Padrões e Desenho de Software

Resumos
2015/2016

João Alegria | 68661

Introdução ao Design de Software

O que é o Design de Software?

→ Design centrado nos objetivos propostos pelo utilizador *(Design bridges the gap)*.

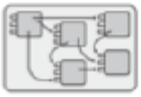
- Saber o que é necessário (requisitos de software)
- Para programar um código eficaz e eficiente (fase de construção)

Requisitos do Software Documento do Design Código

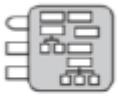
O design é necessário em diferentes níveis do sistema:

Sistema

Subsistemas ou pacotes: interface com o utilizador, armazenamento de dados, criação de classes e gráficos.



Classes dentro de pacotes, relações entre classes, interface de cada classe, métodos públicos



Atributos, métodos privados, classes internas

Implementação de métodos no código fonte

Importância do Design Software

O processo de Design pode tornar-se mais sistemático e previsível através de métodos, técnicas e padrões, todos aplicados de acordos com os princípios e heurísticas.

Importância da Gestão de Complexidade

- Programas mal concebidos são difíceis de entender e alterar;
- Quanto maior for o programa, mais visíveis são as consequências de um fraco design.

Tipos de Complexidade em Software

- Complexidades essenciais: complexidades em que são inerentes os programas;
- Complexidades acidentais/incidentais: complexidades que são os artefactos da solução.

- A soma total da complexidade numa solução de software é:
  Complexidades essenciais + complexidades acidentais

→ O principal propósito do Design é controlar a complexidade
Objetivo: Gerir a complexidade essencial, evitando a introdução de complexidades acidentais adicionais

Lidar com a Complexidade do Software

Modularidade: Subdividir a solução em componentes mais pequenas, para uma fácil gestão (dividir e conquistar).
Abstração: Usar abstrações para suprimir detalhes em lugares onde são desnecessárias.
Esconder Informação: Ocultar detalhes e complexidades atrás de interfaces simples.
Herança: Componentes gerais podem ser reutilizados para definir elementos mais especificados.
Composição: Reutilização de outros componentes para construir uma nova solução.

Características do Design de Software

Não-Determinístico: Não existem dois designs que tendem a produzir o mesmo resultado.
Heurístico: Técnicas de Design tendem a confiar em heurísticas e "regras-de-ouro" ao invés de processos repetidos.
Emergente: O design final evolui/constrói-se a partir de experiência e feedback. O design é um processo iterativo e incrementável, onde um sistema complexo surge de interações simples.

Processo Geral de Design

- Compreender o problema (requisitos de software);
- Construir um modelo de "caixa-preta" da solução (especificação do sistema)
  - Especificações do sistema são tipicamente representadas com "use cases" (especificamente em POO);
- Procurar por soluções já existentes (por exemplo: arquiteturas e padrões) que cobram alguns ou todos os problemas de design de software identificados;
- Considerar a construção de protótipos;
- Documentar e rever o design;
- Iterar sobre a solução (*refactor*);
  - Evoluir o projeto até que cumpra os requisitos funcionais e maximize os requisitos não funcionais.

Inputs para o processo de Design

- Requisitos e especificações do sistema
  - Incluir qualquer restrição sobre as opções de design e implementação
- Conhecimento do domínio
  - Por exemplo, se for uma aplicação de assistência médica o designer precisará de alguns conhecimentos dos termos e conceitos da área da saúde
- Conhecimento da implementação
  - Capacidade e limitações do eventual ambiente de execução

Características Internas para um Design desejável

- **Complexidade mínima**: Manter tudo simples. Talvez não seja preciso grande generalidade
- **Acoplamento fraco**: Minimizar as dependências entre módulos (uso de funções ou classes nos programas deve ser minimizado de maneira a torna-lo simples e independente)
- **Facilidade de Manutenção**: O código será lido com mais regularidade depois de escrito
- **Extensibilidade:** Projete para hoje mas com o olho no amanhã.
  Nota: Esta característica pode estar em conflito com "minimizar complexidade". Engenharia consiste no equilíbrio de objetivos conflituosos.
- **Reutilização**: Reutilizar é uma marca registada de uma grande disciplina de engenharia
- **Portabilidade**: Funciona ou pode ser facilmente desenvolvido para funcionar noutros ambientes.
- **Magreza**: Em caso de dúvida, deixe de fora. O custo de adicionar "uma linha de código a mais" é certamente muito mais do que os minutos que demorou a ser escrito.
- **Estratificação**: Em camadas. Se todo o sistema não segue o estilo de arquitetura por camadas, os componentes individuais podem.
- **Técnicas Padrão**: Por vezes é bom ser conformista! Chato é bom. A implementação de código não é um lugar de experiências

Métodos de Software de Design

- Os métodos de design fornecem uma descrição sobre o procedimento para a obtenção de uma solução.
- A maioria dos métodos incluem:
  - Representação/Notação de problemas e formas intermédias de solução do problema (UML, pseudocódigo)
  - Procedimentos a seguir no desenvolvimento da solução
  - Heurísticas: Diretrizes e boas práticas para a tomada de decisões e avaliação de resultados intermédios e finais. Lembre-se, o design não é determinista.

Design – Formas de Representação

- Diagramas de classe para estruturas estáticas
- Diagramas de sequência para um comportamento dinâmico
- “Use Cases” são usados para criar e validar análises e projetar formas de representação
- Outros métodos UML são também úteis para a compreensão do problema e conceptualizar uma solução (diagramas de máquinas de estado, diagrama de atividade)

Métodos e Padrões

- Métodos e modelos são as principais técnicas para lidar com os desafios do design
- São úteis para:
    - Documentar e projetar um design
    - Transferência de conhecimento do design e experiência entre profissionais

Padrões

- Um padrão de design (*design pattern*) é uma solução reutilizável para um problema de design que ocorre comumente.
- Os padrões de design são adaptados para as características únicas do problema particular
- Assim, como existem níveis de design, existem níveis de Padrões de Design:
    - Estilos/Padrões de Arquitetura
    - Padrões de Design
    - Linguagens de Programação

Princípios GRASP

- General Responsibility Assignment Software Patterns
  - Nome escolhido para sugerir a importância de compreender os princípios fundamentais para projetar com sucesso software orientado a objetos.
  - Descreve os princípios fundamentais do design de objetos e responsabilidades.
- Exemplo:
  - Atribuir a responsabilidade de uma classe
  - Evitar / Minimizar dependências adicionais
  - Maximizar a compreensibilidade

Criador (Creator)

Problema: Quem cria uma instância em A?

Solução: Atribuir à classe B a responsabilidade de criar uma instância da classe A, se uma desta se verificar (quanto mais melhor):

- B contém ou agrega A
- B regista A
- B usa A
- B tem os dados de inicialização para A que passarão ao construtor de A

Discussão do padrão Creator:

- Promove baixo acoplamento, fazendo instâncias de uma classe responsável para a criação de objetos que são necessários para a referência.
- Conecta um objeto para o seu criado quando:
  - Há uma relação de agregação
  - Há uma relação de conteúdo
  - Inicialização de dados passados durante a criação

Contra Indicações ou Advertências:

- A criação pode exigir uma complexidade significativa
  - Reciclar instâncias devido a razões de desempenho
  - Criando condicionalmente instâncias de uma família de classes semelhantes

Informação Especialista (Information Expert)

Problema: Como atribuir responsabilidades a objetos?

Solução: Atribuir a responsabilidade à classe que tem informações necessárias. Atribuir a responsabilidade ao especialista: a classe que tem as informações necessárias para assumir a responsabilidade.

Benefícios: O encapsulamento da informação é mantida uma vez que os objetos usam os seus próprios dados para realizar as tarefas. Isto leva a uma baixo acoplamento entre classes.

- Facilita a informação do encapsulamento
    - Classes usam a sua própria informação para cumprir tarefas
        - Classes altamente coesas
    - Código de fácil percepção
- Promove o baixo acoplamento

MAS

- Pode tornar uma classe excessivamente complexa
- Exemplo: Quem é o responsável por guardar *Vendas* na base de dados? *Vendas* é a 'Information Expert', mas com esta decisão cada classe tem os seus próprios serviços a guardar na base de dados. -> É preciso outro tipo de separação (domínio e persistência).

Baixo Acoplamento (Low Coupling)

Problema: Como reduzir o impacto da mudança e incentivar a reutilização?

Solução: Atribuir a responsabilidade de modo que o acoplamento (entre classes) permaneça baixo. Tentar evitar que uma classe tenha de saber muitos sobre os outros.

- Mudanças são localizadas
- Mais fácil de entender
- Mais fácil de reutilizar

Classes com forte acoplamento:

- Sofrem com as mudanças de classes relacionadas
- São mais difíceis de perceber e sustentar
- São mais difíceis de reutilizar

-> Mas o acoplamento é necessário se quisermos trocar mensagens entre classes

Ponto Chave: "Information Expert" favore o "Low Coupling". O Information Expert pede-nos que entremos o objeto que tem maior parte da informação necessária para assumira a responsabilidade.

Nas programações orientadas a objetos, as formas mais comuns de acoplamento do TipoX para o TipoY incluem:

- O TipoX tem um atributo (membro da memória ou uma variável instanciada) que se refere a uma instancia do TipoY ou mesmo a TipoY.
- O TipoX tem um método que se refere a uma instancia do TipoY, ou até mesmo a TipoY por quaisquer meios. Estes incluem tipicamente um parâmetro ou variável local do TipoY, ou um objeto retornado de uma mensagem a ser instância do TipoY.
- TipoX é uma superclasse direta ou indireta de TipoY.
- TipoY é uma interface e TipoX implementa essa interface.

Benefícios e Contra Indicações:

- Compreensibilidade: As classes são mais fáceis de estudar de forma isolada
- Manutenção: As classes não são afetadas por mudanças em outros componentes
- Reutilização: Mais fácil de agarrar as classes

MAS

- Um grande acoplamento às classes estáveis não é um grande problema
- Exemplo: Bibliotecas e classes bem testadas

- Classes que, por natureza, são genéricas e têm alta probabilidade de reutilização deveriam ter acoplamento baixo
- O caso extremo do baixo acoplamento é o não acoplamento: contraria o princípio da orientação a objetos: objetos conectados, trocando mensagens entre si.
- O acoplamento alto não é problema em si. O problema é o acoplamento a classes que, de alguma forma são instáveis: sua interface, sua implementação ou a sua mera presença.

Alta Coesão (High Cohesion)

Problema: Como manter as classes focadas e geridas? (e consequentemente com baixo acoplamento)

Solução: Atribuir responsabilidades para que a coesão continue elevada.

Benefícios e Contra Indicações:

- Compreensibilidade, manutenção
- Complementa baixo acoplamento

MAS

- As vezes é desejável criar objetos menos coesos
    - Que forneça uma interface para muitas operações, devido à necessidade de desempenho associados aos objetos remotos e comunicação remota.

Controlador (Controller)

Problema: Quem deve ser responsável pelos UI?

Solução: Se um programa receber eventos de fontes externas que não do seu GUI, adicionar uma classe para dissociar a fonte do evento dos objetos que fazem mesmo parte. Atribuir a responsabilidade de manipulação de uma mensagem do sistema para uma classe que represente uma destas opções:

- Os negócios ou o “sistema” global (controlo de Erro - façade controller)
- Uma classe artificial, ‘Pure Fabrication’ representa o caso de uso (um controlador de use cases).

Benefícios e Contra Indicações:

- Aumento do potencial de reutilização
  - Usando um objeto controller, mantém as fontes dos eventos externos e internos do tipo e do comportamento de cada um.
  - Tanto as classes UI como as de domínio do problema / software podem ser alterados sem afetar o outro lado.
- Controller apenas encaminha
  - Pedido de manipulação de evento
  - Pedidos de Output
- Motivo sobre o estado dos Use Cases
  - Garantir que as operações do sistema ocorrem em sequência lógica, ou para ser capaz de raciocinar sobre o estado atual da atividade e operações dentro do use case.

Polimorfismo (Polymorphism)

Problema: Como lidar com o comportamento no tipo (ou seja, classe), mas não com as instruções "if-then-else" ou "switch" que envolva o nome da classe ou um atributo?

Solução: - Quando comportamentos alternativos são selecionados com base no tipo de um objeto, usar o método polimórfico para selecionar o comportamento, ao invés de usar a instrução "if" para testar o tipo.

- Métodos polimórficos: Dar o mesmo nome a serviços em classes diferentes. Serviços são implementados por métodos.

Benefícios e Contra Indicações:

- Mais fácil e mais confiável do que usar a lógica de solução explícita
- Mais fácil de adicionar, posteriormente, comportamentos adicionais

MAS

- Aumenta o número de classes do design
- Pode tornar o código mais complicado de entender

Pura Fabricação (Pure Fabrication)

Problema: - Que objeto deve ter a responsabilidade quando nenhuma classe do domínio do problema pode levá-lo sem violar a alta coesão e baixo acoplamento?

- Nem todos as responsabilidades ajustam-se em classes de domínio, como persistência, comunicação de rede, interação do utilizador, etc…

Solução: Atribuir um conjunto altamente coeso de responsabilidades para uma classe artificial que não representa nada no domínio do problema. (Uma classe fictícia que possibilite alta coesão, baixo acoplamento e reutilização)

Benefícios e Contra Indicações:

- Alta coesão é suportada porque a responsabilidade são fatoradas numa classe que só incide sobre um conjunto muito específico de tarefas relacionadas.
- A reutilização pode ser aumentada devido à presença de classes "Pure Fabrication".

Indireção (Indirection)

Problema: - Como evitar o acoplamento direto?

- Como desacoplar objetos para que o baixo acoplamento é suportado e o potencial alto?

Solução: Atribuir a responsabilidade a um objeto intermédio para mediar entre outros componentes ou serviços, para que eles não são diretamente acoplados.

Benefícios e Contra Indicações:

- Baixo acoplamento
- Promove a reutilização

Variações Protegidas (Protected Variations)

Problema: Como projetar objetos, subsistemas e sistemas, para que variações ou instabilidades nos elementos não tenham um impacto indesejável sobre os outros elementos?

Solução: Identificar pontos de variação previstos ou imprevistos, atribuir responsabilidades para criar uma interface estável em torno deles.

Protected Variations é um princípio fundamental de design que é a base para muitos padrões de design.

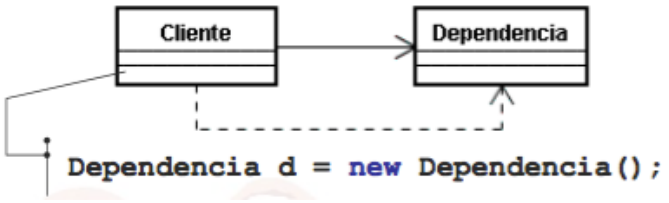
Mecanismos motivados por Protected Variations:

- Mecanismos PV: Encapsulamento de dados, interfaces, polimorfismo, indicação, padrões
- Data-Driven designs: “style sheets”, ficheiros pessoais, outros mecanismos para leitura de dados de configuração em tempo real
- Acesso uniforme

Padrões de Criação

"Abstraem o processo de instanciação, ajudando a tornar o sistema independente da maneira que os objetos são criados, compostos e representados".

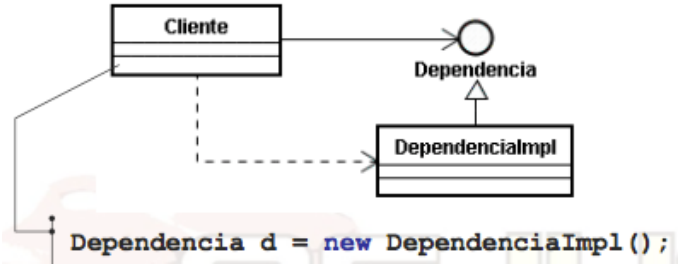
Introdução - Abordagem I



Solução inflexível:

- Cliente refere-se a uma implementação específica da sua dependência;
- Cliente constrói diretamente uma instância específica da sua dependência.

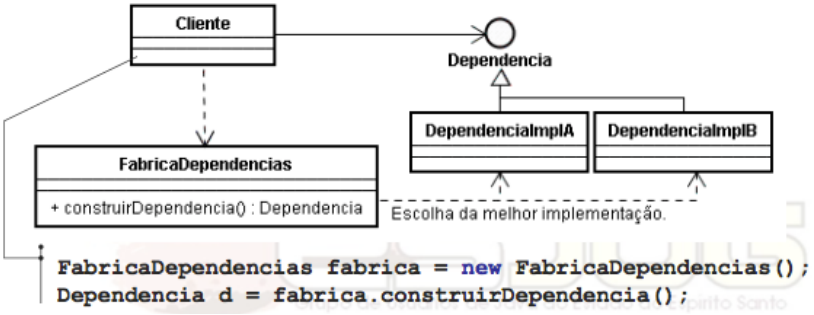
Introdução - Abordagem II



Alguma flexibilidade:

- Cliente já não mais associa-se a uma classe concreta;
- Porém, instancia a mesma diretamente.

Introdução - Abordagem III



Maior flexibilidade:

- A fábrica pode escolher a melhor classe concreta de acordo com um critério:

Maior flexibilização:

- Menor chance de ter que re-projetar;
- Menor desempenho;
- Maior dificuldade de realizar coisas simples;
- Maior dificuldade de compreender o código

Abstract Factory

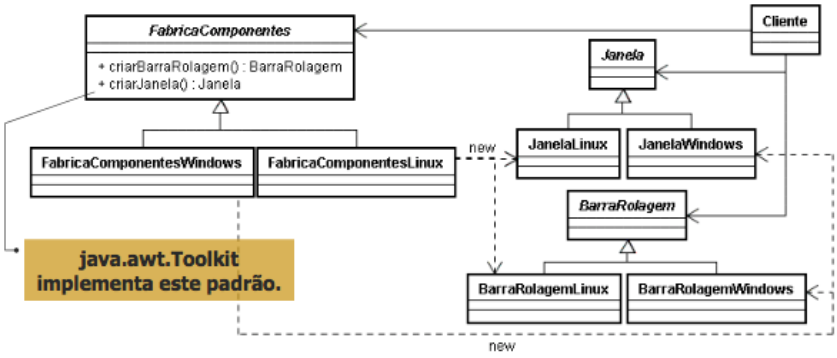
Intenção:

- Fornecer uma interface para criação de famílias de objetos relacionados sem especificar as suas classes concretas.

Problema:

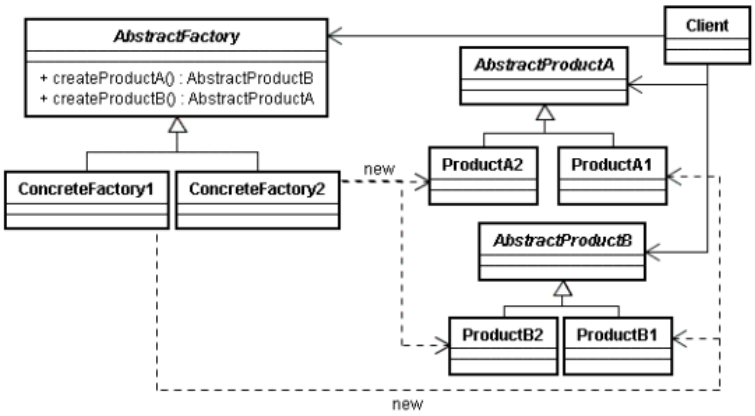
- Uma framework precisa de padronizar um modelo arquitetónico para uma gama de aplicações, mas permitir que aplicações individuais definam os seus próprios objetos e forneçam a sua instanciação.

Solução:



- Um método estático de uma classe retorna um objeto do tipo dessa classe;
- Uma das fábricas de componentes encarregar-se-á da criação dos objetos;
- Cliente não conhece as classes concretas.

Estrutura:



Usar este padrão quando:	Vantagens e Desvantagens:
- O sistema tiver que ser independente de como seus produtos são criados, compostos ou representados; - O sistema tiver que ser configurado com uma ou mais famílias de produtos (classes que devem ser usadas em conjunto); - Você quiser construir uma biblioteca de produtos e quiser revelar apenas suas interfaces e não suas implementações.	- Isola classes concretas: Fábricas cuidam da instanciação, o cliente só conhece interfaces; - Facilita a troca de famílias de classes: Basta trocar de fábrica concreta; - Promove consistência interna: Não dá para usar um produto de uma família com um de outra; - Criar novos produtos é trabalhoso: É necessário alterar as implementações de todas as fábricas para suportar o novo produto.

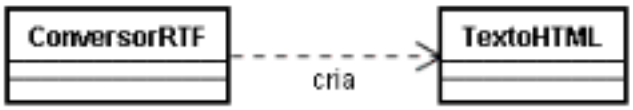
Builder

Intenção:

- Separa o processo de construção de um objeto complexo de sua representação para que o mesmo processo possa criar diferentes representações.

Problema:

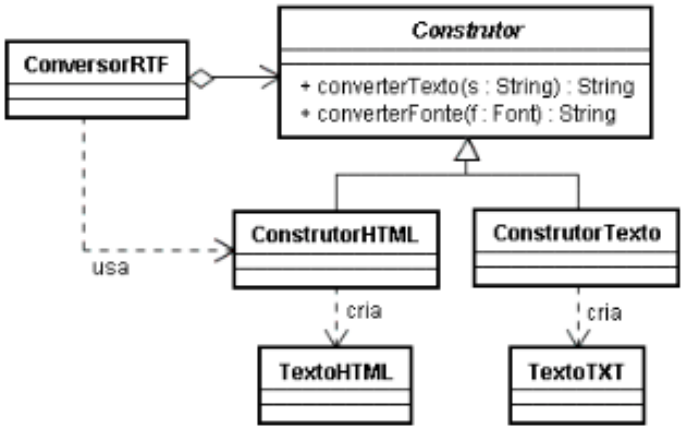
- Uma aplicação precisar criar os elementos de um complexo agregado. A especificação para o agregado existe no armazenamento secundário e uma das muitas representações precisa ser construído no armazenamento primário.



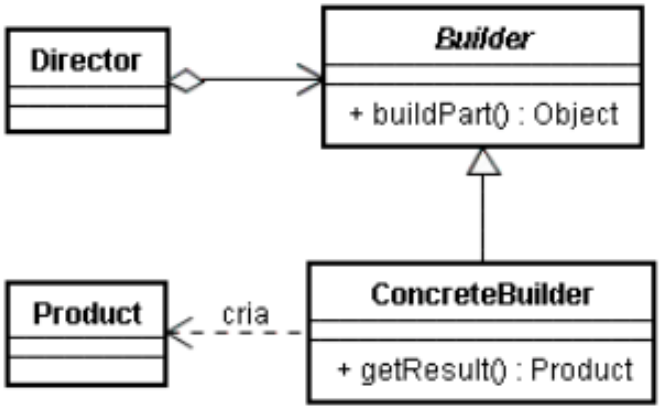
- Um objeto lê um texto em RTF e converte para HTML como produto final de um processamento

- Como fazer para preparar este sistema para uma eventual mudança de formato (TXT, por exemplo)?

Solução:



Estrutura:



- Para um sistema que mostre o resultado em TXT, basta trocar o construtor.

Usar este padrão quando:	Vantagens e Desvantagens:
- O algoritmo para criação de objetos complexos tiver que ser independente das partes que compõem o objeto e como elas são unidas; - O processo de construção tiver que permitir diferentes representações do objeto construído.	- Permite que varie a representação interna de um produto: Basta construir um novo builder - Separa o código da construção: melhora a modularidade, pois o cliente não precisa de saber da representação interna do produto. - Maior controlo do processo de construção: Constrói o produto passo a passo, permitindo o controlo de detalhes do processo de construção.

Factory Method

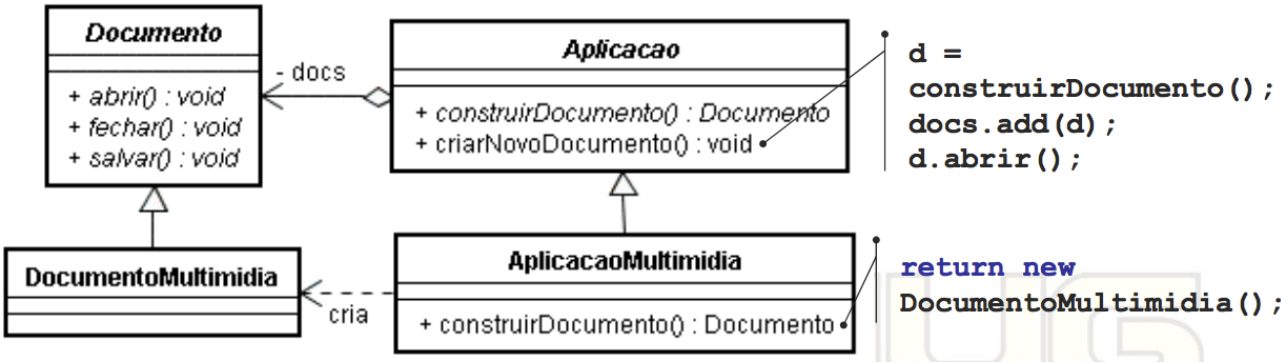
Intenção:

- Definir uma interface para a criação de objetos mas deixar as subclasses decidirem qual classe instanciar. Em outras palavras, delega a instanciação para as subclasses.

Problema:

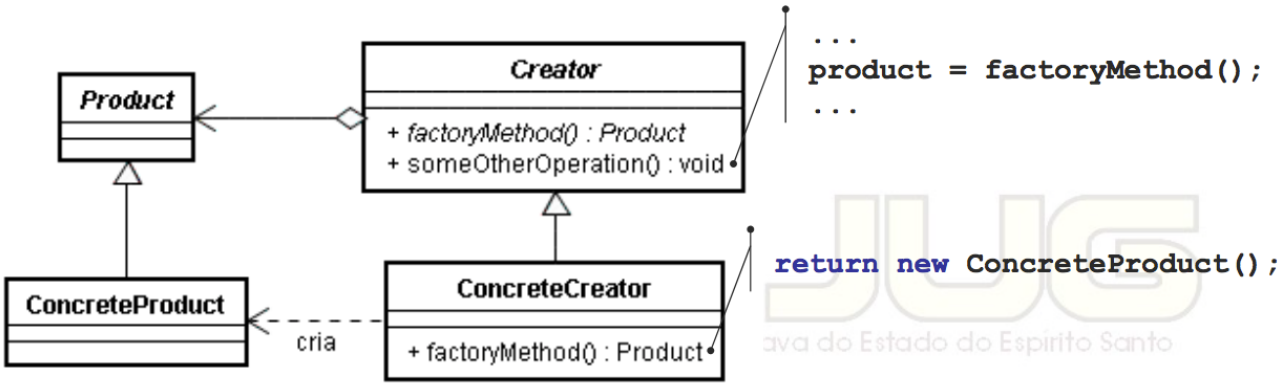
- Framework específico para uma aplicação que manipula documentos multimédia. É possível criar um framework mais genérico, para qualquer aplicação de manipulação de documentos?

Solução:



- Classes abstratas implementam as funções comuns a todo tipo de documento;
- Método fábrica é definido na superclasse e implementado na subclasse.

Estrutura:



Usar este padrão quando:	Vantagens e Desvantagens:
- Uma classe não tem como saber a classe dos objetos que precisará criar; - Uma classe quer que as suas subclasses especifiquem o objeto a ser criado.	- Melhor extensibilidade: Não é necessário saber a classe concreta do objeto para criá-lo. - Obrigatoriedade da subclasse fábrica: Não é possível criar somente um produto novo sem fábrica (excepto no caso do parametrizado)

Prototype

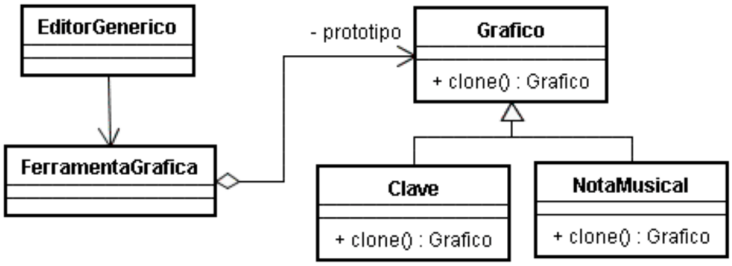
Intenção:

- Especificar o tipo de objeto a ser criado utilizando uma instância como protótipo e criar novos objetos copiando este protótipo.

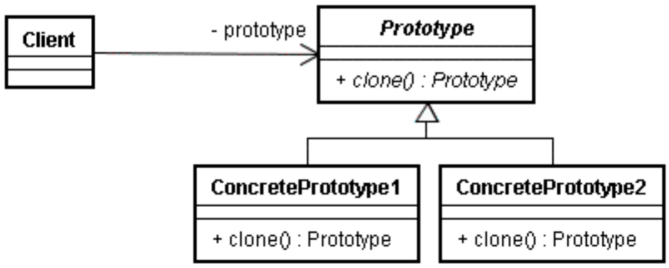
Problema:

- Existe um framework para edição de documentos genéricos. A ferramenta gráfica para um editor musical precisa de conhecer os gráficos específicos, o que anula o benefício conseguido anteriormente.

Solução:



Estrutura:



- A ferramenta gráfica é configurada com protótipos dos objetos gráficos.
- Quando precisar de criar algum deles, chama o método clone(), que retorna uma cópia de si mesmo.

Usar este padrão quando:	Vantagens e Desvantagens:
- O sistema deve ser independente de como seus produtos são criados, compostos e representados e… - As classes que devem ser ciadas são especificadas em tempo de execução; - Ou você não quer construir uma fábrica para cada hierarquia de produtos; - Ou as instâncias da classe clonável só tem alguns (poucos) estados possíveis, e é melhor clonar do que criar objetos	- Esconde a implementação do produto; - Permite adicionar e remover produtos em tempo de execução (configuração dinâmica da aplicação); - Não necessita de uma fábrica para cada hierarquia de objetos; - Implementar clone() pode ser complicado.

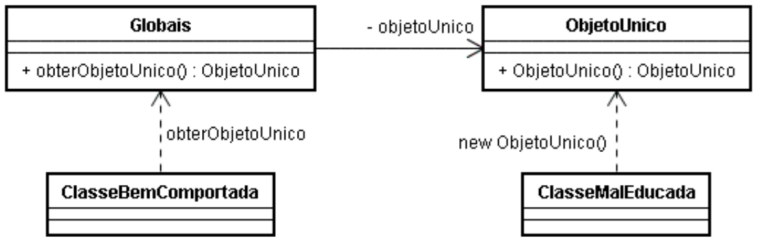
Singleton

Intenção:

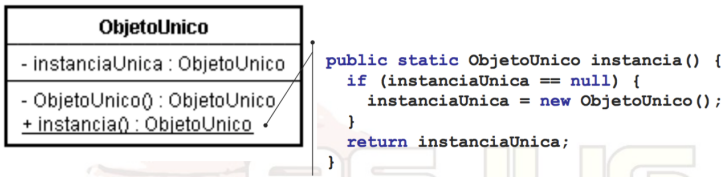
- Garantir que uma classe possui somente uma instância e fornecer um ponto de acesso global a ela.
- Algumas vezes precisa-se que uma classe só tenha uma instância para todo o sistema.
- Providenciar um ponto de acesso *static* não é suficiente, pois classes poderão ainda construir outra instância diretamente.

Problema:

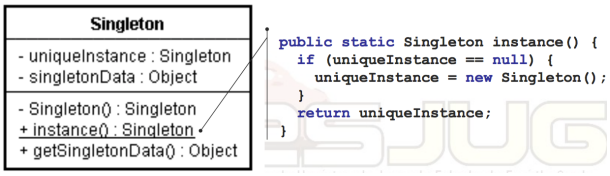
- A aplicação precisa de uma, e apenas uma, instância do objeto.
- Garantir que uma classe possui somente uma instância e fornecer um ponto de acesso global a ela.
- Algumas vezes precisa-se que uma classe só tenha uma instância para todo o sistema.
- Providenciar um ponto de acesso *static* não é suficiente, pois classes poderão ainda construir outra instância diretamente.



Solução:



Estrutura:



- Definir um ponto de acesso global (método static);
- Construtor privado ou protected, dependendo se as subclasses devem ter acesso;
- Bloco de criação poderia ser *synchronized* para maior seguranças em ambientes multithread;
- A instância única poderia ser pré-construída.

Usar este padrão quando:	Vantagens e Desvantagens:
- Tiver que haver exatamente uma instância de uma classe e ela tiver que estar acessível a todos num local bem definido; - Quiser permitir ainda que esta classe tenha subclasses (construtor protected).	- Acesso controlado à instância: A própria classe controla a sua instância única. - Não há necessidade de variáveis globais: Variável globais poluem o espaço dos nomes. - Permite extensão e refinamento: A classe Singleton pode ter subclasses. - Permite número variado de instâncias: Podemos controlar este número. - Mais flexível do que operações de classe: Usar membro static perde flexibilidade.

Object Pool

Intenção:

- Objet Pool pode oferecer um aumento de desempenho significativo. É mais eficaz em situações onde:
    - O custo de inicializar uma instância da classe é alta;
    - A taxa de instanciação de uma classe é alta;
    - O número de instâncias em uso a qualquer momento é baixa.

Problema:

- Os objetos são usados para gerir a cache do objeto. Um cliente com acesso a uma pool de objetos pode evitar a criação de novos objetos simplesmente perguntando à pool por um que já tenha sido instanciado.
- É desejável manter todos os objetos reutilizáveis que não estão atualmente em uso no mesmo pool de objetos para que eles possam ser geridos por uma política coerente.

Resumo:

- É utilizado para a reutilização de objetos e não para os eliminar.
- Verifica se o objeto está na pool e se estiver retorna-o em vez de criar um novo

RESUMIDAMENTE

Abstract Factory: Cria uma instância de várias famílias de classes;
Builder: Separa a construção do objeto da sua representação;
Factory Method: Cria uma instância de várias classes derivadas;
Prototype: Instância totalmente inicializada a serem copiados ou clonados;
Singleton: Uma classe do qual só um exemplo único pode existir
Object Pool: Evita a extensa aquisição e libertação de recursos através da reciclagem de objetos que não estão em uso.

Padrões de Estrutura

"Padrões de estrutura com foco em classes usam herança para compor interfaces ou implementações. Os padrões de estrutura com foco nos objetos descrevem formas de compor objetos para realizar novas funcionalidades".

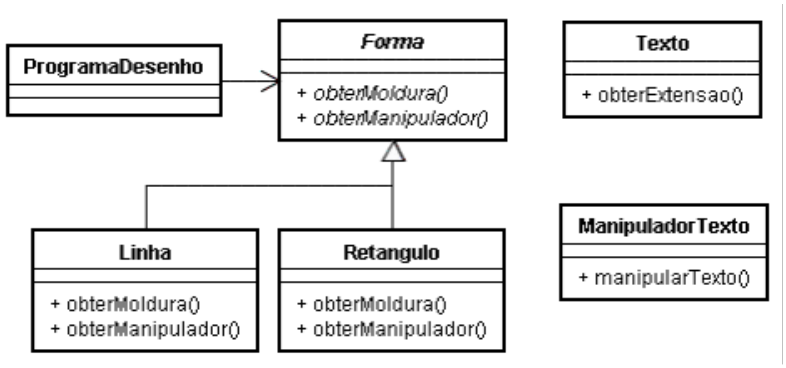
Adapter

Intenção:

- Converter a interface de uma classe numa interface esperada pelo cliente. Permite que as classes com interfaces incompatíveis possam colaborar.
- "Wrap" uma classe existente com uma nova interface.

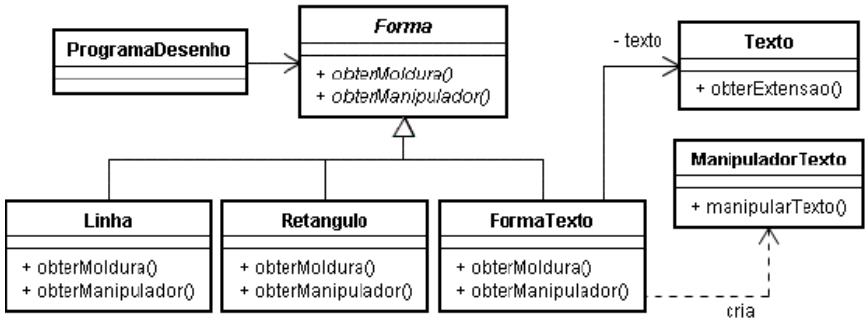
Problema:

- Existe uma ferramenta gráfica de texto pronta, mas o programa de desenho só trabalha com formas.

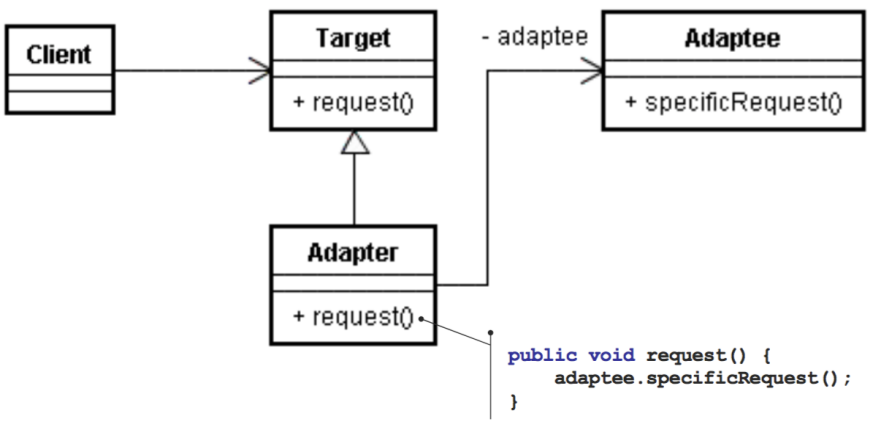


Solução:

- FormaTexto adapta a classe pronta à interface esperada pelo programa de desenho.



Estrutura:



Usar este padrão quando	Vantagens e Desvantagens
- Queremos usar uma classe já pronta que possui uma interface diferente do que precisamos. - Queremos criar uma classe reutilizável já prevendo que a situação acima ocorrerá no futuro.	**Adapter para Classes**: (*subclassing*) - Não funciona bem quando se quer adaptar uma hierarquia de classes - Permite que o *adapter* subescreva alguma funções do adaptado **Adapter para Objetos:** (*delegations*) - Permite o uso de um único *adapter* para uma hierarquia de classes adaptadas - É mais difícil subescrever funções do adaptado

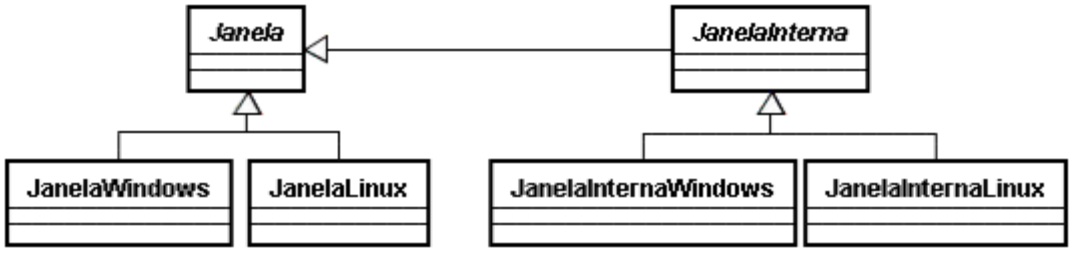
Subclassing
- Fornece automaticamente acesso a todos os métodos á superclasse
- Mais eficiente

Delegation
- Permite remoção de métodos
- Wrappers podem ser adicionados ou removidos automaticamente
- Vários Objetos podem ser compostos
- Mais flexível

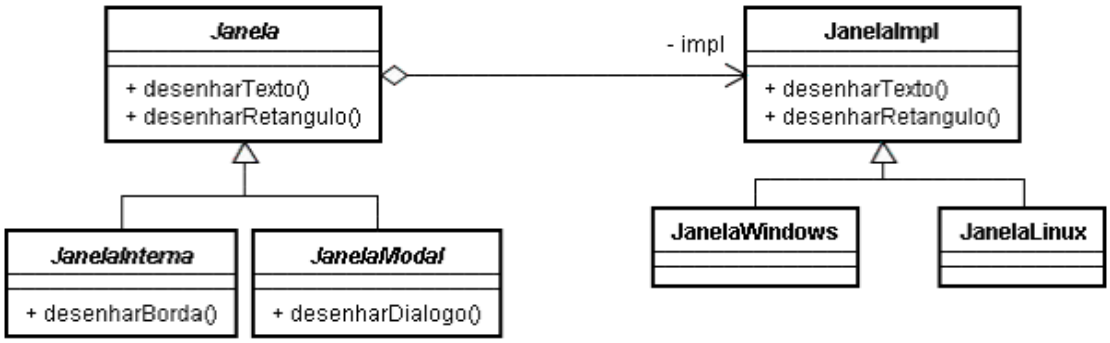
Bridge

Intenção:
- Desacoplar uma abstração da sua implementação para que ambos possam variar independentemente.

Problema:
- O endurecimento das “artérias do software” ocorre usando subclasses de uma classe base abstrata para fornecer implementações alternativas. Isto bloqueia a ligação entre a interface e implementação em “compile-time”. A abstração e implementação não podem ser estendidas ou compostas de forma independente.
  - Componentes gráficos devem ser implementados para várias arquiteturas
  - Cada novo componente exige várias implementações
  - Cada nova arquitetura mais ainda

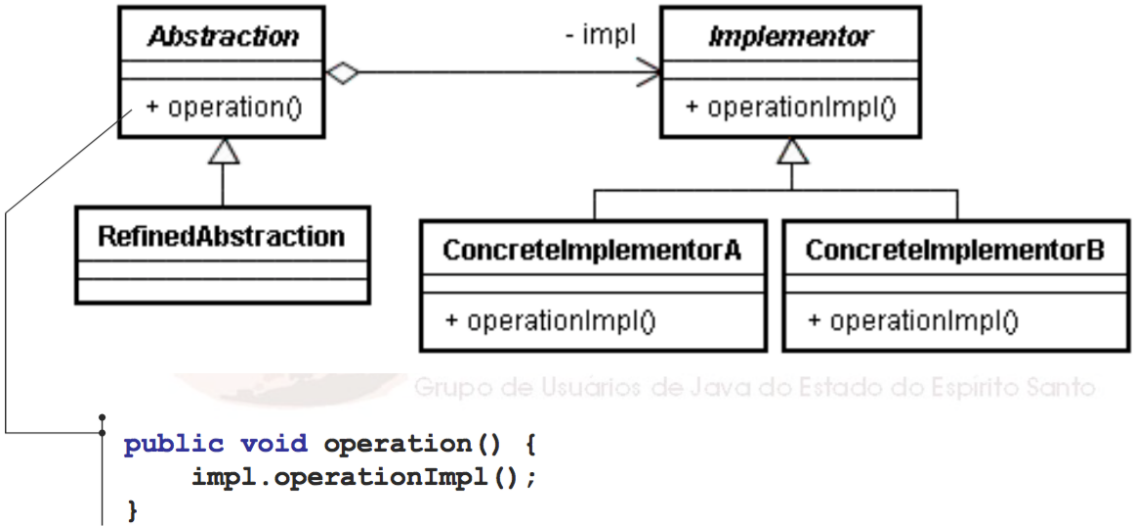


Solução:



- Janela concentra métodos que utilizam recursos específicos da plataforma.
- Subclasses utilizam os métodos de janela para implementar itens específicos.

Estrutura:



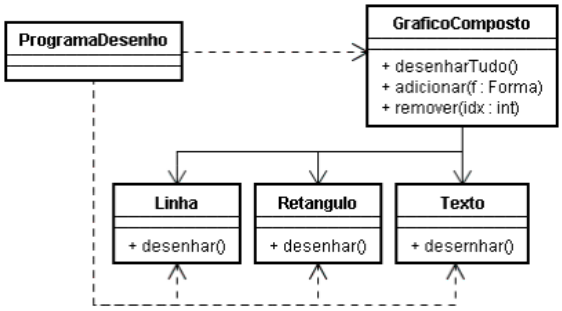
Usar este padrão quando	Vantagens e Desvantagens
- Queremos evitar uma ligação permanente entre a abstração e a implementação. - Tanta a abstração quanto a implementação possuem subclasses. - Mudanças na implementação não devem afetar o código do cliente - A sua atual solução gera uma proliferação de classes	Desacopla a implementação - Podendo até mudá-la em tempo de execução Melhora a extensibilidade - É possível estender a abstração e a implementação separadamente Esconde detalhes de implementações - Clientes não precisam de saber como é implementado

Composite

Intenção:

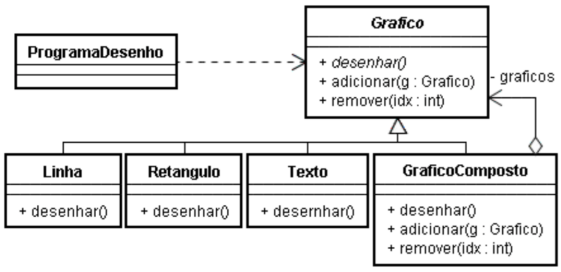
- Compor objetos em estruturas de árvore para representar hierarquias.
- Permite que clientes tratem dos objetos individuais e compostos de maneira uniforme
- Composição recursiva
- Diretórios contêm entradas, cada uma destas pode ser um diretório
- Hierarquia definida de um-para-muitos

Problema:

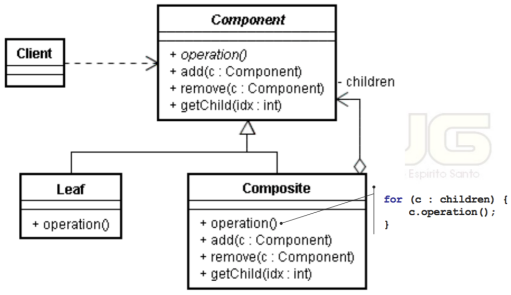


- Existem gráficos que são compostos por outros gráficos.
- O programa tem que conhecer cada um deles, o que complica o código.

Solução:



Estrutura:



- A classe abstrata representa tanto gráficos simples quanto compostos;
- Programa só precisa conhecer Gráfico.

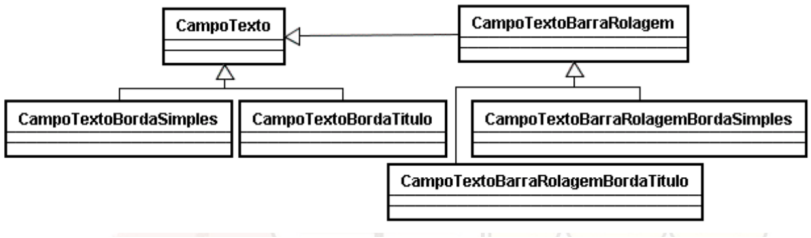
Usar este padrão quando	Vantagens e Desvantagens	Transparência VS. Segurança
- Quisermos representar hierarquias. - Quisermos que os clientes ignorem a diferença entre objetos simples e objetos compostos.	Define hierarquias: - Objetos podem ser compostos de outros objetos e assim sucessivamente Simplifica o cliente: - Clientes não precisam de ter a preocupação se estão a lidar com compostos ou individuais Facilita a criação de novos membros: - Basta estar em conformidade com a interface comum a todos os componentes Pode tornar o objeto muito genérico: - Qualquer componente pode ser criado, não dá para verificar os tipos e aplicar restrições.	O Composite viola o principio de herança: - Leaf IS-A Component é falso, pois add( ), remove( ), etc não fazem sentido para Leaf. - Mais transparência(tratamento uniforme). - Menos segurança(verificação dos tipos). Paleativos(opcionais): - Defina Component como classe abstrata e seus metódos com implementação vazia. - Defina getComposite() para retornar a si memso se for composto e null caso contrário.

Decorator

Intenção:

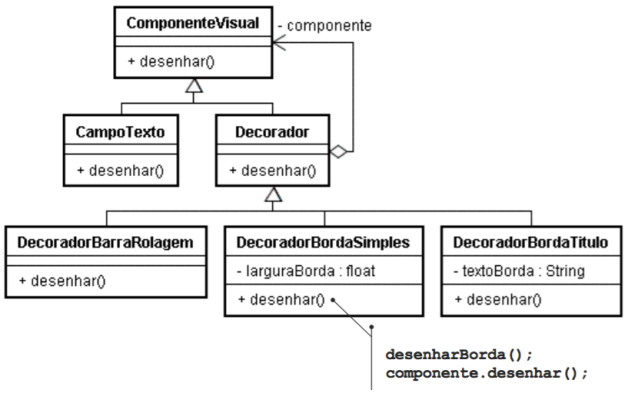
- Anexa funcionalidades adicionais a um objeto dinamicamente. Fornece uma alternativa flexível à herança como mecanismo de extensão.

Problema:

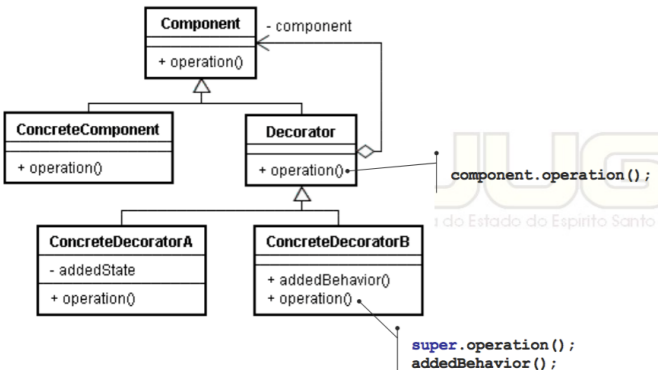


- Desejamos adicionar comportamentos ou estados para objetos individuais em rum-time **MAS** a herança não é viável porque é estática e aplica-se a toda a classe.

Solução:



Estrutura:



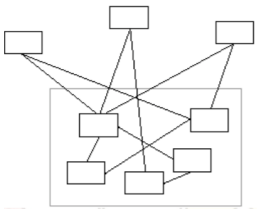
Usar este método quando	Vantagens e Desvantagens
- Queremos adicionar funcionalidades dinamicamente e transparentemente. - Queremos adicionar funcionalidades que podem ser desativados mais tarde. - Extensão por herança é impraticável (não disponível ou produziria uma explosão de subclasses.	- Maior flexibilidade do que herança: • Podem ser adicionados/removidos em tempo de execução • Pode adicionar duas vezes a mesma funcionalidade - O *Decorator* é diferente do *Composite*: • A identidade do objeto não pode ser usada de forma confiável - Muitos objetos pequenos: • Um projeto que utilize decorator pode vir a ter muitos objetos pequenos e parecidos

Façade

Intenção:

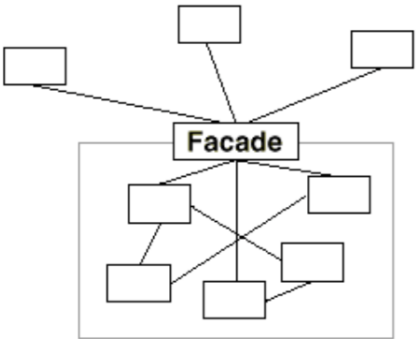
- Fornecer uma interface unificada para um conjunto de interfaces de um subsistema. Façade define uma interface de mais alto nível para tornar o uso dos subsistemas mais fácil.

Problema:

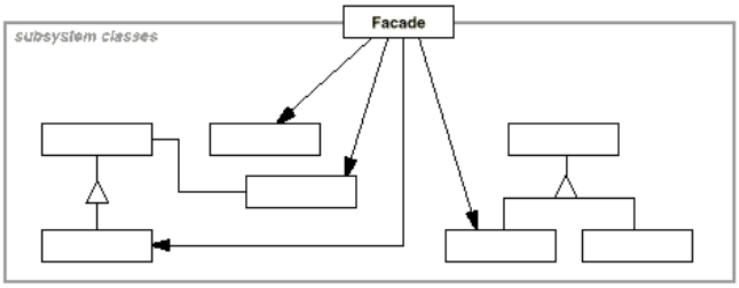


- Clientes acessam vários subsistemas
- Mudanças em algum subsistema demandam alterações em diversos clientes

Solução:



Estrutura:



- Introdução de um objeto façade que fornece uma interface simplificada e única ao sistema.

Façade e Singleton

Usar este padrão quando	Vantagens e Desvantagens
- quisermos fornecer uma interface simples para um subsistema complexo. - diminuir a dependência direta entre o cliente e classes internas do seu sistema. - desenvolver o sistema em múltiplas camadas, cada uma com a sua Façade.	- Facilita a utilização do sistema: • O cliente apenas precisa conhecer o Façade. - Promove o acoplamento fraco: • Pequenas mudanças no sistema não afetam mais o cliente. - Versatilidade: • Quando necessário, clientes ainda podem aceder ao subsistema diretamente (se quiser permitir isso).

- Façade geralmente é implementado como Singleton;
- Pode não ser o caso se o sistema tiver múltiplos utilizadores e cada um usar uma Façade separada.

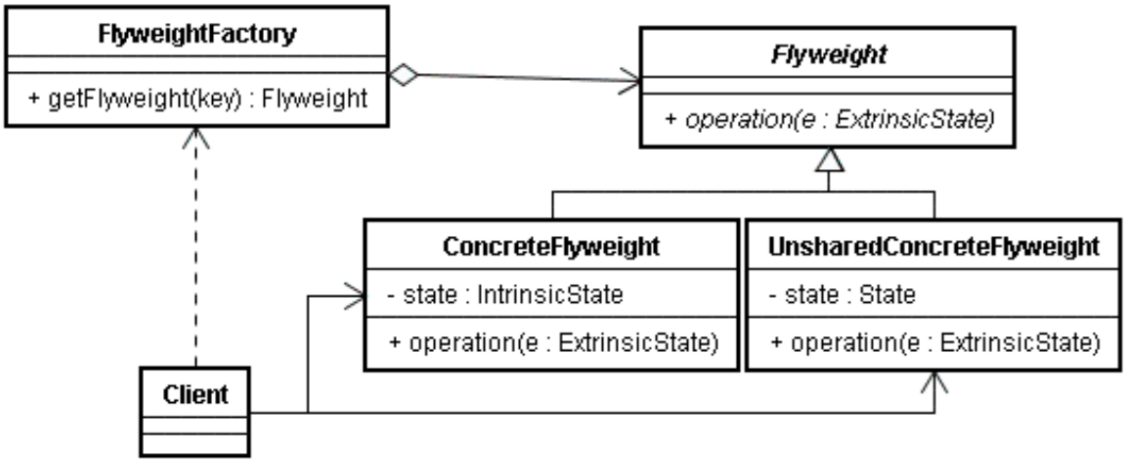
Flyweight

Intenção:
- Estabelecer partilhas de objetos de granularidade muito pequena para dar suporte ao uso eficiente de grande quantidade deles.

Problema:
- Desenvolver um editor de texto onde cada caractere é representado por um objeto:
  - Granularidade muito pequena
  - Não haverá recursos (memória) suficiente para textos grandes

Solução:
- Monta-se uma pool de objetos compartilhados
- Cada caracter tem um objeto

Estrutura:



Usar este padrão quando	Vantagens e Desvantagens
- Todas as seguintes condições forem verdade: • Aplicação usa um grande número de objetos. • O custo de armazenamento é alto por causa desta quantidade. • O estado dos objetos podem ser externos. • Objetos podem ser compartilhados assim que o seu estado é externo. • A aplicação não depende da identidade	- Custo x Beneficio: • Custo de recuperar o objeto compartilhado e transferir o seu estado externo • Benefício de economia de recursos.

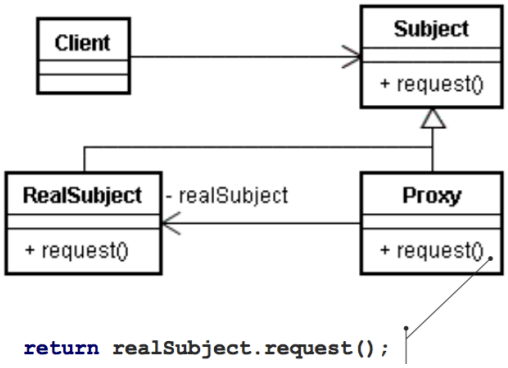
Proxy

Intenção:
- Fornecer um representar ou ponto de acesso que controle o acesso a um objeto.

Problema:
- Necessidade de suportar objetos com "necessidade" de recursos e não se quer instanciar esses mesmos objetos, a menos que sejam realmente solicitados pelo cliente.

Solução:
- Criar um objeto proxy que implemente a mesma interface que o objeto real
  - O objeto proxy (normalmente) contém uma referência para o objeto real
  - Os clientes recebem um referência para o proxy, não o objeto real

Estrutura:



Utilizar este padrão quando	Vantagens e Desvantagens
- Precisamos de um acesso mais versátil a um objeto do que um ponteiro • Remote proxy (acesso remoto) • Virtual Proxy • Protection proxy (controla acesso)	- Adiciona um nível de separação - Transparência na execução de ações de carregamento de objetos.

RESUMIDAMENTE

Adapter: Combina interfaces de diferentes classes;
Bridge: Separa a interface de um objeto a partir da sua implementação;
Composite: Estrutura em árvore de objetos simples e compostos;
Decorator: Adicionar responsabilidades a objetos dinamicamente;
Façade: Uma única classe que representa o subsistema;
Flyweight: Um exemplo de grão fino usado para uma partilha eficiente;
Proxy: Um objeto que representa outro objeto.

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

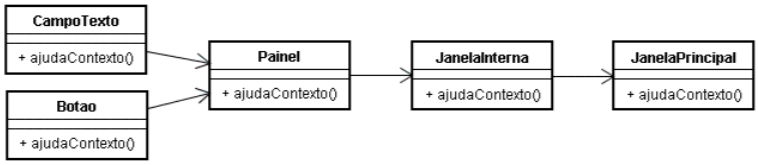
- **Interpreter**

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.

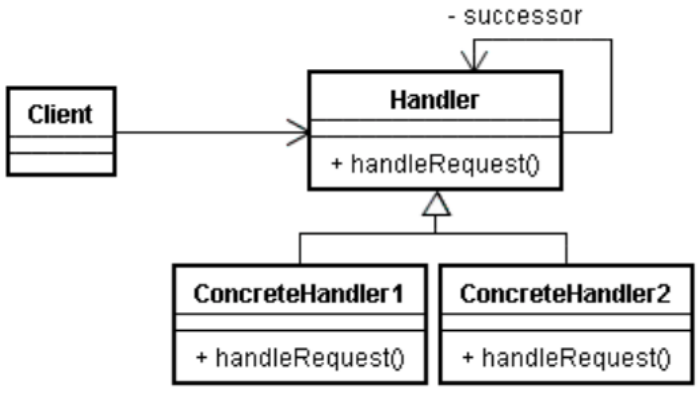
Usar este padrão quando:

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”

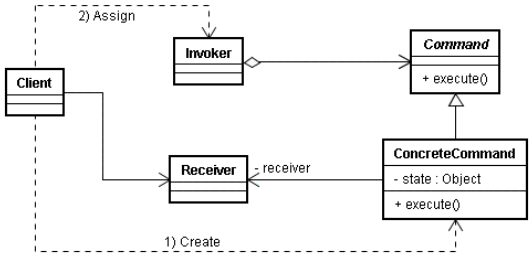
Problema:

- 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. */ }
```

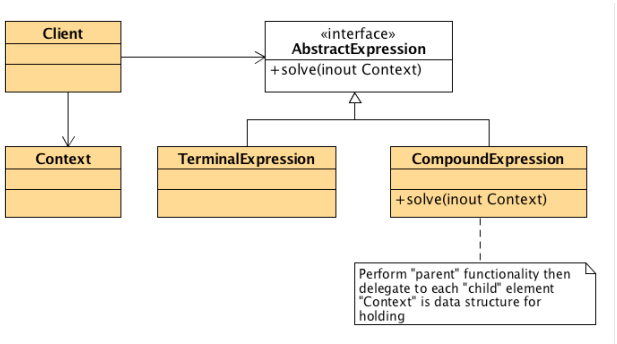
- Algumas tarefas ocorrem com tanta frequência de formas diferentes que é interessante criar uma linguagem só para definir este tipo de problema.

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

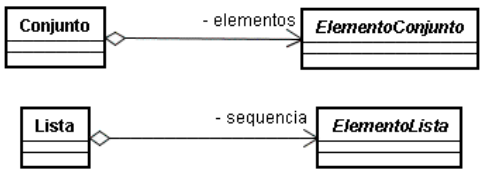
- Interpreter pode ser usado como um refinamento de Command:
  Comandos são escritos numa gramática criada para tal;
  Interpreter lê esta linguagem e cria objetos Command para execução.

• Iterator- Iterador

Intenção:

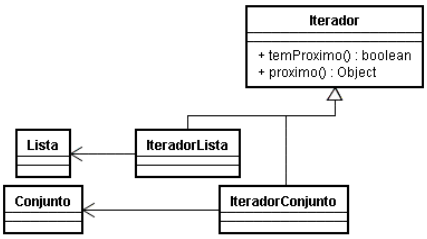
- Prover uma forma de acessar os elementos de um conjunto em sequência sem expor a representação interna deste conjunto.

Problema:



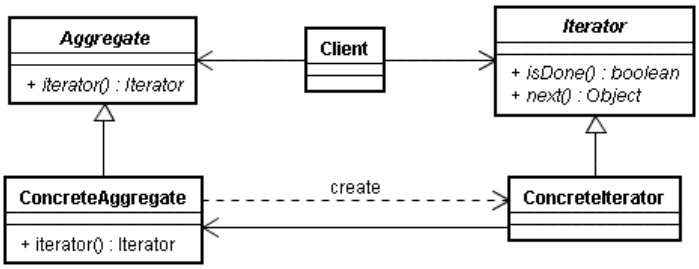
- Cliente precisa acessar os elementos;
- Cada coleção é diferente e não se quer expor a estrutura interna de cada um para o Cliente.

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

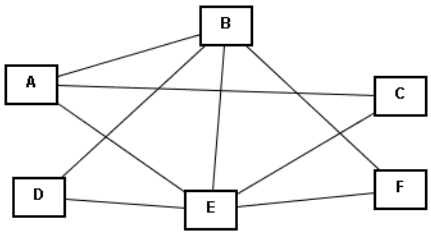
- Múltiplas formas de acesso:
  Basta implementar um novo iterador com uma nova lógica de acesso.
-Interface simplificada:
  Acesso é simples e uniforme para todos os tipos de coleções.
- Mais de um iterador:
  É possível ter mais de um acesso à coleção em pontos diferentes.

- **Mediator- Mediador**

Intenção:

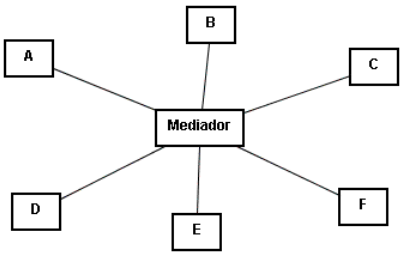
Definir um objeto que encapsula a informação de como um conjunto de outros objetos interagem entre si. Promove o acoplamento fraco, permitindo que você altere a forma de interação sem alterar os objetos que interagem.

Intenção:



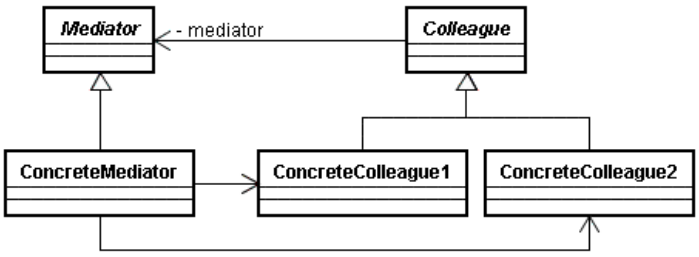
-Modelagem OO encoraja a distribuição de responsabilidades;
- Esta distribuição resulta em um emaranhado de conexões.

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.

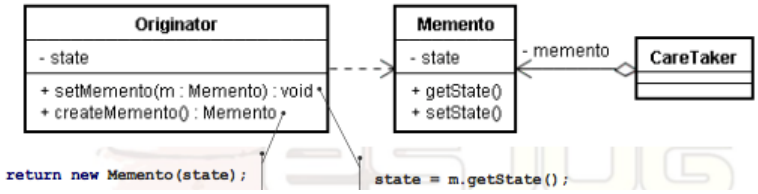
Problema:

- 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

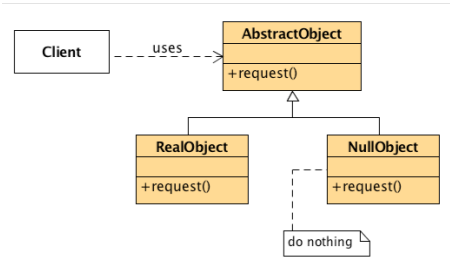
- Identificar os papeis de “carataker” e “originator”
- Originator cria um Memento e copia o seu estado para o mesmo Memento

• 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:

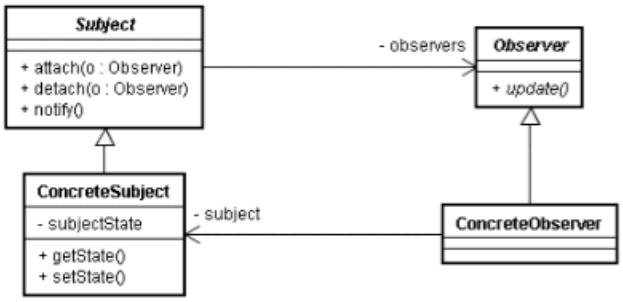
- Definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, os outros são notificados e se atualizam.

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:

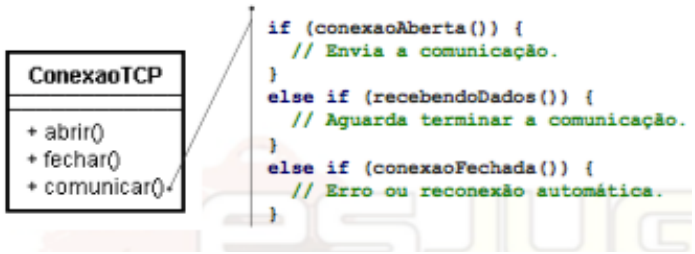
- Diferença entre a funcionalidade independente do núcleo e a funcionalidade opcional.
- Modela a funcionalidade independente com uma abstração “Subject”
- Modela a funcionalidade dependente com uma hierarquia “Observer”.
- Subject é acoplado apenas para a classe base do Observer.

- **State - Estado**

Intenção:

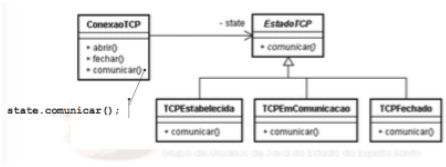
- Permitir que um objeto altere seu comportamento quando muda de estado interno. O objeto aparenta mudar de classe.

Problema:



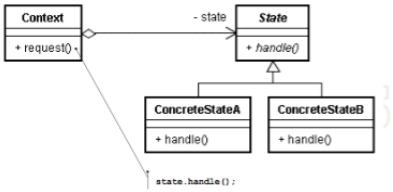
- Um objeto responde diferentemente dependendo do seu estado interno

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

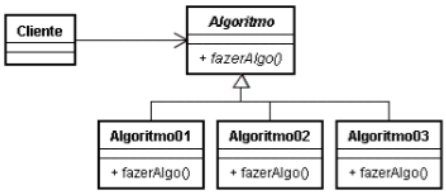
Intenção:

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

Problema:

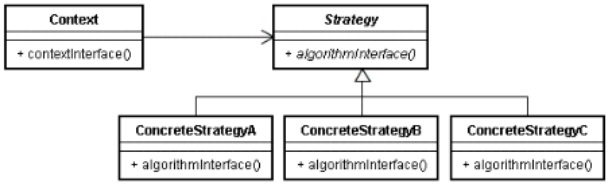
- Existem problemas que possuem vários algorítmos que os solucionam;
  Ex.: quebrar um texto em linhas.
- Seria interessante:
  Separar estes algorítmos em classes específicas para serem reutilizados;
  Permitir que sejam intercambiados e que novos algorítmos sejam adicionados com facilidade.

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

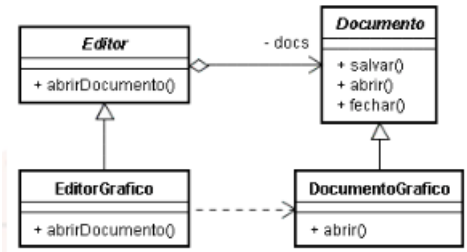
- Famílias de algoritmos:
Beneficiam-se de herança e polimorfismo.
- Alternativa para herança do cliente:
Comportamento é a única coisa que varia.
- Eliminam os grandes condicionais:
Evita código monolítico.
- Escolha de implementações:
Pode alterar a estratégia em runtime.
- Clientes devem conhecer 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:

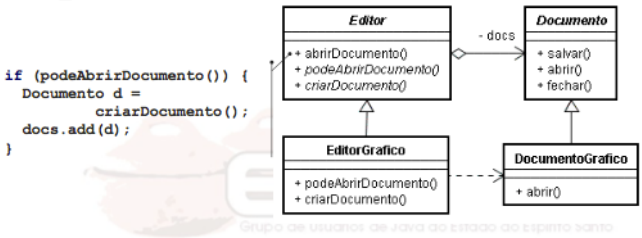
Definir o esqueleto de um algorítmo numa classe, delegando alguns passos às subclasses. Permite que as subclasses alterem partes do algorítmo, sem mudar sua estrutura geral.

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.

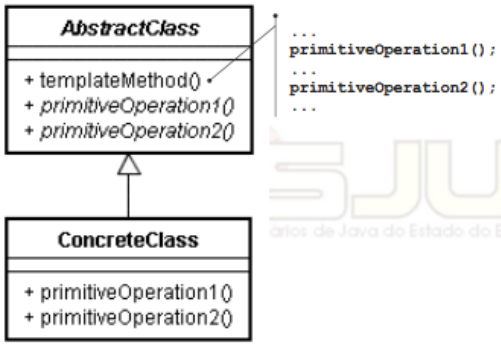
Solução:



(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

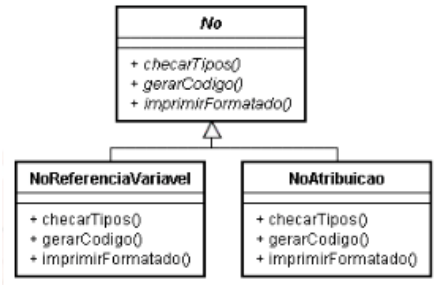
- Reuso de código:
  Partes de um algorítmo são reutilizadas por todas as subclasses.
- Controle:
  É possível permitir o que as subclasses podem estender (métodos finais).
- Comportamento padrão extensível:
  Superclasse pode definir o comportamento padrão e permitir sobrescrita.

- **Visitor - Visitante**

Intenção:

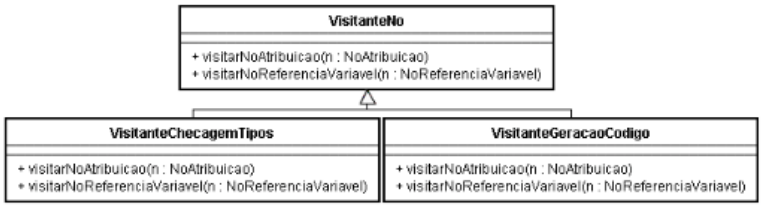
Representar uma operação a ser efetuada em objetos de uma certa classe como outra classe. Permite que seja definido uma nova operação sem alterar a classe na qual a operação é efetuada.

Problema:



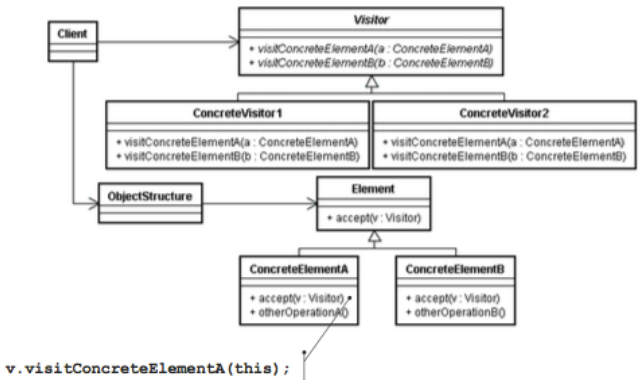
- Um compilador representa o código como uma árvore sintática. Para cada nó precisa de realizar algumas operações
- Misturar estas operações pode ser confuso.

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

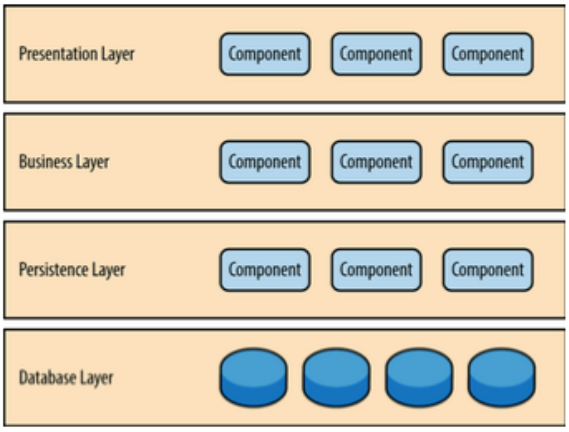
---

Software Arch

Layered Architecture - Arquitetura por Camadas

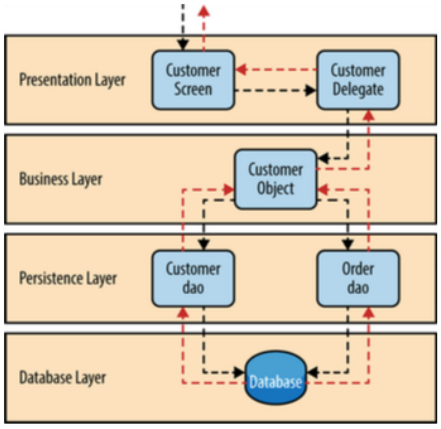
- O padrão de arquitetura mais comum é o padrão de arquitetura em camadas.
- O padrão de arquitetura em camadas aproxima-se da comunicação tradicional de IT e estruturas organizacionais encontradas na maioria das empresas.
  - tornando-se uma escolha natural para a maioria dos esforços de desenvolvimento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



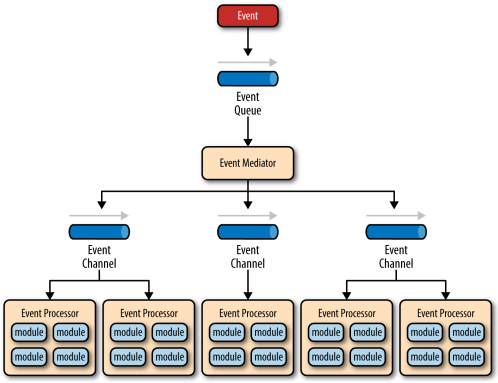
Agilidade geral – classificação: baixo
Facilidade de implementação – classificação: baixo
Capacidade de teste – Avaliação: alta
Desempenho – classificação: baixo
Escalabilidade – Classificação: baixo
Facilidade de desenvolvimento – Rating: alta

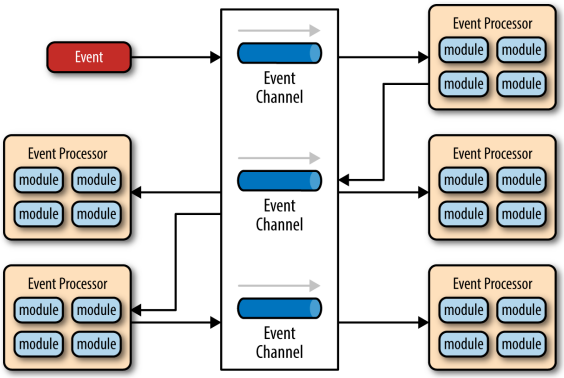
A 'tela do cliente' é responsável por aceitar a solicitação e visualização de informações.
-Não sabe onde os dados são, como ele é recuperado, quais tabelas de banco de dados devem ser consultadas!
– ele encaminha a solicitação para o módulo de representante do cliente.
– Este módulo é responsável por 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.



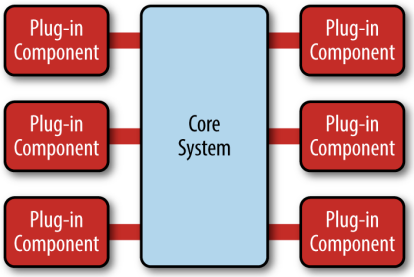


Considerações:

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.

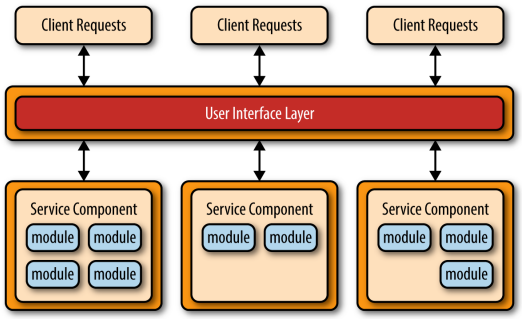
- Dois tipos de componentes de arquitetura:
  – um núcleo do sistema e módulos plug-in
- O núcleo do sistema tradicionalmente contém somente a funcionalidade mínima necessária para tornar o sistema operacional
-Módulos plug-in podem ser conectados ao sistema através de uma variedade de maneiras de núcleo
  – 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.

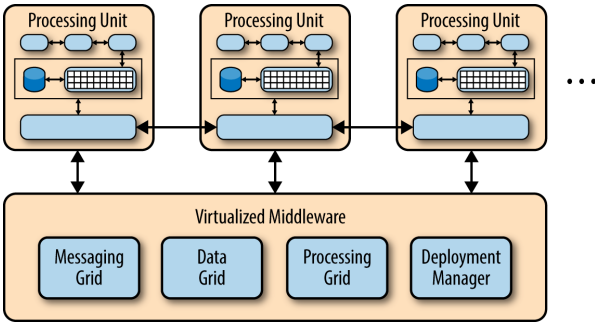
Agilidade geral – classificação: alta
Facilidade de implementação – classificação: alta
Capacidade de teste – Avaliação: alta
Desempenho – classificação: baixa
Escalabilidade – Classificação: alta
Facilidade de desenvolvimento – Rating: alta

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).