Padrões e Desenho de Software

Resumos
2015/2016

João Alegria | 68661

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".

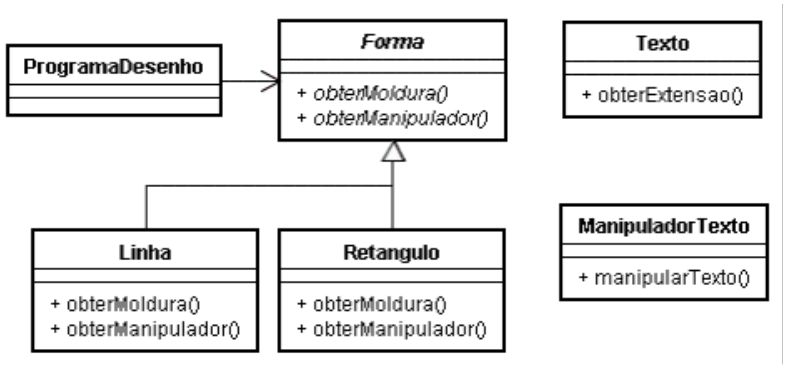
Adapter

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.

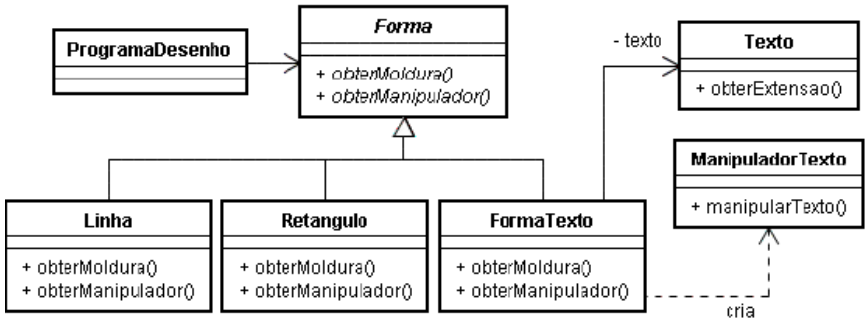
Problema:

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

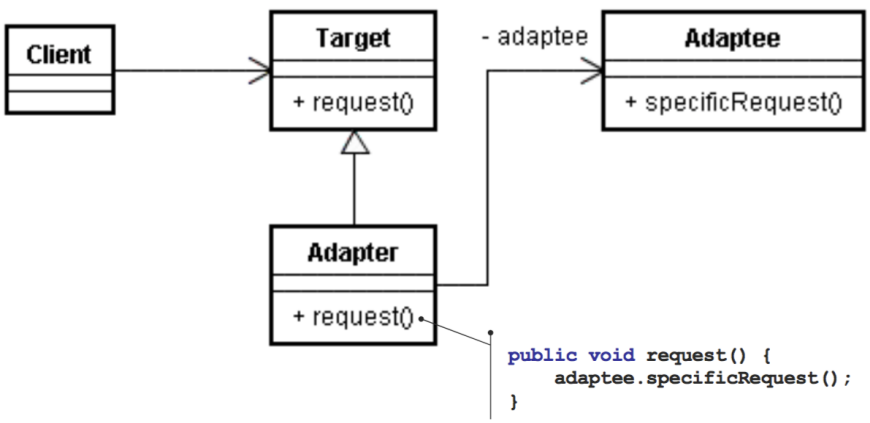


Solução:

- 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

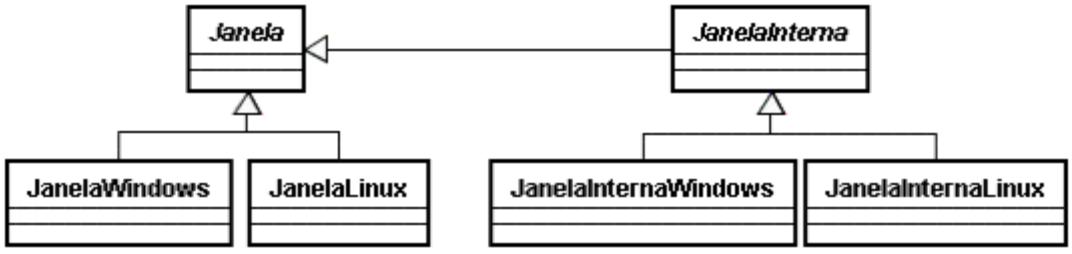
Subclassing
- Fornece automaticamente acesso a todos os métodos á superclasse
- Mais eficiente

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

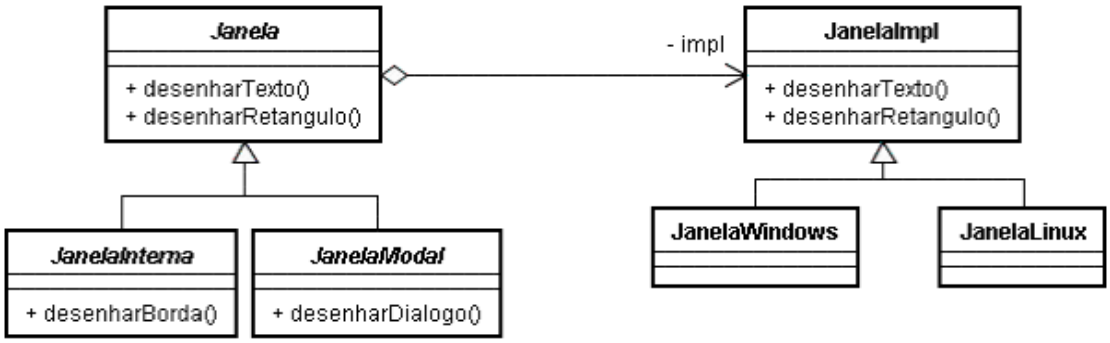
Bridge

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

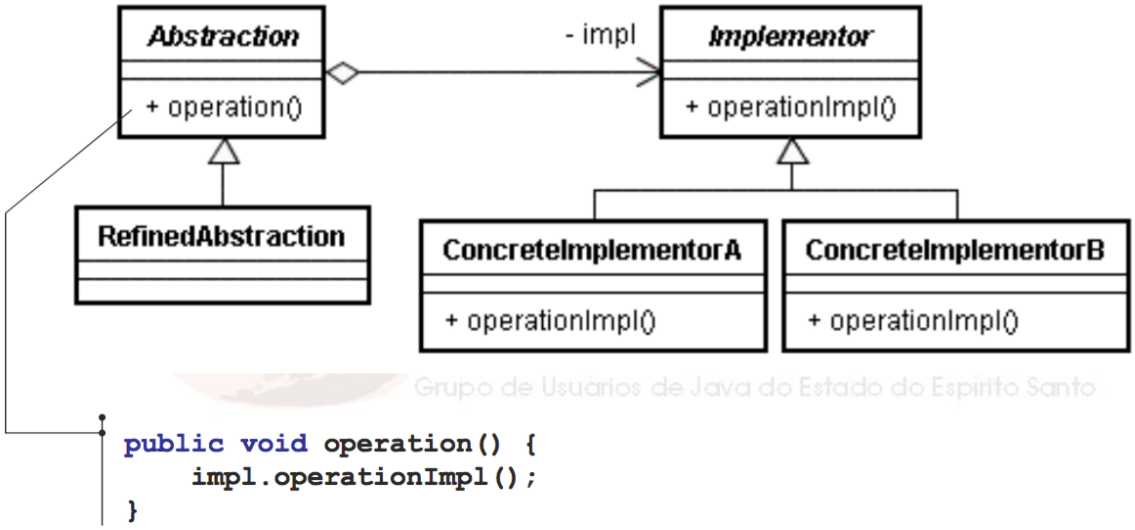


Solução:



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

Estrutura:



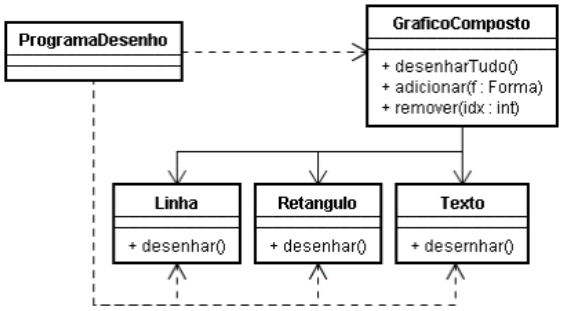
Usar este padrão quando	Vantagens e Desvantagens
- Queremos evitar uma ligação permanente entre a abstração e a implementação. - Tanta a abstração quanto a implementação possuem subclasses. - Mudanças na implementação não devem afetar o código do cliente - A sua atual solução gera uma proliferação de 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:

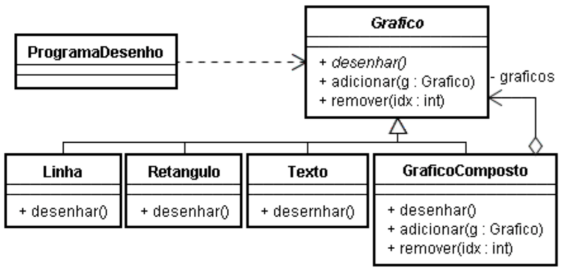
- Compor objetos em estruturas de árvore para representar hierarquias.
- Permite que clientes tratem dos objetos individuais e compostos de maneira uniforme
- Composição recursiva
- Diretórios contêm entradas, cada uma destas pode ser um diretório
- Hierarquia definida de um-para-muitos

Problema:

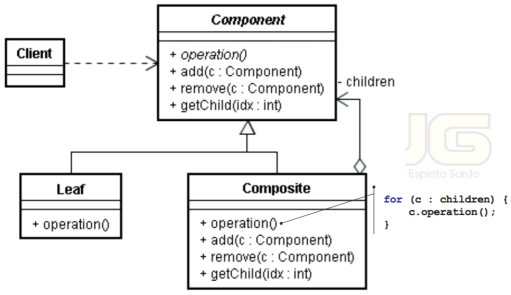


- Existem gráficos que são compostos por outros gráficos.
- O programa tem que conhecer cada um deles, o que complica o código.

Solução:



Estrutura:



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

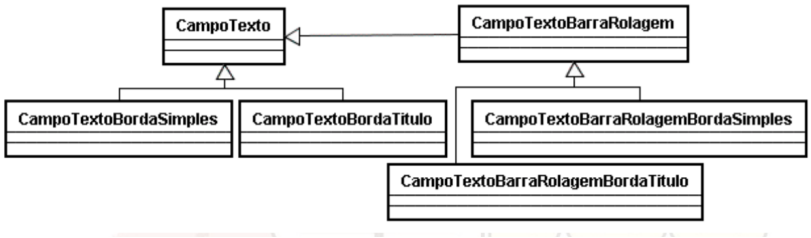
Usar este padrão quando	Vantagens e Desvantagens	Transparência VS. Segurança
- Quisermos representar hierarquias. - Quisermos que os clientes ignorem a diferença entre objetos simples e objetos compostos.	Define hierarquias: - Objetos podem ser compostos de outros objetos e assim sucessivamente Simplifica 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:

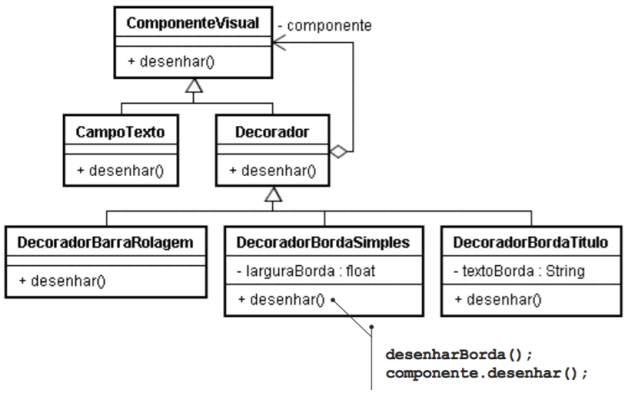
- Anexa funcionalidades adicionais a um objeto dinamicamente. Fornece uma alternativa flexível à herança como mecanismo de extensão.

Problema:

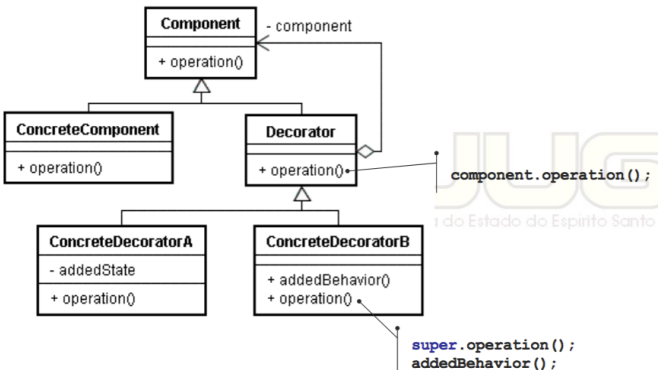


- Desejamos adicionar comportamentos ou estados para objetos individuais em rum-time **MAS** a herança não é viável porque é estática e aplica-se a toda a classe.

Solução:



Estrutura:



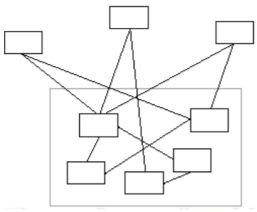
Usar este método quando	Vantagens e Desvantagens
- Queremos adicionar funcionalidades dinamicamente e transparentemente. - Queremos adicionar funcionalidades que podem ser desativados mais tarde. - Extensão por herança é impraticável (não disponível ou produziria uma explosão de subclasses.	- Maior flexibilidade 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:

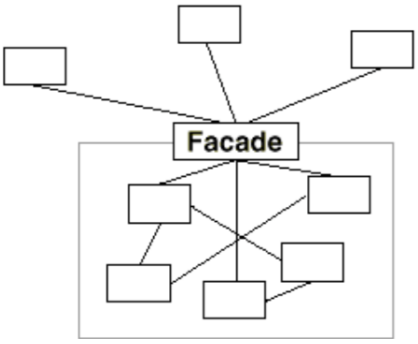
- Fornecer uma interface unificada para um conjunto de interfaces de um subsistema. Façade define uma interface de mais alto nível para tornar o uso dos subsistemas mais fácil.

Problema:

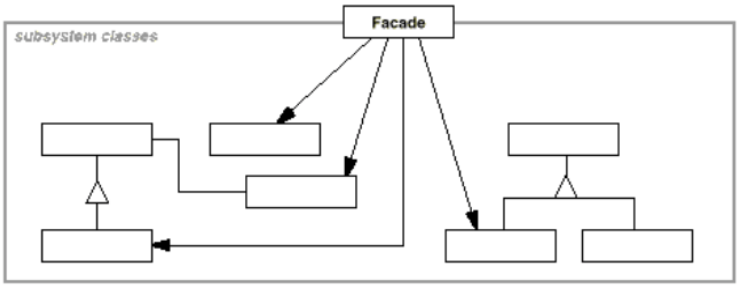


- Clientes acessam vários subsistemas
- Mudanças em algum subsistema demandam alterações em diversos clientes

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.

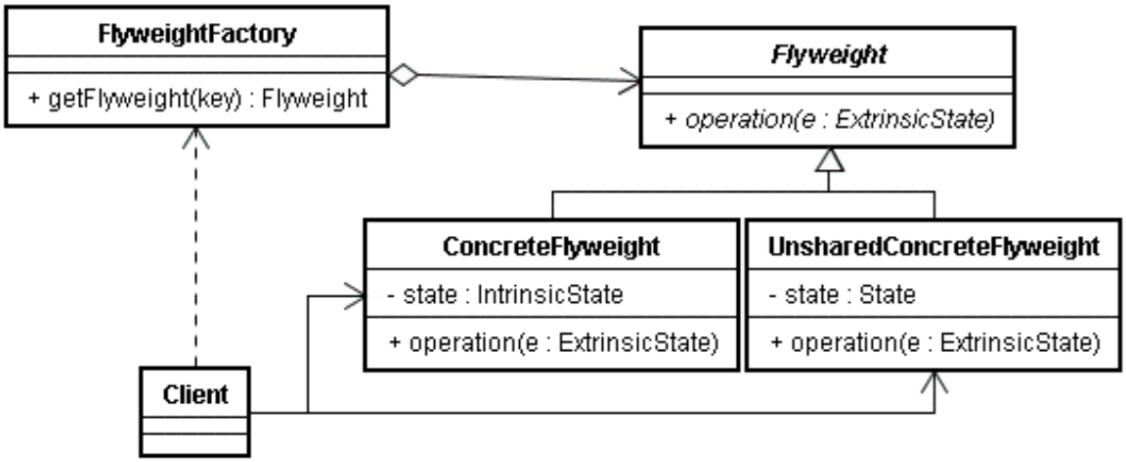
Flyweight

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.

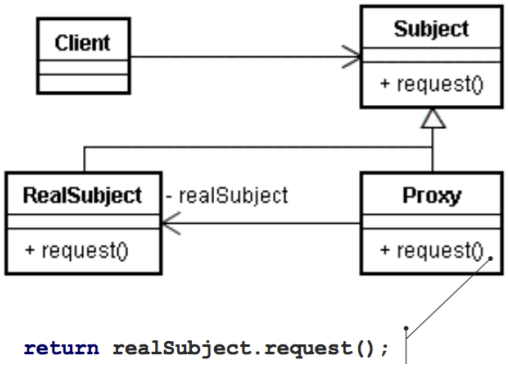
Proxy

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.