Flexibilidade e Reusabilidade em Sistemas Orientados a Objetos - Uma Proposta para Implementação no EB

GLAUBER VOLKMER[4], SILVIO DANILO DE OLIVEIRA[5], MARÇAL DE LIMA HOKAMA[6]

Resumo. Os padrões de projetos tratam de técnicas que visam permitir o desenvolvimento de softwares a partir de estruturas já existentes e testadas, e não a partir do zero. Cada estrutura dessas contém um nome que a identifica, uma descrição do problema ao qual esta se aplica, uma descrição das peças que compõem a estrutura e uma análise das vantagens e desvantagens de utilizá-la. Os padrões dividem-se em categorias de acordo com um critério estabelecido e também características que alguns têm em comum, de forma que estas estruturas formem grupos de padrões. Neste artigo, não serão analisados todos os padrões existentes propostos por Gamma et al (1995), mas somente aqueles considerados os mais relevantes das categorias: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais.

Palavras-chave: padrões, estruturas, categorias, desenvolvimento.

Abstract. Design Patterns are about techniques that allow software development starting from existent and tested structures, and not from scratch. Each structure contains a name, which establishes its identity, a description of the problem where it is applied, a description of the pieces that compose the structure and an analysis of the advantages and disadvantages of using it. These patterns are divided into categories according to an established criterium and also common characteristics. In this article there will be no analysis of all existent patterns proposed by Gamma et al (1995), there will be an analysis only of those considered the most relevant ones in the following categories: Creation Patterns, Structural Patterns and Behavioral Patterns.

Keywords: patterns, structures, categories, development.

1. Introdução

No ambiente de desenvolvimento de software orientado a objetos, muito tem-se discutido e muitas abordagens têm surgido ao longo do tempo, de forma a incrementar o grau de reusabilidade do código escrito. Já existe uma grande aceitação da reutilização de funções através de bibliotecas, mas não é este tipo de reusabilidade que abordamos aqui, nosso objetivo é falar sobre reusabilidade e flexibilidade na arquitetura de sistemas orientados a objetos, conseguida através dos padrões de projeto. Juntamente com os padrões de projeto surgiu, no mercado, uma nova especialidade no ramo da informática que é a "arquitetura de software". Os arquitetos de software utilizam intensamente os padrões de projeto para que seus sistemas possam ser modificados ou acrescentados de forma que os impactos nos seus ciclos de vida sejam amenizados.

Este artigo abordará os padrões de projetos mais utilizados que foram padronizados por Gamma et al (1995) para resolver diversos tipos de problemas observados dentro da modelagem orientada a objetos e a diferença entre trabalhar com herança de classes e composição de objetos. Estes padrões foram divididos em três categorias de acordo com suas finalidades e seguindo o critério do "propósito" destes padrões. Um outro critério estabelecido por Gamma (1995) seria o "escopo".

A título de informação a respeito dos critérios, Buschmann et al (1996)

[4]Escola de Administração do Exército (EsAEx), Salvador, Brasil. glauber_volkmer@hotmail.com
[5]Escola de Administração do Exército (EsAEx), Salvador, Brasil. silviodanilo@yahoo.com
[6]Escola de Administração do Exército (EsAEx), Salvador, Brasil. caplima@esaex.ensino.eb.br

definiu critérios diferentes daqueles propostos por Gamma (1995), como "nível de abstração" e "tipo de problema resolvido", que não serão abordados neste artigo. As categorias apresentadas neste artigo são as seguintes: criação, estrutural ou comportamental.

Os padrões de criação se preocupam com o processo de instanciação de objetos em um sistema. Os padrões estruturais lidam com a composição de classes ou objetos e os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidade. Os padrões de projetos também podem ser divididos segundo o critério de escopo, este critério especifica se o padrão se aplica primariamente a classes ou objetos. Os padrões para classes são estáticos, isto é, fixados em tempo de compilação e são estabelecidos através de um mecanismo de herança. Os padrões para objetos são dinâmicos, isto é, mudados em tempo de execução e lidam com relacionamentos entre objetos.

2. Padrões de Projetos

Os padrões são descritos em certo nível de abstração, não são utilizados para um domínio específico, uma aplicação ou um subsistema, neste texto, seguindo sua bibliografia, são descrições de objetos e classes comunicantes que são configurados para resolver um problema geral de projeto num contexto particular, identificam as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades.

Cada padrão relaciona-se com um determinado problema e apresenta uma solução de forma reutilizável e flexível na construção de softwares orientados a objetos.

Um dos principais objetivos dos padrões de projetos é fazer com que os softwares que serão desenvolvidos não sejam gerados a partir do zero e sim de soluções que já foram implementadas e testadas.

Em um padrão podemos notar quatro elementos essenciais:

- Nome – É a forma de referenciar um determinado padrão, suas soluções e conseqüências, Permite que se discuta sobre as soluções citando apenas o nome.
- Problema – Descreve o porquê da criação do padrão, quando aplicá-lo. Explica o contexto do problema.
- Solução – Descreve todas as peças que compõem o produto, relacionamentos com outros padrões, responsabilidades e colaborações.
- Conseqüência – É uma análise geral sobre o padrão, mostrando as vantagens e desvantagens de se utilizá-lo.

Gamma (1995) divide os padrões de projetos, segundo o critério de herança de classes, nas seguintes categorias: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais.

3. Herança de Classes x Composição de Objetos (Rigidez vs. Flexibilidade)

A herança de classes permite que se criem novas classes em termos de uma classe já existente, este tipo de reutilização é chamado de "reutilização de caixa branca", porque os interiores das classes ancestrais são visíveis nas classes descendentes.

A composição de objetos permite criar classes que são compostas de objetos, conseguindo obter funcionalidades mais complexas, A composição requer que os objetos tenham interfaces bem definidas e é chamada de "reutilização de caixa preta" por esconder os detalhes internos dos objetos.

3.1 Herança de Classes

3.1.1 Vantagens

- É definida em tempo de compilação e é simples de usar.
- É fácil modificar a implementação que está sendo reutilizada. Modificando as operações que não estão sendo redefinidas, valerá para toda a cadeia da hierarquia.

3.1.2 Desvantagens

- Não é possível mudar as implementações herdadas em tempo de execução, já que a herança e definida em tempo de compilação.
- Na herança, as classes ancestrais definem parte da representação física de suas subclasses, expondo a estas detalhes de implementação, por isso se diz que a herança viola o encapsulamento. Com a herança, a implementação de uma subclasse torna-se amarrada à implementação de sua classe mãe.

3.2 Composição de Objetos

3.2.1 Vantagens

- É definida dinamicamente em tempo de execução pela obtenção de referências para outros objetos por um determinado objeto.
- São acessados exclusivamente através de suas interfaces, não violando o encapsulamento.
- Implementação de objetos em termos de suas interfaces, com menos dependências de implementação.

3.2.2 Desvantagens

- Requer que os objetos respeitem as interfaces uns dos outros, o que por sua vez exige interfaces cuidadosamente projetadas.

4. Padrões de Criação

O sumo da utilização destes padrões é o retardamento do processo de instanciação, tornando o sistema flexível na criação de seus produtos. O padrão pode ser de criação de classe, aonde se utiliza a herança para variar a classe que é instanciada ou de criação de objetos, que delega o processo de instanciação a outro objeto.

Os padrões de criação podem ser implementados de maneira rígida utilizando-se herança de classe ou de uma forma flexível através da composição de objetos. Existem dois temas recorrentes destes padrões que são:

- Todos encapsulam o conhecimento sobre quais classes concretas são usadas pelo sistema.
- Ocultam o modo como as instâncias destas classes são criadas e juntadas, tudo o que o sistema sabe sobre os objetos e que suas classes são definidas como classes abstratas.

Dentre os padrões de criação destacamos os seguintes: *Abstract Factory* e *Singleton*.

4.1 Abstract Factory

4.1.1 Intenção

Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

4.1.2 Motivação

Imagine um jogo virtual de carros que precisará ter a capacidade de mudar o estilo da interface sempre que for preciso. Por mudar o estilo da interface, entenda-se mudar a formato do motor, do chassi e outros componentes da interface.

Se no processo de instanciação, especificarmos de forma rígida o estilo dos componentes será muito difícil modificá-los depois e impossível em tempo de execução.

O padrão *Abstract Factory,* através de suas interfaces, fornece uma solução

para este problema criando fábricas de famílias de objetos.

4.1.3 Estrutura

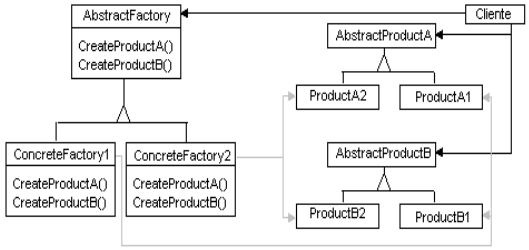

Figura 1: Estrutura da Abstract Factory

4.1.4 Participantes

- AbstractFactory – Declara uma interface para operações que criam objetos-produto abstratos.
- ConcreteFactory – Implementa as operações que criam objetos-produtos concretos.
- AbstractProduct – Declara uma interface para um tipo de objeto-produto.
- ConcreteProduct – Define um objeto-produto a ser criado pela correspondente fábrica concreta; implementa a interface de AbstractProduct.
- Cliente - utiliza as interfaces.

4.1.5 Aplicabilidade

- Um sistema deve ser independente de como os seus produtos são criados, compostos ou representados.
- Um sistema deve ser configurado como um produto de uma família de múltiplos produtos.
- Uma família de objetos-produtos for projetada para ser usada em conjunto e é necessário garantir esta restrição.
- Fornecer uma biblioteca de classes de produtos revelando somente suas interfaces.

Um exemplo da aplicação deste padrão, no Exército Brasileiro, seria por exemplo a disponibilização de objetos em uma biblioteca. Usando-se *Abstract Factory* o acesso à estrutura interna dos objetos seria impedido, sendo liberado o acesso ao objeto somente através de sua interface. Este é um fator primordial que deve ser completamente compreendido, pois é através de interfaces que se mantém a integridade dos objetos em todo o sistema. E a padronização destas interfaces facilita o entendimento e a atualização da arquitetura, já que ela seria previamente conhecida acelerando todo o processo.

4.1.6 Conseqüência

- Isola as classes concretas. A fábrica encapsula a responsabilidade e o processo de criar objetos-produto, as classes concretas ficam isoladas por suas interfaces abstratas.
- Torna fácil a troca de famílias de produtos. A classe de uma fábrica concreta aparece apenas uma vez na aplicação.
- Promove a harmonia entre produtos. Produtos relacionados estarão em uma mesma fábrica.
- Difícil suportar novos tipos de produtos. A interface abstrata da fábrica fixa os tipos de produtos, para modificá-la deve-se também modificar suas subclasses.

4.1.7 Exemplo de Código

```
public interface OmFactory {
  public Om CriaOm()
};

public class BrigadaFactory implements
OmFactory {
  public BriagadaFactory ();
  public Brigada CriaOm() {
    return new Brigada();
  }
};

public class Exercito {
  private OmLista omLista = new OmLista();
  public void Add( OmFactory omf )
  {
        omLista.Add( omf.CriaOm() );
  }
  OmList retornaEstruturaExercito() {
     return omList;
  }
}
```

Método que cria a estrutura do exército retornando uma lista de organizações militares adicionadas anteriormente. Perceba que este simples exemplo já é suficiente para mostrar a flexibilidade do código que utiliza uma *Abstract Factory*. Pois vemos a não dependência da estrutura do exército em relação aos tipos de organizações militares existentes. Para substituir um tipo de organização por outra, basta mudar a instância da fábrica, já que os métodos são definidos em uma interface.

4.2 Singleton

4.2.1 Intenção

Garantir que uma classe tenha somente uma única instância e fornecer um ponto global de acesso à mesma.

4.2.2 Motivação

Em certas aplicações é importante que uma classe apresente apenas uma única instância, podendo esta ser acessada em qualquer parte da aplicação, agindo assim como um componente global do sistema. O padrão *Singleton* oferece esta funcionalidade fazendo com que a classe administre sua única instância.

4.2.3 Estrutura

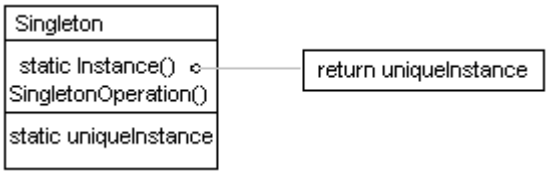

Figura 2: Estrutura do padrão Singleton

4.2.4 Participantes

- Singleton– Define uma operação *Instance* que permite aos clientes acessarem sua única instância. *Instance* é uma operação de classe.

4.2.5 Aplicabilidade

- Deve haver apenas uma instância de uma classe, e essa instância deve dar acesso aos clientes através de um ponto bem conhecido.
- Quando a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usarem uma instância estendida sem alterar o seu código.

4.2.6 Conseqüência

- Acesso controlado à única instância. A classe encapsula e controla sua única instância.
- Espaço de nome reduzido. Evita a poluição do espaço de nomes com variáveis globais de única instância.
- Permite um número variável de instâncias. Com pouca alteração podemos ter ao invés de uma única instância , teremos várias.

4.2.7 Exemplo de Código

```
public class Comandante {
  private static Comandante comandante;
  protected Comandante();

  public static Comandante Instance() {
    if (comandante == null)
      comandante = new Comandante();
    return comandante;
  } // end function
}; //end class
```

Esta classe garante a criação de uma única instância de Comandante, utilizando o padrão *Singleton*.

4.3 Outros Padrões de Criação

4.3.1 Builder

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

4.3.2 Prototype

Especifica os tipos de objetos a serem criados usando uma instância prototípica e criar novos objetos copiando este protótipo.

5. Padrões Estruturais

Estes padrões estudam as formas como classes e objetos são compostos para formarem estruturas maiores.

Eles podem ser:

- Padrões estruturais de classes – Padrões que utilizam a herança para compor interfaces ou implementações, criando uma composição estática de classes.
- Padrões estruturais de objetos – descrevem maneiras de compor objetos para obter novas funcionalidades, criando uma composição flexível.

Dentre os padrões estruturais destacamos os seguintes: *Composite* e *Façade*.

5.1 Padrão Composite

5.1.1 Intenção

Permite aos clientes tratarem de maneira uniforme, objetos individuais e composição de objetos.

5.1.2 Motivação

Imagine que precisemos construir uma aplicação gráfica onde o usuário pode manipular estruturas simples como linhas e textos e também estruturas complexas oriundas da composição de estruturas simples. O problema resolvido pelo padrão *Composite* que apresenta a aplicação citada é como manipular estruturas simples e compostas de uma maneira uniforme.

5.1.3 Estrutura

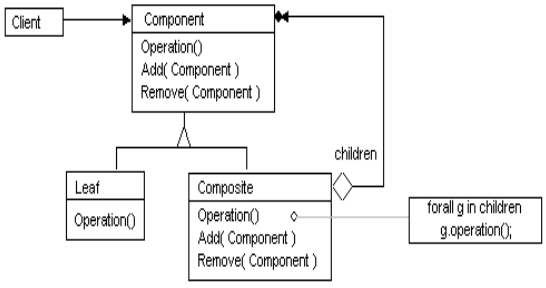

Figura 3: Estrutura do padrão Composite

5.1.4 Participantes

- Component– Declara a interface para os objetos na composição; implementa comportamento por falta para a interface comum a todas as classes; declara uma interface para acessar e gerenciar os seus componentes-filhos.
- Leaf – Representa objetos-folha na composição, define comportamento para objetos primitivos na composição.
- Composite – Define comportamento para os componentes que têm filhos; armazena os componentes filhos; implementa as operações relacionadas com os filhos presentes na interface de *component*.
- Client – Manipula objetos na composição através da interface de *component*.

5.1.5 Aplicabilidade

- Representar hierarquias partes-todo de objetos.
- Clientes capazes de ignorar a diferença entre objetos individuais e composição de objetos.

5.1.6 Conseqüência

- Define hierarquias de classes que consistem de objetos simples e objetos compostos.
- Torna o cliente simples.
- Torna mais fácil de acrescentar novos objetos.
- Torna o projeto excessivamente genérico.

5.1.7 Exemplo de Código

```
public class Graphic {
  public Graphic();
  public void Draw();
}; // end class

public class Line extends Graphic {
  public Line();
  public void Draw() {
    //Código para desenhar a linha
  };
```

```
}; // end class

public class Figure extends Graphic {
  public Figure();
  public void Draw() {
    /*Código para desenhar a figura  */
  }

  public void Add (Graphic g) {
       /*Código para adicionar gráficos
       para
       compor a figura */
  };

  private ArrayList Graphic;
}; //end class
```

5.2 Padrão Façade

5.2.1 Intenção

Fornece uma interface unificada para um conjunto de interfaces em um subsistema.

5.2.2 Motivação

Este padrão diminui a complexidade de sistemas compostos por subsistemas minimizando a comunicação e dependência entre eles. Fornece uma interface única e simplificada para processos complexos envolvendo subsistemas.

5.2.3 Estrutura

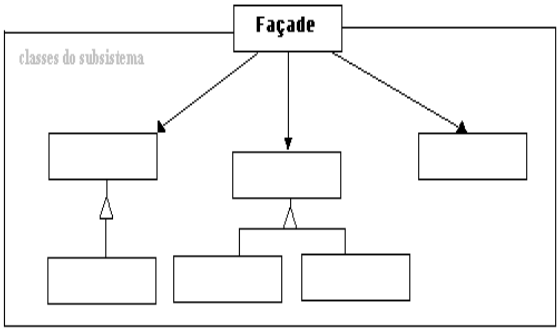

Figura 4: Estrutura do padrão Façade

5.2.4 Participantes

- Façade – Conhece quais as classes do subsistema responsáveis pelo atendimento de uma solicitação.
- Classes do subsistema – Implementam as funcionalidades dos subsistemas.

5.2.5 Aplicabilidade

- Desejar fornecer uma interface simples para um subsistema complexo.
- Existirem muitas dependências entre os clientes e as classes de implementação de uma abstração.
- Desejar estruturar os sistemas em camadas.

5.2.6 Conseqüência

- Isola os clientes dos componentes do subsistema.
- Promove um acoplamento fraco entre o subsistema e seus clientes.

5.2.7 Exemplo de Código

```
public class Serasa {
  public Serasa ();
  public boolean VerificaCliente (Cliente
c) {
    /*Código para verificar o cliente  */
  };
}; //end class

public class SPC {
  public SPC();
  public boolean VerificaCliente( Cliente
c)  {
    /*Código para verificar o cliente */
  };
}; // end class

public class FachadaVisa {
  public FachadaVisa();
  public void CadastraCliente( Cliente c)
{
       SPC spc = new SPC();
       Serasa serasa = new Serasa();
       if ((!serasa.VerificaCliente(c)) &&
       (!spc.VerificaCliente(c)))
            /* cadastra cliente */
};
}; // end class
```

5.3 Outros Padrões Estruturais

5.3.1 Bridge

Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.

5.3.2 Decorator

Atribui responsabilidades adicionais a um objeto dinamicamente. Os *decorators* fornecem uma alternativa flexível a

subclasses para extensão da funcionalidade.

5.3.3 Flyweight

Usa compartilhamento para suportar grandes quantidades de objetos de granularidade fina, de maneira eficiente.

5.3.4 Proxy

Fornece um objeto representante, ou um marcador de outro objeto, para controlar o acesso ao mesmo.

6. Padrões Comportamentais

Estes padrões se preocupam com algoritmos e a atribuição de responsabilidades entre objetos e caracterizam fluxos de controle difíceis de seguir em tempo de execução. Os padrões comportamentais utilizam tanto a herança como a composição de objetos.

Dentre os padrões comportamentais os seguintes:

6.1 Padrão Observer

6.1.1 Intenção

Define uma dependência de um-para-muitos entre objetos, de maneira que quando um objeto muda de estado os outros são notificados e atualizados.

6.1.2 Motivação

Nas aplicações modernas, levamos ao máximo permitido o desacoplamento entre as classes, tornando-as mais reutilizáveis, mas isto nos leva a alguns problemas quando temos classes cooperantes e existe a necessidade de termos a consistência entre seus objetos.

Como exemplo, citamos a representação de uma determinada informação de diferentes maneiras, como em planilhas e gráficos, quando modificamos a informação as suas representações devem acompanhar as modificações. O padrão *Observer* resolve este problema notificando todas as representações sobre a mudança de informação.

6.1.3 Estrutura

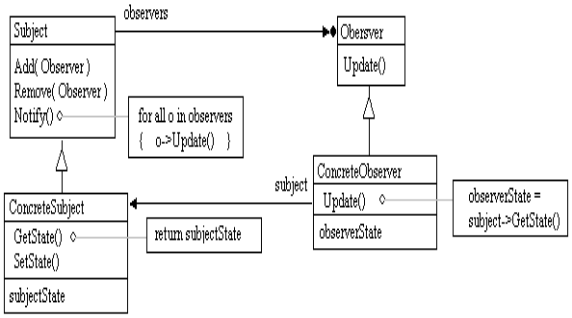

Figura 5: Estrutura do padrão Observer

6.1.4 Participantes

- Subject – Conhece os seus observadores. Um número qualquer de objetos Observer pode conhecer um *Subject* e fornecer uma interface para acrescentar e remover objetos.
- Observer – Define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um *Subject*.
- ConcreteSubject – Armazena estados de interesse para objetos *ConcreteSubject*; envia uma notificação para seus observadores quando seu estado muda.
- ConcreteObserver – Mantém uma referência para um objeto *ConcreteSubject*, armazena estados que deveriam permanecer consistentes com os do *Subject* e implementa a interface de atualização de *Observer*, para manter seu estado consistente com o do *Subject*.

6.1.5 Aplicabilidade

- Quando uma abstração tem aspectos dependentes um do outro.
- Quando uma mudança em um objeto exigir mudanças em outros.
- Quando um objeto deveria ser capaz de notificar outros objetos sem fazer hipóteses, ou usar informações, de quem são estes objetos.

6.1.6 Conseqüência

- Acoplamento abstrato entre *Subject* e *Observer*. Tudo que o *Subject* sabe é que ele tem uma lista de observadores , cada um seguindo a interface simples da classe abstrata *Observer*.
- Suporte para comunicação *brodcast*. Através de um *loop* na sua lista de observadores o *Subject* notifica todos os seus observadores.
- Atualizações inesperadas. Uma operação no *Subject* pode causar uma cascata de atualizações nos seus observadores.

6.1.7 Exemplo de Código

```
public class Observer {
  public void Update();
  protected Observer();
}; // end class

public class Subject {
  public void Add (Observer o) {
        /*Código para ser adicionar um
        observador  */
  };

  public void Remove (Observer o) {
        /*Código para ser remover um
        observador  */
  };

  public void Notify() {
        /*Código para notificar todos o
        observadores executando seu
        Update(); */
  };

  protected Subject();
  private ArrayList Observer obs;
}; / end class

public class Information extends Subject {
  public void ModifyInformation() {
        /* Quando modificada a informação
        os observadores registrados serão
        notificados */
  Notify();
}; / end class

protected Information();

public class Streadsheet extends Observer
{
  public Streadsheet (Information i) {
        /* No construtor recebe o subject e
        registra-se a ele */
        subj = i;
        subj.Add(this);
        //Registrando-se
  };

  public void Update() {
        //Código para atualização
  };

  public void Finalize() {
        //No destrutor desregistra-se
        subj.Remove(this);
        //Desregistrando-se
  };

  protected Information subj;
}; /end class
```

6.2 Padrão Strategy

6.2.1 Intenção

Permite a encapsulação de algoritmos para que variem independentemente dos clientes que o utilizam.

6.2.2 Motivação

Considere uma aplicação que precisa classificar uma massa de dados em um a determinada ordem e queira também parametrizar a forma de como será classificada a informações. O padrão *Strategy* resolve este problema encapsulando os algoritmos de classificação para parametrizar a aplicação.

6.2.3 Estrutura

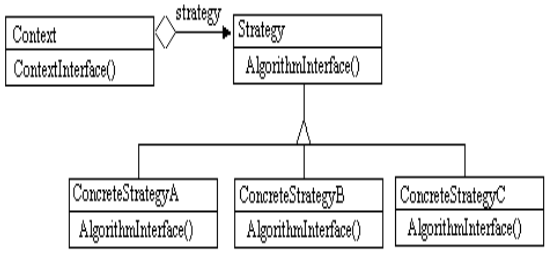

Figura 6: Estrutura do padrão Strategy

6.2.4 Participantes

- Strategy – Define uma interface comum para todos os algoritmos suportados. *Context* usa esta interface para chamar o algoritmo definido por uma *ConcreteStrategy*.
- ConcreteStrategy – Implementa o algoritmo usando a interface de *Strategy*
- Context – É configurado com um objeto *ConcreteStrategy*, mantém

uma referência para um objeto *Strategy* e pode definir uma interface que a permite acessar seus dados.

6.2.5 Aplicabilidade

- Necessidade de variantes de um algoritmo.
- Classes relacionadas diferem apenas no seu comportamento.
- Necessidade de parametrizar um determinado comportamento.

6.2.6 Conseqüência

- Famílias de algoritmos relacionados. Algoritmos para parametrizar as classes.
- Uma alternativa ao uso de subclasses. Ao invés de criar-se uma subclasse com um outro tipo de algoritmo, encapsula apenas o que varia.

6.2.7 Exemplo de Código

```
public class Application {
  public Application(AlgoritmoSort a);
  public void Classifique();
  protected AlgoritimoSort alg;
};

public interface AlgoritimoSort {
  public void Sort();
};

public class ApplicationSort extends
Application  {
  public ApplicationSort( AlgoritmoSort
a){
        alg = a;
  }

  public void Classifique() {
        alg.Sort();
  };
};

public class AlgoQuickSort implements
AlgoritimoSort {
  public  void Sort() {
    //Código para a classificação
  };
};
```

6.3 Outros Padrões Comportamentais

6.3.1 Chain of responsability

Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação.

6.3.2 Interpreter

Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nesta linguagem.

6.3.3 Iterator

Fornece uma maneira de acessar seqüencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

6.3.4 Mediator

Promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.

6.3.5 Memento

Sem violar a encapsulação, captura e externa um estado de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado.

6.3.6 State

Permite que um objeto altere seu comportamento quando seu estado interno muda.

6.3.7 Template Method

Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.

6.3.8 Visitor

Representa uma operação a ser executada sobre os elementos da estrutura de um objeto.

7. Conclusão

Neste artigo, procuramos apresentar as propostas de Gamma et al (1995) sobre Padrões de Projetos com a intenção de difundir o conhecimento destas idéias de forma a incrementar a reutilização e a flexibilidade no desenvolvimento de softwares orientados a objetos dentro do Exército Brasileiro.

Não esperamos que este texto seja o suficiente para convencer desenvolvedores a utilizar ou mesmo entender por inteiro os padrões aqui apresentados. Nosso grande objetivo estará alcançado, se os profissionais que desenvolvem softwares para o exército, sentirem-se impelidos a utilizar os Padrões de Projetos nestes softwares.

Referência Bibliográfica

BUSHMANN, Frank et al. **Pattern-Oriented Software Architecture: A System of Patterns.** [S.L] John Wiley & Sons, 1996. 457 p.

GAMMA, Erich et al. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos.** [S.L] São Paulo:Bookman, 1995. 368 p.

SHALLOWAY, Alan. et al. **Design Patterns Explained: A New Perspective on Object-Oriented Design**. [S.L] Addison-Wesley, 2001, 368 p.