Padrões de Projeto

Prof. Maria Augusta Vieira Nelson
PUC Minas

Prof. Torsten Paul Nelson

Bibliografia

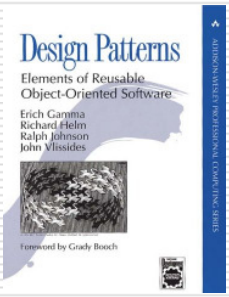
- LARMAN, Graig. Utilizando UML e Padroes:Uma introdução a análise e ao projeto orientados a objetos. Porto Alegre: Bookman, 2a Edição, 2004. capítulo 23
- Gamma; Helm; Johnson; Vlissides. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Bookman, 2000

Nomenclatura

- O termo "*pattern*" tem várias traduções:
  - padrão
    - mas "padrão" também é a tradução de "standard"
  - molde
    - bem mais apropriado, mas não usado na literatura
- O termo "design" também tem várias traduções
  - projeto
    - mas "projeto" também é tradução de "project"
  - desenho
    - feio mas usado.

Padrões de projeto

- O livro "Design Patterns" (1995) aplicou os conceitos introduzidos por Alexander na Arquitetura civil ao projeto de software
  - mais especificamente, projeto orientado a objetos
- Um dos mais importantes avanços da engenharia de software da década de 90.



4

O que é um padrão de projeto?

- Há muitas formas de ver:
  - Uma solução para um pequeno problema de design OO
  - Um conjunto de relacionamentos entre classes que se considera ter boas qualidades
  - Descrições nomeadas de boas heurísticas de design OO
  - Um idioma para se falar sobre designs OO
  - Soluções para deficiências do paradigma OO, sem inventar novas abstrações

5

DANGER!

- Padrões de projeto tornam o código mais flexível
  - nem sempre isso é uma boa coisa
  - evite tornar flexível o que não precisa ser flexível hoje.
- Padrões de projeto tornam o código menos legível
  - Muitas partes do código escrito não são relacionadas com o problema sendo resolvido
  - O número de classes aumenta
  - Os relacionamentos entre classes se tornam mais complexos
- Não é pra ser "programação orientada a padrões"
  - o *aprendiz* usa padrões em tudo para aprender como usá-los;
  - o *graduado* usa padrões apenas onde são necessários;
  - o *mestre* não usa padrões; os padrões surgem naturalmente em seu código.

6

Três categorias

- Padrões de criação
  - maneiras comuns de se instanciar objetos
- Padrões estruturais
  - maneiras comuns de se organizar classes e interfaces
- Padrões comportamentais
  - maneiras comuns nas quais objetos interagem

7

Padrões de criação

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

8

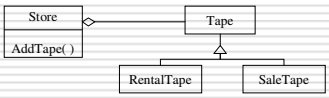
Padrões de criação

- Abstraem o processo de instanciação
- Ajudam a tornar o sistema independente de como objetos são criados
- Encapsulam conhecimento sobre quais classes concretas o sistema usa
- Esconde como instâncias dessas classes são criadas

9

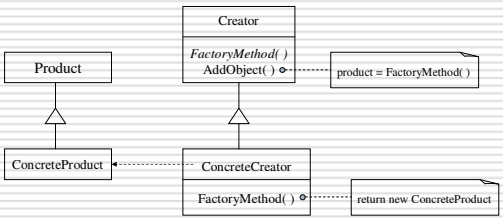
Factory Method

- Intenção
  - Definir uma interface para criar um objeto, mas deixar que as subclasses decidam qual classe deve ser instanciada. Permitir que uma classe delegue a instanciação às suas subclasses.
- Motivação
  - Suponha que você esteja construindo uma aplicação para uma locadora de vídeo que também vende fitas. Como acrescentar uma nova fita à locadora?



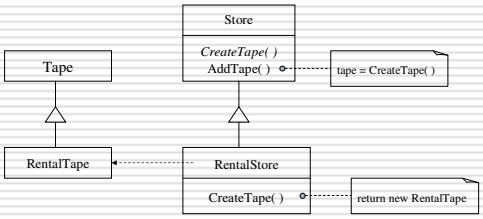
10

Factory Method - Estrutura



11

Factory Method - Exemplo



12

Factory Method - Aplicabilidade

- Use o padrão Factory Method para
  - criar instâncias quando você não sabe de antemão a qual classe as instâncias vão pertencer
  - permitir que uma classe deixe suas subclasses decidirem quais objetos vão criar

13

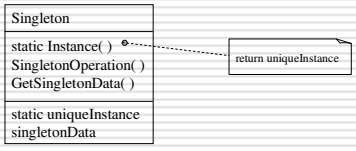
Factory Method - Consequências

- Provê "ganchos" para subclasses
  - Mais flexível do que criar objetos diretamente
- Conecta hierarquias de classes paralelas
  - por exemplo, "RentalStore" (locadora) a "RentalTape" e "SaleStore" (vendedora) a "SaleTape"

14

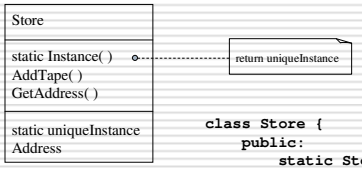
Singleton

- Intenção
  - Garantir que uma classe tenha apenas uma instância
- Motivação
  - Muitas classes devem ter apenas uma instância – escalonadores, window managers, objetos top-level (como a locadora de vídeo) e assim por diante. Deve ser possível garantir essa restrição.



15

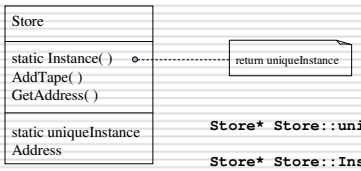
Singleton - Exemplo




```
class Store {
    public:
        static Store* Instance( );
    protected:
        Store();
    private:
        static Store* uniqueInstance;
}
```

16

Singleton - Exemplo




```
Store* Store::uniqueInstance = 0;

Store* Store::Instance( ) {
    if (uniqueInstance == 0) {
        uniqueInstance = new Store;
    }
    return uniqueInstance;
}
```

17

Singleton - Aplicabilidade

- Use o padrão Singleton para
  - garantir que há no máximo uma instância de uma classe
  - permitir que clientes tenham fácil acesso a essa instância
  - permitir que a instância única seja extensível através de subclasses.

18

Singleton – Consequências

- Acesso controlado à única instância
- Espaço de nomes reduzido
  - melhor do que variáveis globais que guardam instâncias únicas
- Pode ter subclasses
- Permite um número variável de instâncias
- Mais flexível do que operações de classe
  - ou seja, métodos *static*
  - operações de classe não podem ser usadas polimorficamente

19

Padrões estruturais

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

20

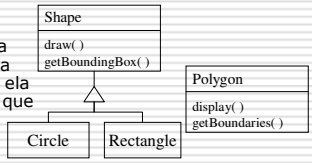
Padrões estruturais

- Lidam com o modo como classes e objetos são compostos para formar estruturas maiores
- Descrevem formas de se compor objetos para obter funcionalidade nova

21

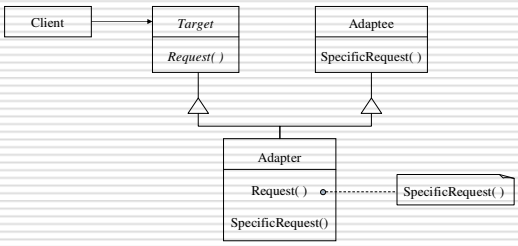
Adapter

- Intenção
  - Converter a interface de uma classe em outra interface que os clientes esperam.
  - Permitir que classes trabalhem juntas quando não poderiam normalmente devido a interfaces incompatíveis.
- Motivação
  - Suponha que você queira reutilizar uma classe como parte da sua hierarquia, mas ela não tem a interface que você precisa.



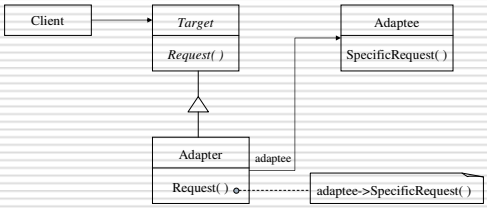
22

Adapter (nível de classes) - Estrutura



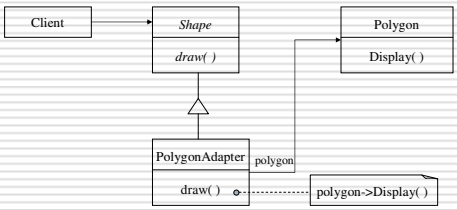
23

Adapter - Estrutura



24

Adapter - Exemplo



25

Adapter - Aplicabilidade

- Use o padrão adapter para
  - usar uma classe existente que não tem uma interface que corresponde às suas necessidades

26

Adapter - Consequências

- Adapters (classes)
  - faz a adaptação usando uma classe adaptadora concreta
    - não funciona se precisarmos adaptar uma classe e suas subclasses
  - permite que o adaptador altere o comportamento do adaptado
  - introduz apenas um objeto
  - não requer o uso de pointers (C++)

27

Adapter - Consequências

- Adapter
  - permite que um único Adapter funcione com muitos "adaptados"
    - o adaptado e todas suas subclasses
    - pode acrescentar funcionalidade a todos os adaptados de uma vez

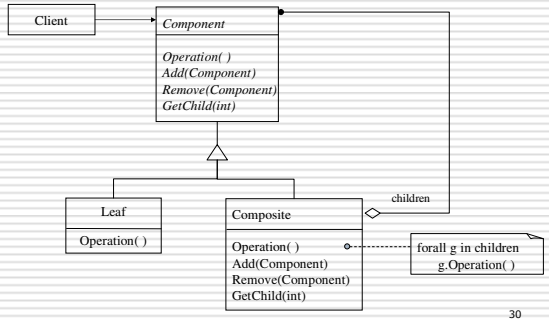
28

Composite

- Intenção
  - Compor objetos em estruturas de árvore para representar hierarquias parte-todo. Composite permite aos clientes tratarem objetos individuas e compostos de maneira uniforme.
- Motivação
  - Suponha que você tenha um editor gráfico que permita que você agrupe figuras para compor figuras mais complexas que podem ser manipuladas como figuras primitivas.

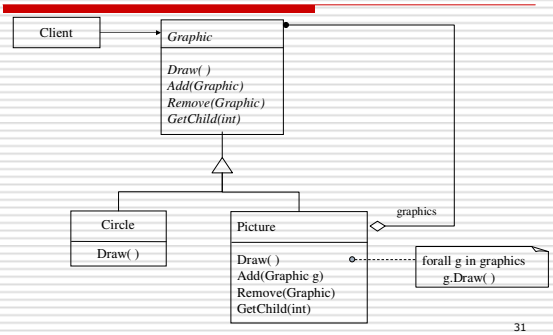
29

Composite - Estrutura



30

Composite - Exemplo



31

Composite - Aplicabilidade

- Use o padrão Composite para
  - representar hierarquias parte-todo recursivamente
  - permitir aos clientes que ignorem a diferença entre composições de objetos e objetos individuais

32

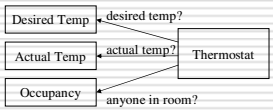
Composite - Consequências

- Define hierarquias de classes que consistem de objetos primitivos e objetos compostos
  - sempre que um cliente esperar um objeto primitivo, pode receber um composto
- Simplifica os clientes
  - os cliente não sabem se estão lidando com objetos simples ou compostos.
- Facilita a adição de novos tipos de componentes
- Pode tornar o projeto geral demais
  - é difícil restringir os componentes de um composite.

33

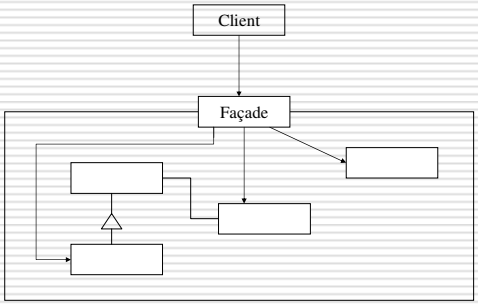
Façade

- Intenção
  - Provê uma interface única para um conjunto de interfaces em um subsistema. Uma façade define uma interface de alto nível que torna o subsistema mais fácil de usar.
- Motivação
  - Suponha que você tenha um termostato que regula um ar condicionado baseado em vários sensores e controladores.



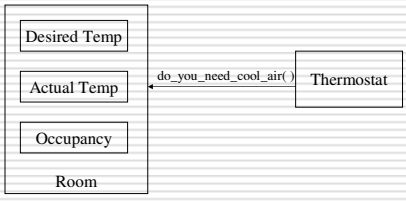
34

Façade - Estrutura



35

Façade - Exemplo



36

Façade - Aplicabilidade

- Use o padrão Façade para
  - fornecer uma interface simples para um subsistema complexo
  - desacoplar um subsistema dos clientes e de outros subsistemas
  - definir um ponto de entrada para cada nível em uma arquitetura de camadas

37

Façade - Consequências

- Isola clientes de componentes do subsistema
  - reduz o número de objetos com os quais o cliente precisa se comunicar
  - torna o subsistema mais simples de se usar
- Promove baixo acoplamento entre o subsistema e seus clientes
  - é possível mudar os componentes do subsistema sem afetar os clientes
- Não impede que as aplicações acessem os componentes se for necessário
  - pode-se escolher entre facilidade de uso e generalidade

38

Padrões comportamentais

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

39

Padrões comportamentais

- Lidam com algoritmos e a distribuição de responsabilidades entre objetos
- Descrevem os padrões de comunicação entre objetos relacionados

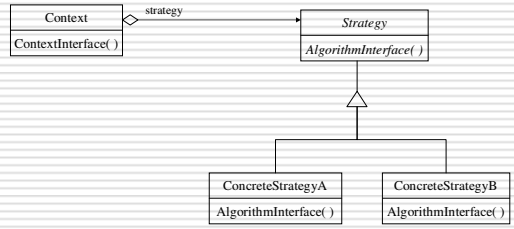
40

Strategy

- Intenção
  - Definir uma família de algoritmos, encapsular cada um, e torná-los intercambiáveis. Esse padrão deixa o algoritmo variar de forma independente dos clientes que o utilizam.
- Motivação
  - Em um buffer síncrono, há muitas formas de se lidar com as situações em que o buffer está vazio ou cheio: pode-se ignorar pedidos, ou bloqueá-los até que possam ser satisfeitos, ou esperar algum tempo antes de desistir.
    - Como você permite que o algoritmo seja escolhido em tempo de execução?

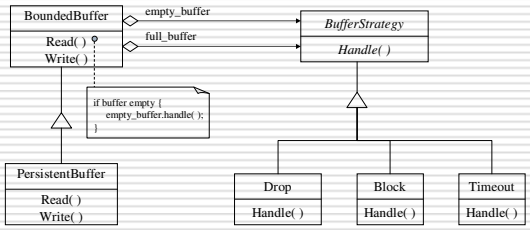
41

Strategy - Estrutura



42

Strategy - Exemplo



43

Strategy - Aplicabilidade

- Use o padrão *Strategy* para
  - configurar uma classe com um de muitos comportamentos possíveis
  - ter diversos algoritmos do mesmo tipo
  - esconder estruturas de dados complexas e específicas a determinados algoritmos
  - evitar o uso de comandos condicionais dentro de classes para selecionar um entre vários comportamentos possíveis

44

Strategy - Consequências

- Famílias de algoritmos relacionados
  - Algoritmos são agrupados em sua própria hierarquia de classes
- Uma alternativa para a herança
  - Evita o problema de explosão de classes
- Elimina comandos condicionais
- Permite escolha de implementações
- Clientes devem estar cientes das estratégias
- Overhead de comunicação
- Maior número de objetos

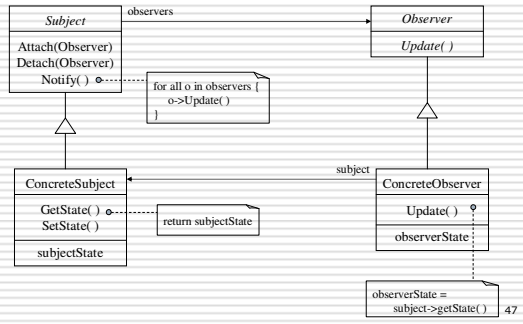
45

Observer

- Intenção
  - Definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, todos seus dependentes são notificados e atualizados automaticamente.
- Motivação
  - Suponha que você tem duas formas de aumentar o volume no seu mp3 player – usando uma escala deslizante, ou digitando uma percentagem do volume máximo. Como você garante que uma é atualizada se a outra for usada?

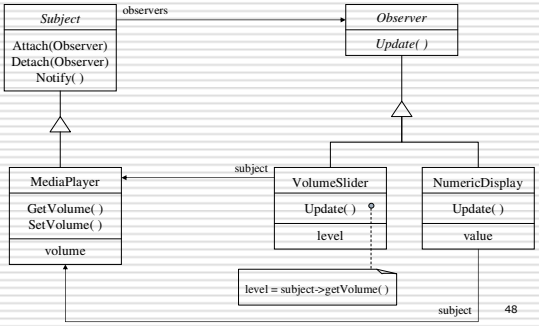
46

Observer - Estrutura



47

Observer - Exemplo



48

Observer - Aplicabilidade

- Use o padrão *Observer* para
  - manter dois ou mais aspectos interdependentes de uma abstração, ao mesmo tempo permitindo que sejam reutilizados independentemente.
  - mudar um objeto causando mudanças em outros objetos, sendo que não se sabe a priori quantos objetos precisam ser mudados.
  - fazer um objeto notificar outros objetos de que seu estado mudou, sem assumir nada sobre os outros objetos.

49

Observer - Consequências

- Acoplamento abstrato entre o sujeito e o observador
  - O sujeito sabe que tem uma lista de observadores, mas não sabe quem são os observadores.
- Suporte para comunicação em broadcast
  - notificações são enviadas a todos os objetos interessados
- Atualizações inesperadas
  - pode ser difícil descobrir exatamente *o que* mudou no sujeito

50