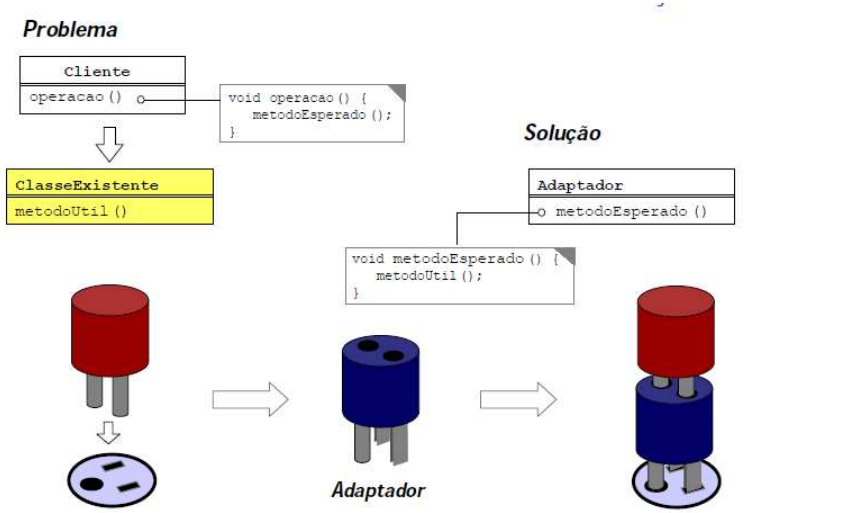
Adapter

> Converter a interface de uma classe em outra interface esperada pelo cliente. Atuar como um intermediário entre duas classes, convertendo a interface de uma para que a mesma possa ser utilizada pela outra.

> Benefícios
  > Permitir dois objetos incompatíveis se comunicar e interagir;
  > Elevar a reusabilidade de sistemas antigos.



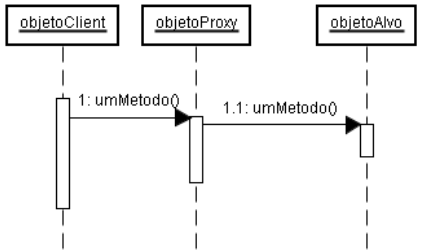
Adapter



Proxy

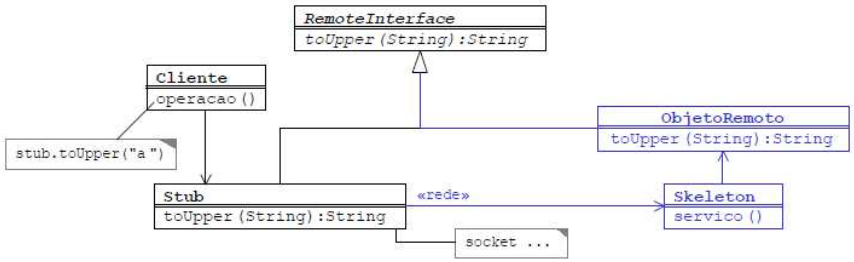
- Prover um objeto substituto para interceptar e controlar o acesso a um outro objeto.

- Benefícios
  - Esconder complexidades relacionadas com o acesso ao objeto destino (acesso remoto);
  - Transparência para o cliente;
  - Permitir maior eficiência com caching no cliente



Proxy

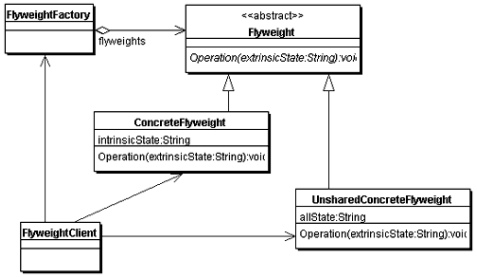
- Exemplo
  - Stubs e Skeletons do Java RMI.



Flyweight

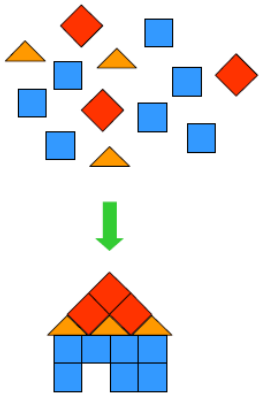
- Utilizar o mecanismo de compartilhamento de instâncias para suportar uma alto número de objetos na aplicação de maneira eficiente.

- Benefícios
  - Reduzir número de objetos a serem tratados pela aplicação;
  - Reduzir utilização de memória;

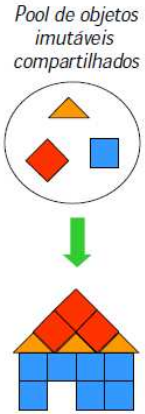


Flyweight

Problema



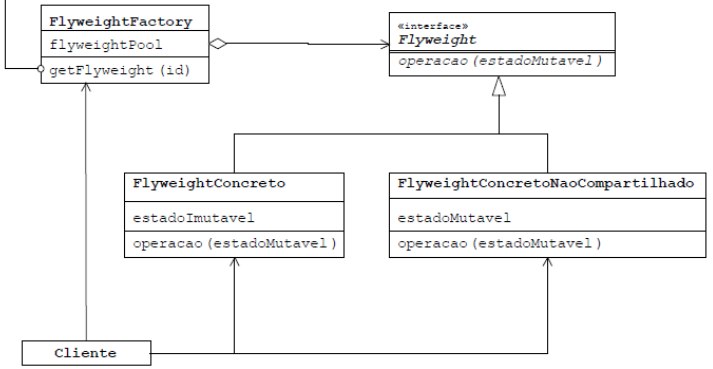
Solução



Flyweight

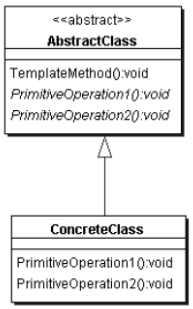
```
if(flyweightPool.containsKey(id )) {
    return (Flyweight)flyweightMap.get(id );
} else {
    Flyweight fly = new FlyweightConcreto ( genKey() );
    flyweightPool.put(fly.getKey (), fly);
    return fly;
}
```

Exemplo Implementação
Usando Caching

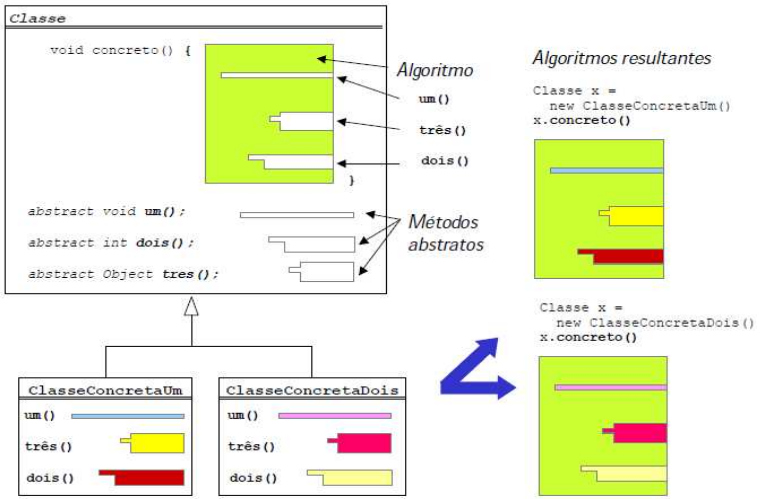


Template Method

> Definir o esqueleto de um algoritmo dentro de uma operação em uma classe, deixando alguns passos a serem preenchidos pelas subclasses.



Template Method



Template Method

```
public abstract class Template {
   protected abstract String link(String texto, String url);
   protected String transform(String texto) { return texto; }
   public final String templateMethod() {
      String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");
      return transform(msg);
   }
```

```
public class XMLData extends Template {
   protected String link(String texto, String url) {
      return "<endereco xlink:href='"+url+"'>"+texto+"</endereco>";
   }
}
```

```
public class HTMLData extends Template {
   protected String link(String texto, String url) {
      return "<a href='"+url+"'>"+texto+"</a>";
   }
   protected String transform(String texto) {
      return texto.toLowerCase();
   }
}
```


Padrões Não-GoF

- *Value Object*
- *Value List Handler*
- *Advanced Search*

Value Object

Intenção:

Reduzir tráfego de rede e melhorar o tempo de resposta no acesso a dados (somente leitura).

Categoria:

Padrão de comportamento.

Value Object

Motivação:

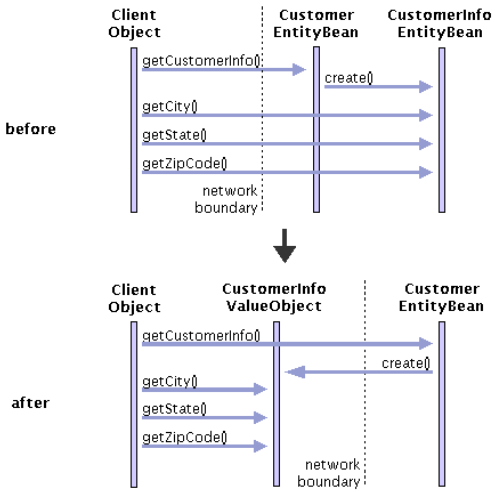
Gerar um objeto a partir de composição de dados de outros, que são frequentemente usados juntos.

Esse objeto composto é mantido no lado cliente (cópia local).

Reduz os acessos ao servidor.

O agrupamento tende a ser por Caso de Uso.

Value Object



Value Object

Aplicabilidade:

Usar quando a classe tem:

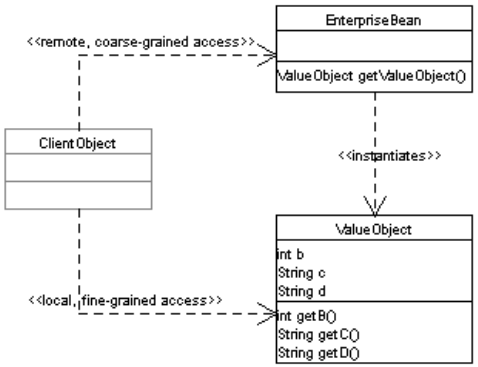
✔dados imutáveis

✔ciclo de vida controlado por outro objeto

✔tamanho relativamente pequeno

Value Object

Estrutura:



Value Object

Participantes:

✔EnterpriseBean

✔ValueObject

Value Object

Colaborações:

No primeiro uso o cliente requisita o Value Object que é instanciado, serializado e devolvido pelo EnterpriseBean.

A partir disso o cliente passa a fazer acesso local aos dados.

A passagem do objeto é feita por valor (serialização)

Value Object

Consequências:

✔Melhora o tráfego de rede e o tempo de resposta:

✔Menor número de chamadas remotas e menor quantidade de dados transferidos.

✔Reduz a carga no servidor (EnterpriseBean).

Value Object

Implementação:

✔O ValueObject é uma estrutura de dados simples que contem propriedades privadas e métodos *Get.*

✔Deve ser *Serializable.*

✔Deve ser imutável para reforçar a idéia de que não é um objeto remoto e que qualquer alteração de seu estado não refletirá no servidor.

Value Object - Referências

Site SUN J2EE Design Patterns

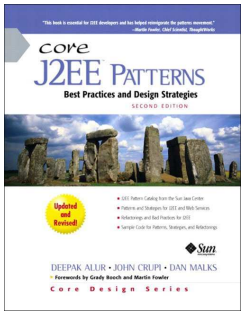
http://java.sun.com/blueprints/patterns/j2ee_patterns/value_object/

Site Catálogo de Patterns Aplicação exemplo J2EE Blueprints

http://www.iplanet.ne.jp/developers/ias-samples/jps1.1.1 /docs/patterns/ValueObject.html

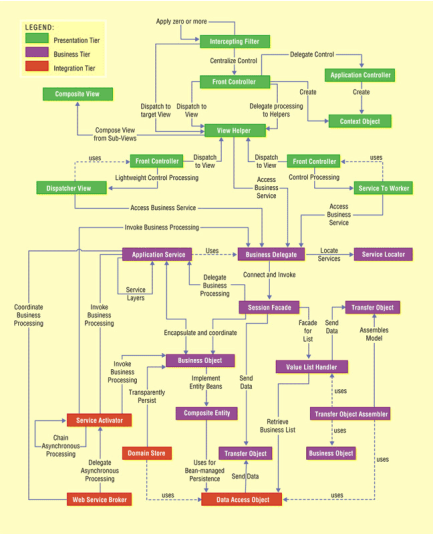
Core J2EE Patterns

- Surgiu com a publicação do livro Core J2EE Patterns em 2001;
- Descreve um catálogo de 25 padrões específicos para plataforma Java EE;
- Produto de anos de experiência aplicados em consultoria em projetos Java EE, documentados por consultores da Sun Microsystems.
- Atualmente este livro encontra-se publicado em segunda edição, com alguns “novos” design patterns;



Core J2EE Patterns

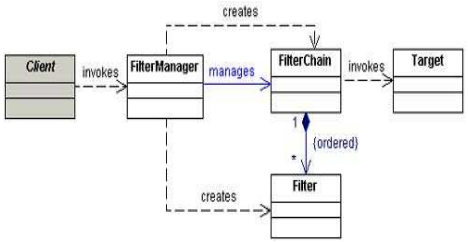
- Os padrões encontram-se sub-divididos em três categorias:
  - Apresentação
  - Negócio
  - Integração



Intercepting Filter

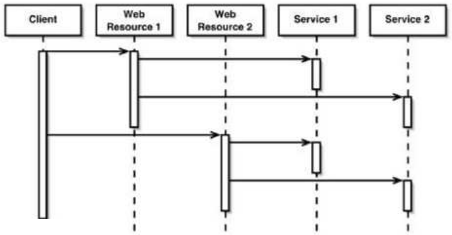
- Permitir o pré e/ou pós processamento de uma requisição para um determinado componente, possibilitando a facilidade na configuração de ativação e desativação deste processamento.

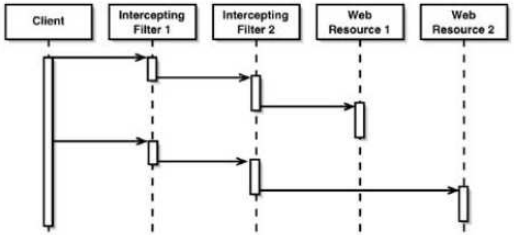
- Benefícios
  - Centralizar controle;
  - Promover a reusabilidade;
  - Fornecer flexibilidade através de configurações declarativas;



Intercepting Filter

Problema ->



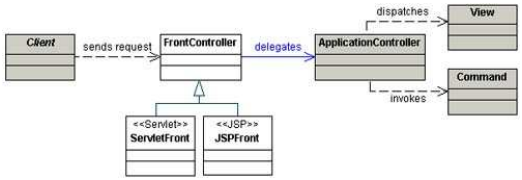


<- Solução

Front Controller

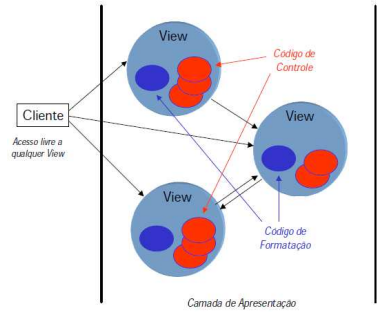
- Centralizar o processamento de requisições em uma único e centralizado componente. Redirecionar o processamento após sua finalização, para a view respectiva.

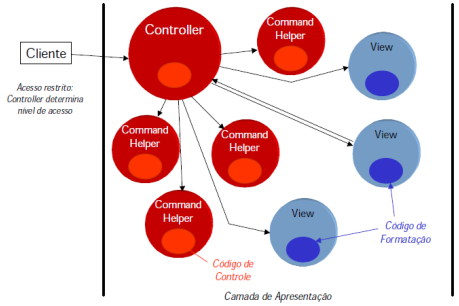
- Benefícios
  - Controle centralizado;
  - Melhorar gerenciamento de segurança;
  - Promover reuso;



Front Controller

Problema ->

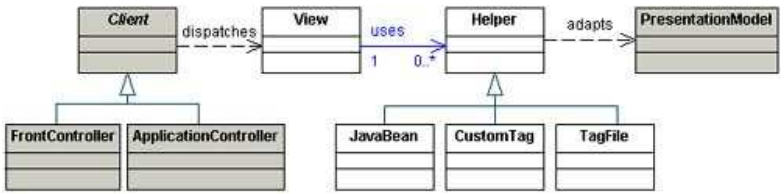




<- Solução

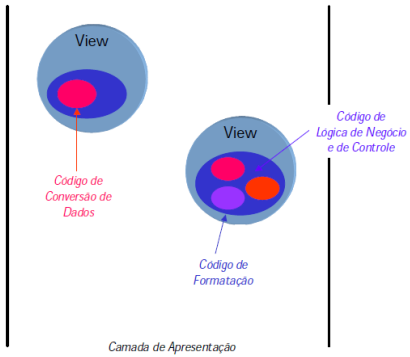
View Helper

- Separar do código as responsabilidades de formatação da interface do usuário, do processamento de dados necessário à construção da view.

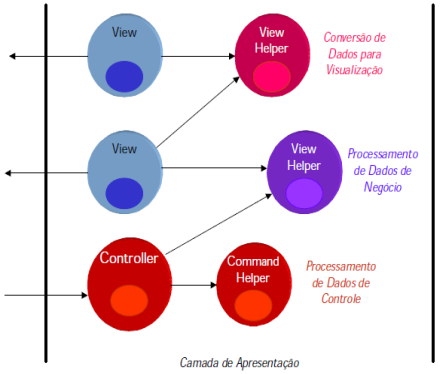


View Helper

Problema

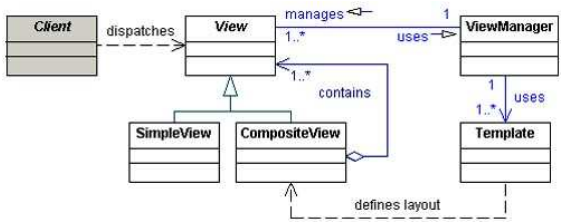


Solução



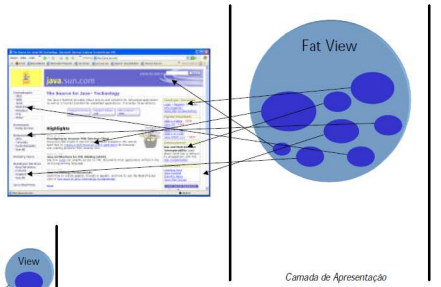
Composite View

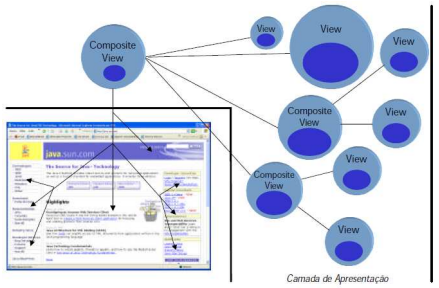
- Componentizar a view para a partir de views menores dividir as responsabilidades, simplificar a construção da interface e promover o reuso.



Composite View

Problema ->

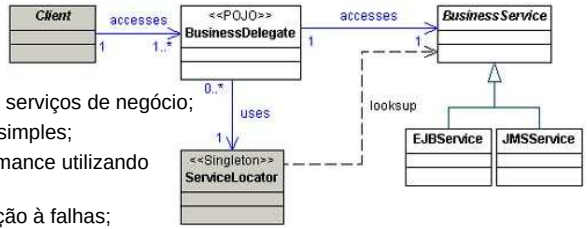




<- Solução

Business Delegate

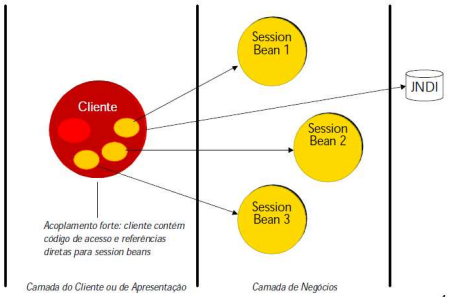
- Esconder dos clientes detalhes acerca da camada de negócios, fornecendo uma interface de serviços semelhantes aos serviços de negócio.

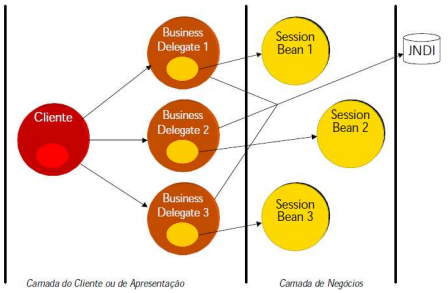


- Benefícios
  - Reduzir acoplamento;
  - Traduzir exceções dos serviços de negócio;
  - Expor interfaces mais simples;
  - Poder melhorar performance utilizando estratégias de cache;
  - Implementar recuperação à falhas;
  - Ocultar o fato dos objetos de negócio estarem remotos.

Business Delegate

Problema ->

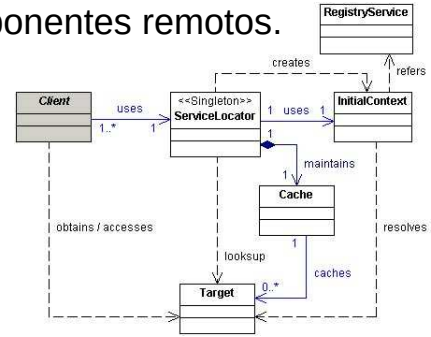




<- Solução

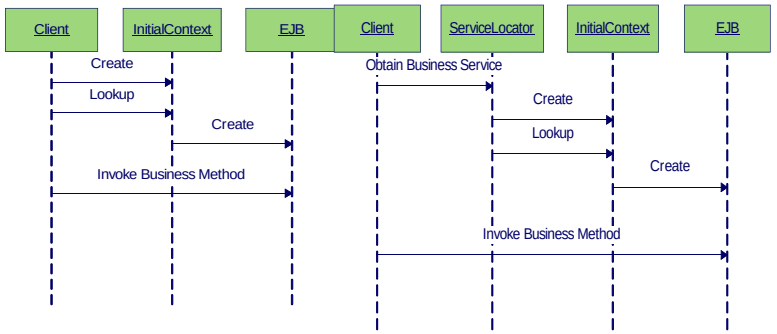
Service Locator

- Esconder dos clientes a necessidade do conhecimento dos serviços de localização (JNDI) e da lógica necessária para utilização do mesmo, fornecendo uma interface simplificada para recuperar os componentes remotos.



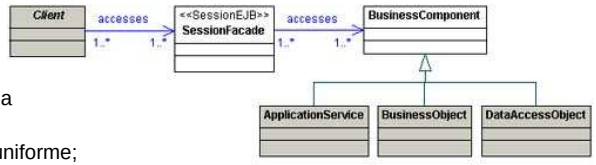
Service Locator

Problema **Solução**



Session Façade

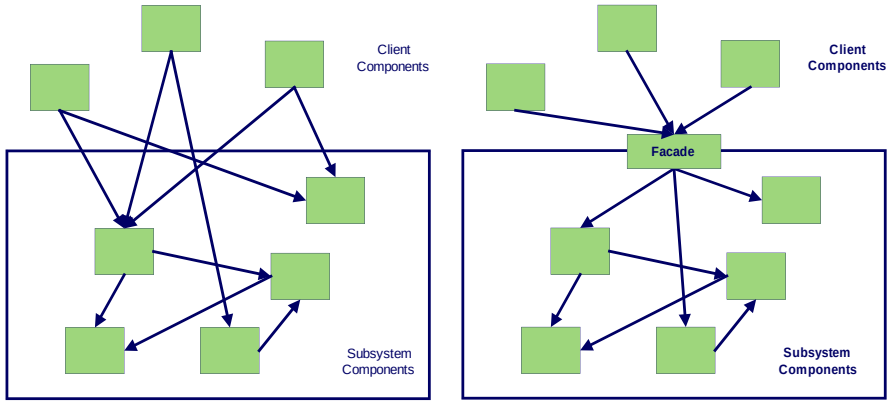
- Simplificar a interface do cliente dos componentes de negócio e controlar o acesso e a lógica de negócio entre os componentes existentes.



- Benefícios
  - Introduzir uma camada controladora;
  - Expor uma interface uniforme;
  - Reduzir o acoplamento do cliente;
  - Melhorar a performance
  - Centralizar o controle de segurança e transações;
  - Reduzir a interface visível para o cliente.

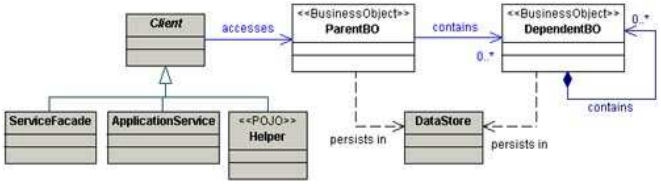
Session Facade

Problema **Solução**



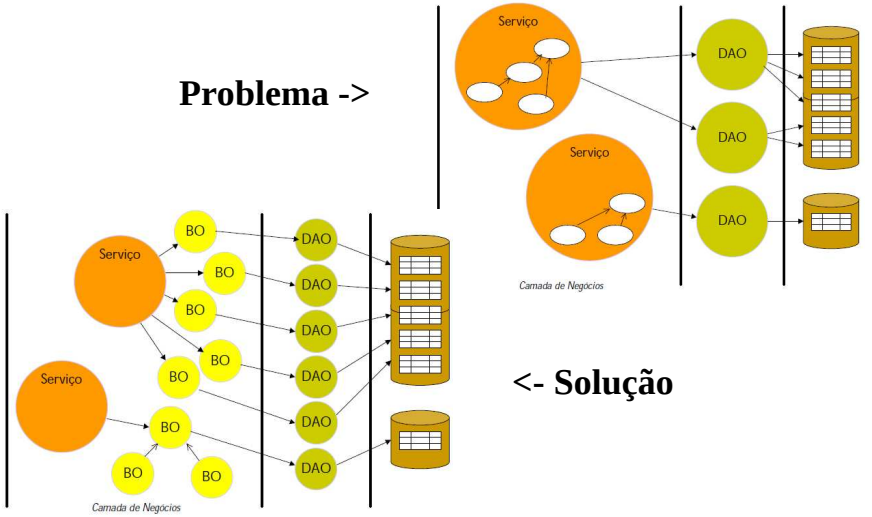
Business Object

> Separar dados de negócio da lógica usando um modelo de objetos. Abstrair os dados de negócio da aplicação, representando uma entidade.



Business Object

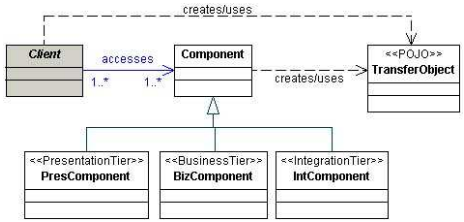
Problema ->



<- Solução

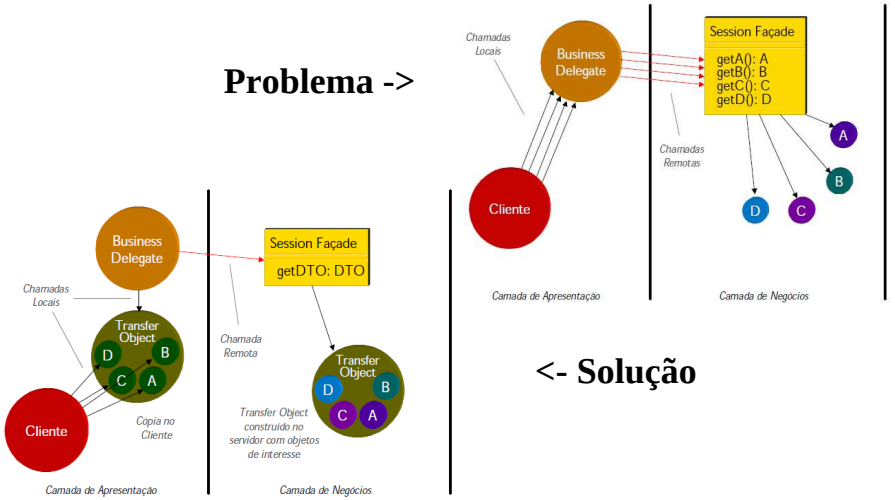
Transfer Object

> Reduzir a quantidade de requisições necessárias para recuperar um objeto. Encapsular um subconjunto de dados a ser utilizado pelo cliente, afim de retorná-los em somente uma requisição remota.



Transfer Object

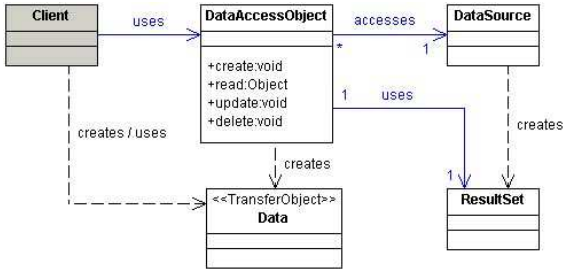
Problema ->



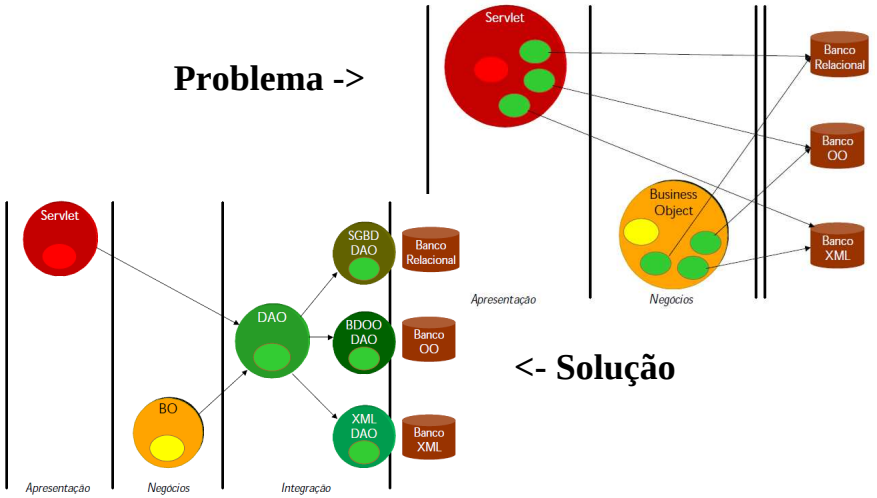
<- Solução

Data Access Object

> Abstrair e encapsular todo o acesso a uma fonte de dados, separando-a do código de negócio e visualização da aplicação.

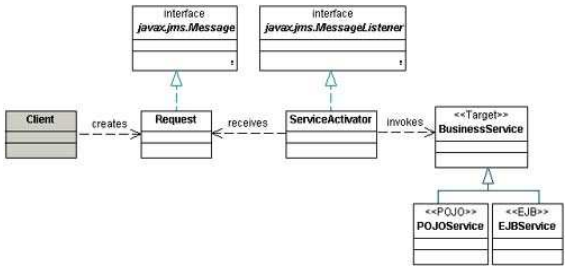


Data Access Object

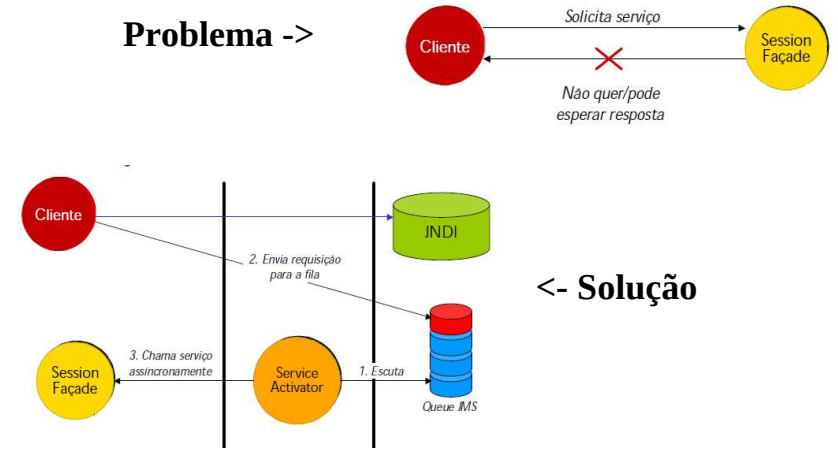


Service Activator

> Receber requisições e mensagens assíncronas do cliente. Localizar e chamar os métodos de negócio para atender as requisições de forma assíncrona.



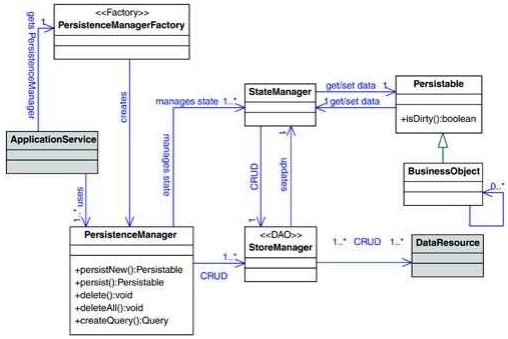
Service Activator



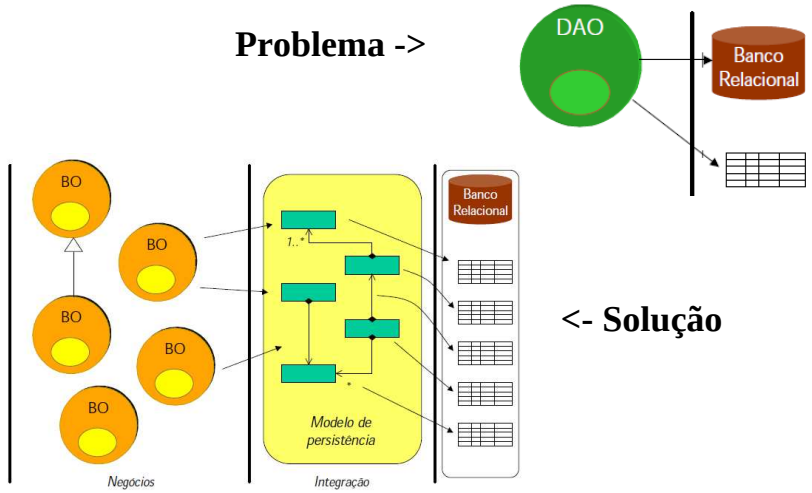
Domain Store

> Oferecer um mecanismo transparente para persistência dos objetos de negócio. Abstrair o repositório de dados do cliente, afim de fornecer um mecanismo de persistência automático.

> Benefícios
  > Separar modelo de objetos de negócio da lógica de persistência;
  > Melhorar testabilidade da camada de persistência;

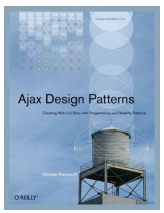


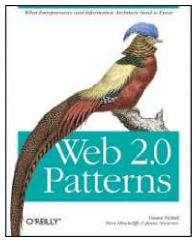
Domain Store

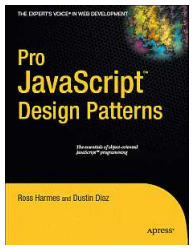


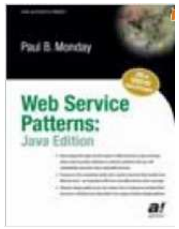
<- Solução

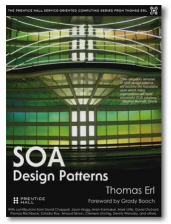
Outros Design Patterns...

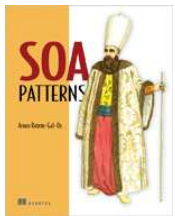


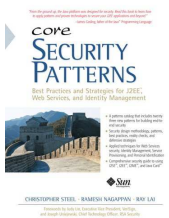


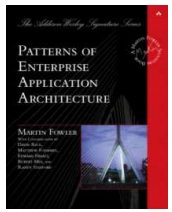












Conclusões

> Será que alguns padrões de projetos morreram com a evolução das novas tecnologias?

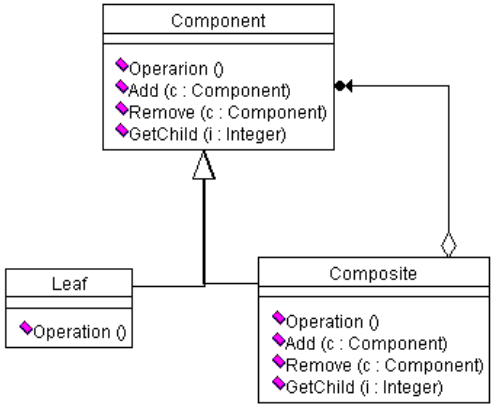
> Devo realmente utilizar padrões de projetos na minha aplicação?

> Qual será o futuro dos padrões de projetos? Terão eles um fim?

Conceitos e Exemplos

- *Composite*
- *Template Method*
- *Singleton*
- *State*

Composite

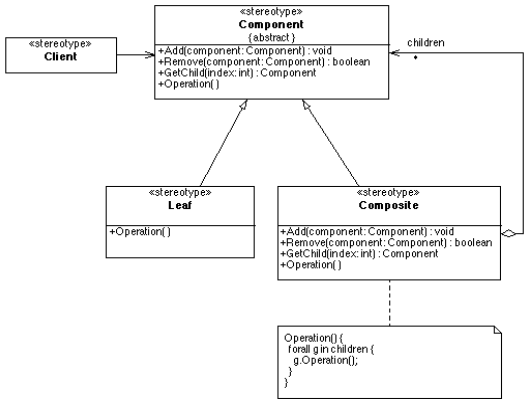


Composite

- Objetivo:
  - Representação de hierarquias partes-todo de objetos:
    - »permitir a composição de estruturas em árvores na representação de composição de objetos;
    - »possibilitar o tratamento uniforme tanto a objetos individuais como compostos
  - Utilização da composição recursiva para evitar a necessidade de distinção entre os objetos:
    - » útil quando se quer que os clientes ignorem as diferenças entre composições de objetos e objetos individuais, tratamento uniforme aos objetos;
  - apropriado para modelagem de interfaces gráficas.

Composite

Estrutura:



Composite

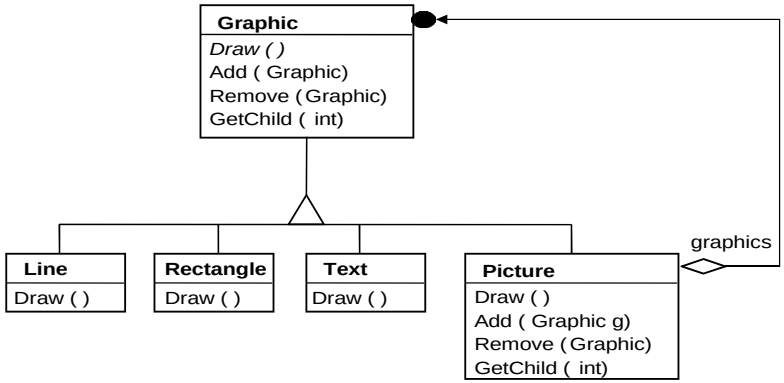
- Component
  - declara a interface para os objetos na composição
  - implementa comportamento por falta para a interface comum a todas as classes
  - declara uma interface para acessar e gerenciar os seus componentes filhos
  - define uma interface para acessar o pai de um componente na estrutura recursiva e a implementa (opcional)
- Leaf (folha)
  - representa N objetos-folha (possivelmente 1) na composição
  - define comportamento para objetos primitivos na composição

Composite

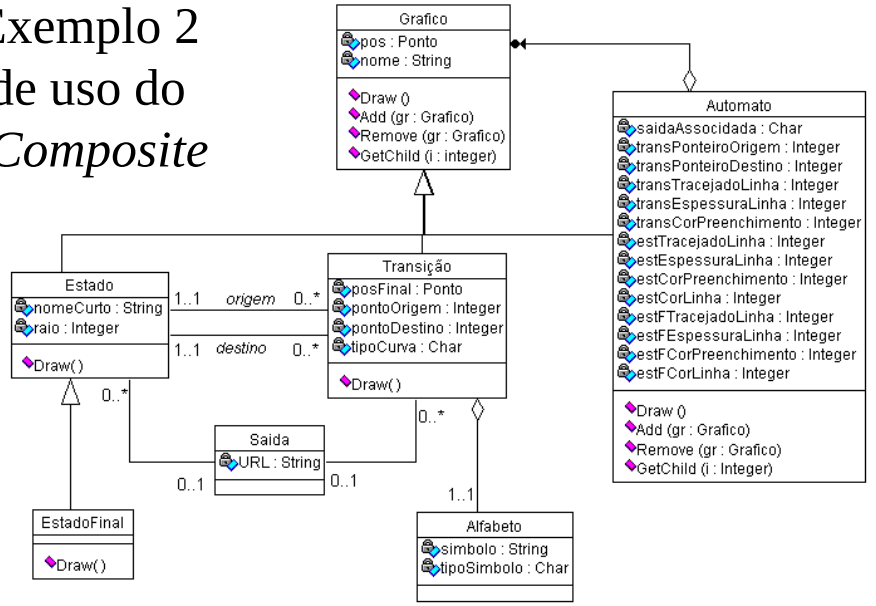
- Composite
  - define comportamento para objetos que têm filhos
  - armazena os componentes-filho
  - implementa as operações relacionadas com os filhos presentes na interface de component
- Client
  - manipula os objetos na composição através da interface de component

Exemplo 1 de uso do *Composite*

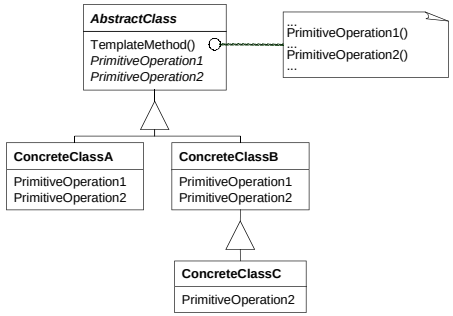
Editores de desenhos: construir diagramas complexos a partir de componentes simples



Exemplo 2 de uso do *Composite*



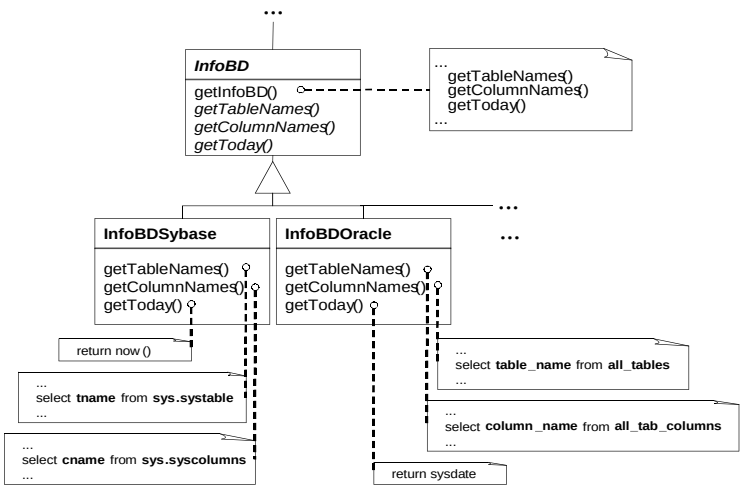
Template Method



Template Method

- Objetivo:
  - Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses: subclasses redefinem certos passos de um algoritmo sem mudar a estrutura do mesmo;
- O padrão *Template Method* pode ser usado:
  - ➢ para implementar as partes invariantes de um algoritmo uma só vez e deixar para subclasses a implementação do comportamento variável;
  - ➢ para fatorar o comportamento comum entre subclasses e concentrá-lo numa classe comum para evitar a duplicação de código;
  - ➢ para controlar extensões de subclasses. O usuário pode definir um método *template* que chama operações "gancho" em pontos específicos, desta forma permitindo extensões somente nesses pontos.

Exemplo de uso do *Template Method*



Singleton

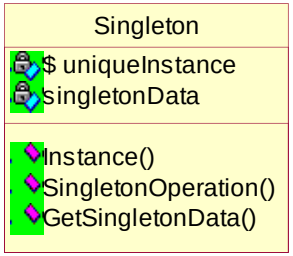
- Definição
  - Classe com somente uma instância
  - Fornece um único ponto de acesso
- Motivação
  - Muitas vezes há necessidade de garantir a existência de uma única instância
  - Exemplos
    - Gerenciador de impressão
    - Sistema de arquivos

Singleton

- Aplicabilidade
  - necessidade de existir somente uma instância de uma classe
  - deve ser acessada a partir de um ponto conhecido
  - instância única tem que ser extensível às subclasses
  - utilizar a instância estendida sem modificar seus códigos.

Singleton

- Estrutura



Singleton

- Conseqüências
  - Acesso controlado da instância única.
  - Evita a utilização de variáveis globais para o controle da instância única.
  - Permite um refinamento de operações e da representação uma vez que é possível subclasses de Singleton utilizarem sua única instância.

Exemplo 1 - *Singleton*

```
public class DBConnection {
        static private DBConnection instance=null;
        private Connection con[];
        private ResultSet rs[][];
        private Statement stmt[];
        final int MAX_CONNECTIONS = 10;
        final int INCR_CONNECTIONS = 5;
        final static int INIT_CONNECTIONS = 2;
        private int totalCon;
        private Hashtable conexoes;

        public static DBConnection getInstance(){
            if (instance==null){
                    instance = new DBConnection(INIT_CONNECTIONS);
            }
          return instance;
        }
}
```

Exemplo 2 - *Singleton (Spool)*

Definição da Classe

```
class PrintSpooler
{
//spool de impressora
static boolean
instance_flag=false; //Verdadeiro se 1 instancia
public PrintSpooler() throws SingletonException
{
if (instance_flag)
throw new SingletonException(´So um spool permitido´);
else
instance_flag = true; //liga flag para 1 instancia
System.out.println(´Spool aberto´);
}
//----------------------------------------
public void finalize()
{ instance_flag = false; //limpa se destruido }
}
```


Exemplo 2 - *Singleton (Spool)*

Instanciação da Classe

```
public class singleSpooler
{
static public void main(String argv[])
{
PrintSpooler pr1, pr2;
//Abrir um spool - deve SEMPRE funcionar
System.out.println(´Abrindo um spool´);
try{
pr1 = new PrintSpooler();
}
catch (SingletonException e)
{System.out.println(e.getMessage());}
//Tentar abrir outro spool - deve falhar
System.out.println(´Abrindo 2 spools´);
try{
pr2 = new PrintSpooler();
}
catch (SingletonException e)
{System.out.println(e.getMessage());}
}
}
```

Executando o programa:

```
Abrindo um spool
Spool aberto
Abrindo 2 spools
So um spool permitido
```


State

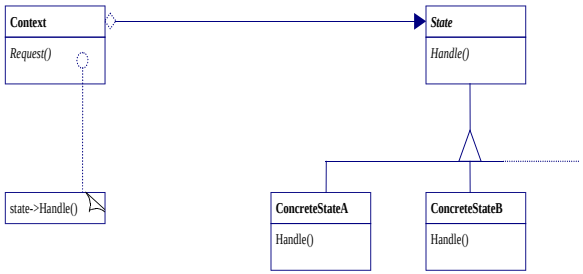
Intenção:

- Permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.

Aplicabilidade:

- quando o comportamento de um objeto depende de seu estado e ele pode mudar em tempo de execução, dependendo desse estado;
- quando operações têm comandos condicionais grandes, de várias alternativas que dependem do estado do objeto.

State



State

Participantes:

Context: define a interface de interesse para os clientes, mantém uma instância de uma subclasse ConcreteStage que define o estado corrente;

Stage: define uma interface para encapsulamento associado com um determinado estado da classe Context;

Subclasses ConcreteState: implementa um comportamento associado com um estado da classe Context.

- A classe Context delega solicitações específicas de estados para o objeto corrente ConcreteState. Um contexto pode passar a si próprio como um argumento para o objeto State que trata a solicitação, permitindo que esse acesse o contexto necessário. Tanto Context quanto as subclasses de ConcreteStage podem decidir qual o estado sucede o outro, e sob quais circunstâncias.

State

- **Conseqüências:**

- Confina o comportamento específico de estados e particiona o comportamento para estados diferentes;
- Também torna explícitas as transições de estado;
- Objetos State podem ser compartilhados.

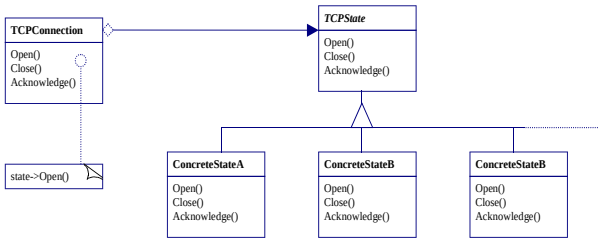
- **Implementação:**

- Ao implementar um padrão **State**, alguns aspectos particulares devem ser considerados:
  - Quem define as transições de estado, Context ou as subclasses de State;
  - O uso de tabelas para mapear entradas para transições de estados;
  - Quando criar e destruir os objetos;
  - Utilizar herança dinâmica, se a linguagem de programação utilizada permitir.

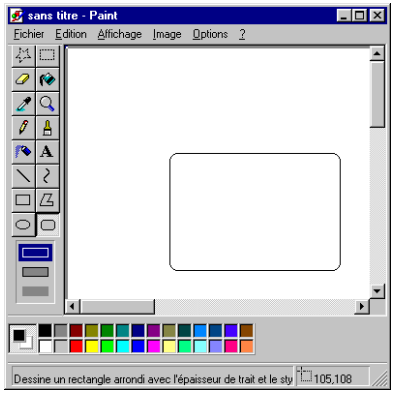
Exemplo de uso do *State*

- Considere uma classe TCPConnection que representa uma conexão em uma rede. Esta classe pode estar em diversos estados: estabelecida, na escuta, na transmissão e fechada. A forma como o objeto TCPConnection irá responder depende de seu estado corrente. A idéia-chave deste padrão é introduzir uma classe abstrata TCPState para representar os estados da conexão na rede. Cada estado de conexão implementa um comportamento específico, através de subclasses concretas da classe TCPState abstrata. A classe TCPConnection instancia uma subclasse de TCPState que representa o estado corrente da conexão. Todas as solicitações são executadas pelo objeto da subclasse TCPState instanciada. Quando a conexão muda de estado, o objeto TCPConnection subtitui a instância do objeto da subclasse TCPState utilizada.

Exemplo de uso do *State*

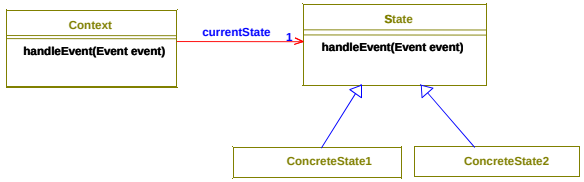


The "State" pattern



- Objetivo: permitir a um objeto mudar comportamento quando seu estado interno muda; o objeto comporta-se como se sua classe mudasse.
- Exemplo típico: editor de diagramas com uma barra de ferramentas
  - Diferentes "modos de operação"
  - Sempre manipula os mesmos eventos (de mouse) mas de formas diferentes

General structure



- Participantes
  - Contexto :
    - Define os eventos possíveis
    - Mantém uma instância de State (classe abstrata) que define qual é o estado corrente
  - State
    - Classe abstrata definindo a interface de cada classe concreta
  - ConcreteState (derivada de State)
    - Cada subclasse define o comportamento associado a um particular estado do contexto

Consequências

- Comportamento dependente de estado é localizado em subclasses específicas
  - Não há mais "if" or "switch" no controlador
- Fácil introduzir novo estado (p.ex. novas ferramentas na barra de ferramentas do editor de diagramas)
  - Definir uma nova subclasse
  - Instanciá-la propriamente
- Faz as mudanças de estado mais explícitas

Detalhes Específicos

- Como as mudanças de estado são realizadas?
  - Pelo contexto (o contexto implementa um diagrama de estados finitos)
  - Por cada subestado
    - Cada estado sabe seu(s) estado(s) seguinte(s)
    - Implentação distribuída da máquina de estados
- Como os estados/objetos são instanciados?
  - Pré-instanciação
  - Instanciação dinâmica quando necessário

Mais Tentativas de aumentar reuso

- Frameworks
  - estrutura inacabada de classes que colaboram para um dominio de aplicação
  - usada para através de especialização e intanciação de metodos e classes especificas gerar mais rapidamente aplicações do dominio do framework.
- Componentes
  - caixa preta de funcionalidades programadas para serem usadas de forma a serem combinadas com outras para a geração de aplicações

Frameworks

Definição

- *Conjunto de objetos que colaboram com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação.*

  Erich Gamma,Ralph Johnson, John Vlissides e Richard Helm

- *Software parcialmente completo (subsistema) projetado para ser instanciado.*

  Frank Buschmann

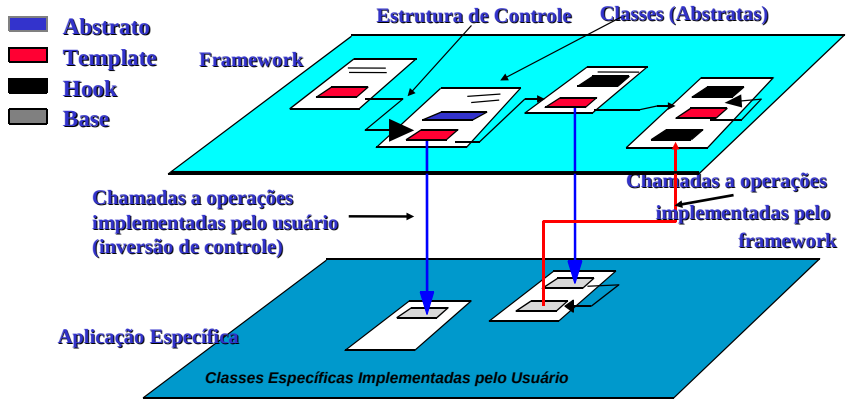
Frameworks

Classificação de Frameworks de aplicação

- Frameworks de Infra-Estrutura
- Frameworks de Domínio Específico
- Frameworks de Aplicação Corporativos
  - Frameworks Próprios (feitos pela coprporação)
  - Frameworks de Mercado (adotados pela corporação)

Frameworks Orientados a Objetos

Estrutura de aplicação pré-implementada e reusável, constituída de um conjunto de classes abstratas



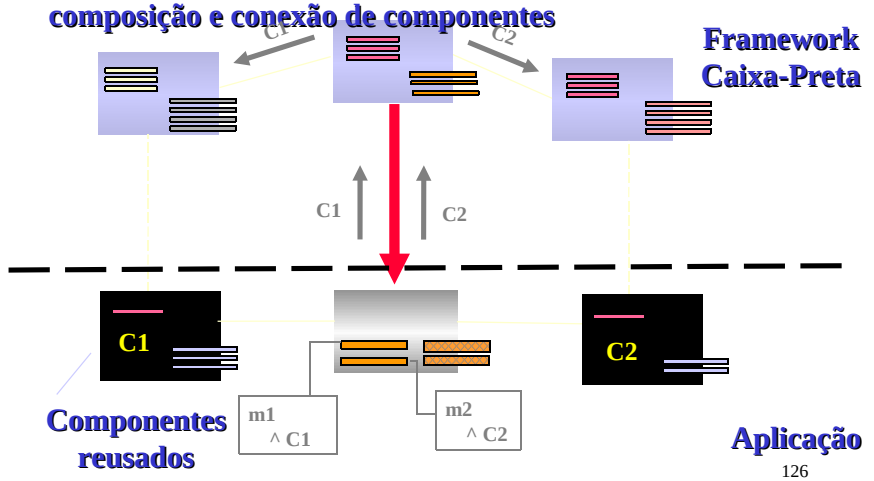
Visão estrutural de aplicação construída a partir de framework

Frameworks: Conceitos

- **Um framework é específico para um domínio de aplicação**
- **Áreas de aplicação:**
  - ***Frameworks de infra-estrutura***
  - ***Frameworks de aplicação***
  - ***Frameworks para domínios específicos***
- ***Instanciação do framework* = construção de uma aplicação específica**
- ***Frameworks caixa-branca:* Instanciados predominantemente através da escrita de subclasses**

Frameworks: Conceitos

- ***Frameworks caixa-preta:* São instanciados através da composição e conexão de componentes**



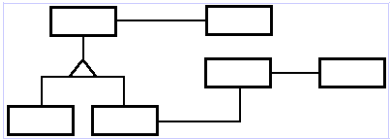
Frameworks: Visão Geral

- **Estrutura de Classes (Enfoque OO)**
- **Software Incompleto (com lacunas a serem completadas pelo projetista)**
- **Domínio da aplicação**

Frameworks: Visão Geral

Desenvolvendo aplicações OO do “zero”

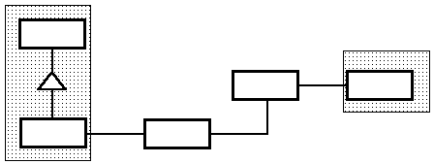
- Definir todas classes e todas colaborações



Frameworks: Visão Geral

Desenvolvendo aplicações reusando classes

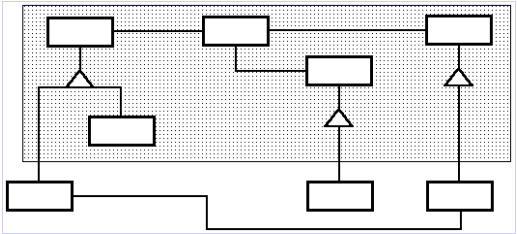
- Definir algumas classes mas todas colaborações



Frameworks: Visão Geral

Desenvolvendo aplicações a partir de um framework

- Definir/redefinir algumas classes
- Definir algumas colaborações



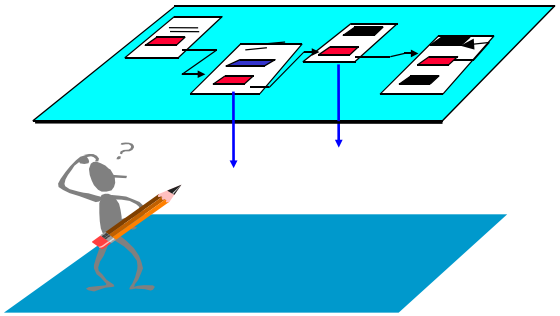
Frameworks: Visão Geral

Framework:

- tem embutido o projeto da aplicação
- estabelece o fluxo de controle da aplicação

Frameworks: O Problema

O usuário precisará estender e adaptar uma estrutura de software, a qual não criou!



A inversão de controle, implica na necessidade de compreende previamente o framework para construção de aplicações

Frameworks: O Problema

Conhecimento que o usuário precisa ter sobre o framework depende:

1. **Quantidade de adaptações necessárias no framework**
2. **A forma como as mudanças poderão ser feitas**

Compreender frameworks implica em entender:

- **Mapeamento de conceitos do domínio de aplicação em estruturas de classe**
- **Protocolos de colaboração e fluxo de controle**
- **Pontos adequados para adaptação**

Frameworks: O Problema

A organização do framework nem sempre é obvia

- **Com objetivo de flexibilidade e extensibilidade, projetistas utilizam soluções de projeto sofisticadas e abstratas - exemplo: padrões de projeto**

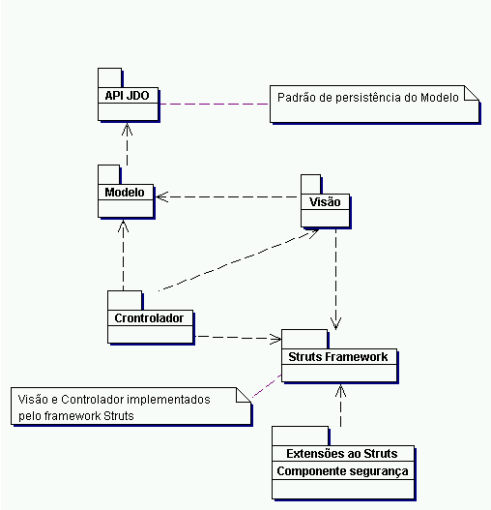
A compreensão detalhada do fluxo de controle é mais complexa em programas orientados a objetos

- **Diferentes linhas de execução (*threads*)**
- **Numerosa troca de mensagens entre objetos**

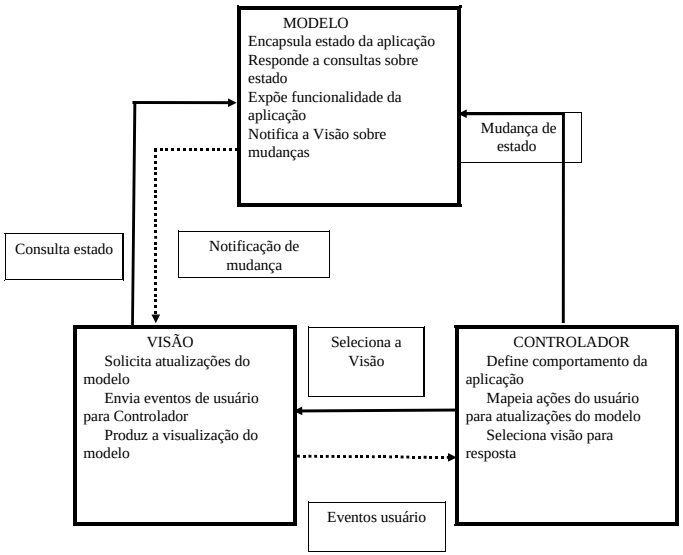
Frameworks: Revisão

- **Estrutura de Classes**
- **Esquemas de colaboração definidos**
- **Software Incompleto (com lacunas a serem completadas pelo projetista)**
- **Reuso através de especialização e instanciação de métodos e classes específicas**
- **Domínio da aplicação**.

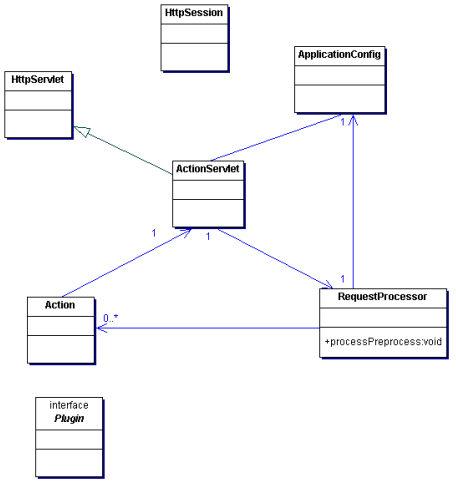
Ex. de fmwk: Struts



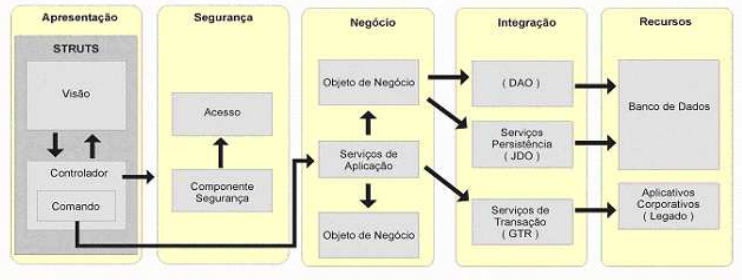
MVC: base do Struts



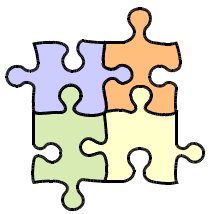
Controladores do Struts



Arq. Aplic transacionais na Web com Struts



Desenvolvimento Baseado em Componentes (DBC)



Desenvolvimento Baseado em Componentes (DBC)

- Visão clássica de desenvolvimento
  - SW = "Blocos" monolíticos
  - Grande número de partes inter-relacionadas
  - num mundo "integral"
- ...um cenário em que DBC
  - "quebra" tais blocos {simplificar!}…
  - para reduzir complexidade e custo de desenvolvimento…
  - num mundo distribuído…

Componentes de Software

- Definições Diversas/Divergentes
  - Bertrand Meyer [1999]: "a software *element* that must be usable by developers who are not personally known to the component's author to build a project that was not foreseen by the component's author."
- Visões de um componente
  - Elemento arquitetural
  - Elemento implementacional
  - Componente de negócio

Componentes de Software

- Aspectos de um componente
  - Descrever ou realizar uma função específica
  - Estar em conformidade e prover um conjunto de interfaces definidas
  - Ter uma documentação adequada
  - Estar inserido no contexto de um modelo que oriente a composição deste componente com outros
- Categorias [Williams, 2001]
  - Componentes GUI
  - Componentes de Serviços
  - Componentes do Domínio [Negócio]

Componentes de Software

"Componentes reutilizáveis são artefatos autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras em conformidade com um dado modelo de arquitetura de software, documentação apropriada e um grau de reutilização definido."

Sametinger, 1997

a idéia NÃO é nova...

- ***Mass Produced Software Components***, Doug McIlroy
- **NATO Software Engineering Conf., Garmisch, Germany, 1968**

http://cm.bell-labs.com/cm/cs/who/doug/components.txt

Mass Produced Software Components, '68

- McIlroy e a indústria de componentes:

  ...to see **standard catalogues of routines**, **classified** by precision, robustness, time-space performance, size limits, and binding time of parameters

  ...to apply routines in the catalogues to any one of a larger class of often quite different machines; **to have confidence in the quality of the routines**

  ...[I want] different types of routine in the catalogue that are similar in purpose to be engineered uniformly, so that two similar routines should be available with similar options and two options of the same routine should be **interchangeable** in situations indifferent to that option

De McIlroy até agora...

- pesquisa/desenvolvimento/evolução de
  - Domain Engineering
  - Component-Based Development
  - Repository Systems
  - Product Lines
  - ...
- cujo principal objetivo é atingir...

Systematic Software Reuse

Systematic Software Reuse

"Systematic Software Reuse is **domain** focused, based on a **repeatable** process, and concerned primarily with the **reuse of higher level lifecycle artifacts**, such as requirements, design, and subsystems"

William B. Frakes, 1994

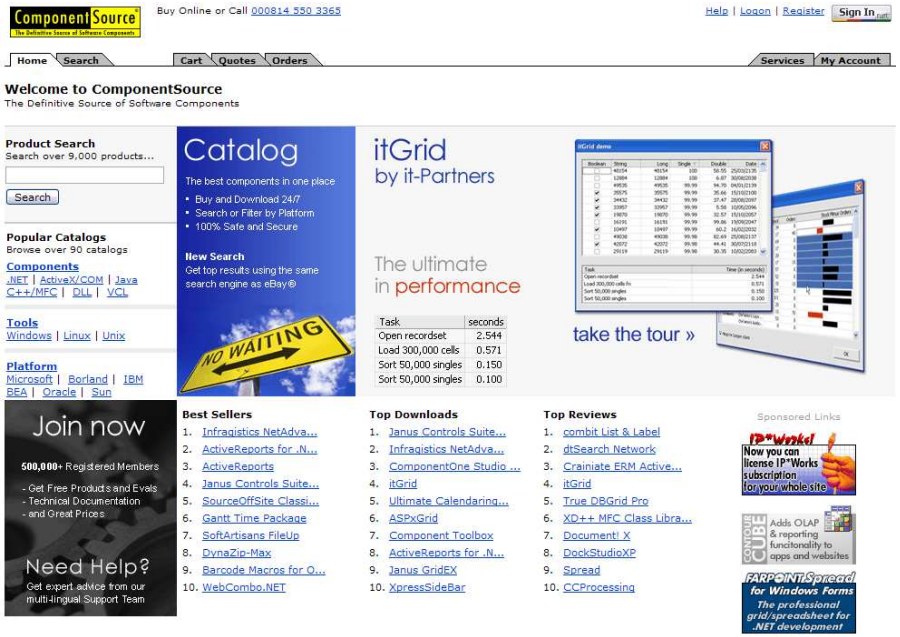
Porque ainda não decolou.....

- Pesquisa conduzida pelo Software Engineering Institute (SEI) durante 1999-2000 [SEI, 2000]
  - Economistas, analistas industriais, gerentes e engenheiros de software
- Análise de componentes de software
  - Visão técnica e de negócio

os inibidores são...

- Carência de componentes disponíveis
  - para 30%... falta uma indústria
  - para 20%... faltam componentes em domínios
- Carência de padrões para tecnologia de componentes
  - 30% lembraram a instabilidade dos padrões de componentes
- Carência de componentes certificados
- Carência de métodos para CBSE

mesmo assim…



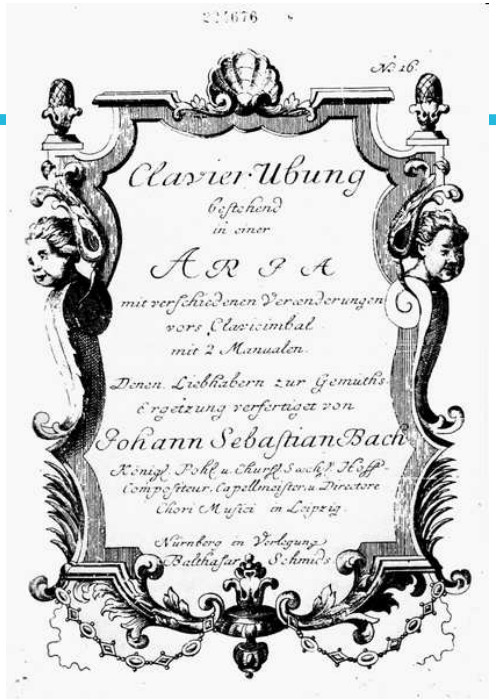
Reuso de Software: Linha de Produtos de Software (LPS)

Roteiro

- Motivação
- Definição
- Exemplos
- Estrutura
- Desafios
- Bibliografia

Motivação

Frans Post.
Instituto Ricardo Brennand, Recife



J.S. Bach.
Variações de Goldberg

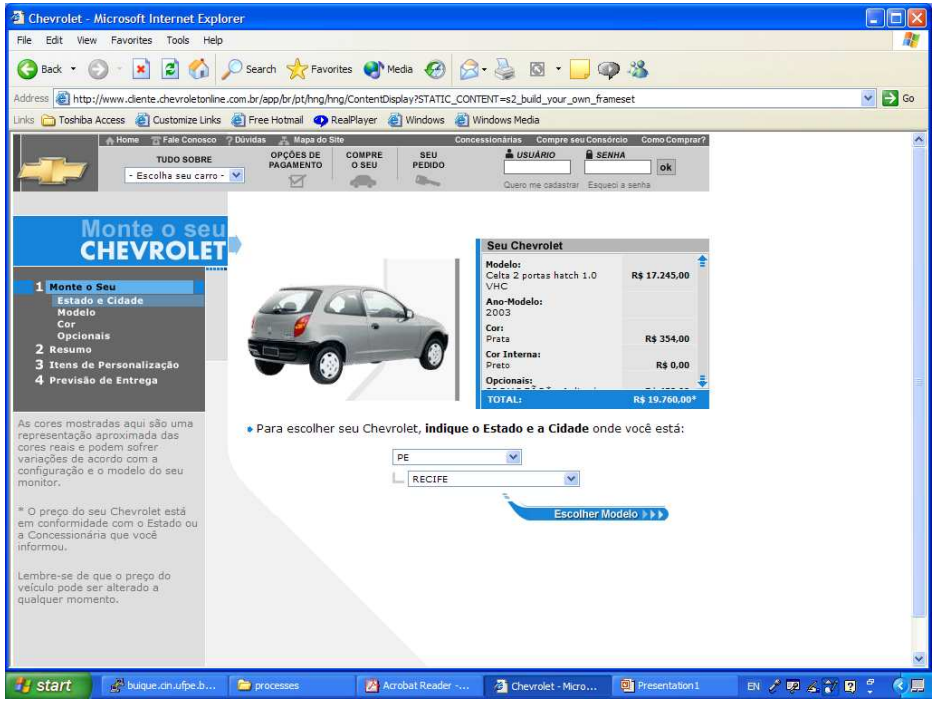
Revolução Industrial

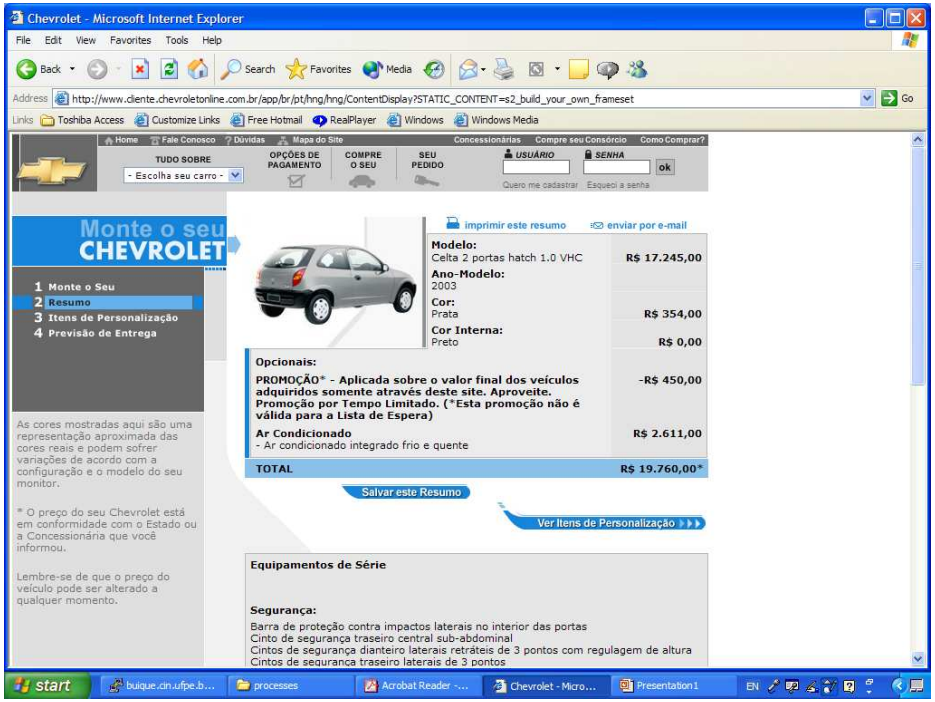
1826
interchangeable parts

1901
assembly lines
1913 Henry Ford

1980s
automated assembly lines
1961 General Motors

Czarnecki, 2000





Engenharia de único sistema

- Ausência de suporte explícito ao reuso (ex. RUP).
- Geralmente, não há escopo do domínio
- Pouco suporte a variabilidade
- Modelagem de variações amarrada à implementação
- Uso objeto -> Reuso de software ?

Tentativas de remediar...

- Padrões de projeto
- Frameworks
  - "we've got the car body, you the rest"
- Componentes
  - "we deliver the car, you assemble"

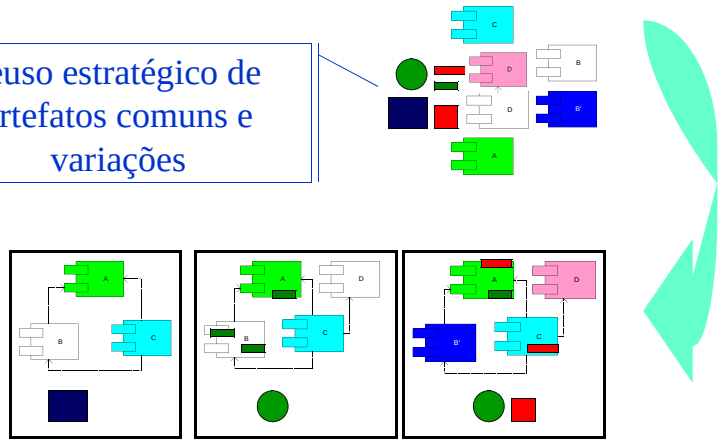
Linha de Produtos de Software (LPS): **Definição**

Conjunto de sistemas compartilhando um conjunto comum e gerenciado de funcionalidades (*features*) que satisfazem necessidades específicas de um segmento, e desenvolvidos a partir de um conjunto comum de artefatos base e de uma forma determinada.

[Clemens e Northrop 2001]

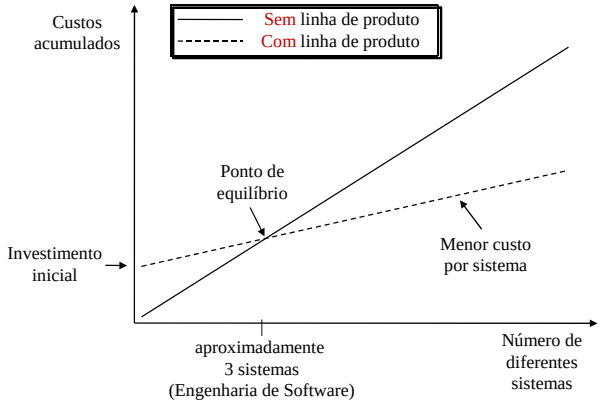
LPS

reuso estratégico de artefatos comuns e variações



Custo de desenvolvimento

Sistemas únicos vs Famílias de Produtos



Fonte: Clemens e Northrop 2001

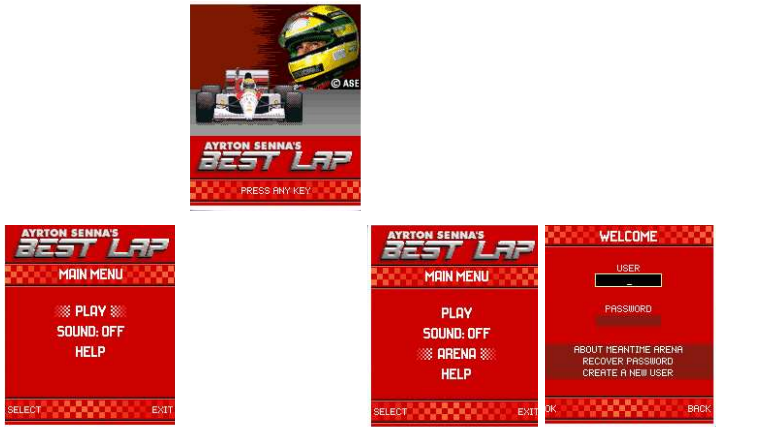
Exemplos de LPS

- Jogos móveis
- SAP
- Air bag & engine controller (Bosch)
- Medical Imaging (Philips, Siemens)
- Imposto de Renda (TurboTax, EUA)
- Vários outros...

Jogos móveis

meantime games

Features de jogo



meantime games

Variações de plataforma

produto 1 produto 2

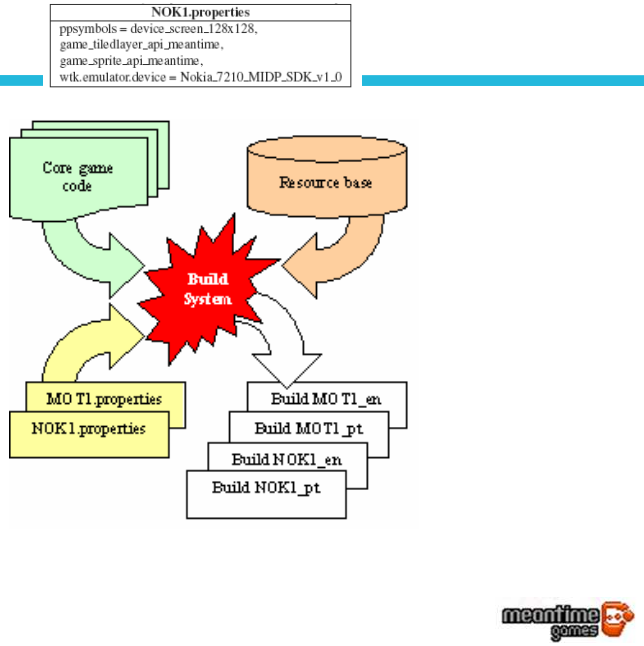
produto 3 produto 4

meantime games

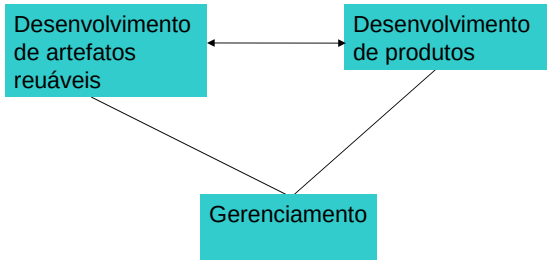
Variabilidade -

```
public class GameController {...
  public static int BOARD_SOFT_LEFT = 0;
  public static int BOARD_SOFT_RIGHT = 0;...
  static{
        //#ifdef KEYS_C650
          BOARD_SOFT_LEFT = -21;
          BOARD_SOFT_RIGHT = -22;
        //#elifdef KEYS_T720
        //BOARD_SOFT_LEFT = -6;
        //BOARD_SOFT_RIGHT = -7;
        //#elifdef KEYS_V300
          ...
        //#elifdef KEYS_SIEMENS
          ...
        //#endif
       } ...}
```

Properties File: KEYS_C650



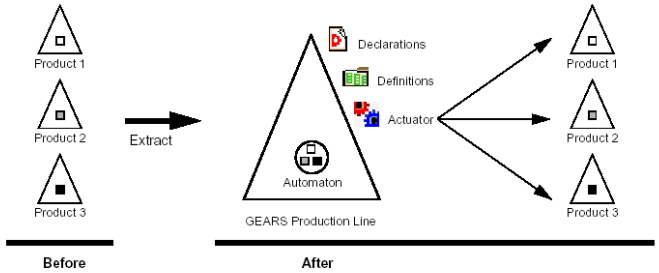
Estrutura



Desafios em LPS

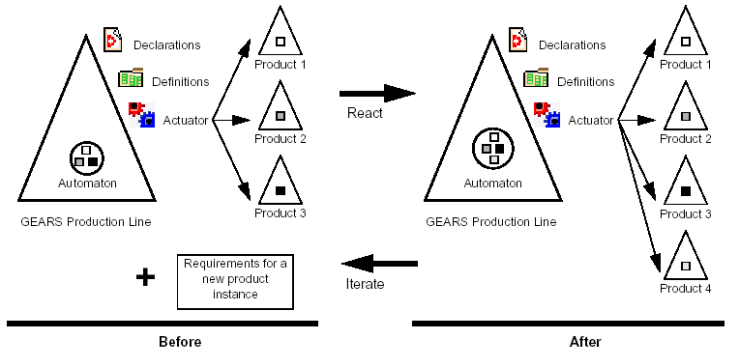
- Diferentes artefatos
- Testes
- Mudança gerencial necessária
- Estratégia de adoção realista

Estratégia Extrativa



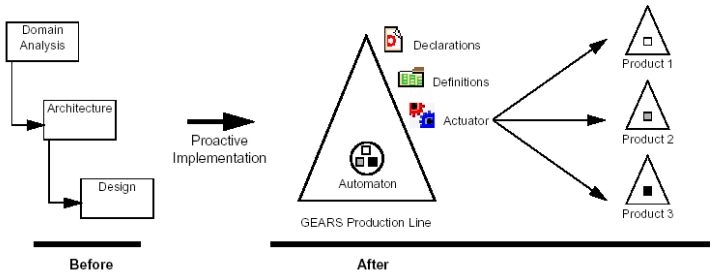
Fonte: Krueger, PFE'01

Estratégia Reativa



Fonte: Krueger, PFE'01

Estratégia Proativa



Fonte: Krueger, PFE'01