

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/341327459>

Project Standards (GoF)

Article · January 2017

CITATIONS

0

READS

22

1 author:



[Alvaro F. Pinheiro](#)

University of Pernambuco

29 PUBLICATIONS 5 CITATIONS

SEE PROFILE

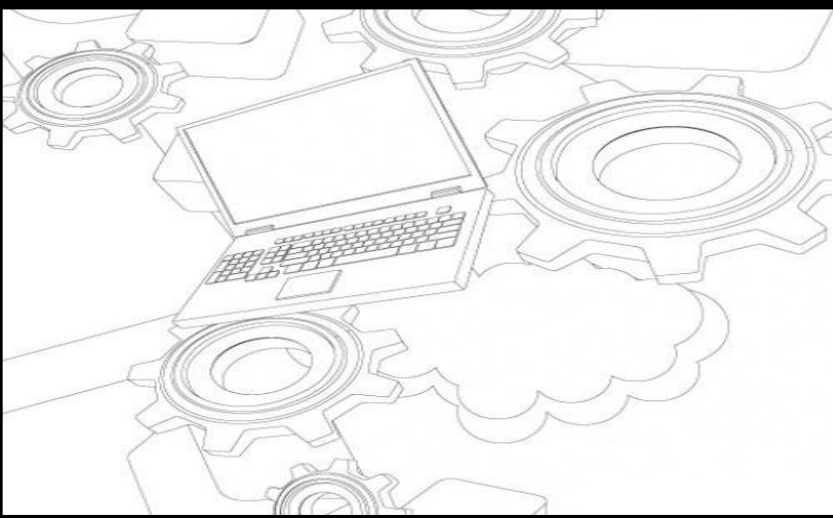
Álvaro Farias Pinheiro

Revista Fundamentos da Engenharia de Software

Padrões de Projetos

GoF

Ano II – Número 2
Fevereiro de 2017



Série Fundamentos da Engenharia de Software

Padrões de Projetos

PINHEIRO, Álvaro Farias
Autor

Série Fundamentos da Engenharia de Software

Padrões de Projetos

Publicação 2017

O autor acredita que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, por ventura, mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação. Eventuais erratas estarão disponíveis para download no site de publicação.

As imagens utilizadas neste livro foram obtidas na Internet.

Dados da Publicação

Pinheiro, Álvaro Farias

Série Fundamentos da Engenharia de Software: Padrões de Projetos

Ano II – Número 2 – Recife, Fevereiro de 2017

Selo Editorial: Publicação Independente

1. GoF Criacional
2. GoF Estrutural
3. GoF Comportamental

Série Fundamentos da Engenharia de Software

Padrões de Projetos

Publicação Independente
Revista em português com o título
Padrões de Projetos
Série Fundamentos da Engenharia de Software
Ano II – Número 2
Recife – Pernambuco – Brasil
Fevereiro de 2017

Introdução

Um padrão é a solução para um determinado problema em um determinado contexto. Um padrão codifica conhecimento específico obtido em uma experiência em um determinado domínio. Um sistema bem estruturado estará cheio de padrões: de linguagem; de projeto; e de arquitetura. Segundo Fowler, podem ser utilizados para melhorar o entendimento ou comunicação dos problemas e decisões arquiteturais. Podem ser vistos como uma tentativa de criar um vocabulário comum para comunicação.

Um padrão que se deve ter conhecimento na orientação a objetos é o GRASP que significa *General Responsibility Assignment Software Patterns* e que descreve os princípios fundamentais do design orientado a objetos e a atribuição de responsabilidades. Outro padrão é o de Gamma, *et al* que descreve vinte e três padrões clássicos na orientação a objetos. O padrão de Gamma é mais conhecido como padrão da gangue dos quatro (*Gang of Four patterns*, ou apenas GoF).

Segue um exemplo de *template* dos Padrões de Projetos:

- Nome - o nome que serve como referencia para o padrão;
- Problema - explica o problema que ocorre em um contexto, com sintomas e em condições;
- Solução - elementos que constituem o design, seus relacionamentos, responsabilidades e colaborações;
- Consequências - resultados e compromissos decorrentes da aplicação do padrão. Impactos sobre a flexibilidade, extensibilidade, portabilidade ou desempenho do sistema. Fundamentam a escolha do padrão mais apropriado;
- Nome e Classificação - identificam unicamente o padrão e o classifica para catalogação;
- Intenção - uma breve descrição que deve responder o que o padrão faz qual sua intenção e que problema ele trata;
- Também Conhecido Como - outros nomes pelo qual o padrão é conhecido;
- Motivação - um cenário que ilustra o problema e como as classes deste padrão o solucionam;
- Aplicabilidade - em que situações o padrão pode ser aplicado;
- Estrutura - representação gráfica do padrão com suas classes e colaborações;
- Participantes - classes e objetos que participam no padrão, incluindo suas responsabilidades;
- Colaborações - como os participantes colaboram entre si;
- Consequências - como o padrão atende a seus objetivos e que “efeitos colaterais” seu uso pode provocar;
- Implementação - dicas, técnicas e erros comuns ao implementar este padrão;
- Exemplo de Código - fragmentos de código ilustrando o padrão;
- Usos Conhecidos - exemplos de uso do padrão em sistemas reais; e
- Padrões Relacionados - padrões que estão fortemente relacionados a este, incluindo suas diferenças, ou utilizados por este.

1.1 Categorias dos Padrões de Projetos

Existem basicamente três categorias, as criacionais, estruturais e comportamentais. Padrões Criacionais: auxiliam na criação de objetos, tornando o programa menos dependente do modo como os objetos são criados e compostos. Assim, estes padrões permitem que se mudem as classes dos objetos criados em tempo de execução com pouco esforço de programação; Padrões Estruturais: Descrevem formas flexíveis para compor classes e objetos; e Padrões Comportamentais: Estes padrões são relacionados a algoritmos e responsabilidades associados a cada objeto. Mudando-se os objetos e classes utilizados, pode-se mudar o comportamento do programa. Acoplando-se um objeto a outro, pode-se adicionar comportamento ao segundo objeto.

1.2 Critérios dos Padrões de Projetos

Os padrões de projeto são classificados por dois critérios, o de objetivo e o de escopo. Objetivo reflete as categorias (criação, de estrutura ou de comportamento). Escopo especifica quando o padrão é aplicado (classes ou a objetos).

Padrões com escopo de classe lidam com relacionamentos estabelecidos através de herança, ou seja, estáticos e definidos em tempo de compilação. Padrões com escopo de objeto lidam com relacionamentos de objeto, que podem ser modificados em tempo de execução e são mais dinâmicos. Praticamente todo padrão utiliza herança de alguma forma, por isto apenas os padrões que focam em relacionamentos através de herança devem ser classificados com escopo de classe. Os padrões mais importantes estão em escopo de objeto.

1.3 Gang of Four (GoF)

Gamma *et al* descrevem vinte e três padrões que podem ser utilizados em praticamente qualquer área de programação. Estes padrões se tornaram clássicos da orientação a objetos e são chamados de padrões da gangue dos quatro (*Gang of Four patterns*, ou apenas GoF). Esse padrão se divide em dois critérios: Critério Objetivo que refletem as categorias de criação, estrutura e comportamento, por sua vez esse critério se divide em três categorias. Categoria dos Criacionais usados para determinar como os objetos são criados; Categoria dos Estruturais que descrevem formas flexíveis para compor classes e objetos; e a Categoria dos Comportamentais relacionados aos algoritmos e responsabilidades associados a cada objeto. O Critério Escopo que especifica quando o padrão é aplicado à classe e/ou objeto. Como pode ser visto logo abaixo.

Figura 0.1 Padrões GoF

Criacional	Estrutural	Comportamental
1. <i>Factory Method</i> (Classe)	1. <i>Adapter</i> (Classe/Objeto)	1. <i>Interpreter</i> (Classe)
2. <i>Abstract Factory</i> (Objeto)	2. <i>Bridge</i> (Objeto)	2. <i>Template Method</i> (Classe)
3. <i>Builder</i> (Objeto)	3. <i>Composite</i> (Objeto)	3. <i>Chain Responsibility</i> (Objeto)

4. <i>Prototype</i> (Objeto)	4. <i>Decorator</i> (Objeto)	4. <i>Command</i> (Objeto)
5. <i>Singleton</i> (Objeto)	5. <i>Facade</i> (Objeto)	5. <i>Strategy</i> (Objeto)
	6. <i>Flyweight</i> (Objeto)	6. <i>State</i> (Objeto)
	7. <i>Proxy</i> (Objeto)	7. <i>Observer</i> (Objeto)
		8. <i>Memento</i> (Objeto)
		9. <i>Mediator</i> (Objeto)
		10. <i>Visitor</i> (Objeto)
		11. <i>Iterator</i> (Objeto)

Fonte: Próprio Autor

Descrição Resumida dos Padrões GoF:

- *Abstract Factory*-Permitir que um cliente crie famílias de objetos sem especificar suas classes concretas;
- *Adapter*-Classe adaptadora entre a classe interna e um componente;
- *Bridge*-Desassociar a evolução de duas classes;
- *Builder*-Encapsular a construção de um comportamento e permitir que ele seja construído em etapas;
- *Chain Responsibility*-Define uma forma de passar uma solicitação entre objetos;
- *Command*-Encapsular um pedido de comando em um objeto;
- *Composite*-Criar uma hierarquia parte todo;
- *Decorator*-Estender a funcionalidade dinamicamente;
- *Facade*-Definir uma interface de alto nível;
- *Factory Method*-Subclasses decidem quais classes concretas serão criadas;
- *Flyweight*-Definir ajustes finos em subclasses;
- *Iterator*-Fornecer forma de acessar sequencialmente uma coleção de objetos sem expor a sua implementação;
- *Interpreter*-Permitir a inclusão de elementos de linguagem em um aplicativo;
- *Mediator*-Definir comunicação simplificada entre classes;
- *Memento*-Salvar e restaurar o estado interno de um objeto;
- *Observer*-Definir um regime de notificação de objetos de mudanças para um outro objeto;
- *Prototype*-Permitir criar novas instancias simplesmente copiando instancias existentes;
- *Proxy*-Um classe que controla o acesso para outra classe;
- *Singleton*-Assegurar que somente um objeto de uma determinada classe seja criado em todo o projeto;
- *State*-Alterar o comportamento de um objeto quando seu estado muda;
- *Strategy*-Encapsular um algoritmo dentro de uma classe;
- *Template Method*-Permitir que subclasses redefinam os passos de um algoritmo; e
- *Visitor*-Define uma nova operação em uma classe sem trocá-la.

1.4 Padrão de Projeto GoF Criacional

Padrões criacionais são usados para determinar como os objetos serão criados ou instanciados. Padrões de projeto abstraem o processo de instanciação. Eles ajudam a tornar o sistema independente de como os objetos são criados, compostos e representados. Um padrão criacional de

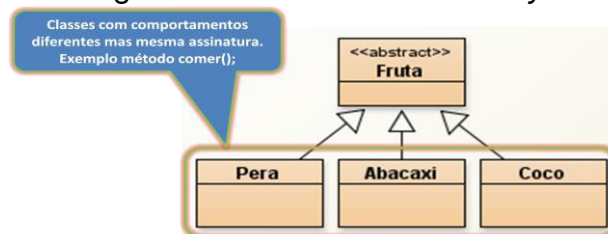
classe utiliza herança para variar a classe que será instanciada. Um padrão criacional de objeto irá delegar a instanciação de um objeto para outro objeto. Padrões criacionais tornam-se importantes à medida que os sistemas evoluem e passam a depender mais de composição de objetos do que de herança de classes. À medida que isto acontece, a ênfase migra da codificação de um conjunto de comportamentos para a codificação de conjuntos menores de comportamento, que podem ser combinados em vários conjuntos mais complexos.

1.4.1 Abstract Factory

Intenção: Provê uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Aplicabilidade: Este padrão deve ser utilizado quando o programa precisa de independência de como seus objetos são criados, compostos e representados. Ou quando o sistema precise ser configurado para uma ou muitas famílias de classes ou objetos. Ou quando uma família de objetos relacionados é projetada para ser utilizada em conjunto e esta premissa deve ser garantida. Ou quando se deseja prover uma biblioteca de classes, e deseja-se disponibilizar apenas as interfaces, e não as implementações.

Figura 1.2 GoF *Abstract Factory*



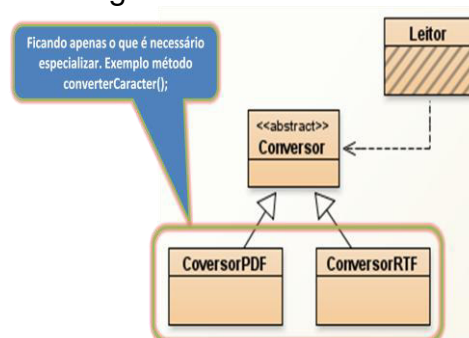
Fonte: Próprio Autor

1.4.2 Builder

Intenção: Separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

Aplicabilidade: O padrão *builder* deve ser utilizado quando o algoritmo para criação de objetos complexos deve ser independente das partes que compõem o objeto e da forma como este objeto é “montado”. Ou quando o processo de construção deve permitir diferentes representações para o objeto que está sendo construído.

Figura 1.3 GoF *Builder*

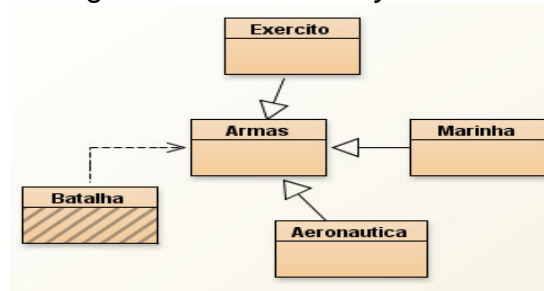


Fonte: Próprio Autor

1.4.3 Factory Method

Intenção: Define uma interface para criação um objeto, mas deixa as subclasses decidirem que classe instanciar. Permite que uma classe delegue a instanciação a suas subclasses.

Aplicabilidade: Este padrão deve ser utilizado quando uma classe não pode antecipar a classe dos objetos que deve criar. Ou quando a classe deseja que suas subclasses especifiquem o objeto que será criado. O quando a classe delega a responsabilidade de criação para um de muitas classes auxiliares, e deseja-se localizar o “conhecimento” de que classe auxiliar deve ser delegada.

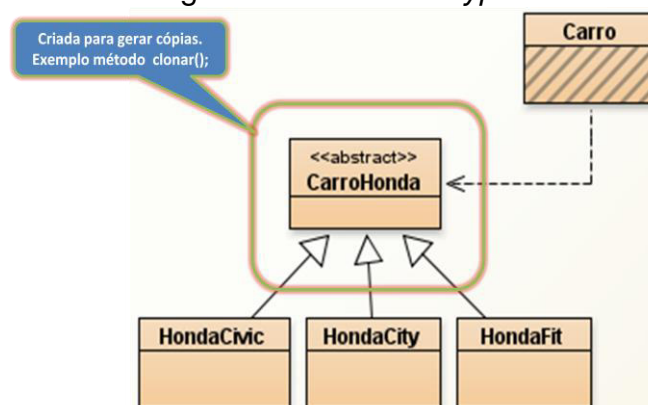
Figura 1.4 GoF *Factory Method*

Fonte: Próprio Autor

1.4.4 Prototype

Intenção: Especifica que tipos de objetos criarem usando uma instância prototípica do objeto. Cria novos objetos através da cópia deste protótipo.

Aplicabilidade: Este padrão deve ser utilizado quando a aplicação deve ser independente de como os produtos são criados, compostos, e representados, e, adicionalmente: As classes a serem instanciadas são definidas em tempo de execução (por exemplo, *dynamic loading*); Deseja-se evitar criar uma hierarquia de fábricas paralelas a hierarquia de classes; Quando a instanciação da classe pode ter um de algumas poucas combinações diferentes de estado. Isto pode ser mais conveniente criar um número correspondente de protótipos e cloná-los ao invés de instanciar a classe manualmente a cada vez com o estado apropriado.

Figura 1.5 GoF *Prototype*

Fonte: Próprio Autor

1.4.5 Singleton

Intenção: Garantir que uma classe possui apenas uma única instância. Provê um ponto de acesso global a ela.

Aplicabilidade: Este padrão deve ser utilizado quando deve haver apenas uma instância de cada classe e esta instância deve ser acessível a todos os clientes a partir de um ponto de acesso conhecido. Ou quando uma única instância deve ser extensível apenas por subclasses, e os clientes devem apenas utilizar a instância estendida, sem modificar seu código.

```
public class Singleton {  
    private static Singleton instancia;  
    private Singleton() {  
    }  
    public static Singleton getInstancia() {  
        if (instancia == null)  
            instancia = new Singleton();  
        return instancia;  
    }  
}
```

1.5 Padrão de Projeto GoF Estrutural

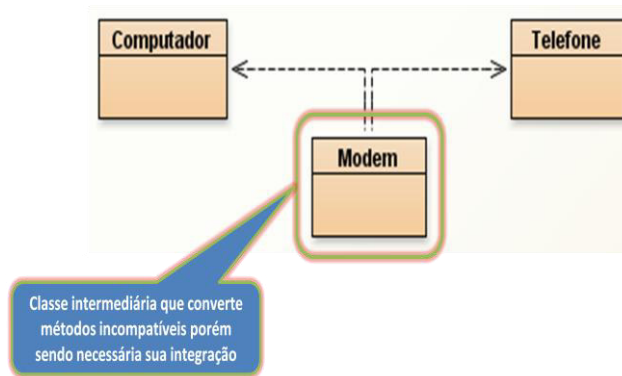
Padrões estruturais que descrevem as formas para relacionar as classes e objetos. Padrões estruturais focam em como criar estruturas de maior porte através da composição de classes e objetos. Padrões estruturais de classe utilizam herança para compor interfaces ou implementações. Padrões estruturais de objeto utilizam composição de objetos para prover novas funcionalidades. A flexibilidade adicional da composição de objetos vem da habilidade de mudar esta composição em tempo de execução, o que é impossível na composição estática de classes.

1.5.1 Adapter

Intenção: Converte a interface de uma classe na interface esperada pelo cliente. Compatibiliza classes de interfaces não compatíveis, permitindo que trabalhem em conjunto.

Aplicabilidade: Este padrão deve ser utilizado quando se deseja utilizar uma classe já existente, e sua interface não atende a interface que você precisa. Quando se deseja criar uma classe reusável que coopera com classes ainda não conhecidas ou não criadas, ou seja, classes que não necessariamente possui interfaces compatíveis. Necessita-se usar diversas subclasses já existentes, mas é impraticável adaptar suas interfaces através da criação de subclasses para cada uma delas. Um objeto adaptador pode adaptar a interface da superclasse.

Figura 1.6 GoF *Adpater*



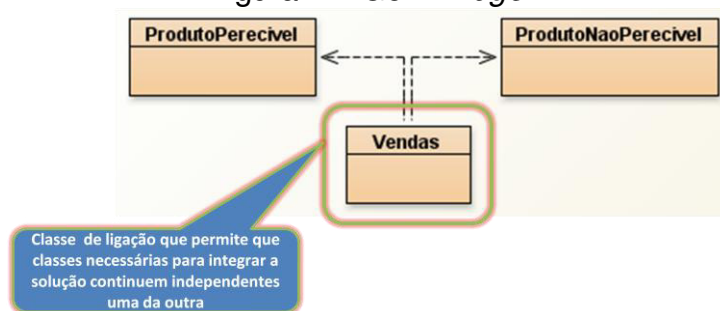
Fonte: Próprio Autor

1.5.2 Bridge

Intenção: Desacopla uma abstração de sua implementação, de modo que as duas possam variar independentemente.

Aplicabilidade: Este padrão pode ser utilizado nas seguintes situações: Deseja-se evitar uma dependência direta entre a abstração e sua implementação. Este pode ser o caso, por exemplo, quando a implementação tiver de ser selecionada ou substituída em tempo de execução; Ambas as abstrações e suas implementações devem ser extensíveis através da criação de subclasses. Neste caso o padrão *bridge* permite que diferentes abstrações e implementações possam ser combinadas e estendidas independentemente. Mudanças na implementação de uma abstração não deve impactar em seus clientes, isto é, seu código não deve ser recompilado.

Figura 1.7 GoF *Bridge*



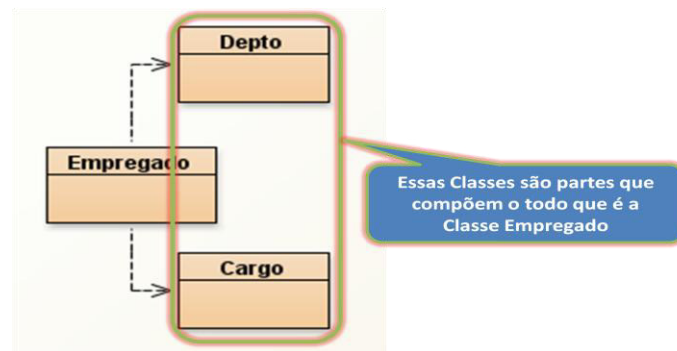
Fonte: Próprio Autor

1.5.3 Composite

Intenção: Compõe objetos em estruturas de árvores para representar hierarquias parte-todo. Permite que clientes tratem de modo uniforme tanto objetos individuais quanto suas composições.

Aplicabilidade: Este padrão deve ser utilizado quando se deseja representar hierarquias parte-todo. Ou quando se deseja que clientes sejam capazes de ignorar as diferenças entre composições dos objetos e os objetos individualmente. Clientes irão tratar uniformemente todos os objetos na estrutura composta.

Figura 1.8 GoF *Composite*



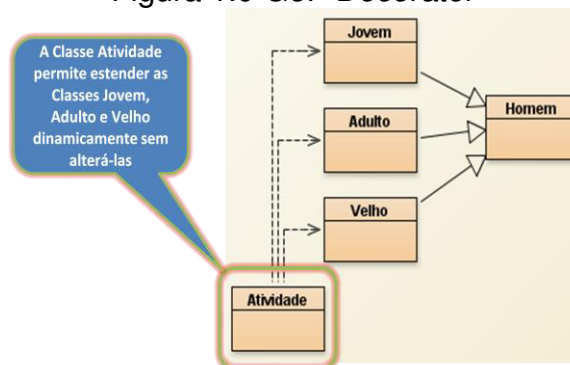
Fonte: Próprio Autor

1.5.4 Decorator

Intenção: Anexa dinamicamente responsabilidade adicional a um objeto. Provê uma alternativa flexível ao uso de herança como mecanismo de extensão de funcionalidade.

Aplicabilidade: Utilize este padrão para adicionar responsabilidades individuais a objetos dinamicamente e transparentemente, isto é, sem afetar outros objetos. Utilize este padrão para responsabilidades que podem ser “removidas”. Ou ainda quando a extensão de funcionalidade através da criação de subclasses é impraticável. Em algumas situações um grande número de extensões independentes é possível e isto poderá produzir uma grande quantidade de subclasses para suportar cada uma das combinações.

Figura 1.9 GoF *Decorator*



Fonte: Próprio Autor

1.5.5 Facade

Intenção: Provê uma interface unificada para o conjunto de interfaces de um subsistema. Define uma interface de mais alto nível que torna um subsistema de mais fácil uso.

Aplicabilidade: Utilize este padrão quando: Deseja-se prover uma interface simples para um subsistema complexo. A fachada pode prover uma visão padrão do subsistema que é boa o suficiente para a maior parte dos clientes. Apenas clientes que necessitem de customização terão que olhar além da fachada. Existem muitas dependências entre clientes e as classes que implementam as abstrações. A introdução da fachada desacopla o subsistema dos clientes e outros subsistemas, promovendo a independência e portabilidade do subsistema. Deseja-se quebrar o sistema em camadas. Utilize a fachada para definir um ponto de entrada para cada um dos subsistemas. Se

os subsistemas são dependentes, podem-se simplificar as dependências entre eles fazendo com que se comuniquem apenas através de suas fachadas.

Figura 1.10 GoF *Facade*



Fonte: Próprio Autor

1.5.6 Flyweight

Intenção: Usa compartilhamento para suportar um grande número de pequenos objetos de forma eficiente.

Aplicabilidade: Este padrão deve ser utilizado quando: Uma aplicação utiliza um grande número de objetos; Armazenamento tem custo elevado devido à grande quantidade de objetos; Muitos grupos de objeto podem ser substituídos por relativamente poucos objetos compartilhados. A aplicação não depende da identidade do objeto. Uma vez que os objetos podem ser compartilhados, testes de identidade irão retornar *true* para objetos conceitualmente distintos.

Figura 1.11 GoF *Flyweight*



Fonte: Próprio Autor

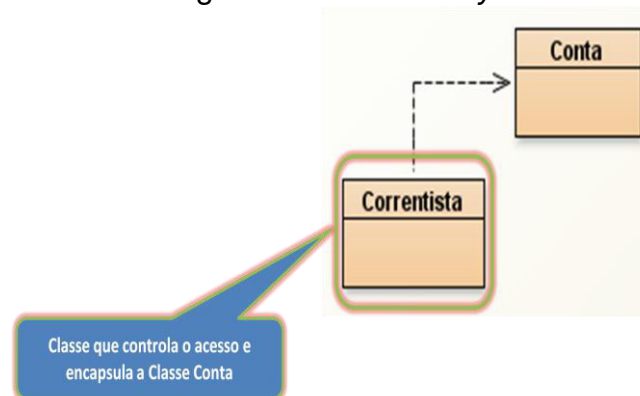
1.5.7 Proxy

Intenção: Provê um ponto de atendimento para que outro objeto possa controlar o acesso ao primeiro.

Aplicabilidade: Proxy é aplicável sempre que existe a necessidade de referências mais versáteis ou sofisticadas que um ponteiro para um objeto. Exemplos comuns são: Proxy remoto provendo um representante local para um objeto em um espaço de memória diferente; Proxy virtual criando objetos custosos sobre demanda; Proxy de proteção controlando o acesso ao objeto

original; Referência inteligente que executa ações adicionais quando o objeto original é acessado.

Figura 1.12 GoF *Proxy*



Fonte: Próprio Autor

1.6 Padrão de Projeto GoF Comportamental

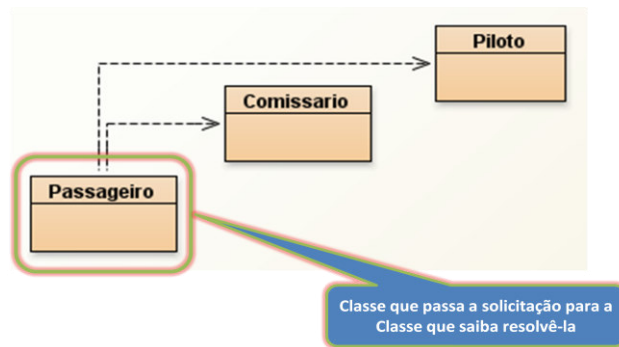
Padrões comportamentais são relacionados aos algoritmos e responsabilidades associados a cada objeto. Padrões comportamentais estão relacionados com algoritmos e atribuição de responsabilidades entre objetos. Estes padrões não descrevem apenas os padrões de classes e objetos, mas também os padrões de comunicação entre estas classes e objetos. Caracterizam complexos fluxos de controle, difíceis de acompanhar em tempo de execução. Eles mudam o foco do fluxo de controle e permite que se concentre apenas na forma como os objetos estão interconectados. Padrões comportamentais de classes utilizam herança para distribuir o comportamento entre classes, que inclui os padrões *Template Method* – o mais simples deles, e o *Interpreter*. Padrões comportamentais de objetos utilizam composição de objetos, ao invés de herança, para realização de tarefas que um único objeto não poderia realizar. Estes objetos podem manter referências explícitas entre si, porém isto trás um maior acoplamento. Outros padrões permitem um menor nível de acoplamento, como o *Memento*, *Chain of Responsibility* e *Observer*.

1.6.1 Chain of Responsibility

Intenção: Evita acoplamento entre solicitantes e atendentes permitindo que mais de um objeto tenha chance de tratar a solicitação. Encadeia os atendentes e passa a solicitação através desta cadeia permitindo que todos eles a tratem.

Aplicabilidade: Este padrão deve ser usado quando: mais de um objeto pode tratar uma solicitação e o objeto que a tratará não é conhecido a priori. O objeto que trata a solicitação deve ser escolhido automaticamente; deve-se emitir uma solicitação para um dentre vários objetos, sem especificar explicitamente o receptor; o conjunto de objetos que pode tratar uma solicitação deveria ser especificado dinamicamente.

Figura 1.13 GoF *Chain Responsibility*



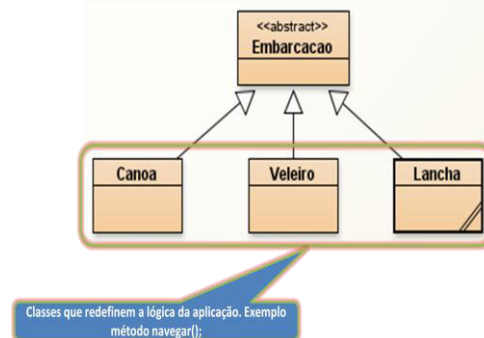
Fonte: Próprio Autor

1.6.2 Template Method

Intenção: Permite subclasses redefinir os passos de um algoritmo.

Aplicabilidade: Este padrão deve ser utilizado: Para implementar partes invariantes de um algoritmo uma única vez e deixar que as subclasses implementem o comportamento que varia. Quando o comportamento padrão entre subclasses devam ser fatorados e localizados em uma classe comum para evitar duplicação de código; Para controlar extensões por subclasses. Pode-se definir um *template method* que chama operações gancho em pontos específicos, permitindo extensões apenas nestes pontos.

Figura 1.14 GoF *Template Method*



Fonte: Próprio Autor

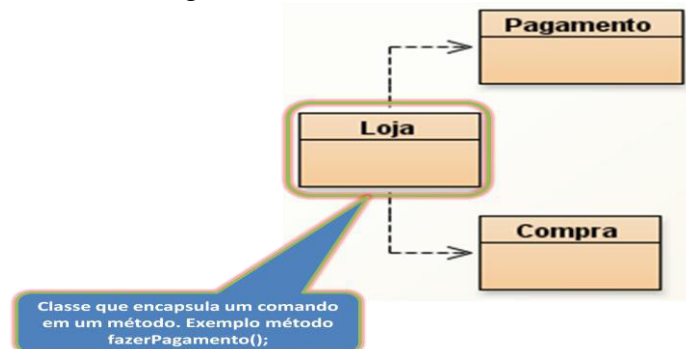
1.6.3 Command

Intenção: Encapsula uma solicitação em um objeto, permitindo que se parametrize clientes com diferentes solicitações, filas ou registros de solicitações, suportando ainda que operações sejam desfeitas.

Aplicabilidade: Utilize este padrão para: Parametrizar objetos por uma ação a ser executada. Você pode expressar tal parametrização numa linguagem procedural através de uma função callback, ou seja, uma função que é registrada em algum lugar para ser chamada em um momento mais adiante. Os *Commands* são uma substituição orientada a objetos para callbacks; Especificar, enfileirar e executar solicitações em tempos diferentes. Um objeto *Command* pode ter um tempo de vida independente da solicitação original. Se o receptor de uma solicitação pode ser representado de uma maneira independente do espaço de endereçamento, então você pode transferir um objeto *Command* para a solicitação para um processo diferente e lá atender a solicitação; Suportar desfazer operações. A operação *Execute*, de *Command*, pode armazenar estados para reverter seus efeitos no próprio comando. A

interface do *Command* deve ter acrescentada uma operação *Unexecute*, que o reverte efeitos de uma chamada anterior de *Execute*. Os comandos executados são armazenados em uma lista histórica. O nível ilimitado de desfazer e refazer operações são obtidos percorrendo esta lista para trás e para frente, chamando operações *Unexecute* e *Execute*, respectivamente.

Figura 1.15 GoF *Command*



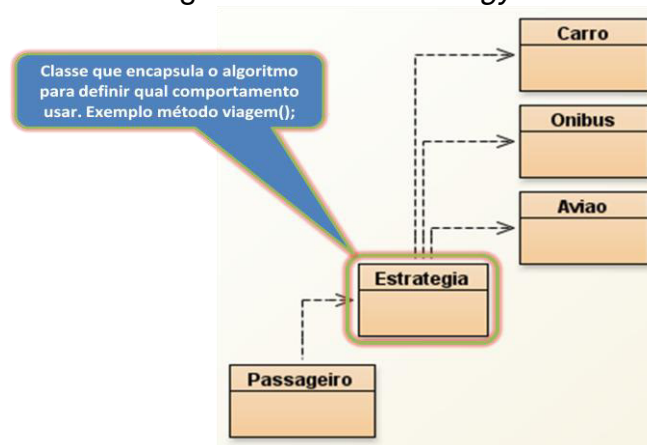
Fonte: Próprio Autor

1.6.4 Strategy

Intenção: Encapsular um algoritmo dentro de uma classe.

Aplicabilidade: Este padrão deve ser utilizado quando: Diversas classes relacionados diferem apenas em comportamento. Estratégias provêm formas de configurar a classe com um dos muitos comportamentos; Necessita-se de diversas variações de um algoritmo; Um algoritmo utiliza dados que não devem ser conhecidos pelo cliente. Utilize este padrão para evitar expor estruturas de dados complexas e específicas do algoritmo. Um classe define diversos comportamentos, e estes aparecem como instruções condicionais múltiplas. Ao invés de manter estas condicionais, mova os trechos condicionais para sua própria classe de estratégia.

Figura 1.16 GoF *Strategy*



Fonte: Próprio Autor

1.6.5 State

Intenção: Alterar o comportamento de um objeto quando seu estado muda.

State Aplicabilidade: Este padrão deve ser utilizado quando: O comportamento de um objeto depende fortemente do seu estado e ele deve alterar o seu

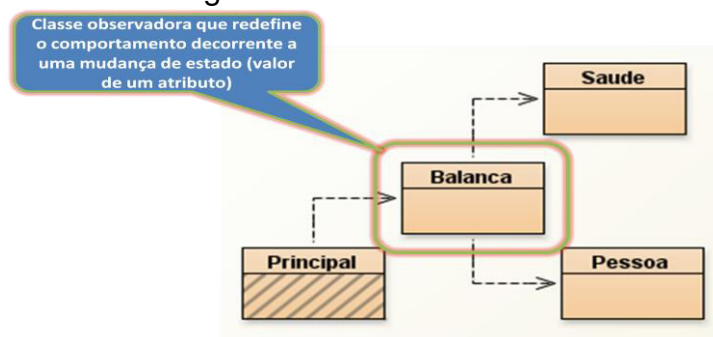
comportamento em tempo de execução dependendo do seu estado. Os métodos têm instruções condicionais grandes em que as condições dependem do estado do objeto. Este estado é normalmente representado por uma ou mais constantes do tipo enumerado. Frequentemente, vários métodos contêm esta mesma estrutura condicional. O padrão State coloca cada ramo da instrução condicional numa classe separada. Desta forma, o estado do objeto pode ser tratado como um objeto ele próprio, o qual pode variar independentemente.

1.6.6 Observer

Intenção: Define um regime de notificação de objetos de mudanças para outro objeto.

Aplicabilidade: Use este padrão quando uma mudança em um objeto deve causar mudança em outros e não se sabem quais e quantos são os outros objetos; quando um objeto deve ser capaz de notificar outros objetos sem assumir nada sobre que são estes outros objetos. Em outras palavras, você não quer que estes objetos estejam fortemente acoplados entre si.

Figura 1.17 GoF *Observer*



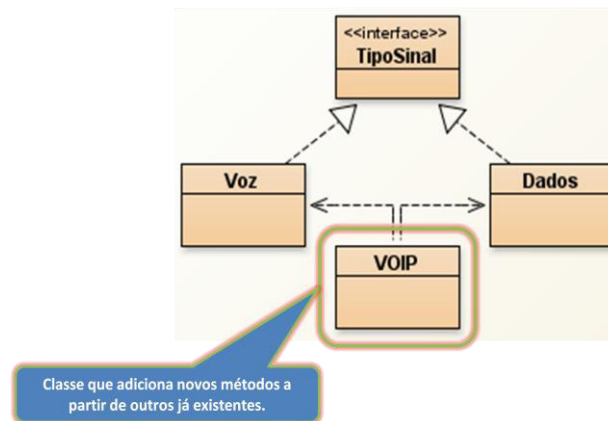
Fonte: Próprio Autor

1.6.7 Interpreter

Intenção: Dada uma linguagem, cria uma representação de sua gramática, juntamente com um interpretador que utiliza esta representação para interpretar sentenças na linguagem.

Aplicabilidade: Este padrão deve ser utilizado quando existe uma linguagem a ser interpretada e é possível representar expressões nesta linguagem como árvores sintáticas abstratas. Esse padrão funciona melhor quando: A gramática é simples. Para gramáticas complexas, a hierarquia de classes se torna muito grande e não gerenciável. Outras ferramentas como geradores de parsers são melhores alternativas nestas situações, pois podem interpretar expressões sem construir árvores sintáticas abstradas, o que pode salvar espaço e possivelmente tempo; Eficiência não é crítico. Os interpretadores mais eficientes são usualmente não implementados através da interpretação de árvores de parser diretamente, mas primeiro é feita uma tradução para um outro formato.

Figura 1.18 GoF *Interpreter*



Fonte: Próprio Autor

1.6.8 Memento

Intenção: Salvar e restaurar o estado interno de um objeto.

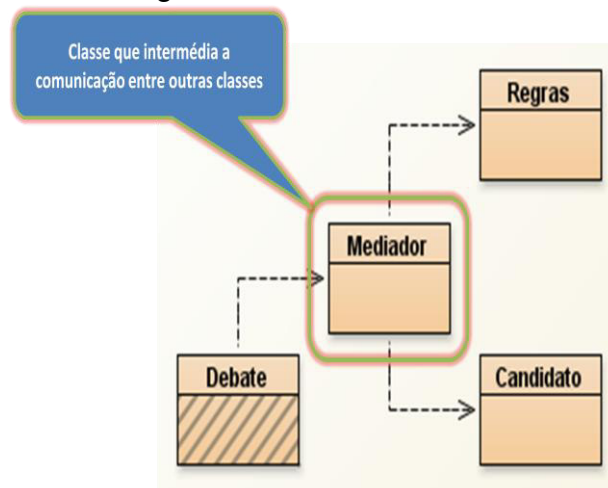
Aplicabilidade: Use este padrão quando uma parte ou todo o estado de um objeto deve ser guardado e possivelmente recuperado posteriormente; a obtenção direta do estado quebraria a proteção de informação expondo detalhes de implementação.

1.6.9 Mediator

Intenção: Define um objeto que encapsula o modo como um conjunto de objetos interage. Promove um acoplamento fraco entre objetos, evitando que referenciem explicitamente um ao outro, e com isto permitindo que se possa variar independentemente a interação entre eles.

Aplicabilidade: Use este padrão quando um conjunto de objetos se comunica de maneira complexa; o reuso de objetos é dificultado porque este referencia e se comunica com muitos outros objetos; o comportamento operações deve ser customizável sem a criação de inúmeras subclasses.

Figura 1.19 GoF *Mediator*

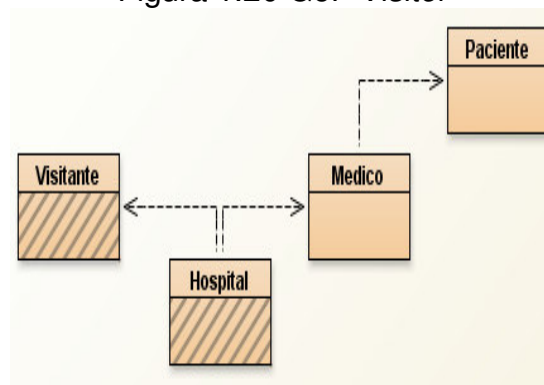


Fonte: Próprio Autor

1.6.10 Visitor

Intenção: Define uma nova operação em uma classe sem trocá-la.

Aplicabilidade: Este padrão deve ser utilizado quando: Uma estrutura de objetos contém muitas classes de objetos com interfaces distintas, e deseja-se executar operações nestes objetos que dependem de sua classe concreta; Muitas operações distintas e não relacionadas precisam ser executadas em objetos em uma estrutura de objetos, e deseja-se evitar poluir estas classes com estas operações. *Visitor* permite que operações relacionadas sejam mantidas juntas definindo-as em uma classe. Quando a estrutura do objeto é compartilhada por muitas aplicações, utilize *Visitor* para colocar as operações apenas nas aplicações que necessitem dela; As classes que definem a estrutura do objeto raramente mudam, porém frequentemente deseja-se adicionar novas operações sobre a estrutura. Modificar a classe da estrutura de objetos requer redefinir a interface para todos os *visitors*, o que é potencialmente custoso. Se as classes da estrutura de objetos mudam constantemente, provavelmente é melhor definir as operações nestas classes.

Figura 1.20 GoF *Visitor*

Fonte: Próprio Autor

1.6.11 Iterator

Intenção: Provê uma forma de acessar seqüencialmente os elementos de um agregado de objetos, sem expor a sua representação interna.

Aplicabilidade: O padrão *iterator* deve ser utilizado para acessar agregações de objetos sem expor sua representação interna; para suportar diversas “varreduras transversais” em agregados de objetos; para prover uma interface uniforme para varrer estruturas agregadas diferentes.

.

Livros da série Fundamentos da Engenharia de Software



Fundamentos da Engenharia de Software: Conceitos Básicos é uma coletânea de disciplinas que integradas servem para fundamentar o

entendimento da construção de projetos de software com qualidade, isto é, baseado em processos maduros e reconhecidos pela comunidade tecnológica. O objetivo deste livro é fornecer ao leitor as bases necessárias para o desenvolvimento de aplicações sejam Desktop, Web ou Mobile. Iniciando a leitura na Teoria da Computação, passando por Processos, Linguagens, Bancos de Dados e finalizando com Sistemas de Informação e Colaboração. Este livro pode ser lido capítulo a capítulo ou somente a disciplina desejada, pois sua elaboração consiste na compilação das disciplinas fundamentais da Engenharia de Software que são independentes, mas ao mesmo tempo se integram objetivando o desenvolvimento de aplicações.



Introdução à Banco de Dados. Neste são abordados os conceitos básicos de bancos de dados e seus sistemas gerenciadores, mas com o foco

na arquitetura relacional, porque ainda hoje o mercado faz uso em larga escala desses bancos de dados, mesmo que o paradigma predominante seja o orientado a objetos e que, já existam a um bom tempo bancos orientados a objeto, até mesmo os bancos objetos-relacionais que são um híbrido entre essas duas arquiteturas, o que predomina ainda é o relacional, assim, este material é focado na linguagem de consulta estruturada para os SGBD-Rs do mercado, com foco na comparação de cinco dos mais utilizados bancos relacionais, os quais são: Oracle, SQLServer, MySQL, SQLBase e Interbase.



Este livro é sobre processos de desenvolvimento de software, evidenciando a necessidade de qualidade na construção de sistemas, conceituando a diferença entre desenvolvimento Adhoc e com processo. Para isso é realizado a introdução à engenharia de requisitos abordando as técnicas para a elicitação de requisitos que forneçam subsídios necessários para uma construção de software com maior qualidade, enfatizando a necessidade de se aplicar na construção de qualquer sistema as técnicas de análise e modelagem, evidenciando o uso da linguagem da Linguagem de Modelagem Unificada (UML) para diagramar um projeto de software, explicando a necessidade do uso de modelos na construção, entrando com detalhes na análise orientada a objetos, com o objetivo de explorar os seus conceitos de requisitos e modelagem integrados. Este material é finalizado com a introdução à medidas de esforço de desenvolvimento, técnica necessária para responder as perguntas básicas de qualquer desenvolvimento: Qual o prazo e custo? E para responder a essas questões é abordado o uso da métrica análise de ponto de função.



Este livro aborda os sistemas que são classificados como informação, a exemplo, sistemas de apoio a decisão, sistemas estratégicos, sistemas

gerenciais e sistemas transacionais. A produção deste material que compõe o volume 4 da coleção Fundamentos da Engenharia de Software é resultado da compilação das aulas produzidas nas disciplinas que compõem os capítulos deste livro.



A motivação deste livro é exemplificar os conceitos de Padrões de Projetos utilizando a linguagem de programação Java, sendo a construção uma

compilação das aulas produzidas com o intuito de facilitar o entendimento do assunto abordando os seguintes temas: Paradigma Orientado a Objetos que introduz o leitor nos conceitos do POO; Linguagem de Modelagem Unificada para apresentar a simbologia UML dos conceitos de POO; Linguagem de Programação Java apresentando essa poderosa linguagem de programação orientada a objetos para exemplificar os padrões de projeto; e Padrões de Projetos que neste livro aborda os mais referenciados nas academias, sendo eles o GRASP e GoF.



Este livro é o resultado do uso da ferramenta MS Project da Microsoft utilizada na aplicação dos conceitos de gestão de projetos do PMBOK com as premissas da engenharia de testes para aquisição de qualidade nos produtos de software.



desenvolvimento e lógica de programação.

Este livro aborda basicamente os conceitos básicos de programação como autômatos, tipos de linguagens, princípios dos compiladores, paradigmas de



dicas de codificação para a linguagem de marcação ASPX, para a linguagem de script mais utilizada pelos navegadores o JavaScript com exemplos de CSS e principalmente dicas de código para a linguagem de programação CSharp e de banco de dados SQL com foco no SQLServer.

Este livro introduz nas tecnologias Web abordando os conceitos básicos para desenvolvimento para Internet com a apresentação da plataforma Dot Net e exibindo