J930

Padrões de Projeto

1 Introdução

Helder da Rocha (helder@acm.org)

argonavis.com.br

O que é um padrão?

- Maneira testada ou documentada de alcançar um objetivo qualquer
  - Padrões são comuns em várias áreas da engenharia
- Design Patterns, ou Padrões de Projeto
  - Padrões para alcançar objetivos na engenharia de software usando classes e métodos em linguagens orientadas a objeto
  - Inspirado em "A Pattern Language" de Christopher Alexander, sobre padrões de arquitetura de cidades, casas e prédios
  - "Design Patterns" de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, conhecidos como "The Gang of Four", ou GoF, descreve 23 padrões de projeto úteis.

2

O que é um padrão?

"Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que pode-se usar essa solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes"

Christopher Alexander, sobre padrões em Arquitetura

"Os padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema genérico de design em um contexto específico"

Gamma, Helm, Vlissides & Johnson, sobre padrões em software

3

Por que aprender padrões?

- Aprender com a experiência dos outros
  - Identificar problemas comuns em engenharia de software e utilizar soluções testadas e bem documentadas
  - Utilizar soluções que têm um nome: facilita a comunicação, compreensão e documentação
- Aprender a programar bem com orientação a objetos
  - Os 23 padrões de projeto "clássicos" utilizam as melhores práticas em OO para atingir os resultados desejados
- Desenvolver software de melhor qualidade
  - Os padrões utilizam eficientemente polimorfismo, herança, modularidade, composição, abstração para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento

4

Por que aprender padrões?

- *Vocabulário comum*
  - *Faz o sistema ficar menos complexo ao permitir que se fale em um nível mais alto de abstração*
- *Ajuda na documentação e na aprendizagem*
  - *Conhecendo os padrões de projeto torna mais fácil a compreensão de sistemas existentes*
  - *"As pessoas que estão aprendendo POO freqüentemente reclamam que os sistemas com os quais trabalham usam herança de forma convoluida e que é difícil de seguir o fluxo de controle. Geralmente a causa disto é que eles não entendem os padrões do sistema" [GoF]*
  - *Aprender os padrões ajudam um novato a agir mais como um especialista*

5

Por que aprender padrões?

- *Uma prática adjunta aos métodos existentes*
  - *Mostram como usar práticas primitivas*
  - *Descrevem mais o porquê do design*
  - *Ajudam a converter um modelo de análise em um modelo de implementação*
- *Um alvo para refatoramento*
  - *Captura as principais estruturas que resultam do refatoramento*
  - *Uso de patterns desde o início pode diminuir a necessidade de refatoramento*

6

Elementos de um padrão

- *Nome*
- *Problema*
  - *Quando aplicar o padrão, em que condições?*
- *Solução*
  - *Descrição abstrata de um problema e como usar os elementos disponíveis (classes e objetos) para solucioná-lo*
- *Conseqüências*
  - *Custos e benefícios de se aplicar o padrão*
  - *Impacto na flexibilidade, extensibilidade, portabilidade e eficiência do sistema*

7

Formas de classificação

- *Há várias formas de classificar os padrões. Gamma et al [1] os classifica de duas formas*
  - *Por propósito: (1) criação de classes e objetos, (2) alteração da estrutura de um programa, (3) controle do seu comportamento*
  - *Por escopo: classe ou objeto*
- *Metsker [2] os classifica em 5 grupos, por intenção (problema a ser solucionado):*
  - *(1) oferecer uma interface,*
  - *(2) atribuir uma responsabilidade,*
  - *(3) realizar a construção de classes ou objetos*
  - *(4) controlar formas de operação*
  - *(5) implementar uma extensão para a aplicação*

8

Classificação dos 23 padrões segundo GoF*

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

* Padrões "clássicos" selecionados e organizados por Gamma et al. "Design Patterns" [1]

9

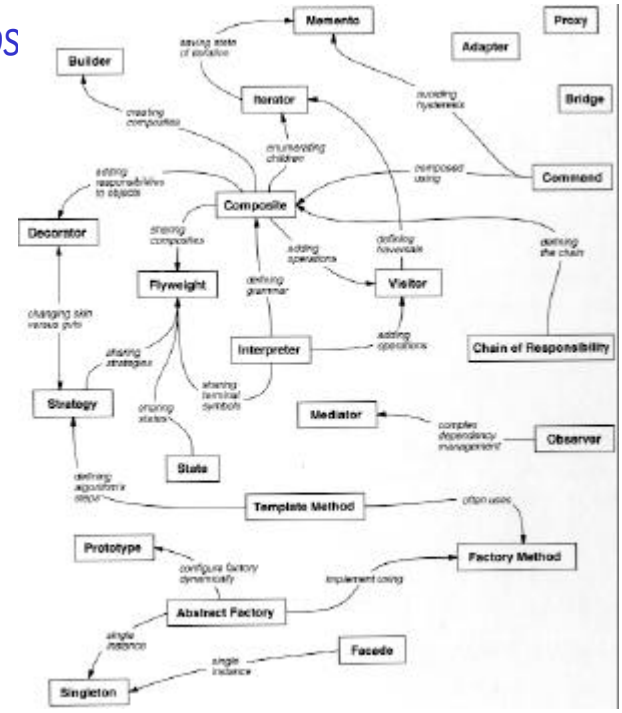
Classificação dos padrões GoF segundo Metsker [2]

Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento
4. Operações	Template Method, State, Strategy, Command, Interpreter
5. Extensões	Decorator, Iterator, Visitor

- Neste curso usaremos esta classificação

10

Relacionamentos entre os 23 padrões "clássicos"



Fonte: [1]

Finalidade dos 23 padrões: ***Interface***

- *1. Adapter*
  - *Converter a interface de uma classe em outra interface esperada pelos clientes.*
- *2. Façade*
  - *Oferecer uma interface única de nível mais elevado para um conjunto de interfaces de um subsistema*
- *3. Composite*
  - *Permitir o tratamento de objetos individuais e composições desses objetos de maneira uniforme*
- *4. Bridge*
  - *Desacoplar uma abstração de sua implementação para que os dois possam variar independentemente*

12

Finalidade dos padrões: **Responsabilidades**

- *5. Singleton*
  - *Garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela*
- *6. Observer*
  - *Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, os seus dependentes sejam notificados e atualizados automaticamente*
- *7. Mediator*
  - *Definir um objeto que encapsula a forma como um conjunto de objetos interagem*

13

Finalidade dos padrões: **Responsabilidades**

- *8. Proxy*
  - *Prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro*
- *9. Chain of Responsibility*
  - *Compor objetos em cascata para, através dela, delegar uma requisição até que um objeto a sirva*
- *10. Flyweight*
  - *Usar compartilhamento para suportar eficientemente grandes quantidades de objetos complexos*

14

Finalidade dos 23 padrões: **Construção**

- *11. Builder*
  - *Separar a construção de objeto complexo da representação para criar representações diferentes com mesmo processo*
- *12. Factory Method*
  - *Definir uma interface para criar um objeto mas deixar que subclasses decidam que classe instanciar*
- *13. Abstract Factory*
  - *Prover interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas*
- *14. Prototype*
  - *Especificar tipos a criar usando uma instância como protótipo e criar novos objetos ao copiar este protótipo*
- *15. Memento*
  - *Externalizar o estado interno de um objeto para que o objeto possa ter esse estado restaurado posteriormente*

15

Finalidade dos 23 padrões: **Operações**

- *16. Template Method*
  - *Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses*
- *17. State*
  - *Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar*
- *18. Strategy*
  - *Definir uma família de algoritmos, encapsular cada um, e fazê-los intercambiáveis*
- *19. Command*
  - *Encapsular requisição como objeto, para clientes parametrizarem diferentes requisições, filas, e suportar operações reversíveis*
- *20. Interpreter*
  - *Dada uma linguagem, definir uma representação para sua gramática junto com um interpretador*

16

Finalidade dos 23 padrões: **Extensão**

- *21. Decorator*
  - *Anexar responsabilidades adicionais a um objeto dinamicamente*
- *22. Iterator*
  - *Prover uma maneira de acessar elementos de um objeto agregado seqüencialmente sem expor sua representação interna*
- *23. Visitor*
  - *Representar uma operação a ser realizada sobre os elementos de uma estrutura de objetos*

17

Como os padrões solucionam problemas

- *Problema 1: quais os objetos mais apropriados?*
  - *A tarefa de decompor um sistema em objetos não é trivial*
  - *É preciso levar em conta fatores como encapsulamento, granularidade, dependência, flexibilidade, performance, reuso, etc.*
  - *Muitos objetos são descobertos na fase de análise, mas muitos não têm paralelo no mundo real*
- *Design patterns ajudam a identificar as abstrações menos óbvias e objetos que podem representá-las*
  - *Exemplo: objetos que representam um algoritmo ou um estado (raramente aparecem na fase de análise)*

18

Como os padrões solucionam problemas

- *Problema 2: qual a granularidade ideal?*
  - *Objetos podem representar qualquer coisa*
  - *Um objeto pode representar todos os detalhes até o hardware ou ser a aplicação inteira*
- *Design patterns oferecem várias soluções*
  - *Façade descreve como representar subsistemas inteiros como um único objeto*
  - *Flyweight descreve como suportar grandes quantidades de objetos nas menores granularidades*
  - *Abstract Factory, Builder, Visitor e Command limitam a responsabilidade de objetos*

19

Como os padrões solucionam problemas

- *Problema 3: como especificar interfaces?*
  - *Uma interface é o conjunto de todas as assinaturas* definidas pelas operações de um objeto*
  - *Objetos são conhecidos apenas através de suas interfaces em sistemas orientados a objetos*
  - *A interface de um objeto nada diz sobre sua implementação, que pode ser determinada em tempo de execução*
- *Design patterns ajudam a definir interfaces ao identificar seus elementos-chave e tipos de dados que são passados*
  - *Podem restringir o que se pode colocar em uma interface*
  - *Podem especificar relacionamentos entre interfaces*
  - *Podem estabelecer regras para criação de interfaces*

** Nome da operação, objetos que recebe como parâmetros e valor de retorno*

20

Como os padrões solucionam problemas

- *Problema 4: como especificar implementações*
  - *Objetos só devem ser manipulados em termos de uma interface definida por classes abstratas (ou interfaces Java)*
  - *Clientes não devem conhecer os tipos concretos dos objetos, nem das classes que implementam esses objetos. Só devem conhecer as classes abstratas que definem a interface*
  - *Princípio de design reutilizável: programe para uma interface, nunca para uma implementação*
- *Design patterns oferecem formas de instanciar classes concretas em outras partes do sistema*
  - *Padrões de construção abstraem o processo de criação de objetos oferecendo um meio transparente para associar uma interface com uma implementação.*

21

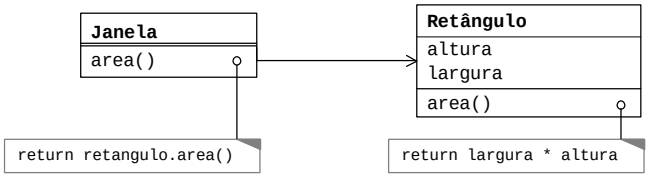
Como os padrões solucionam problemas

- *Problema 5: Como fazer o reuso funcionar*
  - *Deve-se usar herança de classes com cautela. Há quebra de encapsulamento na herança porque ela expõe a subclasse a detalhes da implementação da superclasse*
  - *Dando preferência à composição de objetos sobre herança de classes ajuda a manter o encapsulamento e o foco de cada classe em uma única tarefa*
  - *Princípio de design reutilizável: dê preferência à composição de objetos sobre herança de classe*
- *Design patterns usam delegação para tornar a composição tão poderosa para reuso quando a herança*
  - *Dois objetos estão envolvidos para tratar uma requisição: o objeto que recebe a requisição passa uma referência de si mesmo para o objeto delegado*

22

Delegação

- *Exemplo: Janela tem um retângulo (em vez de ser um)*



- *Vantagem: facilita a composição de comportamentos em tempo de execução*
- *Desvantagem: possível performance menor; código mais difícil de acompanhar.*
  - *Delegação é uma boa escolha de design somente quando ela simplifica mais que complica!*
  - *Funciona melhor quando usada de forma padrão: Patterns!*

23

Como os padrões solucionam problemas

- *Problema 6: como distinguir estruturas estáticas (compile-time) e dinâmicas (run-time)*
  - *A estrutura em tempo de execução de um programa orientado a objetos mantém pouca semelhança com sua estrutura de código: código não revela como sistema funciona!*
  - *Estrutura estática: hierarquias fixas e imutáveis*
  - *Estrutura dinâmica: redes mutáveis de objetos interagindo*
  - *Exemplo: agregação e associação são implementadas da mesma forma (em código) mas se mostram muito diferentes em tempo de execução*
- *Vários design patterns capturam a distinção entre estruturas run-time e compile-time*
  - *As estruturas não são óbvias pelo código. É preciso entender os padrões!*

24

Problema 7: como antecipar mudanças?

- Os padrões viabilizam o desenvolvimento de código mais robusto diante de possíveis mudanças e refatoramento do código
- Padrões promovem desacoplamento e permitem que algum aspecto da estrutura do sistema varie independentemente de outros aspectos
  - Evita redesign e readaptação de código nas situações previstas pelo padrão aplicado
  - Reduz risco e possíveis custos futuros
  - Na maior parte dos casos, o investimento não implica em altos custos (risco) no presente, já que contribuem para a legibilidade e qualidade do código.

25

Oito causas comuns de redesign e padrões que os evitam [1]

1. Criação de objeto especifica classe explicitamente
   - O sistema está preso a uma implementação específica
   - Solução: criar objetos indiretamente com Abstract Factory, Factory Method ou Prototype
2. Dependência em operações específicas
   - O sistema só tem uma forma de satisfazer uma requisição
   - Solução: evitar ações "hard-coded" com Chain of Responsibility ou Command
3. Dependência em plataforma de H/W ou S/W
   - O software precisa ser portado a outras plataformas
   - Solução: limitar dependências com Abstract Factory ou Bridge
4. Dependência em representações ou implementações de objetos
   - Clientes que sabem como um objeto é implementado, representado ou armazenado podem ter que ser alterados se o objeto mudar
   - Solução: isolar cliente com Abstract Factory, Bridge, Memento ou Proxy

[1] *Pags. 24 e 25*

26

Oito causas comuns de redesign e padrões que os evitam [1]

5. Dependências de algoritmo
   - Mudanças de algoritmo são freqüentes. Objetos que dependem de um algoritmo precisam mudar quando o algoritmo mudar.
   - Solução: isolá-los com Builder, Iterator, Strategy, Template Method ou Visitor
6. Forte acoplamento
   - Classes fortemente acopladas são difíceis de reusar, testar, manter, etc.
   - Solução: enfraquecer o acoplamento com Abstract Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator ou Observer
7. Extensão de funcionalidade através de subclasses
   - Herança é difícil de usar; composição dificulta compreensão.
   - Solução: usar padrões que implementam bem herança e composição como Bridge, Chain of Responsibility, Composite, Decorator, Observer ou Strategy
8. Incapacidade de alterar classes convenientemente
   - Classes inacessíveis, incompreensíveis ou difíceis de alterar
   - Solução: usar Adapter, Decorator ou Visitor

27

Tipos de software

- Aplicações
  - Prioridades: reuso interno, manutenção, extensão
- Toolkits, APIs, bibliotecas
  - Conjunto de classes reutilizáveis de propósito geral. Não impõem design
  - Prioridade: amplo reuso de código
- Frameworks
  - Dita a arquitetura da aplicação. Requer que usuário aprenda o framework e inclua código e configuração.
  - Prioridade: amplo reuso de design
  - Geralmente são fortemente baseados em padrões. Quem conheçe os padrões entende o framework mais facilmente.

28

Aspectos de design que padrões permitem variar

Design Pattern	Aspecto(s) que pode(m) variar
Abstract Factory	famílias de objetos de produtos
Builder	como um objeto composto é criado
Factory Method	subclasse de objeto que é instanciado
Prototype	classe de objeto que é instanciado
Singleton	a única instância da classe
Adapter	interface para um objeto
Bridge	implementação de um objeto
Composite	estrutura e composição de um objeto
Decorator	responsabilidades de um objeto sem recorrer a subclasses
Façade	interface para um subsistema
Flyweight	custos de armazenamento de objetos
Proxy	como um objeto é acessado; sua localização
Chain of Responsibility	objeto que pode satisfazer uma requisição
Command	quando e como uma requisição é satisfeita
Interpreter	gramática e interpretação de uma linguagem
Iterator	como os elementos de um agregado são acessados
Mediator	como e quais objetos interagem uns com os outros
Memento	que informação privativa é armazenada fora de um objeto, e quando
Observer	número de objetos que dependem de outro objeto e como eles se mantém em dia
State	estados de um objeto
Strategy	um algoritmo
Template Method	passos de um algoritmo
Visitor	operações que podem ser aplicadas a objetos sem mudar suas classes

Fonte: [1] *Pag. 30*

29

Como selecionar um padrão?

1. *Considere como os padrões solucionam os problemas de projeto*
2. *Analise seu problema e compare com o objetivo de cada padrão*
3. *Veja como os padrões envolvidos se relacionam entre si*
4. *Estude padrões de propósito ou intenção similar (veja formas de classificação)*
5. *Examine causas comuns que podem forçar o redesign do seu sistema*
6. *Considere o que deve variar no seu design*

30

Fontes

[1] *Steven John Metsker, Design Patterns Java Workbook. Addison-Wesley, 2002*

[2] *Erich Gamma et al. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995. Parte I foi principal referência para este capítulo.*

31

Curso J930: Design Patterns

Versão 1.1

www.argonavis.com.br

(helder@acm.org)