Padrões e Desenho de Software

Resumos
2015/2016

João Alegria | 68661

Padrões Comportamentais

Descrevem padrões de comunicação entre objetos;
.Fluxos de comunicação complexos;
.Foco na interconexão entre objetos.
Foco na classe: herança;
Foco no objeto: composição.

Foco na classe

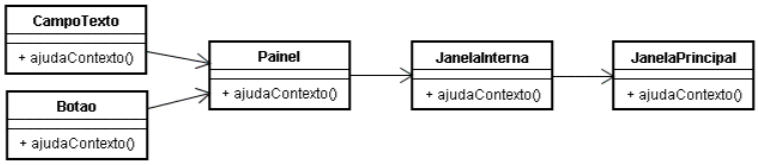
- **Interpreter**

Foco no Objeto

- **Chain of Responsibility - Cadeia de Responsabilidades**

Formar uma cadeia de objetos receptores e passar um pedido pelo mesmo, dando a chance a mais de um objeto a responder a requição ou colaborar de alguma forma na resposta.

Solução:



. Componentes colocados em cadeia na ordem filho -> pai;
. Se não há ajuda de contexto para o filho, ele delega ao pai e assim sucessivamente.

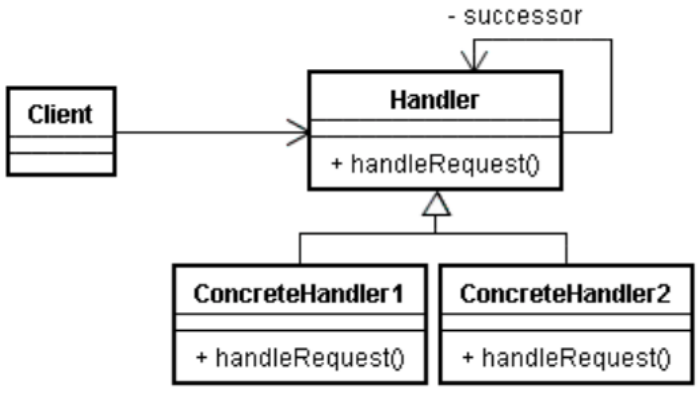
Usar este padrão quando:

Mais de um objeto pode responder a um pedido e:
. não se sabe qual a priori;
. não se quer especificar o receptor explicitamente;
. estes objetos são especificados dinamicamente.

Vantagens e Desvantagens:

- Acoplamento reduzido:
Não se sabe a classe ou estrutura interna dos participantes. Pode usar Mediator para desacoplar ainda mais.
- Delegação de responsabilidade:
Flexível, em tempo de execução.
- Garantia de resposta:
Deve ser uma preocupação do desenvolvedor!
- Não utilizar identidade de objetos.

Estrutura:



Chain of Responsibility	Decorator
Analogia com uma fila de filtros	Analogia com camadas de um mesmo objeto
Os objetos na cadeia são do “mesmo nível”	O objeto decorado é “mais importante”. Os demais são opcionais
O normal é quando um objeto atender, interromper a cadeia	O normal é ir até o final

- **Command - Comando**

Intenção:

- Encapsular um pedido como um objeto, permitindo parametrização, enfileiramento, suporte a histórico, etc.
- Promove a “invocação de um método num objeto”

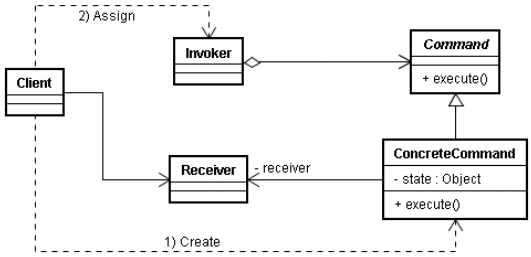
Problema:

- A cada pedido:
  - Uma funcionalidade diferente deve ser executada
  - Conjuntos de dados diferentes são enviados

Solução:

- O Cliente escolhe uma ação que encapsula os dados e o que deve ser executado
- Servidor executa esta ação.

Estrutura:



Usar este padrão quando:

-quiser parametrizar ações genéricas;
-quiser enfileirar e executar comandos de forma assíncrona, em outro momento;
-quiser executar comandos de forma assíncrona, numa outra altura
-quiser permitir o retrocesso de operações, dando suporte a históricos;
-quiser fazer log dos comandos para refazê-los em caso de falha de sistema;
-quiser estruturar um sistema em torno de operações genéricas, como transações.

Vantagens e Desvantagens:

- Desacoplamento:
  Objeto que evoca a operação e o que executa são desacoplados.
- Extensibilidade:
  Comandos são objetos, passíveis de extensão, composição, etc.;
  Pode ser usado junto com Composite para formar comandos complexos;
  É possível definir novos comandos sem alterar nada existente.

- **Interpreter - Interpretador**

Intenção:

- Definir a gramática de uma linguagem e criar um interpretador que leia instruções nesta linguagem e interprete-as para realizar tarefas.

Problema:

```
// Verificar se o e-mail é válido.
int pos = email.indexOf('@');
int length = email.length();
if ((pos > 0) && (length > pos)) {
    // Ainda verificar se tem ".com", etc.
}
else { /* E-mail inválido. */ }
```

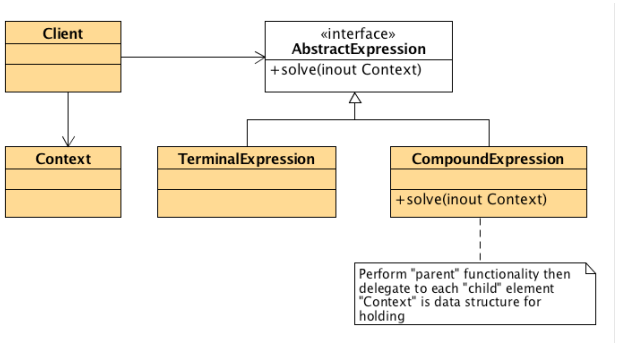
- Algumas tarefas ocorrem com tanta frequência de formas diferentes que é interessante criar uma linguagem só para definir este tipo de problema.

Solução:

```
// Verificar se o e-mail é válido.
regexp = "[a-zA-Z0-9_.-]@[a-zA-Z0-9_.-].com.br";
if (email.matches(regexp)) {
      // E-mail válido.
}
else {
      // E-mail inválido.
}
```

- Expressões regulares são um exemplo: gramática criada somente para verificar padrões em Strings;
- É criado um interpretador para a nova linguagem.

Estrutura:



Usar este padrão quando…

- Existe uma linguagem a ser interpretada que pode ser descrita como uma árvore sintática;
- Funciona melhor quando:
  A linguagem é simples;
  Desempenho não é uma questão crítica.

Vantagens e Desvantagens

- É fácil mudar e estender a gramática:
  Pode alterar expressões existentes, criar novas expressões, etc.;
  Implementação é simples, pois as estruturas são parecidas.
- Gramáticas complicadas dificultam:
  Se a gramática tiver muitas regras complica a manutenção.

Interpreter e Command

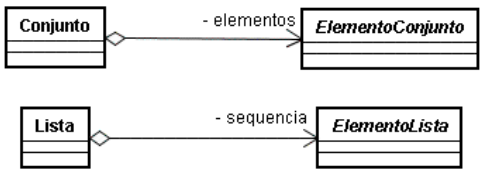
- Interpreter pode ser usado como um refinamento de Command:
  Comandos são escritos numa gramática criada para tal;
  Interpreter lê esta linguagem e cria objetos Command para execução.

• Iterator- Iterador

Intenção:

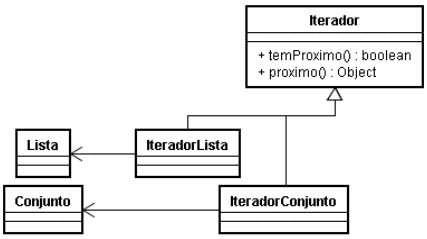
- Prover uma forma de acessar os elementos de um conjunto em sequência sem expor a representação interna deste conjunto.

Problema:



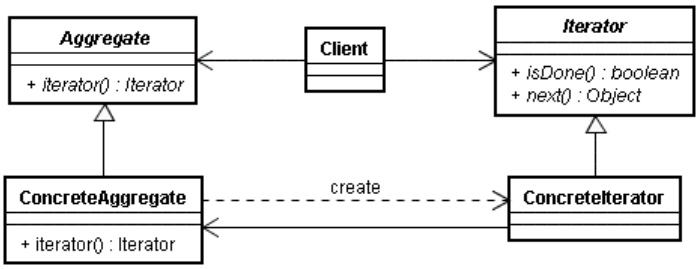
- Cliente precisa acessar os elementos;
- Cada coleção é diferente e não se quer expor a estrutura interna de cada um para o Cliente.

Solução:



- Iterator fornece acesso sequencial aos elementos, independentemente da coleção

Estrutura:



Usar este padrão quando…

-quiser acessar objetos agregados (coleções) sem expor a estrutura interna;
-quiser prover diferentes meios de acessar tais objetos;
-quiser especificar uma interface única e uniforme para este acesso.

Vantagens e Desvantagens

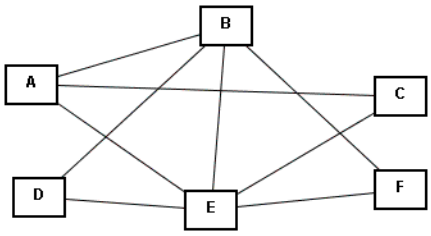
- Múltiplas formas de acesso:
  Basta implementar um novo iterador com uma nova lógica de acesso.
-Interface simplificada:
  Acesso é simples e uniforme para todos os tipos de coleções.
- Mais de um iterador:
  É possível ter mais de um acesso à coleção em pontos diferentes.

- **Mediator- Mediador**

Intenção:

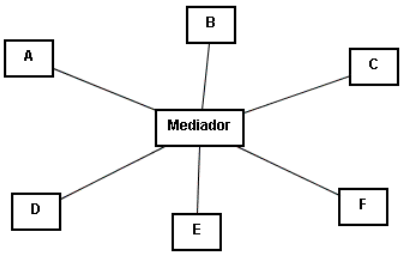
Definir um objeto que encapsula a informação de como um conjunto de outros objetos interagem entre si. Promove o acoplamento fraco, permitindo que você altere a forma de interação sem alterar os objetos que interagem.

Intenção:



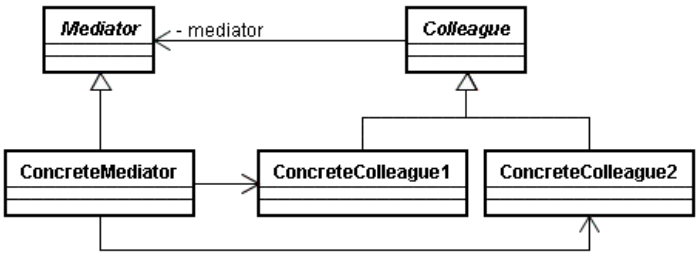
-Modelagem OO encoraja a distribuição de responsabilidades;
- Esta distribuição resulta em um emaranhado de conexões.

Solução:



- Um mediador assume a tarefa de realizar a comunicação entre os muitos objetos

Estrutura:



Usar este padrão quando...

-um conjunto de objetos se comunica de uma forma bem determinada, porém complexa;
-reutilizar uma classe é difícil pois ela tem associação com muitas outras;
-um comportamento que é distribuído entre várias classes deve ser extensível sem ter que criar muitas subclasses.

Vantagens e Desvantagens

- Limita extensão por herança:
  Para estender ou alterar o comportamento, basta criar uma subclasse do mediador.
- Desacopla objetos:
  Desacoplamento promove o reuso.
- Simplifica o protocolo:
  Fica mais claro como os objetos interagem.
- Exagero pode levar a sistema monolítico.

CheckList:

- Identificar uma coleção interativa de objetos que beneficiam com a dissociação mutua
- Criar uma instancia dessa nova classe e retrabalhar todos os objetos “peer” para interagir apenas com um Mediator.
- Equilibrar uniformemente o principio da dissociação com o principio da distribuição de responsabilidade.

• Memento - Recordação

Intenção:

- Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto para que possa ser restaurado posteriormente.

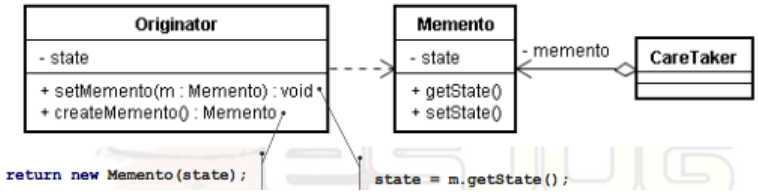
Problema:

- Para dar suporte a operação de undo, é preciso armazenar o estado do(s) objeto(s) antes de uma determinada operação.

Solução:

- Se o estado (pontos x,y; tamanho, etc.) de cada objeto (cada classe e linhas) foi armazenado, basta restaurá-los.

Estrutura



Usar este padrão quando…

- o estado do objeto (ou de parte dele) deve ser armazenado para ser recuperado no futuro;
- uma interface direta para obtenção de tal estado iria expor a implementação e quebrar o encapsulamento.

Vantagens e Desvantagens

- Preserva o encapsulamento:
  Retira do objeto original a tarefa de armazenar estados anteriores;
  Caretaker não pode expor a estrutura interna do objeto, a qual tem acesso;
  No entanto pode ser difícil esconder este estado em algumas linguagens.
- Pode ser caro:
  Dependendo da quantidade de estado a ser armazenado, pode custar caro.

CheckList

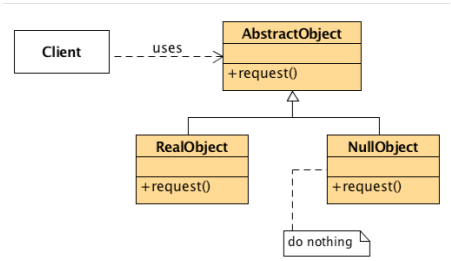
- Identificar os papeis de “carataker” e “originator”
- Originator cria um Memento e copia o seu estado para o mesmo Memento

• Null Object

Intenção:

- Encapsular a ausência de um objeto fornecendo uma substituição alternativa que não tenha nenhum comportamento.
- As referências devem ser verificados para garantir que não são nulos antes de chamar qualquer método, porque os métodos geralmente não podem ser invocados com referências nulas.
- Usar o padrão quando se quer abstrair a manipulação null longe do cliente.

Estrutura:



• Observer - Observador

Intenção:

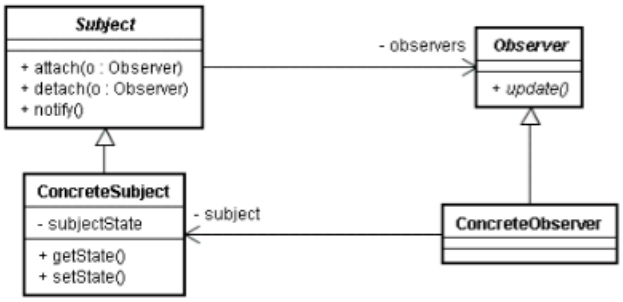
- Definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, os outros são notificados e se atualizam.

Solução:

- Definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, os outros são notificados e se atualizam.

Objeto observável registra os observadores e os notifica sobre qualquer alteração.

Estrutura



Usar este padrão quando…

-uma abstração possui dois aspectos e é necessário separá-los em dois objetos para variá-los;
-alterações num objeto requerem atualizações em vários outros objetos não-determinados;
-um objeto precisa notificar sobre alterações em outros objetos que, a princípio, ele não conhece.

Vantagens e Desvantagens:

Flexibilidade:
- Observável e observadores podem ser quaisquer objetos;
- Acoplamento fraco entre os objetos: não sabem a classe concreta uns dos outros;
- É feito broadcast da notificação para todos, independente de quantos;
- Observadores podem ser observáveis de outros, propagando em cascata.

CheckList:

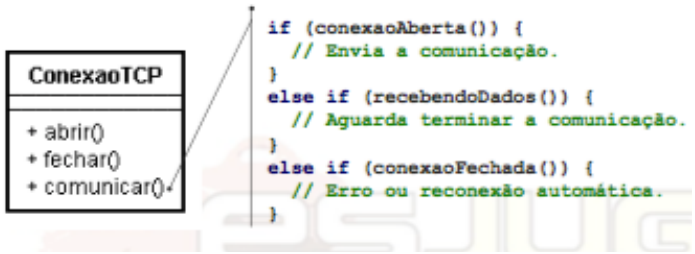
- Diferença entre a funcionalidade independente do núcleo e a funcionalidade opcional.
- Modela a funcionalidade independente com uma abstração “Subject”
- Modela a funcionalidade dependente com uma hierarquia “Observer”.
- Subject é acoplado apenas para a classe base do Observer.

- **State - Estado**

Intenção:

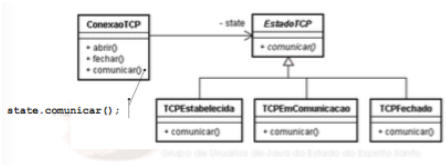
- Permitir que um objeto altere seu comportamento quando muda de estado interno. O objeto aparenta mudar de classe.

Problema:



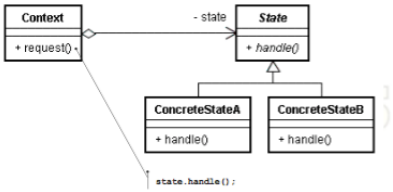
- Um objeto responde diferentemente dependendo do seu estado interno

Solução:



- Conexão possui um objeto que representa o seu estado e implementa o método que depende desse estado.

Estrutura



Usar este padrão quando…

-o comportamento de um objeto depende do seu estado, que é alterado em tempo de execução;
-operações de um objeto possuem condicionais grandes e com muitas partes (sintoma do caso anterior).

Vantagens e Desvantagens:

- Separa comportamento dependente de estado:
  Novos estados/comportamentos podem ser facilmente adicionados.
- Transição de estados é explícita:
  Fica claro no diagrama de classes os estados possíveis de um objeto.
- States podem ser compartilhados:
  Somente se eles não armazenarem estado em atributos.

CheckList:

- Identificar uma classe existente, ou criar uma nova classe, que servirá como a "máquina de estado" na perspectiva do cliente. Essa classe é a classe "wrapper".
- Criar uma classe base de estado que replica os métodos da interface de máquina de estado. Cada método tem um parâmetro adicional: uma instância da classe wrapper.
- Crie uma classe derivada de estado para cada Estado do domínio. Essas classes derivadas apenas substituem os métodos precisam substituir.
- A classe wrapper mantém um objeto de estado "atual".

- Todas as solicitações de cliente para a classe de wrapper simplesmente são delegadas para o objeto do estado atual e do objeto wrapper esse ponteiro é passado.
- Os métodos de estado altera o estado "atual" no objeto wrapper, conforme o caso.

• Strategy - Estratégia

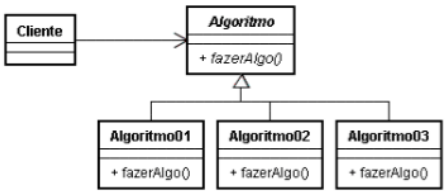
Intenção:

Definir uma família de algoritmos e permitir que um objeto possa escolher qual algorítmo da família utilizar em cada situação.

Problema:

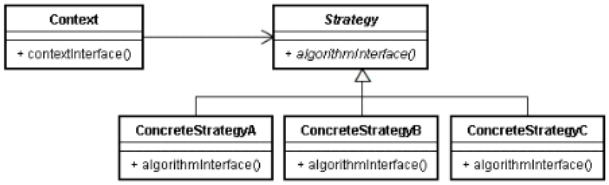
- Existem problemas que possuem vários algorítmos que os solucionam;
  Ex.: quebrar um texto em linhas.
- Seria interessante:
  Separar estes algorítmos em classes específicas para serem reutilizados;
  Permitir que sejam intercambiados e que novos algorítmos sejam adicionados com facilidade.

Solução:



- Comportamento é encapsulado em objetos de uma mesma família;
- Similar ao padrão State, no entanto não representa o estado do objeto.

Estrutura:



Usar este padrão quando…

-várias classes diferentes diferem-se somente no comportamento;
-você precisa de variantes de um mesmo algorítmo;
-um algoritmo utiliza dados que o cliente não deve conhecer;
-uma classe define múltiplos comportamentos, escolhidos num grande condicional.

Vantagens e Desvantagens

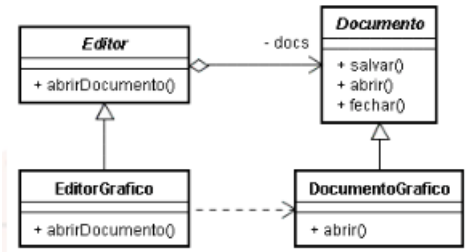
- Famílias de algoritmos:
Beneficiam-se de herança e polimorfismo.
- Alternativa para herança do cliente:
Comportamento é a única coisa que varia.
- Eliminam os grandes condicionais:
Evita código monolítico.
- Escolha de implementações:
Pode alterar a estratégia em runtime.
- Clientes devem conhecer as estratégias:
Eles que escolhem qual usar a cada momento.
-Parâmetros diferentes para algoritmos diferentes:
Há possibilidade de duas estratégias diferentes terem interfaces distintas.
- Aumenta o número de objetos:
Este padrão aumenta a quantidade de objetos pequenos presentes na aplicação.

- **Template Method - Método Modelo**

Intenção:

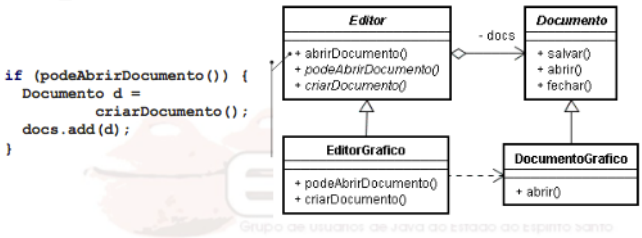
Definir o esqueleto de um algorítmo numa classe, delegando alguns passos às subclasses. Permite que as subclasses alterem partes do algorítmo, sem mudar sua estrutura geral.

Problema:



(Duas componentes diferentes têm semelhanças significativas, mas não demonstram a reutilização de interface ou de uma aplicação comum).
-Alguns passos de abrirDocumento() e abrir() são iguais para todo Editor e Documento;
-EditorGrafico e DocumentoGrafico têm que sobrescrever todo o método.

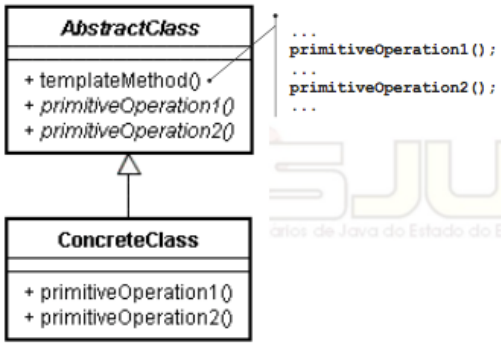
Solução:



(Chamar métodos abstratos)

- Método é implementado em Editor, chamando métodos abstratos que são implementados em EditorGrafico

Estrutura:



Usar este padrão quando…

- quiser implementar partes invariantes de um algorítmo na superclasse e deixar o restante para as subclasses;
- comportamento comum de subclasses deve ser generalizado para evitar duplicidade de código;
- quiser controlar o que as subclasses podem estender (métodos finais).

Vantagens e Desvantagens

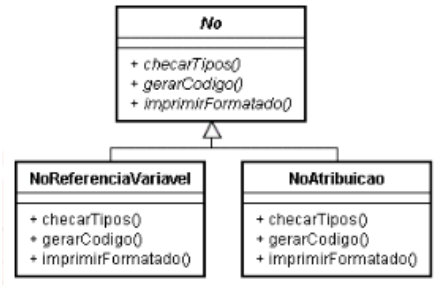
- Reuso de código:
  Partes de um algorítmo são reutilizadas por todas as subclasses.
- Controle:
  É possível permitir o que as subclasses podem estender (métodos finais).
- Comportamento padrão extensível:
  Superclasse pode definir o comportamento padrão e permitir sobrescrita.

- **Visitor - Visitante**

Intenção:

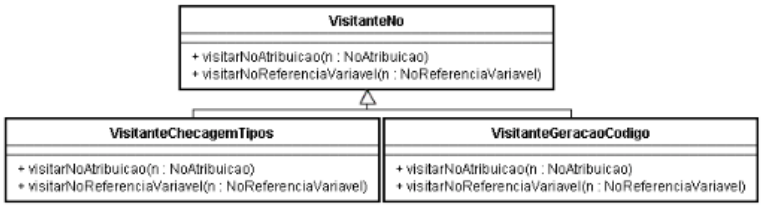
Representar uma operação a ser efetuada em objetos de uma certa classe como outra classe. Permite que seja definido uma nova operação sem alterar a classe na qual a operação é efetuada.

Problema:



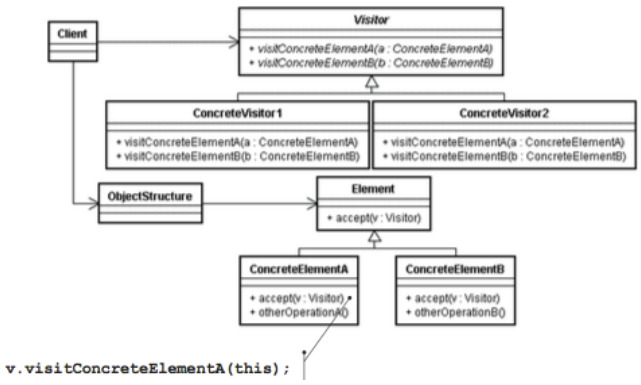
- Um compilador representa o código como uma árvore sintática. Para cada nó precisa de realizar algumas operações
- Misturar estas operações pode ser confuso.

Solução:



- As operações possíveis viram classes
- Cada uma deve tratar todos os parâmetros (nós) possíveis para aquela operação
- Nós agora possuem somente uma operação: aceitar(v: VisitanteNo)

Estrutura:



Usar este padrão quando:

- uma estrutura de objetos contém muitas classes com muitas operações diferentes;
- quiser separar as operações dos objetos-alvo, para não “poluir” seu código;
- o conjunto de objetos-alvo raramente muda, pois cada novo objeto requer novos métodos em todos os visitors.

Vantagens e Desvantagens:
- Organização:
Visitor reúne operações relacionadas.
- Fácil adicionar novas operações:
Basta adicionar um novo Visitor.
- Difícil adicionar novos objetos:
Todos os Visitors devem ser mudados.
- Transparência:
Visite toda a hierarquia transparentemente.
- Quebra de encapsulamento:
Pode forçar a exposição de estrutura interna para que o Visitor possa manipular.

RESUMIDAMENTE

Padrões permitem variar um comportamento ou estendê-lo:

Strategy: implementações de um algorítmo;
State: comportamento diferente para estados diferentes;
Mediator: como um conjunto de objetos se comunica;
Iterator: como apresentar um conjunto de objetos agregados em sequência;
Chain of Responsibility: quantidade de objetos que colabora para uma requisição;
Template Method: partes de um algorítmo;
Command: parâmetros e comportamento das ações;
Interpreter: Uma maneira de incluir elementos de linguagem em um programa
Memento: Captura e restaura o estado interno do objeto
Null Object: Projetado para agir como um valor padrão de um objeto
Observer: Uma maneira de notificar a mudança para um número de classes
Visitor: Define uma nova operação para uma classe sem alteração

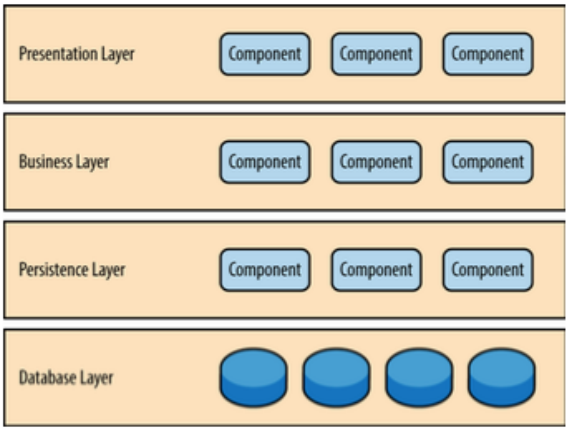
---

Software Arch

Layered Architecture - Arquitetura por Camadas

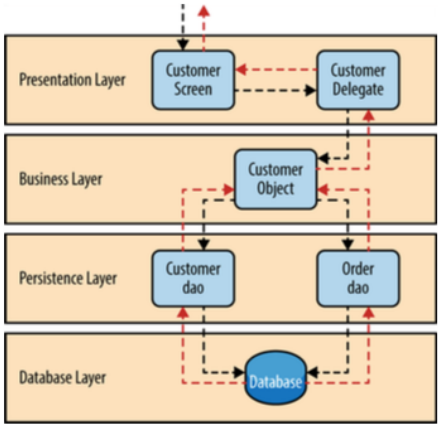
- O padrão de arquitetura mais comum é o padrão de arquitetura em camadas.
- O padrão de arquitetura em camadas aproxima-se da comunicação tradicional de IT e estruturas organizacionais encontradas na maioria das empresas.
  - tornando-se uma escolha natural para a maioria dos esforços de desenvolvimento de aplicativo de negócios.

- A arquitetura por camadas consiste em quatro camadas padrão:
  apresentação, negócio, persistência, base de dados
- Cada camada do padrão de arquitetura em camadas tem um papel específico e responsabilidade dentro do aplicativo.
- Um dos recursos poderosos do padrão de arquitetura em camadas é a separação de preocupações entre os componentes.
- Componentes dentro de uma camada específica lidam apenas com a lógica que diz respeito a essa camada.



As camadas do conceito de isolamento significa que as alterações feitas em uma camada da arquitetura geralmente não afetam componentes em outras camadas.
Desde que a camada de serviços é aberta, a camada de negócios é permitida para contorná-lo e ir diretamente para a camada de persistência.

Exemplo:



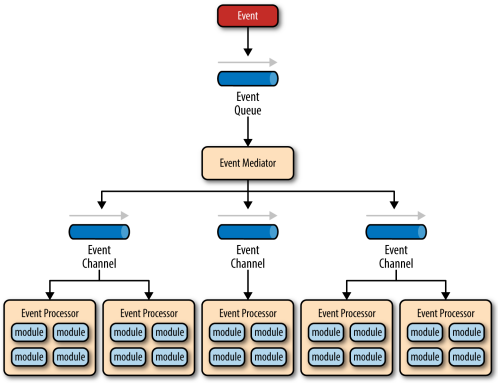
Agilidade geral – classificação: baixo
Facilidade de implementação – classificação: baixo
Capacidade de teste – Avaliação: alta
Desempenho – classificação: baixo
Escalabilidade – Classificação: baixo
Facilidade de desenvolvimento – Rating: alta

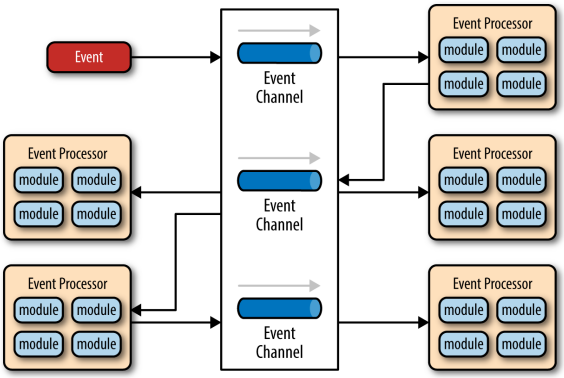
A 'tela do cliente' é responsável por aceitar a solicitação e visualização de informações.
-Não sabe onde os dados são, como ele é recuperado, quais tabelas de banco de dados devem ser consultadas!
– ele encaminha a solicitação para o módulo de representante do cliente.
– Este módulo é responsável por conhecer quais módulos na camada de negócios
pode processar esse pedido.
O objeto do cliente é responsável por agregar todas as informações necessárias para a solicitação de negócios.

Event-Driven Architecture - Arquitetura orientada a eventos

Padrão de arquitetura assíncrona distribuída popularmente usada para produzir aplicações altamente escaláveis.
– Também é altamente adaptável e pode ser usado para aplicações pequenas, bem como, grandes e complexos.
Ela é composta de componentes que forma assíncrona recebem e processam eventos de processamento de eventos altamente dissociado, finalidade única.
O padrão de arquitetura orientada a eventos é composto por duas topologias principais, o **mediator** e o **broker**.

Topologia Mediator	Topologia Broker
A topologia do mediador é útil para eventos que tem várias etapas e requerem um certo nível de orquestração para processar o evento. – Por exemplo, um único evento para colocar um estoque comércio pode exigir que primeiro seja necessário validar o comércio, em seguida, verificar a conformidade do que o comércio das ações contra várias regras de conformidade, atribuir o comércio de um corretor, calcular a Comissão e finalmente colocar o comércio com o broker	Não há nenhum mediador do evento central – o fluxo de mensagem é distribuído entre os componentes do processador de evento em uma cadeia como forma através de um corretor de mensagem leve.
Existem quatro tipos principais de componentes de arquitetura dentro da topologia do mediador: -filas de eventos, um mediador do evento, evento canais e processadores de eventos.	Esta topologia é útil quando você tem um evento relativamente simples de processamento de fluxo e você não quer (ou precisa) orquestração evento central. -Existem dois tipos principais de componentes de arquitetura dentro da topologia do corretor: - a corretor e um componente evento processador.



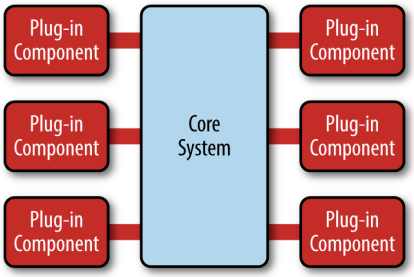


Considerações:

O padrão de arquitetura orientada a eventos é um padrão relativamente complexo de implementar, principalmente devido à sua natureza assíncrona distribuída.
Falta de transações atômicas para um processo de negócio único.
– Componentes do processador evento são altamente dissociado e distribuída,
– é muito difícil manter uma unidade transacional de trabalho através deles.
Um aspecto fundamental é a criação, manutenção e governança dos contratos componente processador de evento.

Agilidade geral – classificação: alta
Facilidade de implementação – classificação: alta
Capacidade de teste – Avaliação: baixa
Desempenho – classificação: alta
Escalabilidade – Classificação: alta
Facilidade de desenvolvimento – Rating: baixa

Microkernel Architecture



O padrão de arquitetura microkernel permite que você adicione recursos adicionais às aplicações como plug-ins para o núcleo da aplicação, fornecendo extensibilidade, bem como recurso separação e isolamento.
Muitos sistemas operativos implementam a arquitetura microkernel, daí a origem do nome a esse padrão.

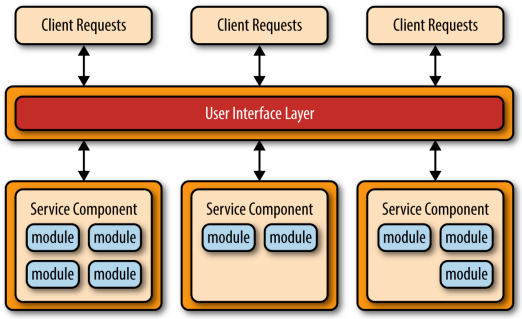
- Dois tipos de componentes de arquitetura:
  – um núcleo do sistema e módulos plug-in
- O núcleo do sistema tradicionalmente contém somente a funcionalidade mínima necessária para tornar o sistema operacional
-Módulos plug-in podem ser conectados ao sistema através de uma variedade de maneiras de núcleo
  – OSGi (iniciativa de gateway de serviço aberto), de mensagens, serviços web, ou até mesmo direta ligação ponto a ponto (ou seja, instanciação de objeto)

Considerações:
- pode ser incorporado ou usado como parte de outro padrão de arquitetura.
- fornece grande apoio para design evolutivo e desenvolvimento incremental.
- Para aplicativos baseados no produto deve ser sempre a primeira escolha como uma arquitetura de partida
  - especialmente para aqueles produtos onde nós estará liberando recursos adicionais ao longo do tempo, e quer controlar ao longo do qual os usuários ficar que apresenta.
  - – Nós sempre pode refatorar o aplicativo para outro padrão de arquitetura mais adequado para suas necessidades específicas.

Agilidade geral – classificação: alta
Facilidade de implementação – classificação: alta
Capacidade de teste – Avaliação: alta
Desempenho – classificação: alta
Escalabilidade – Classificação: baixa
Facilidade de desenvolvimento – Rating: baixa

Microservices Architecture

Ele está a ganhar terreno na indústria como uma alternativa viável para aplicações monolíticos e arquiteturas orientadas para serviços.



A primeira característica é a noção de unidades implementadas separadamente.
– Cada componente do serviço é implantado como uma unidade separada, permitindo a implantação mais fácil e dissociação.
– de um único módulo de uma grande parte do aplicativo.
Arquitetura distribuída:
– todos os componentes são totalmente dissociados
– comunicação através de JMS, AMQP, descansar, sabão, RMI, etc.
O estilo de arquitetura microservices naturalmente evoluído a partir de duas fontes principais:
– aplicações monolíticos desenvolvidos usando o padrão de arquitetura em camadas e
– aplicações distribuídos desenvolvidos através do serviço - padrão de arquitetura orientada.

Considerações:
Aplicações são geralmente mais robustas, fornecem escalabilidade melhor e podem mais facilmente suportar entrega contínua.
- Capacidade de fazer implantações de produção em tempo real.
- Somente os componentes de serviço que mudar precisa ser implantado.

Mas... arquitetura distribuída
– partilha alguns dos mesmos problemas complexos encontrados no padrão de arquitetura orientada a eventos, incluindo o contrato de criação, manutenção e governo, disponibilidade de sistema remoto e autorização e autenticação de acesso remoto.

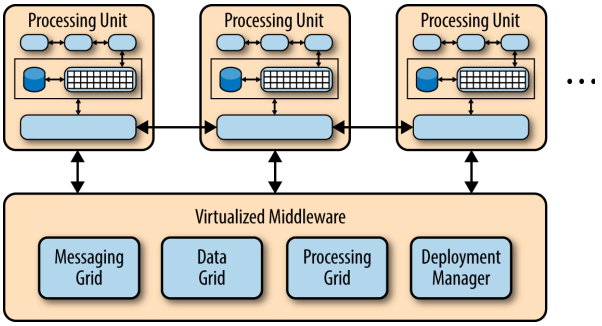
Agilidade geral – classificação: alta
Facilidade de implementação – classificação: alta
Capacidade de teste – Avaliação: alta
Desempenho – classificação: baixa
Escalabilidade – Classificação: alta
Facilidade de desenvolvimento – Rating: alta

Space-Based Architecture

Aplicativos de negócios baseados na web mais seguem o mesmo fluxo de pedido geral:
– um pedido de um navegador atinge o servidor web, um servidor de aplicativos, e depois finalmente o servidor de banco de dados.
O padrão de arquitetura baseada em espaço é projetado especificamente para enfrentar e resolver problemas de escalabilidade e simultaneidade.
- Muitas vezes é uma aproximação melhor do que tentar expandir um banco de dados ou retrofit cache tecnologias em uma arquitetura não dimensionável.

Este padrão recebe o nome do conceito de espaço de tupla, a idéia de memória compartilhada distribuída.
– Também conhecido como o padrão de arquitetura de nuvem
-Alta escalabilidade é conseguida substituindo o central banco de dados com redes de dados replicados na memória.
– Dados de aplicativos são mantidos na memória e replicados entre todas as unidades de processamento ativo.
Unidades de processamento podem ser dinamicamente iniciou-se e desligar-se como carga de usuário aumenta e diminui.
– Gargalo o banco de dados é removido, fornecendo escalabilidade quase infinita dentro do aplicativo.

Existem dois componentes principais dentro deste padrão de arquitetura: uma unidade de processamento e middleware virtualizado.
O componente de processamento-unidade contém os componentes do aplicativo.
– Isso inclui componentes baseados na web, bem como a lógica de negócios de back-end.
O componente middleware virtualizados lida com tarefas domésticas e comunicações.



Considerações:

O padrão de arquitetura baseada no espaço é um padrão complexo e caro de implementar.
-É uma escolha boa arquitetura para aplicações web-based menores com carga variável (por exemplo, sites de mídias sociais, sites de licitação e leilão).
No entanto, não é adequado para aplicações de tradicional banco de dados relacional em larga-escala com grandes quantidades de dados operacionais.

Agilidade geral – classificação: alta
Facilidade de implementação – classificação: alta
Capacidade de teste – Avaliação: baixa
Desempenho – classificação: baixa
Escalabilidade – Classificação: alta
Facilidade de desenvolvimento – Rating: baixa

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

Model-View-Controller

Model-view-controller (**MVC**), é um padrão de arquitetura de software que separa a representação da informação da interação do utilizador com ele.

Vantagens do MVC:
- Facilita a manutenção do software;
- Simplifica a inclusão de um novo elemento de visão (ex.: cliente);
- Melhora a escalabilidade;
- Possibilita desenvolvimento das camadas em paralelo, se forem bem definidas.

Desvantagens do MVC:
- Requer análise mais aprofundada (mais tempo);
- Requer pessoal especializado;
- Não aconselhável para aplicações pequenas (custo benefício não compensa).