Ministério da Educação
Instituto Federal do Paraná

Campus Londrina

Padrões de Projeto
Catálogo GoF

Romualdo Rubens de Freitas
romualdo.freitas@ifpr.edu.br

Licença

O uso deste material está condicionado à licença **Creative Commons Attribution-NonCommercial-Share Alike 3.0 Brazil**, que pode ser obtida mediante o acesso ao endereço http://creativecommons.org/licenses/by-nc-sa/3.0/br/.

Resumidamente, com este material é possível realizar:

Cópia, distribuição, exibição e execução;

Criação de obras derivadas.

Tendo em mente as seguintes condições:

Deve-se dar crédito ao autor original;

É proibido o uso deste material com fins financeiros;

Se este material for alterado, transformado ou outra obra for criada a partir dele, a obra resultante somente poderá ser distribuída mediante licença idêntica a esta.

Apresentação

Muito se discute quanto à utilização dos padrões de projeto. Alguns argumentam que a proliferação de classes acaba por ”inchar” o código da aplicação. Outros, no entanto, atestam que os padrões de projeto fazem parte das assim chamadas “boas práticas” e que seu uso proporciona maior clareza no código, traz facilidades de manutenção, promove a separação de atribuições e garante extensibilidade à medida que novas funcionalidades são agregadas ao sistema.

Independente de opiniões fica claro que a utilização de boas práticas em projetos de software garante inúmeras vantagens, tais como melhor entendimento do problema que se deseja resolver com a aplicação; divisão de responsabilidades, sendo que cada parte ou camada executa um conjunto de funcionalidades inter-relacionadas; facilidade na inclusão de novos processos e, não menos importante, manutenibilidade. Estas vantagens são apresentadas tendo em vista a codificação. Normalmente, o desenvolvimento de uma aplicação de software deve levar em consideração, além do desenvolvimento, igualmente as etapas de Análise e Projeto, nas quais o uso de boas práticas leva a um projeto de software, como um todo, que será bem entendido por todos da equipe de desenvolvimento, bem como por aqueles que venham a integrá-la posteriormente, além, claro, daqueles dos próprios usuários.

Os padrões de projeto permitem a reusabilidade de projetos e arquiteturas e evitam seu comprometimento, além de auxiliar na documentação e na manutenção de sistemas existentes. De um modo geral, um padrão tem 4 (quatro) elementos principais: **nome**, uma ou duas palavras usadas para descrever o problema no qual o padrão é aplicável, suas soluções e conseqüências; **problema**, que explica o problema e seu contexto e quando aplicar o padrão; **solução**, descreve os elementos que constituem o projeto, seus relacionamentos, responsabilidades e colaborações; e **conseqüências**, descrevem os resultados e benefícios da aplicabilidade do padrão. Desta

forma, um padrão de projeto nomeia, abstrai e identifica aspectos chave de uma estrutura de projeto comum que o torna útil na criação de um projeto orientado a objetos reusável.

Padrões de projeto possibilitam aplicar soluções a problemas freqüentes, também denominados de recorrentes. Mas, na maioria dos casos, muitos consideram o entendimento destas soluções um passo a mais na complexidade do desenvolvimento de aplicações de software e terminam negligenciando seu uso. Na verdade, esta negligência acaba por gerar um contraponto no uso de padrões de projeto, pois, na maioria dos casos, uma solução semelhante à apresentada pelos padrões de projeto é utilizada. Este contraponto se dá pelo fato de o problema ser resolvido com uma codificação muito semelhante àquela que seria feita se os padrões de projeto fossem aplicados. Esta prática é conhecida como ***anti-pattern*** (anti-padrão), que é, basicamente, o uso de práticas não recomendas. Isto se dá pelas seguintes características (a) uma dada solução de código que é utilizada com freqüência e que, a princípio, representa ser uma boa solução, mas que, ao longo do tempo, torna-se um complicador na evolução da aplicação de software ou em sua manutenção; e (b) tal solução não adequada pode ser substituída por um padrão de projeto.

Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides organizam os 23 padrões de projeto apresentados no livro **Design Patterns: Elements of Reusable Object-Oriented Software** de acordo com seus propósitos e escopo. Esta classificação é apresentada na Tabela 1.

		Propósito		
		Criação	Estrutural	Comportamental
Escopo	Classe	Factory Method	Adapter (classe)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (objeto) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabela 1: Classificação dos Padrões de Projeto.

Esta classificação tornou-se conhecida como Catálogo GoF, sendo que GoF significa *Gang of Four* ou Grupo dos Quatro, em inglês, como alusão aos quatro autores do célebre livro sobre padrões de projetos publicado em 1995.

Quando se fala em projeto de software uma notação gráfica para expressar as intenções do projetista é importante. Entretanto, uma notação gráfica é útil para demonstrar o produto, que é o resultado final do processo de projeto na forma de relacionamentos entre classes e objetos. Para se obter um projeto reutilizável é preciso registrar decisões, alternativas e benefícios, bem como criar exemplos concretos, que permitam verificar o projeto em ação.

No livro de Erich Gamma et al, além da notação gráfica e exemplos[1], os padrões de projeto são descritos de acordo com uma estrutura textual considerada consistente. O objetivo desta estrutura é tornar os padrões de projeto mais fáceis de aprender, comparar e usar e é apresentada no Quadro 1.

[1] Os exemplos apresentados no livro estão escritos na linguagem Smalltak e C++.

Nome e Classificação	O nome descreve a essência do padrão e a classificação reflete as divisões apresentadas na Tabela 1.
Intenção	Uma breve declaração sobre o objetivo do padrão, sua lógica e intenção e a que questão de projeto ou problema ele resolve.
Conhecido como	Outros nomes, caso existam, pelos quais o padrão é conhecido.
Motivação	Apresenta uma cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão resolvem o problema.
Aplicabilidade	Situações nas quais o padrão é aplicável, exemplos de projetos ruins que o padrão pode resolver e como reconhecer tais situações.
Estrutura	Apresenta a representação gráfica das classes usando uma notação baseada na OMT (*Object Modeling Technique*), entre outras.
Participantes	Classes e/ou objetos que participam no padrão e suas responsabilidades.
Colaborações	Como os participantes colaboram para conduzir suas responsabilidades.
Conseqüências	Como o padrão suporta seus objetivos, quais os benefícios e resultados do uso do padrão e que aspecto do sistema o padrão permite que possa variar independentemente.
Implementação	Que cuidados, dicas e técnicas devem ser utilizadas quando o padrão for implementado e se há questões específicas a uma determinada linguagem.
Exemplo	Fragmentos de código de como o padrão pode ser implementado.
Usos Conhecidos	Exemplos do padrão em sistemas reais.
Padrões Relacionados	Que padrões de projeto se relacionam, importantes diferenças e com quais padrões pode ser usado.

Quadro 1: Estrutura para descrição dos padroes de projeto utilizada no Catálogo GoF.

Além da descrição textual – e outras já citadas – apresentada por Erich Gamma et al, várias referências apresentam os padrões do Catálogo GoF de acordo com seus propósitos, descrevendo os padrões de criação, estruturais e, por fim, os comportamentais. Este material, por outro lado, utiliza outra ordem de apresentação. Tal ordem baseia-se no grau de complexidade de cada padrão, permitindo que o leitor possa ganhar entendimento sobre os padrões de projeto do Catálogo GoF aos poucos. Mesmo assim, ao lado de cada

padrão é informado qual o seu propósito se de criação, estrutural ou comportamental.

Assim, este material proporciona ao leitor informações sobre Padrões de Projeto provenientes de várias fontes, entre livros, artigos e *sites* na Internet, a partir dos quais se fez uma compilação de tal forma que o leitor possa, com certa facilidade, obter conhecimento sobre soluções para problemas que ocorrem com freqüência. As referências consultadas encontram-se ao final deste material, sob o título Referências.

SUMÁRIO

Introdução

O Projeto Orientado a Objetos enfatiza na definição de objetos de software e em como eles colaboram entre si. Esta colaboração depende de como os objetos necessitam saber sobre questões internas uns dos outros.

Além de vários conceitos e princípios que são empregados no projeto de um software orientado a objetos, dois princípios: **coesão** e **acoplamento**, voltados à qualidade do projeto, devem ser considerados para assegurar a sua qualidade. Tais princípios envolvem uma constante procura pela melhor forma de organizar o código para que ele seja fácil de escrever e de compreender, além de facilitar mudanças futuras.

Algumas questões devem ser levadas em consideração quando se projeta um software orientado a objetos:

- Manter código propício a mudanças tão próximo quanto possível;
- Permitir que código não relacionado sofra mudanças independentes;
- Diminuir a duplicidade de código.

As questões mencionadas têm uma relação muito próxima com qualidade de código e, neste sentido, a coesão e o acoplamento são os princípios empregados com o intuito de se chegar a um projeto de qualidade, sendo que o objetivo destes princípios é o de se atingir a maior coesão e o menor acoplamento possível.

A coesão mede quão bem as partes de um objeto suportam o mesmo propósito. Esta medida considera uma boa definição do propósito do objeto e se cada uma de suas partes contribui diretamente para atingir o propósito desejado. Para que um objeto atinja uma alta coesão ele deve ter um propósito bem definido e todas as suas partes devem contribuir para que este propósito seja alcançado. Na baixa coesão ou o objeto possui um propósito ambíguo,

executando múltiplas tarefas, por exemplo, ou nem todas as suas partes contribuem para o propósito desejado ou ambos.

Quando uma classe é projetada para realizar uma determinada tarefa, seus comportamentos, dados e associações permitem que sua tarefa seja executada a contento. Se uma classe possui comportamentos não relacionados ao seu propósito, dados também não relacionados possivelmente serão acrescentados à sua definição e seus comportamentos deverão decidir se devem ou não realizar determinada tarefa. Se um comportamento freqüentemente gasta mais tempo em descobrir o que fazer e não em como fazer, provavelmente, a classe desse objeto tem baixa coesão.

Infelizmente, altos níveis de coesão são difíceis de serem obtidos, nem sempre é possível descrever uma classe com um único propósito. Neste caso, é importante saber o custo benefício de quando e por que sacrificar a coesão.

O acoplamento mede o grau de dependência entre objetos. Se um objeto é capaz de realizar a tarefa para a qual ele foi concebido sem a interferência de outros objetos no sistema, então se diz que há um fraco acoplamento, ou seja, quanto menor o acoplamento maior é a independência de um determinado objeto em relação aos outros objetos do sistema; ao passo que, quando um objeto necessita de um ou mais objetos para realizar suas tarefas se diz que há um forte acoplamento. Lidar com um objeto é problemático, pois suas conexões com outros objetos devem ser consideradas.

Em outras palavras, o acoplamento trata dos relacionamentos e interações entre objetos com o intuito de se saber se um objeto precisa de outro para executar sua tarefa. Se um objeto é modificado todos os que dependem dele, provavelmente, sofrerão alguma modificação.

A dependência entre objetos é medida sob quatro aspectos dos relacionamentos entre classes:

- **O número de associações**: cada associação que uma classe possui requer manutenção. As associações devem ser criadas, mantidas e excluídas ao longo do tempo;
- **O volume de comunicação**: enquanto um objeto se comunica ele não realiza seu trabalho. A comunicação é importante, mas qual sua real necessidade? Toda comunicação requer um resposta. À medida que o volume de comunicação aumenta, mais tempo é gasto pelo objeto na interpretação da informação e na decisão de como responder;
- **A complexidade da comunicação**: mesmo que haja pouca comunicação entre objetos, se ela é complexa há um forte acoplamento, pois ambos os lados necessitam manter regras adequadas para que a informação seja interpretada corretamente. Se as regras são modificadas de um lado e não do outro a comunicação falha;
- **Um objeto necessita conhecer da estrutura interna ou o estado de outro objeto**: a pior forma de acoplamento é quando um objeto precisa conhecer o estado de outro objeto antes de realizar a comunicação, pois este caso viola o conceito de encapsulamento.

Promover uma alta coesão e um baixo acoplamento é o objetivo de todo projeto de software. Algumas vezes, a solução para problemas de acoplamento está em uma melhor coesão. Em outras vezes, a coesão e o acoplamento têm de ser balanceados para se chegar a um projeto de software adequado.

Singleton – Criação

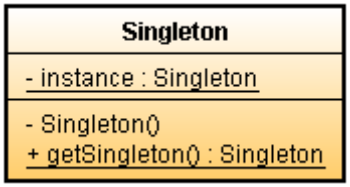
Apresentação

Este padrão garante que haja somente uma instância para uma classe e, assim, fornece um ponto único de acesso a ela.

Aplicabilidade

Grupos (*pools*) de objetos para o gerenciamento de *threads* e conexões com bases de dados; caixas de diálogo; configurações de preferências e de registros; objetos para *logging* e objetos que agem como *drivers* para dispositivos, tais como impressoras e placas gráficas.

Estrutura



Descrição

Para que não haja a possibilidade de criação de objetos arbitrários o construtor é declarado como `private`. A variável estática `instance` é utilizada para armazenar uma instância da classe `Singleton`. Como o construtor da classe é privado, o único ponto de entrada para a obtenção de uma referência para a classe `Singleton` é através do método `getInstance()`, que será invocado para a obtenção desta referência.

Exemplo de Código

```
// Singleton
public class SystemConfig {
   /**
    * Este é o objeto "singleton" que será retornado pelo método getInstance e
    * a partir do qual os métodos getXXX, abaixo, serão invocados.
    */
   private static SystemConfig instance = null;

   public String getUserName() { return System.getProperty("user.name"); }
   public String getUserHome() { return System.getProperty("user.home"); }
   public String getUserDir() { return System.getProperty("user.dir"); }
   /**
    * O construtor DEVE ser 'private' para evitar que objetos sejam criados por
    * outras classes.
    */
   private SystemConfig() {}
   /**
    * Cria o objeto "singleton" a ser utilizado com double-checked locking.
    *
    * @return instance o objeto "singleton"
    */
   public static SystemConfig getInstance() {
      if (instance == null) instance = new SystemConfig();

      return instance;
   } // public static SystemConfig getInstance()
} // public class SystemConfig
```

Exemplo na API Java

A classe `java.lang.Runtime` é uma classe *singleton*, cuja instância é obtida mediante uma chamada ao método `getRuntime()`. Toda aplicação Java possui somente uma instância desta classe, que permite acesso ao ambiente no qual a mesma encontra-se em execução.

Observações

Algumas características deste padrão de projeto merecem atenção especial. Devido ao fato de usar um construtor privado, o conceito de **polimorfismo** torna-se limitado. Outro ponto importante diz respeito a aplicações com múltiplas *threads*. Neste caso, o método `getInstance()` deve ser escrito de tal forma que várias *threads* concorrendo por ele possam aguardar o momento correto para acessá-lo. Assim, sua assinatura passa a ser:

```
public synchronized static Singleton getInstance()
```

O sincronismo de métodos faz com que a aplicação perca desempenho, tornando-a lenta demais, dependendo do número de vezes em que o método sincronizado é invocado.

Uma alternativa é criar a instância da classe assim que ela é carregada pela Máquina Virtual:

```
19 // Singleton
20 public class SystemConfig {
21    /**
22     * Este é o objeto "singleton" que será retornado pelo método getInstance e
23     * a partir do qual os métodos getXXX, abaixo, serão invocados.
24     */
25    private static SystemConfig instance = new SystemConfig();
26
27    public String getUserName() { return System.getProperty("user.name"); }
28    public String getUserHome() { return System.getProperty("user.home"); }
29    public String getUserDir() { return System.getProperty("user.dir"); }
30    /**
31     * O construtor DEVE ser 'private' para evitar que objetos sejam criados por
32     * outras classes.
33     */
34    private SystemConfig() {}
35    /**
36     * Cria o objeto "singleton" a ser utilizado com double-checked locking.
37     *
38     * @return instance o objeto "singleton"
39     */
40    public static SystemConfig getInstance() { return instance; }
41 } // public class SystemConfig
```

Outra opção para minimizar o impacto no uso de *threads* é denominada ***double-checked locking***. Com esta técnica[2] o sincronismo acontece somente se a instância ainda não foi criada:

[2] Alguns autores consideram esta técnica como um padrão.

```
19 // Singleton
20 public class SystemConfig {
21    /**
22     * Este é o objeto "singleton" que será retornado pelo método getInstance e
23     * a partir do qual os métodos getXXX, abaixo, serão invocados.
24     */
25    private static SystemConfig instance = null;
26
27    public String getUserName() { return System.getProperty("user.name"); }
28    public String getUserHome() { return System.getProperty("user.home"); }
29    public String getUserDir() { return System.getProperty("user.dir"); }
30    /**
31     * O construtor DEVE ser 'private' para evitar que objetos sejam criados por
32     * outras classes.
33     */
34    private SystemConfig() {}
35    /**
36     * Cria o objeto "singleton" a ser utilizado com double-checked locking.
37     *
38     * @return instance o objeto "singleton"
39     */
40    public static SystemConfig getInstance() {
41       if (instance == null)
42          synchronized(SystemConfig.class) {
43             if (instance == null)
44                instance = new SystemConfig();
45          } // synchronized(SystemConfig.class)
46
47       return instance;
48    } // public static SystemConfig getInstance()
49 } // public class SystemConfig
```

Esta técnica pode apresentar problemas devido à otimização realizada pelo compilador, que pode retornar a referência a um objeto a partir de um construtor antes mesmo que o construtor possa instanciar o objeto. Uma forma de contornar este possível problema é tomar vantagem do carregador de classes (*class loader*) da aplicação:

```
19 // Singleton
20 public class SystemConfig {
21    /**
22     * Classe interna para criar o objeto 'singleton', que será carregada uma
23     * única vez.
24     */
25    private static class Instance {
26       static final SystemConfig instance = new SystemConfig();
27    } // private static class Instance
28
29    public String getUserName() { return System.getProperty("user.name"); }
30    public String getUserHome() { return System.getProperty("user.home"); }
31    public String getUserDir() { return System.getProperty("user.dir"); }
32    /**
33     * O construtor DEVE ser 'private' para evitar que objetos sejam criados por
34     * outras classes.
35     */
36    private SystemConfig() {}
37    /**
38     * Cria o objeto "singleton" a ser utilizado com double-checked locking.
39     *
40     * @return instance o objeto "singleton"
41     */
42    public static SystemConfig getInstance() { return Instance.instance; }
43 } // public class SystemConfig
```

O que acontece neste caso é que a classe interna (*inner class*) `Instance` é carregada uma única vez.

Padrões Relacionados

Abstract Factory, *Builder* e *Prototype*.

Anti-Pattern

O *anti-pattern* do padrão *Singleton* diz respeito ao fato de haver duas ou mais instâncias da classe em questão, uma para cada classe que queira acessar suas informações, o que acabaria levando a um aumento na criação de objetos e na sua conseqüente remoção pelo coletor de lixo (*garbage collector*), quando for o caso.

Factory Method – Criação

Apresentação

Conhecido também como **Construtor Virtual** ou simplesmente **Factory**, este padrão fornece uma interface para a criação de objetos, permitindo que as classes descendentes decidam qual objeto instanciar. A criação de objetos com um método de *factory* permite maior flexibilidade em relação à criação realizada antecipadamente, pois tal criação é condicional ao objeto que realmente se deseja criar. Em contrapartida, sempre há a necessidade de se derivar uma nova classe específica a partir da interface `Creator`.

Métodos *factory* podem ser utilizados de duas formas: (1) `Creator` é uma classe abstrata e não fornece uma implementação para o método *factory*, forçando as classes derivadas a fazê-lo, ou fornece uma implementação padrão, permitindo que as classes derivadas sejam diferentes de sua classe base, se necessário; (2) a forma mais comum de implementação deste padrão é a utilização de um método *factory* parametrizado, que criará o objeto adequado de acordo com o parâmetro que o identifica. Neste caso, todos os objetos implementam a interface `Product`.

Quando se programa para uma interface é possível que novas classes possam ser agregadas ao sistema sem a necessidade de maiores mudanças. Isto leva a um princípio muito importante no Paradigma Orientado a Objetos: **Uma classe deve estar fechada para modificações, porém aberta para extensões** (*Open-Closed Principle*). Outro aspecto importante é o fato de que a aplicação tem pouco ou nenhum conhecimento de como a classe com a qual ela está lidando foi implementada, concordando com outro princípio: **Programe para uma interface e não para uma implementação**, conforme apresentado por Erich Gamma, et al em **Design Patterns: elements of reusable object-oriented software**.

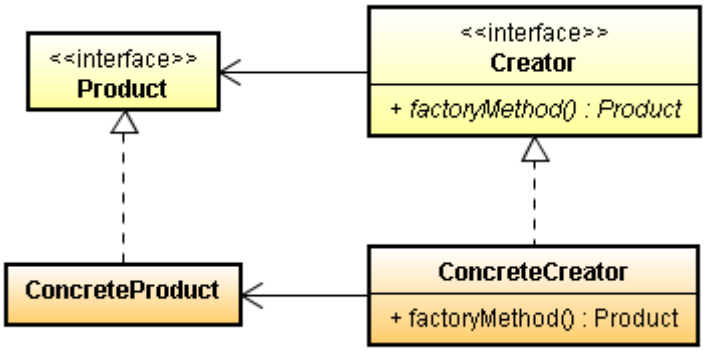
De uma forma geral, não há nada de errado em se usar o operador `new` para a criação de instâncias concretas de classes. O problema reside no fato

de se estar violando o último princípio apresentado no parágrafo anterior. Quanto maior o número de objetos criados a partir de classes concretas, maior é o **acoplamento** entre estas classes, o que dificulta, em longo prazo, a manuteniblidade e extensibilidade de uma aplicação.

Aplicabilidade

A maioria dos *frameworks* faz uso deste padrão possibilitando que uma porção de código existente seja genérica o suficiente para satisfazer alguns requisitos e deixando por conta do desenvolver o código específico, implementado a partir da base do *framework*, que vá de encontro aos requisitos para uma dada aplicação.

Estrutura



Descrição

A interface `Creator` contém, além de outros, o método *factory*. Este método deve ser codificado pelas classes concretas que implementam, no caso de interface, ou herdam, no caso de classe abstrata, da classe `Creator`.

Todas as classes `ConcreteProduct` devem implementar a mesma interface de modo que estas sejam referenciadas por sua interface e não por classes concretas.

Exemplo de Código

```
16 // ConcreteCreator
17 public class DocumentFactory {
18    /*
19     * Tipos de documentos para os quais uma referência é retornada.
20     */
21    public static final String WORD_DOC  = "WORD_DOC";
22    public static final String EXCEL_DOC = "EXCEL_DOC";
23    public static final String XML_DOC   = "XML_DOC";
24    public static final String TEXT_DOC  = "TEXT_DOC";
25    /**
26     * Esta implementação assume que as classes para as quais os objetos serão
27     * criados encontram-se disponíveis na aplicação ou mediante o uso de uma
28     * biblioteca (arquivo .JAR). Além disto, este tipo de implementação é
29     * conhecido como método 'factory' estático, pois, para cada "produto" real,
30     * deve haver código que realize a criação de um objeto.
31     *
32     * @param tipoDoc descreve o tipo do documento a ser criado
33     *
34     * @return o documento criado.
35     */
36    public static Document createDocumentFactory(String tipoDoc) {
37       Document doc = null;
38
39       if (WORD_DOC.equals(tipoDoc)) doc = new WordDocument();
40       else
41          if (EXCEL_DOC.equals(tipoDoc)) doc = new ExcelDocument();
42          else
43             if (XML_DOC.equals(tipoDoc)) doc = new XmlDocument();
44             else
45                if (TEXT_DOC.equals(tipoDoc)) doc = new TextDocument();
46
47       return doc;
48    } // public static Document createDocumentFactory(String)
```

```java
   /**
    * Esta implementação obém o nome da classe para a qual um objeto será
    * criado a partir do arquivo de propriedades e cria a instância do
    * documento utilizando reflexão. Este tipo de implementação é denominado
    * método 'factory' dinâmico, pois, para um novo "produto", não há
    * necessidade de alteração no código, bastando que uma nova linha contendo
    * o nome qualificado da classe seja acrescentada ao arquivo de propriedades.
    */
   public static Document createDynamicDocumentFactory(String tipoDoc) {
      Properties prop      = loadProperties();
      String     className = prop.getProperty(tipoDoc);

      Class clazz = null;
      try { clazz = Class.forName(className); }
      catch (ClassNotFoundException ex) { ex.printStackTrace(); }

      Object instance = null;
      try { instance = clazz.newInstance(); }
      catch (InstantiationException ex) { ex.printStackTrace(); }
      catch (IllegalAccessException ex) { ex.printStackTrace(); }

      return (Document) instance;
   } // public static Document createDynamicDocumentFactory(String)

   private static Properties loadProperties() {
      Properties prop = new Properties();
      try {
         prop.load(DocumentFactory.class.getResourceAsStream(
                                   "/br/edu/utfpr/resource/classes.properties"));
      }
      catch (FileNotFoundException ex) { ex.printStackTrace(); }
      catch (IOException ex) { ex.printStackTrace(); }

      return prop;
   } // private static Properties loadProperties()
} // public class DocumentFactory
```

```java
// Product
public abstract class Document {
   protected String docName;

   public void open(String docName) { this.docName = docName; }
   public abstract void read();
   public abstract void write();
   public abstract void close();
} // public abstract class Document

```

```
7  //ConcreteProduct
8  public class XmlDocument extends Document {
9     @Override
10    public void open(String docName) {
11       super.open(docName);
12       System.out.println("Abrindo documento XML: " + docName);
13    } // public void open(String)
14
15    public void read() {
16       System.out.println("Lendo documento XML: " + docName);
17    }
18
19    public void write() {
20       System.out.println("Gravando documento XML: " + docName);
21    }
22
23    public void close() {
24       System.out.println("Fechando documento XML: " + docName);
25    }
26 } // public class XmlDocument extends Document
```

Exemplo na API Java

O método `getInstance()` da classe `java.util.Calendar`, o método `iterator()` das classes descendentes da interface `java.util.List` são exemplos de *factories*. O primeiro, que também é um *singleton*, retorna uma referência para o calendário representado pela *time zone* (zona horária) e localidade padrão. O segundo devolve uma referência que permite percorrer a coleção de objetos.

Padrões Relacionados

Abstract Factory, *Template Method* e *Prototype*.

Anti-Pattern

Para este padrão, o *anti-pattern* apresenta um forte acoplamento entre o aplicativo e as várias classes que descrevem os “produtos” a serem criados, pois haverá uma referência para cada `ConcretProduct`. Isto implica que modificações futuras podem acrescentar *bugs* ao código já existente e testado.

Abstract Factory – Criação

Descrição

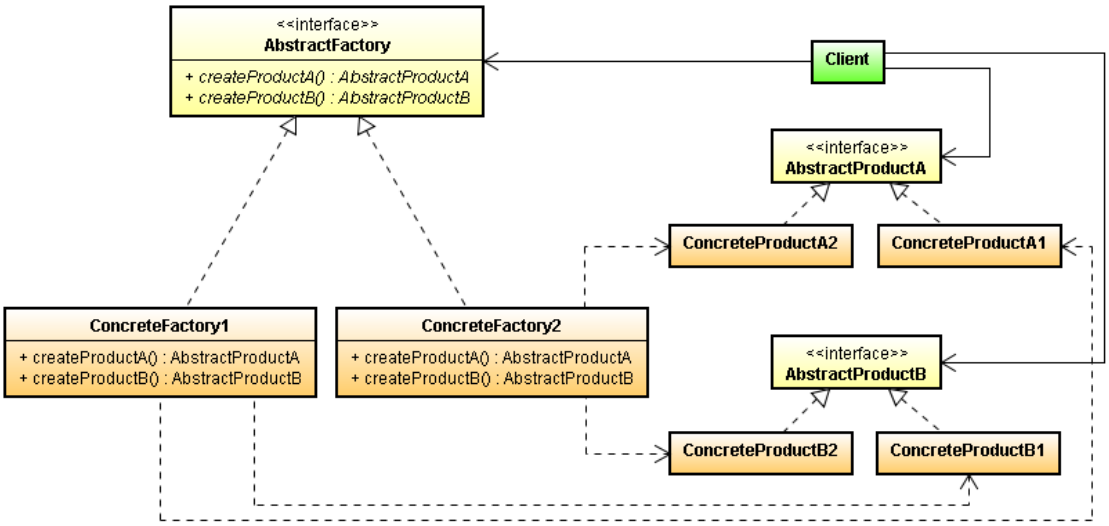
Este padrão provê uma interface, mediante um método *factory* por família, para a criação de famílias de objetos que possuem algum tipo de dependência ou relação sem, no entanto, especificar suas classes concretas.

Aplicabilidade

Quando se deseja criar famílias de objetos (**produtos**) relacionados ou que possuam algum tipo de dependência entre si. Um conjunto de componentes para a interface gráfica com usuário é um bom exemplo. Cada sistema operacional possui um sistema gráfico diferente, como, por exemplo, Windows GDI, Motif, GTK+ e Qt, entre outros. O mesmo acontece com *drivers* para *mouse*, placas de vídeo e impressoras. Para cada sistema operacional a aplicação pode selecionar o conjunto de *drivers* de acordo com a plataforma na qual está em execução.

Aplicações multi-SGBD (Sistema Gerenciador de Banco de Dados) também são exemplos do uso deste padrão de projeto. A aplicação pode selecionar o conjunto de classes para o modelo de dados de acordo com o SGBD.

Estrutura



Descrição

`AbstractFactory` define a interface que todas as fábricas concretas (`ConcreteFactory1`, `ConcreteFactory2`) devem implementar. As fábricas concretas implementam diferentes famílias de produtos (`AbstractProductA`, `AbstractProductB`). Desta forma, a classe que representa um "produto", propriamente dita, jamais tem que ser instanciada.

As interfaces `AbstractProductA` e `AbstractProductB` formam as famílias de **produtos** que podem ser criados.

A classe `Client` faz uso de `AbstractFactory` para obter referências a `AbstractProductA` e a `AbstractProductB`. Assim, famílias de **produtos** podem ser criadas independentes de suas classes concretas.

Exemplo de Código

```
12 // AbstractFactory
13 public interface ContinenteAbstractFactory {
14    HerbivoroAbstractProduct createHerbivoroFactory();
15    CarnivoroAbstractProduct createCarnivoroFactory();
16 } // public interface ContinenteAbstractFactory
17
```

```
14 // ConcreteFactory
15 public class AmericaConcreteFactory implements ContinenteAbstractFactory {
16    public HerbivoroAbstractProduct createHerbivoroFactory() {
17       return new Bufalo();
18    } // public HerbivoroAbstractProduct createHerbivoroFactory()
19
20    public CarnivoroAbstractProduct createCarnivoroFactory() {
21       return new Lobo();
22    } // public CarnivoroAbstractProduct createCarnivoroFactory()
23 } // public class AmericaConcreteFactory implements ContinenteAbstractFactory
24
```

```
14 // ConcreteFactory
15 public class AfricaConcreteFactory implements ContinenteAbstractFactory {
16    public HerbivoroAbstractProduct createHerbivoroFactory() {
17       return new Gnu();
18    } // public HerbivoroAbstractProduct createHerbivoroFactory()
19
20    public CarnivoroAbstractProduct createCarnivoroFactory() {
21       return new Leao();
22    } // public CarnivoroAbstractProduct createCarnivoroFactory()
23 } // public class AfricaConcreteFactory implements ContinenteAbstractFactory
24
```

```java
// AbstractProduct
public interface CarnivoroAbstractProduct {
    void alimentar(HerbivoroAbstractProduct h);
} // public interface CarnivoroAbstractProduct

```

```java
// AbstractProduct
public interface HerbivoroAbstractProduct {
} // public interface HerbivoroAbstractProduct

```

```java
// ConcreteProduct
public class Leao implements CarnivoroAbstractProduct {
   public void alimentar(HerbivoroAbstractProduct h) {
      System.out.println(this.getClass().getSimpleName() + " aliementa-se de " +
                         h.getClass().getSimpleName());
   } // public void alimentar(HerbivoroAbstractProduct)
} // public class Leao implements CarnivoroAbstractProduct

```

```java
// Client
public class MundoAnimal {
   private CarnivoroAbstractProduct carnivoro;
   private HerbivoroAbstractProduct herbivoro;

   public MundoAnimal(ContinenteAbstractFactory abstractFactory) {
      carnivoro = abstractFactory.createCarnivoroFactory();
      herbivoro = abstractFactory.createHerbivoroFactory();

      cadeiaAlimentar();
   } // public MundoAnimal(ContinenteAbstractFactory)

   public void cadeiaAlimentar() { carnivoro.alimentar(herbivoro); }
} // public class MundoAnimal
```

Exemplo na API Java

A classe abstrata `java.awt.Toolkit` é a superclasse para todas as implementações de *Abstract Window Toolkit*, permitindo que os componentes da interface gráfica com o usuário sejam vinculados a implementações nativas (dependentes da interface gráfica específica, tais como, Motif e GTK+, entre outras).

Observações

Padrões Relacionados

Factory Method, *Prototype* e *Singleton*.

Anti-Pattern

O *anti-pattern* para o padrão de projeto *Abstract Factory* assemelha-se ao do padrão *Factory Method*, pois a aplicação manteria um forte acoplamento com as classes concretas, `Leao` e `Lobo`, por exemplo, o que tornaria sua manutenção muito difícil.

Command – Comportamento

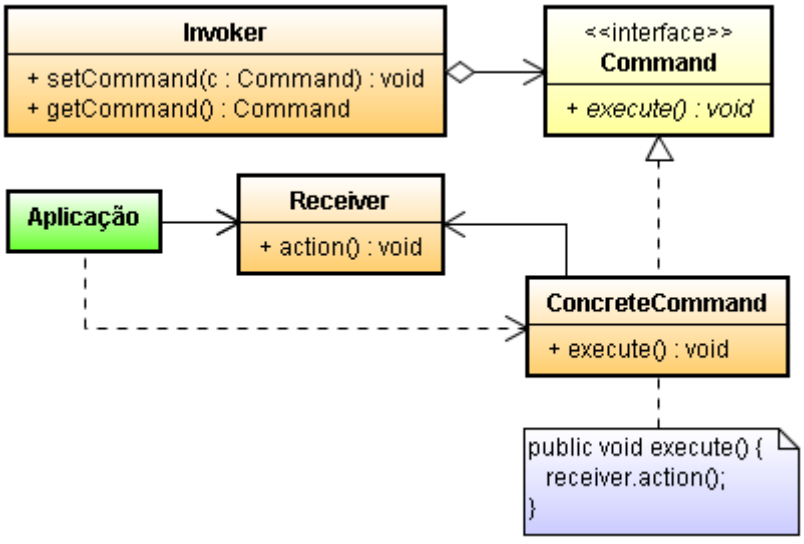
Descrição

Este padrão de projeto representa comandos como objetos. Isto possibilita realizar requisições a objetos sem o conhecimento de como a operação é executada ou o destinatário (*receiver*) da requisição.

Aplicabilidade

Objetos podem ser parametrizados pela ação a executar, como, por exemplo, uma opção de menu ou um botão em uma barra de ferramentas. Opcionalmente, um comando pode especificar uma operação de desfazer (*undo*), permitindo reverter os efeitos no próprio comando. Assim, a interface `Command` deve declarar uma operação (método) – `undo()` – para realizar tal operação. Neste caso, as ações realizadas por `execute()` são armazenadas em uma lista de histórico e o nível de execuções e reversões pode ser ilimitado simplesmente percorrendo a lista de histórico do fim para o início ou do início para fim.

Estrutura



Descrição

A classe `Aplicação` cria um comando concreto e configura o *receiver* para que este possa executá-lo. A classe `Receiver`, que pode ser qualquer classe na aplicação, sabe como executar o trabalho necessário. `ConcreteCommand` é responsável por manter um vínculo entre uma ação (método `action()`) e um objeto da classe `Receiver`. Assim, um objeto `Invoker` invoca o método `execute()` e `ConcreteCommand` realiza uma ou mais ações no objeto `Receiver`. O papel de `Command`, que pode ser uma interface ou classe abstrata, é o de disponibilizar uma interface comum a todas as classes concretas que a implementam.

Exemplo de Código

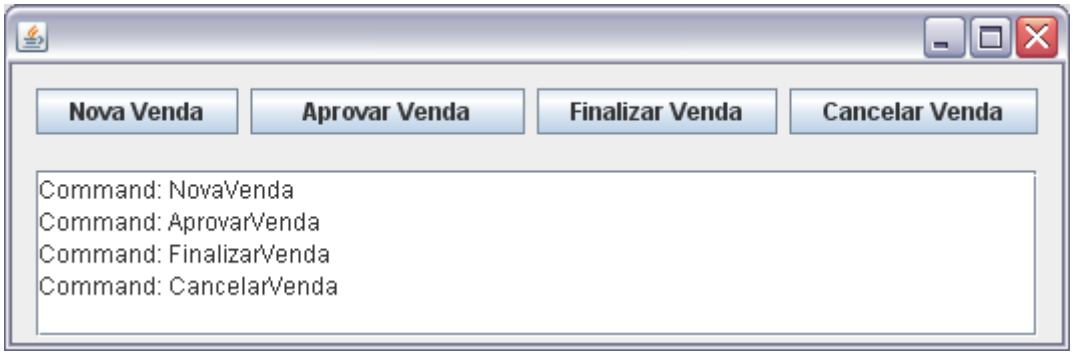
```
14 public class VendaFrame extends javax.swing.JFrame {
15    private ProcessarVenda pv = new ProcessarVenda();
16    private CommandFactory cf = new CommandFactoryProperties();
17
```

```
83
84        jButton1.setActionCommand(CommandFactory.NOVA_VENDA);
85        jButton1.addActionListener(pv);
86        jButton2.setActionCommand(CommandFactory.APROVAR_VENDA);
87        jButton2.addActionListener(pv);
88        jButton3.setActionCommand(CommandFactory.FINALIZAR_VENDA);
89        jButton3.addActionListener(pv);
90        jButton4.setActionCommand(CommandFactory.CANCELAR_VENDA);
91        jButton4.addActionListener(pv);
92
93        java.awt.Dimension screenSize = java.awt.Toolkit.getDefaultToolkit().getSc
94        setBounds((screenSize.width-454)/2, (screenSize.height-173)/2, 454, 173);
95    }// </editor-fold>//GEN-END:initComponents
96
97    private class ProcessarVenda implements ActionListener {
98       public void actionPerformed(ActionEvent e) {
99          String commandAction = e.getActionCommand();
100         jTextArea1.append("Command: " + commandAction + "\n");
101         Command command = cf.createCommandFactory(commandAction);
102         command.execute();
103      } // public void actionPerformed(ActionEvent)
104   } // private ProcessarVenda implements ActionListener
```

```
7  // Command
8  public interface Command {
9     void execute();
10 } // public interface Command
11
```

```
// ConcreteCommand
public class AprovarVendaCommand implements Command {
   public void execute() {
      System.out.println("Verificando credito do cliente.");
   }
} // public class AprovarVendaCommand implements Command

```



Exemplo na API Java

A interface `java.swing.Action` é utilizada quando uma mesma funcionalidade deve ser utilizada por mais de um componente (`JMenuItem` e `JButton`, por exemplo). A classe `java.swing.AbstractAction` é utilizada como base para a implementação da classe-comando que realizará a operação necessária e uma referência a esta é passada ao componente em questão através do método `setAction()`.

Observações

Classes `Command` desacoplam objetos que invocam operação daqueles que a executam. Além disto, utilizando o padrão de projeto *Composite*, é possível criar conjuntos compostos de comandos que são executados em seqüência.

Padrões Relacionados

Composite, *Memento* e *Prototype*.

Anti-Pattern

Separar os comandos a serem executados em classes distintas mediante uma interface comum é tido como uma boa prática em projetos de software. Classes que implementam métodos para realizarem suas tarefas

tornam-se “inchadas”, de difícil manutenção e pouco extensíveis. Com isto, tais classes passam a ter um forte acoplamento com as tarefas (comandos) a serem realizadas.

Chain of Responsibility – Comportamento

Apresentação

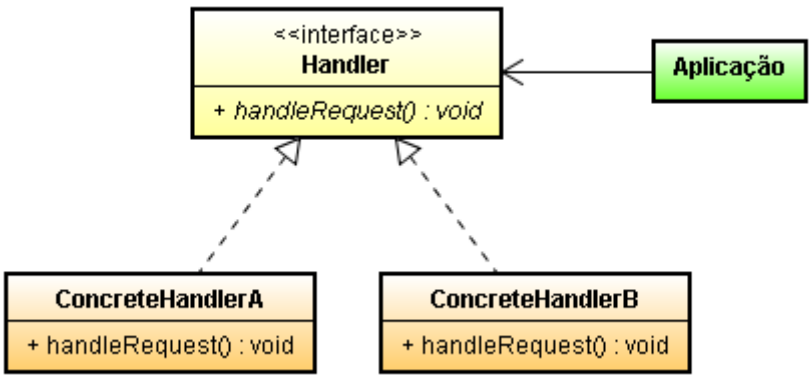
Normalmente, este padrão de projeto é utilizado em situações nas quais se deseja que mais de um objeto possa tratar (manipular) um pedido (requisição), evitando o acoplamento entre o remetente de um pedido e o seu destinatário. Assim, os objetos recebedores são encadeados e o pedido é passado por este encadeamento até que um objeto o trate.

Aplicabilidade

Este padrão de projeto é comumente utilizando quando mais de um objeto é capaz de tratar um pedido e o mesmo não é conhecido antecipadamente. O objeto tratador do pedido deve ser reconhecido automaticamente.

Um pedido deve ser tratado por um entre vários objetos sem que o objeto tratador seja especificado explicitamente.

Estrutura



Descrição

Cada objeto no encadeamento (`ConcreteHandlerA`, `ConcreteHandlerB`) age como um tratador de pedidos e conhece seu sucessor. Se um dado objeto é capaz de tratar o pedido ele o faz, caso contrário o pedido é passado ao próximo tratador no encadeamento de objetos.

Exemplo de Código

```
 9 // Handler
10 public abstract class Aprovador {
11    protected Aprovador proximo;
12
13    public void setProximo(Aprovador proximo) { this.proximo = proximo; }
14
15    public abstract void processarRequisicao(Compra compra);
16 } // public abstract class Aprovador
```

```
10 // ConcreteHandler
11 public class Vendedor extends Aprovador {
12    @Override
13    public void processarRequisicao(Compra compra) {
14       if (compra.getQtde() < 10000.0)
15          System.out.println(this.getClass().getSimpleName() + " aprovou " +
16                             "compra No. " + compra.getNumero());
17       else
18          if (proximo != null)
19             proximo.processarRequisicao(compra);
20    } // public void processarRequisicao(Compra)
21 } // public class Vendedor extends Aprovador
```

```
10 // ConcreteHandler
11 public class GerenteVendas extends Aprovador {
12    @Override
13    public void processarRequisicao(Compra compra) {
14       if (compra.getQtde() < 15000.0)
15          System.out.println(this.getClass().getSimpleName() + " aprovou " +
16                             "compra No. " + compra.getNumero());
17       else
18          if (proximo != null)
19             proximo.processarRequisicao(compra);
20    } // public void processarRequisicao(Compra)
21 } // public class GerenteVendas extends Aprovador
```

```
10 // ConcreteHandler
11 public class Diretor extends Aprovador {
12    @Override
13    public void processarRequisicao(Compra compra) {
14       if (compra.getQtde() < 50000.0)
15          System.out.println(this.getClass().getSimpleName() + " aprovou " +
16                             "compra No. " + compra.getNumero());
17       else
18          System.out.println("Compra muita alta.");
19    } // public void processarRequisicao(Compra)
20 } // public class Diretor extends Aprovador
```

```
12 public class ChainPrincipal {
13     /**
14      * @param args os argumentos de linha de comando
15      */
16     public static void main(String[] args) {
17        Vendedor       v  = new Vendedor();
18        GerenteVendas gv = new GerenteVendas();
19        Diretor        d  = new Diretor();
20
21        v.setProximo(gv);
22        gv.setProximo(d);
23
24        Compra c = new Compra(1,7500);
25        v.processarRequisicao(c);
26
27        c = new Compra(2,11500);
28        v.processarRequisicao(c);
29
30        c = new Compra(3,62130.5);
31        v.processarRequisicao(c);
32     } // public static void main(String[])
33 } // public class ChainPrincipal
```

Exemplo na API Java

Observações

O projeto Apache Commons disponibiliza, entre outros, um *framework*, denominado **Commons Chain** (http://commons.apache.org/chain/), que implementa e estende o padrão de projeto *Chain of Responsibility*. Este *framework* é disponibilizado gratuitamente junto com seu código fonte.

Padrões Relacionados

Composite.

Anti-Pattern

Dado um conjunto de serviços, como, por exemplo, validação de dados, *log* e filtro de dados e segurança, que devem ser executados por um grupo de operações (processamentos), como, por exemplo, um sistema de venda na qual cada etapa de sua realização constitui-se em uma operação, cada processamento estaria acoplado a todos os serviços e estes seriam totalmente dependentes das várias etapas do processamento do pedido (realização de um venda).

Observer – Comportamento

Apresentação

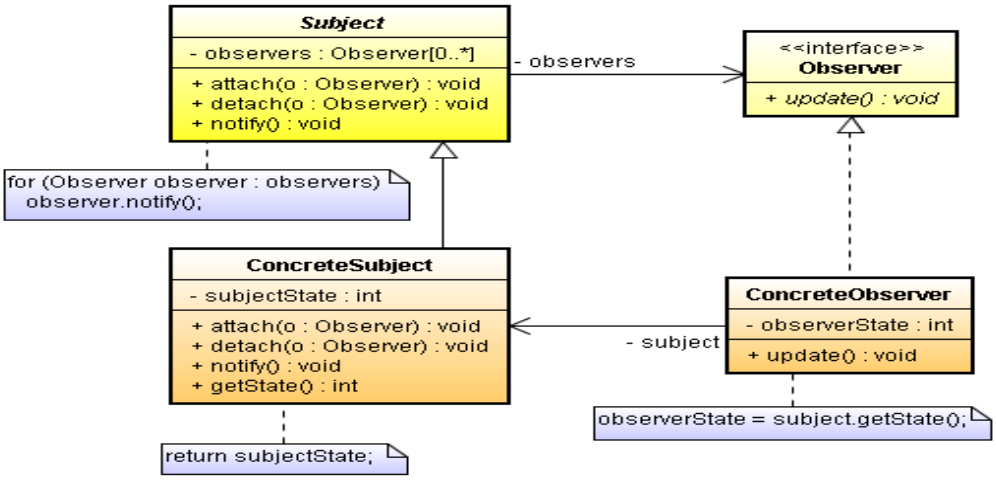
O objetivo deste padrão de projeto é definir uma relação de dependência de um para muitos entre um conjunto de objetos. Desta forma, quando um objeto muda de estado todos seus dependentes são notificados de tal mudança. De outra forma, se pode dizer que um ou mais objetos, denominados **observadores** (*observers*) ou **ouvintes** (*listeners*) são registrados (ou se registram) para **observervar** um determinado evento que pode acontecer por meio de um objeto observado (**subject**). Normalmente, este objeto observado mantém uma coleção de **observadores**.

Aplicabilidade

Este padrão de projeto é utilizado quando se deseja observar um evento externo, tal como, uma ação do usuário, o que é muito empregado em programação orientada a eventos. Mudanças no valor de propriedades (variáveis) também empregam o uso deste padrão de projeto.

A arquitetura Modelo-Visão-Controlador[3] (MVC – *Model-View-Controller*) faz uso freqüente deste padrão, pois ele é empregado de forma a criar um baixo acoplamento entre o Modelo e a Visão fazendo com que uma mudança no Modelo alerte os seus observadores, que são, neste caso, as visões.

Estrutura



[3] Alguns autores consideram o MVC como um padrão de projetos.

Descrição

A classe `Subject` é utilizada por classes concretas como interface comum para que objetos possam se registrar como **observadores**, bem como para que possam ser removidos, sendo que cada classe concreta pode ter uma coleção de observadores. Qualquer classe pode ser um observador desde que implemente a interface `Observer` e se registre com uma classe concreta de `Subject`. Assim, os observadores serão notificados sempre que o estado da classe concreta de `Subject` que os tenha registrado mude.

Exemplo de Código

```
// Observer
public interface Investidor {
   public void update(AcaoAbstract acao);
} // public interface Investidor
```

```
// ConcreteObserver
public class InvestidorImpl implements Investidor {
   private String       nome;
   private AcaoAbstract acao;

   public InvestidorImpl(String nome) { setNome(nome); }
   // setters
   public void setNome(String nome) { this.nome = nome; }
   public void setAcao(AcaoAbstract acao) { this.acao = acao; }
   // getters
   public String       getNome() { return nome; }
   public AcaoAbstract getAcao() { return acao; }

   @Override
   public void update(AcaoAbstract acao) {
      System.out.println("A acao " + acao.getSimbolo() + " do investidor " +
                         nome + " mudou seu valor para $" + acao.getPreco());
   } // public void update(AcaoAbstract)
} // public class InvestidorImpl implements Investidor
```

```java
// Subject
public abstract class AcaoAbstract {
   private String           simbolo;
   private BigDecimal       preco;
   private List<Investidor> investidores; // observers

   protected AcaoAbstract(String simbolo, BigDecimal preco) {
      this.investidores = new ArrayList<Investidor>();
      setSimbolo(simbolo);
      setPreco(preco);
   } // protected AcaoAbstract(String, BigDecimal)
   // setters
   public void setSimbolo(String simbolo) { this.simbolo = simbolo; }
   public void setPreco(BigDecimal preco) {
      this.preco = preco;
      // toda vez que o preço de uma ação se modificar, os observadores desta
      // registrados a esta ação (objeto observável) serão notificados.
      change();
   } // public void setPreco(BigDecimal)
   // getters
   public String     getSimbolo() { return simbolo; }
   public BigDecimal getPreco() { return preco; }

   // registro um observador.
   public void attach(Investidor investidor) { investidores.add(investidor); }
   // remove (desregistra) um observador.
   public void detach(Investidor investidor) { investidores.remove(investidor);}

   public void change() {
      // para cada observador registrado o método update() é invocado para
      // informar o observador da modificação do estado do objeto observável.
      for (Investidor investidor : investidores)
         investidor.update(this);
   } // public void change()
} // public abstract class AcaoAbstract
```

```java
// ConcreteSubject
public class Google extends AcaoAbstract {
   public Google(String simbolo, BigDecimal preco) { super(simbolo,preco); }
} // public class Google extends AcaoAbstract

```

Exemplo na API Java

A interface `java.util.Observer` representa este padrão de projeto em Java, juntamente com a classe `java.util.Observable`, que representa um objeto observável.

As interfaces “ouvintes” (*listeners*) no pacote `java.awt.event` permitem que uma aplicação Java que faça uso da API Swing para a construção de interfaces gráficas com o usuário (GUI – *Graphical User Interface*) instalem a si mesmas ou classes auxiliares como ouvintes de eventos, como, por exemplo, eventos de *mouse*, teclado e janela. Outra característica da API Swing é que esta segue a arquitetura MVC, na qual o

modelo avisa sobre notificações de mudanças utilizando a infra-estrutura `PropertyChangeNotification`. Classes Modelo em Java seguem a especificação JavaBeans, que se comportam como `Subject`. Classes Visão são associadas com algum item na interface gráfica com o usuário e se comportam como observadores. Assim, enquanto a aplicação está em execução, mudanças realizadas no modelo são percebidas pelo usuário porque as visões (componentes gráficos) são atualizadas por meio de notificações.

Observações

Padrões Relacionados

Singleton e *Mediator*.

Anti-Pattern

Quando uma classe se encarrega de observar um determinado evento e tomar todas as ações necessárias fere o **Princípio Aberto-Fechado**, pois a classe deverá ser modificada para comportar novas ações.

Adapter – Estrutura

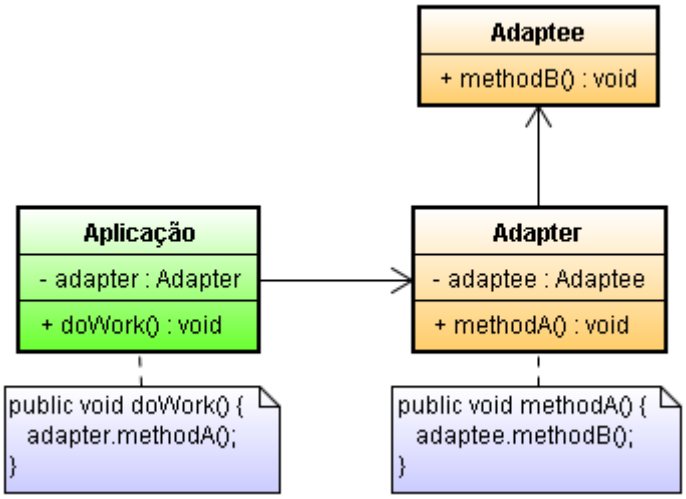
Descrição

Padrão de projeto que possibilita converter a interface de uma classe em outra necessária à aplicação. Assim, classes com interfaces distintas ou incompatíveis podem trabalhar juntas.

Aplicabilidade

Quando há a necessidade de se usar uma classe existente e sua interface não é aquela esperada. Proporciona empregar o princípio de composição, no qual um objeto é adaptado com uma interface modificada. Além disto, é possível vincular uma aplicação cliente a uma determinada interface, permitindo acrescentar novas funcionalidades após a implementação deste padrão de projeto.

Estrutura



Descrição

De acordo com o Diagrama de Classes acima, a aplicação – `Aplicação` – possui uma referência à classe `Adapter` (classe adaptadora). De posse desta referência, basta que o método `doWork()` seja invocado para que este possa realizar a chamada ao método `methodA()`, na classe

`Adapter`, que, por sua vez, invocará o método `methodB()` presente na clase `Adaptee` (classe adaptada).

Exemplo de Código

```
11 // Adapter
12 public class EnumerationIterator implements Iterator {
13    private Enumeration enumeration;
14
15    public EnumerationIterator(Enumeration enumeration) {
16       this.enumeration = enumeration;
17    } // public EnumerationIterator(Enumeration
18
19    public boolean hasNext() { return enumeration.hasMoreElements(); }
20
21    public Object next() { return enumeration.nextElement(); }
22
23    public void remove() {
24       throw new UnsupportedOperationException("Operacao nao implementada.");
25    } // public void remove()
26 } // public class EnumerationIterator implements Iterator
```

Exemplo na API Java

Este padrão é muito utilizado na API Swing com as classes `java.awt.event.WindowAdapter` e `java.awt.event.MouseAdapter`, entre outras.

Observações

Há dois tipos de adaptadores: **objetos adaptadores** e **classes adaptadoras**. No primeiro caso, a adaptação é realizada pelo princípio da composição; enquanto, no segundo caso, a adaptação é propiciada pelo conceito de **herança múltipla**, conceito este inexistente na linguagem Java.

Padrões Relacionados

Bridge, *Decorator* e *Proxy*.

Anti-Pattern

O *anti-pattern* deste padrão de projeto dá-se pela existência de uma ou mais classes cuja interface difere das interfaces de classes existentes e têm certa relação com a nova classe ou classes. A codificação do sistema deverá levar em consideração estas possíveis novas classes e isto acarretará um

inchamento do código à medida que tais classes passem a integrar o sistema existente. O mesmo caso aplica-se quando classes de uma aplicação, possivelmente legada, têm de ser integradas a novos sistemas.

Façade – Estrutura

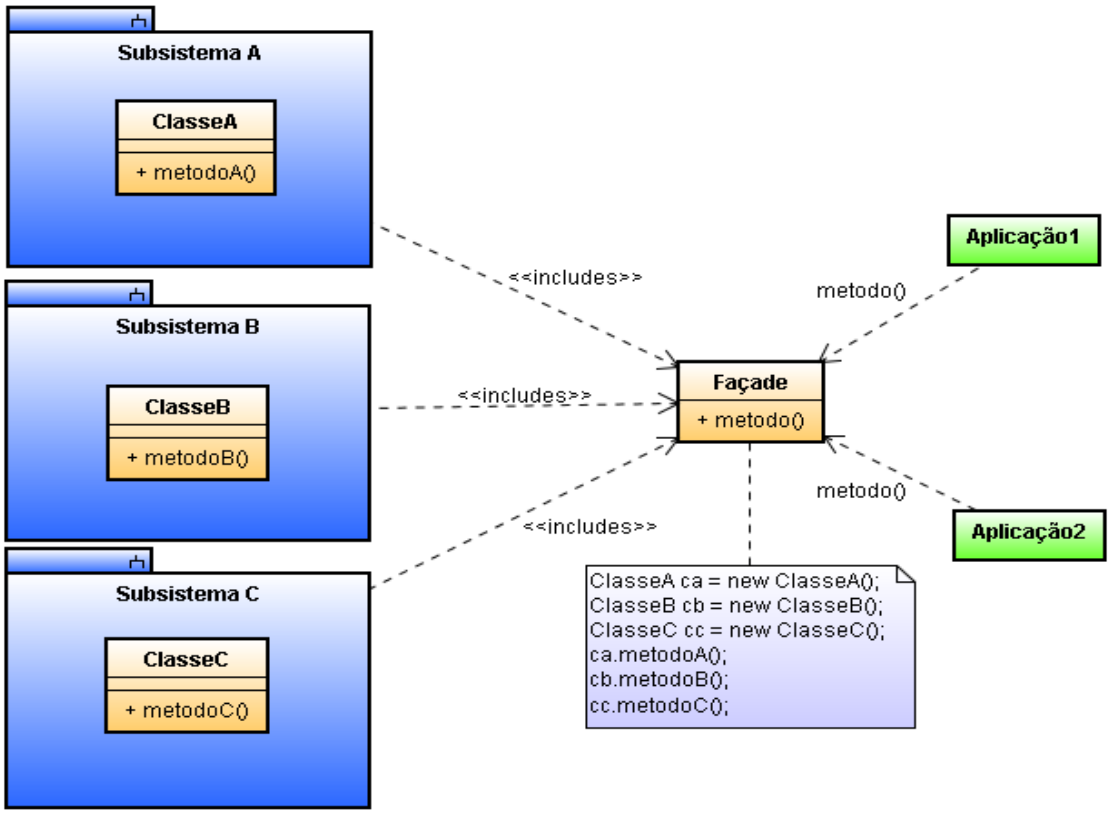
Apresentação

Este padrão de projeto fornece uma interface única para um conjunto de interfaces em um subsistema, definindo uma interface de nível mais alto que facilita o uso do subsistema. Desta forma, o acoplamento entre a aplicação cliente e os subsistemas torna-se minimizado.

Aplicabilidade

À medida que os sistemas de software evoluem, eles se tornam mais complexos devido à necessidade de se executar um conjunto de operações de uma única vez ou em uma dada seqüência. Quando há necessidade de se criar um ponto comum de execução para um conjunto de operações, o padrão de projeto *Façade* é aplicado para a criação deste ponto comum (ou interface) para um determinado subsistema, simplificando e unificando um grupo de classes acessíveis mediante o uso de uma única classe.

Estrutura



Descrição

`Aplicação1` e `Aplicação2` faz uso apenas de `Façade`, mediante uma chamada ao método `metodo()`, para acesso aos subsistemas para os quais a "fachada" foi implementada.

Exemplo de Código

```
12 // Façade
13 public class DataFacade {
14    // Subsistema A
15    private Calendar         cal = Calendar.getInstance();
16    // Subsistema B
17    private SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
18
19    public DataFacade(String data) throws ParseException {
20       // Subsistema A
21       Date d = sdf.parse(data);
22       cal.setTime(d);
23    } // public DataFacade(String) throws ParseException
24
25    public void add(int dias) { cal.add(Calendar.DAY_OF_MONTH,dias); }
26
27    public void subtract(int dias) { add(dias * -1); }
28
29    @Override
30    public String toString() { return sdf.format(cal.getTime()); }
31 } // public class DataFacade
```

Exemplo na API Java

A classe `java.net.URL` é usada como uma "fachada" (*Façade*) para um recurso qualquer na Internet.

Observações

A utilização deste padrão de projeto permite a aplicação do princípio denominado **Princípio do Menor Conhecimento** (***Principle of Least Knowledge***). Este princípio atesta que se deve cuidar com o número de classes com as quais um determinado objeto se relaciona e como este relacionamento é realizado. Evitar o alto acoplamento garante que mudanças terão pouco ou nenhum impacto na relação entre objetos.

Padrões Relacionados

Abstract Factory e *Mediator*.

Anti-Pattern

A realização de uma venda implica em uma série de operações necessárias à sua efetivação. Estas operações podem ser, por exemplo, a baixa no estoque de produtos, a verificação de viabilidade financeira do cliente para efetuar a compra e a realização de lançamentos nos registros de contas a receber e de caixa. Neste caso, o que se deseja é a efetivação da venda, mas este processo está fortemente acoplado a outros subsistemas.

Prototype – Criação

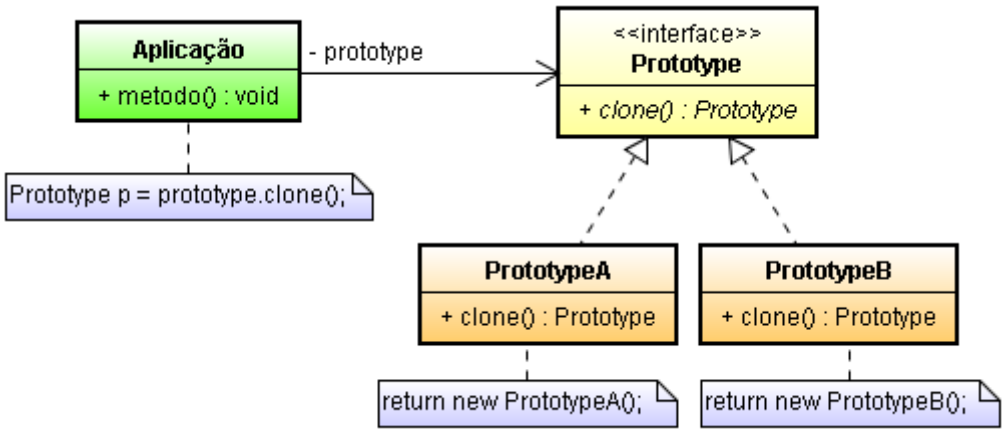
Apresentação

Especifica os tipos de objetos a criar usando uma instância de protótipo, criando-os mediante cópia (ou clonagem) deste protótipo. Esta operação é realizada sem que a aplicação saiba qual classe específica está sendo instanciada.

Aplicabilidade

Em situações nas quais criar instâncias de objetos a partir de hierarquias de classes complexas e quando a classe de um objeto é conhecida somente em tempo de execução são exemplos do uso deste padrão de projeto.

Estrutura



Descrição

A utilização deste padrão de projeto dá-se pela existência de uma classe base (ou interface) contendo a declaração do método `clone()` e, dependendo da implementação, mantém um coleção (na forma de um dicionário, por exemplo) de classes concretas derivadas da classe base (ou que implementem a interface).

Exemplo de Código

```
 7 // Prototype
 8 public interface Alguem {
 9    public Alguem duplicate();
10 } // public interface Alguem
```

```
 9 // ConcretePrototypeA
10 public class Beltrano implements Alguem {
11    @Override
12    public Alguem duplicate() { return new Beltrano(); }
13
14    @Override
15    public String toString() { return "Beltrano"; }
16 } // public class Beltrano implements Alguem
```

```
 9 // ConcretePrototypeB
10 public class Cicrano implements Alguem {
11    @Override
12    public Alguem duplicate() { return new Cicrano(); }
13
14    @Override
15    public String toString() { return "Cicrano"; }
16 } // public class Cicrano implements Alguem
```

```
 9 // ConcretePrototypeC
10 public class Fulano implements Alguem {
11    @Override
12    public Alguem duplicate() { return new Fulano(); }
13
14    @Override
15    public String toString() { return "Fulano"; }
16 } // public class Fulano implements Alguem
```

Exemplo na API Java

Este padrão de projeto é a essência da especificação JavaBean. A interface `java.lang.Cloneable` existe para que classes a implementem, e, assim, atestem que podem ser clonadas, mediante a sobre-escrita do método `clone()`, disponível na classe `java.lang.Object`.

Observações

Eventualmente, padrões de projeto de criação competem entre si em uso. Às vezes pode-se utilizar ou o padrão de projeto *Prototype* ou *Abstract Factory*. Enquanto em outros casos, ambos são complementos um do outro.

Padrões Relacionados

Abstract Factory, *Composite* e *Decorator*.

Anti-Pattern

O uso deste padrão de projeto destina-se a criar cópias (clones) a partir de uma instância denominada protótipo. Assim, o seu *anti-pattern* é a situação na qual toda vez que uma cópia de um determinado objeto é necessária sua criação deve ser realizada, comprometendo, possivelmente, o desempenho da aplicação e a quantidade de memória para comportar um número elevado de objetos.

Decorator – Estrutura

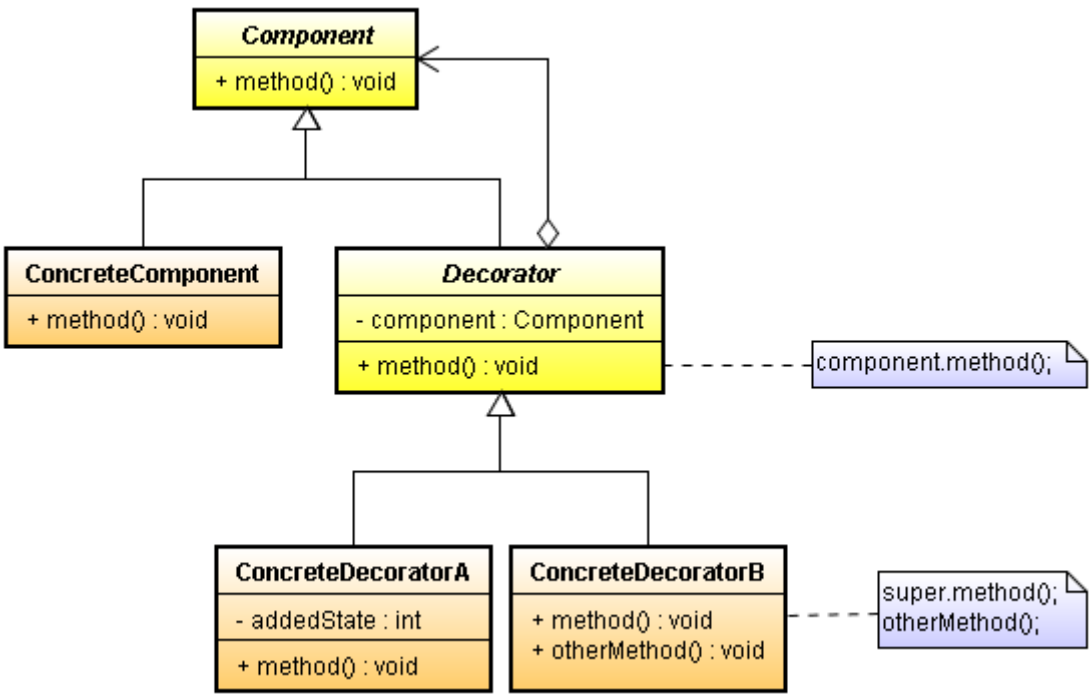
Apresentação

Este padrão de projeto possibilita o acréscimo de funcionalidades a um objeto, e não a uma classe, em tempo de execução, provendo uma alternativa bem flexível como mecanismo de extensão de classes. O acréscimo de tais funcionalidades é realizado pelo encadeamento de um conjunto de objetos, sendo que os primeiros tratam das novas funcionalidades e o último da funcionalidade original.

Aplicabilidade

O uso deste padrão de projeto torna possível estender (decorar) a funcionalidade de uma classe em tempo de execução, ao contrário do mecanismo de extensão, que acrescenta funcionalidade em tempo de compilação. Isto acontece quando a classe decoradora cria um “invólucro” na classe original recebendo como parâmetro em sua construção um objeto da classe original.

Estrutura



Descrição

Objetos `Component` podem ser usados por si só ou decorados e cada `Decorator` possui uma referência a `Component`, que é estendido (classe abstrata) por `Decorator`. Além desta referência, `ConcreteDecorators` podem ampliar o estado de `Component`, acrescentando novos métodos, por exemplo. A ampliação do estado de `Component`, com o acréscimo de novos métodos, normalmente é realizada antes ou depois dos métodos de `Component`.

Exemplo de Código

```
 9 // Decorator
10 public abstract class Decorator extends ItemBibliotecaAbstract {
11    protected ItemBibliotecaAbstract item;
12
13    public Decorator(ItemBibliotecaAbstract item) { this.item = item; }
14
15    @Override
16    public void mostrar() { item.mostrar(); }
17 } // public abstract class Decorator extends ItemBibliotecaAbstract
```

```
12 // ConcreteDecorator
13 public class Emprestimo extends Decorator {
14    private List<String> devedores = new ArrayList<String>();
15
16    public Emprestimo(ItemBibliotecaAbstract item) { super(item); }
17
18    public void emprestar(String devedor) {
19       devedores.add(devedor);
20       item.setExemplares(item.getExemplares() - 1);
21    } // public void emprestar(String)
22
23    public void devolver(String devedor) {
24       devedores.remove(devedor);
25       item.setExemplares(item.getExemplares() + 1);
26    } // public void devolver(String)
27
28    @Override
29    public void mostrar() {
30       super.mostrar();
31       for (String devedor : devedores)
32          System.out.println("Devedor " + devedor);
33    } // public void mostrar()
34 } // public class Emprestimo extends Decorator
```

```java
// Component
public abstract class ItemBibliotecaAbstract {
   protected int exemplares;

   // setter
   public void setExemplares(int exemplares) { this.exemplares = exemplares; }
   // getter
   public int getExemplares() { return exemplares; }

   public abstract void mostrar();
} // public abstract class ItemBibliotecaAbstract
```

```java
// ConcreteComponent
public class Livro extends ItemBibliotecaAbstract {
   private String titulo;
   private String autor;

   public Livro(String titulo, String autor, int exemplares) {
      this.titulo     = titulo;
      this.autor      = autor;
      this.exemplares = exemplares;
   } // public Livro(String, String, int)

   @Override
   public void mostrar() {
      System.out.println(getClass().getSimpleName() + ":\nTitulo " + titulo +
                         "\nAutor " + autor + "\nExemplares " + exemplares);
   }
} // public class Livro extends ItemBibliotecaAbstract
```

```java
// ConcreteComponent
public class Periodico extends ItemBibliotecaAbstract {
   private String titulo;
   private String editora;
   private int    volume;
   private int    edicao;

   public Periodico(String titulo, String editora, int volume, int edicao,
                    int exemplares) {
      this.titulo     = titulo;
      this.editora    = editora;
      this.volume     = volume;
      this.edicao     = edicao;
      this.exemplares = exemplares;
   } // public Periodico(String, String, int)

   @Override
   public void mostrar() {
      System.out.println(getClass().getSimpleName() + ":\nTitulo " + titulo +
                         "\nEditora " + editora + "\nVolume " + volume +
                         "\nEdicao " + edicao + "\nExemplares " + exemplares);
   }
} // public class Periodico extends ItemBibliotecaAbstract
```

Exemplo na API Java

As classes de entrada e saída de fluxos (*I/O stream*) contidas no pacote `java.io` são um bom exemplo de classes decorativas. Ou seja, é possível, a

partir de um objeto criado para um *stream* decorá-lo com um objeto que converte bytes em caracteres e, por fim, decorá-lo com um objeto que recupera um conjunto de caracteres, como é apresentado no quadro a seguir:

```
BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream("receitas.doc")));
```

Observações

Padrões Relacionados

Adapter, *Composite* e *Strategy*.

Anti-Pattern

Em um cenário que conta com um conjunto de classes derivas de uma mesma classe pode ser difícil ou mesmo impraticável criar uma nova classe que faça uso de funcionalidades de classes existentes nesta hierarquia. Mesmo que isto seja possível, a proliferação de classes acabaria por criar uma quantidade muito grande de classes com funcionalidades repetidas.

Builder – Criação

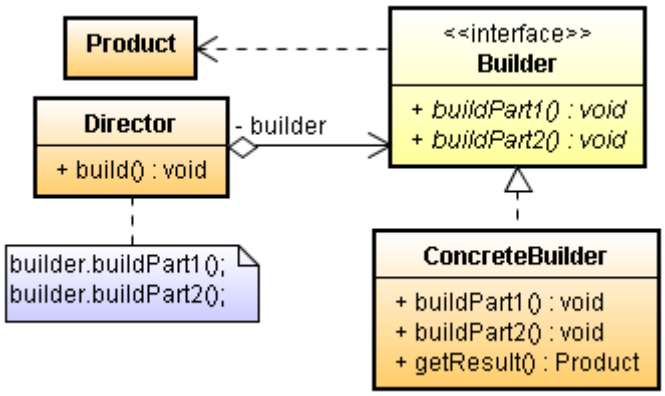
Apresentação

Este padrão de projeto é apropriado quando se deseja criar um “produto” em partes. O cliente, utilizando um objeto *builder* pede que o objeto “produto” seja criado. O objeto *builder* invoca um conjunto de métodos que possibilita a criação do objeto “produto”, que é, então, retornado. Assim, é possível que um objeto complexo possa ser criado sem que a aplicação (cliente) tenha conhecimento de como suas várias partes são criadas.

Aplicabilidade

Na criação de objetos complexos que devem ser “montados” independentes de suas partes ou de sua montagem. Além disto, o processo de construção deve permitir diferentes representações para o objeto que é construído.

Estrutura



Descrição

O objeto `Director` realiza a construção de um objeto `Product` mediante a construção de suas várias partes, descritas pela interface `Builder` e que são implementadas por `ConreteBuilder`.

Exemplo de Código

```
 7 // Director
 8 public class VeiculoDirector {
 9    public void buildVeiculo(VeiculoBuilder vb) {
10       vb.buildCarroceria();
11       vb.buildMotor();
12       vb.buildPortas();
13       vb.buildRodas();
14    } // public void buildVeiculo(VeiculoBuilder)
15 } // public class VeiculoDirector
```

```
 9 // Builder
10 public abstract class VeiculoBuilder {
11    protected Veiculo veiculo;
12
13    public Veiculo getVeiculo() { return veiculo; }
14
15    public abstract void buildCarroceria();
16    public abstract void buildMotor();
17    public abstract void buildRodas();
18    public abstract void buildPortas();
19 } // public abstract class VeiculoBuilder
```

```
10 // ConcreteBuilder
11 public class Carro extends VeiculoBuilder {
12    @Override
13    public void buildCarroceria() {
14       veiculo = new Veiculo("Carro");
15       veiculo.addParte("carroceria","Carroceria de carro de passeio");
16    }
17
18    @Override
19    public void buildMotor() { veiculo.addParte("motor","2000 HP"); }
20
21    @Override
22    public void buildRodas() { veiculo.addParte("rodas","4"); }
23
24    @Override
25    public void buildPortas() { veiculo.addParte("portas","2"); }
26 } // public class Carro extends VeiculoBuilder
```

```
10 // ConcreteBuilder
11 public class Minivan extends VeiculoBuilder {
12    @Override
13    public void buildCarroceria() {
14       veiculo = new Veiculo("Minivan");
15       veiculo.addParte("carroceria","Carroceria de Sprint");
16    }
17
18    @Override
19    public void buildMotor() { veiculo.addParte("motor","4800 HP"); }
20
21    @Override
22    public void buildRodas() { veiculo.addParte("rodas","6"); }
23
24    @Override
25    public void buildPortas() { veiculo.addParte("portas","3"); }
26 } // public class Minivan extends VeiculoBuilder
```

Exemplo na API Java

Entre os vários exemplos, podem ser citadas as classes `javax.xml.parsers.DocumentBuilder` e `java.lang.ProcessBuilder`. A primeira define uma API para obter um objeto DOM (*Document Object Model*) a partir de um documento XML (*eXtensible Markup Language*) e a segunda possibilita a criação de processos junto ao sistema operacional.

Observações

Padrões Relacionados

Abstract Factory e *Composite*.

Anti-Pattern

O *anti-pattern* deste padrão de projeto lida com o problema da criação de objetos complexos de forma aleatória e espalhada pelo código. Se um novo objeto precisa ser criado a modificação na aplicação para realizar esta nova operação é onerosa em termos de tempo e propensa à inclusão de erros em código que já havia sido testado e validado.

Template Method – Comportamento

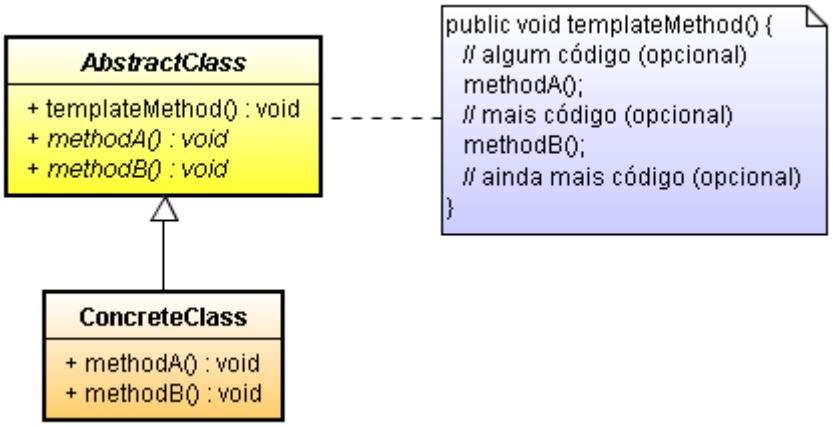
Descrição

Padrão de projeto cuja função é generalizar um processo, em um nível mais genérico, em um conjunto de passos, permitindo que etapas comuns sejam implementadas e que etapas específicas tenham suas implementações realizadas por classes descendentes. Em outras palavras, este padrão permite que subclasses de uma classe comum possam variar partes de um algoritmo mais geral.

Aplicabilidade

Partes de um algoritmo genérico são implementadas em uma classe comum a partir da qual suas subclasses podem ou devem implementar as partes específicas.

Estrutura



Descrição

A classe `AbstractClass` possui o método `templateMethod()` e as assinaturas de métodos abstratos, usados pelo método *template*, que devem ser implementados por classes descendentes. Uma vez que a classe contendo o método com os passos comuns (ou genéricos) tenha sido criada, várias classes concretas podem existir, cada uma implementando as partes específicas do algoritmo.

Algumas vezes um algoritmo genérico pode-se valer de passos que, se não implementados por classes descendentes, podem não existir ou são padronizados. Neste caso, as classes descendentes podem escolher ou não realizar suas implementações.

Outras vezes, dependendo do projeto, `AbstractClass` pode ser, na verdade, uma classe concreta, cujos passos específicos de um algoritmo podem ou devem ser implementadas por classes descendentes.

Exemplo de Código

```
 8 // AbstractClass
 9 public abstract class Venda {
10    private   String estadoDestino;
11    protected double icms;
12
13    public Venda() {}
14
15    public double getIcms() { return icms; }
16
17    public void finalizarVenda() {
18       // algum código necessário é finalização da venda, porém executado ANTES
19       // do cálculo do ICMS.
20       calcularICMS();
21       // algum código necessário é finalização da venda, porém executado DEPOIS
22       // do cálculo do ICMS.
23    } // public void finalizarVenda()
24
25    public abstract void calcularICMS();
26 } // public abstract class Venda
```

```
 9 // ConcreteClass
10 public class VendaPR extends Venda {
11    @Override
12    public void calcularICMS() { icms = 0.1; }
13 } // public class VendaPR extends Venda
```

```
10 public class VendaRS extends Venda {
11    @Override
12    public void calcularICMS() { icms = 0.2; }
13 } // public class VendaRS extends Venda
```

Exemplo na API Java

Observações

Um dos conceitos mais importantes em projetos de software trata do reuso de código e o padrão de projeto *Template Method* é fundamental neste

quesito, pois código (passos de um algoritmo) necessário à execução de uma tarefa pode ser deixado a cargo de classes que têm como base uma classe comum que realizará a tarefa.

Padrões Relacionados

Factory Method e *Strategy*.

Anti-Pattern

Se o código necessário a uma tarefa é composto de partes que são específicas a determinadas situações o programador escreverá estas partes utilizando um conjunto de **if-else-if-else...**. Mais uma vez, uma atitude como esta torna o código final um emaranhado de condições, que continuarão a crescer à media que novas condições forem impostas como requisitos à aplicação.

Iterator – Comportamento

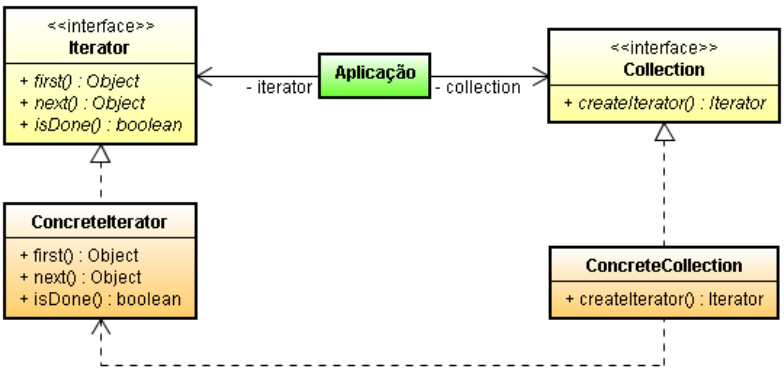
Apresentação

Este padrão de projeto fornece uma interface comum para que uma coleção de objetos possa ser percorrida sem, no entanto, que a aplicação tome conhecimento de como tais objetos estão agrupados.

Aplicabilidade

A utilização deste padrão de projeto possibilita à aplicação ter acesso ao conteúdo de um objeto agregado sem que sua representação interna seja exposta. A coleção de objetos pode ser percorrida em ambas as direções, de acordo com a declaração da interface.

Estrutura



Descrição

`Collection` é a interface comum que disponibiliza a assinatura do método `createIterator()`, entre outros, que deve ser implementado pelas classes descendentes e cujas implementações permitem o percorrimento da coleção de objetos (agregado). Assim, cada coleção concreta implementa o algoritmo de percorrimento de acordo com a representação de seus dados, retornando à aplicação uma referência à interface `Iterator`, que apresenta uma interface comum utilizada para percorrer da coleção de objetos.

Exemplo de Código

```
 7 // Iterator
 8 public interface IteratorGOF {
 9    public Object first();
10    public Object next();
11    public boolean isDone();
12 } // public interface IteratorGOF
```

```
 9 // ConcreteCollection
10 public class Vetor {
11    private static final int DEFAULT_SIZE = 16;
12    private Object[] valores;
13    private int      atual, total_geral, total_atual;
14
15    public Vetor() { this(DEFAULT_SIZE); }
16    public Vetor(int size) {
17       atual = total_atual = 0;
18       total_geral = size;
19       valores     = new Object[total_geral];
20    } // public Vetor()
21
22    public void add(Object obj) {
23       if (atual >= total_geral) {
24          total_geral += DEFAULT_SIZE;
25          Object[] temp = new Object[total_geral];
26          System.arraycopy(valores,0,temp,0,valores.length);
27          valores = temp;
28       } // if (atual >= total_geral)
29       valores[atual++] = obj;
30       ++total_atual;
31    } // public void add(Object)
32
33    public IteratorGOF createIterator() { return new VetorIterator(); }
34    // ConcreteIterator
35    private class VetorIterator implements IteratorGOF {
36       private int indice = Integer.MAX_VALUE;
37       @Override
38       public Object first() { indice = 0; return next(); }
39       @Override
40       public Object next() { return valores[indice++]; }
41       @Override
42       public boolean isDone() { return indice > total_atual; }
43    } // private class VetorIterator implements IteratorGOF
44 } // public class Vetor
```

Exemplo na API Java

A interface `Enumeration` e `Iterator`, ambas presentes no pacote `java.util`, são utilizadas pelas classes de coleção para fornecer um interface comum de percorrimento de seus objetos. Além do nome mais curto, a interface `Iterator` possui um método que permite a remoção de elementos da coleção.

Observações

Padrões Relacionados

Composite, *Factory Method* e *Memento*.

Anti-Pattern

Neste caso, o *anti-pattern* é implementado diretamente na classe que descreve a coleção de objetos, deixando que a aplicação tenha acesso à representação destes dados.

Composite – Estrutura

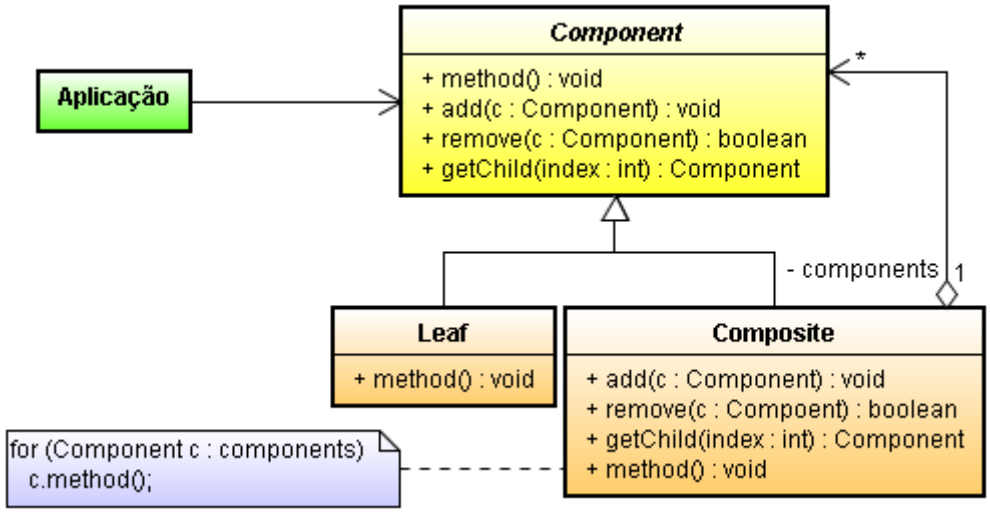
Apresentação

Objetos são compostos em uma estrutura semelhante a uma “árvore” para representar hierarquias **todo-parte**. Assim, este padrão de projeto permite que elementos primitivos e composições de elementos sejam tratados de forma semelhante.

Aplicabilidade

Quando se deseja modelar relações de componentes e estes, por sua vez, também são constituídos de outros componentes.

Estrutura



Descrição

A aplicação cliente faz uso de `Component` para ter acesso aos objetos na composição. A interface `Component`, por sua vez, define uma interface padrão para todos os objetos na composição, tanto para elementos simples – ou primitivos – (`Leaf`), quanto para elementos compostos (`Composite`). `Leaf` implementa os métodos para os elementos da composição, enquanto `Composite` implementa os métodos relacionados aos elementos na composição.

Tanto `Leaf` quanto `Composite` implementam `Component`, mas, no entanto, alguns dos métodos de `Component` não fazem sentido a `Leaf`, da mesma forma que outros não fazem sentido a `Component`. Neste caso, é possível realizar uma implementação padrão, como lançar uma exceção, por exemplo, nos métodos definidos em `Component`. Assim, os métodos que interessam a `Leaf` serão sobrescritos, da mesma forma serão sobrescritos os que interessam a `Composite`. Se um método que não possui relação com `Leaf`, por exemplo, for invocado, a exceção será lançada.

Exemplo de Código

```
// Component
public abstract class Component {
   public abstract void      add(Component component);
   public abstract boolean   remove(Component component);
   public abstract Component getChild(int index);

   public abstract void      processar();
} // public abstract class Component
```

```
// Leaf
public class Funcionario extends Component {
   private String     nome;
   private BigDecimal salario;

   public Funcionario(String nome, BigDecimal salario) {
      setNome(nome);
      setSalario(salario);
   } // public Funcionario(String, BigDecimal)
   // setters
   public void setNome(String nome) { this.nome = nome; }
   public void setSalario(BigDecimal salario) { this.salario = salario; }
   // getters
   public String     getNome() { return nome; }
   public BigDecimal getSalario() { return salario; }

   @Override
   public void processar() { System.out.println(nome + " ganha " + salario); }
   // ------------------------------------------------------------------------
   // Fica a critério do desenvolvedor implementar os métodos que não interessam
   // ao elemento folha (leaf). Neste caso, uma exceção será lançada caso algum
   // dos métodos a seguir seja invocado.
   // ------------------------------------------------------------------------
   @Override
   public void add(Component component) {
      throw new UnsupportedOperationException();
   } // public void add(Component)
   @Override
   public boolean remove(Component component) {
      throw new UnsupportedOperationException();
   } // public boolean remove(Component)
   @Override
   public Component getChild(int index) {
      throw new UnsupportedOperationException();
   } // public Component getChild(int)
} // public class Funcionario extends Component
```

```
11 // Composite
12 public class Departamento extends Component {
13    private String          nome;
14    private List<Component> funcs;
15
16    public Departamento(String nome) {
17       setNome(nome);
18       funcs = new ArrayList<Component>();
19    } // public Departamento(String)
20    // setters
21    public void setNome(String nome) { this.nome = nome; }
22    // getters
23    public String getNome() { return nome; }
24
25    public void      add(Component c) { funcs.add(c); }
26    public boolean   remove(Component c) { return funcs.remove(c); }
27    public Component getChild(int index) { return funcs.get(index); }
28
29    @Override
30    public void processar() {
31       System.out.println("Departamento: " + nome);
32       for (Component f : funcs) f.processar();
33    } // public void processar()
34 } // public class Departamento extends Component
```

Exemplo na API Java

Os componentes gráficos (classes) `JFrame`, `JDialog` e `JPanel`, disponíveis no pacote `javax.swing` e que fazem parte da API Swing, são exemplos de componentes do tipo contêiner, aqueles que armazenam uma coleção de componentes gráficos, como, por exemplo, `JButton` e `JTextField`, entre outros. No caso de `JPanel`, alem de contêiner, ele também é um componente que pode ser adicionado a um `JFrame`, por exemplo.

Observações

O padrão de projeto *Composite* pode ser dividido, basicamente, em dois grupos: **Composição de Relação** e **Composição Taxonômica**. No primeiro caso, a relação todo-parte dá-se pela composição de objetos complexos feita de objetos menores, que podem, também, ser compostos de objetos ainda menores. Já no segundo caso, a relação de composição existe entre objetos de tipos que são mais ou menos gerais.

A Composição Taxonômica geralmente é utilizada quando se deseja percorrer os seus elementos, encontrar alguma informação ou processar um

elemento com base em algum contexto. Este processamento é realizado pela aplicação cliente.

A Composição de Relação é implementada levando-se em consideração o comportamento desejado, o que terá diferentes implicações quanto ao encapsulamento, que pode ser: **Comportamento por Agregação** cuja implementação do comportamento está vinculada ao percorrimento de todos os elementos filhos para que se possa realizar o processamento da composição; **Comportamento Caso Melhor/Pior** quando a implementação do comportamento leva em consideração o processamento dos elementos filhos e, caso algum não satisfaça determinada condição, a composição como um todo não a satisfará também; **Comportamento Investigatório** caso em que a aplicação cliente realizará o percorrimento na composição, pois somente esta saberá se um determinado elemento satisfaz ou não um dado critério.

Padrões Relacionados

Decorator, *Flyweight*, *Iterator* e *Visitor*.

Anti-Pattern

No *anti-pattern* deste padrão de projeto a composição é substituída por diversas referências aos elementos que uma dada classe deve possuir. Para que esta classe possa processar um novo elemento, outra referência deve ser acrescentada, o que, muito provavelmente, acarretará a mudança no processamento das varias partes.

State – Comportamento

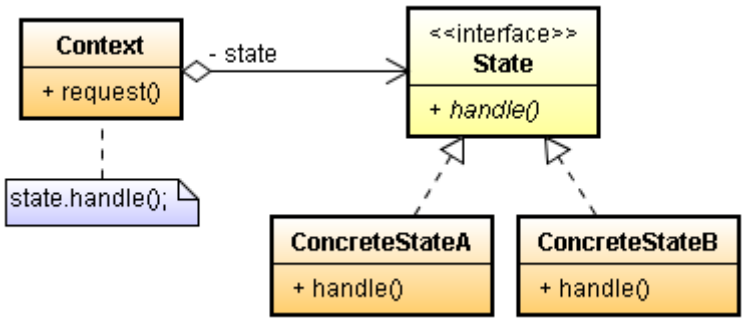
Descrição

Este padrão de projeto permite que um objeto modifique seu comportamento de acordo com a mudança de seu estado interno. Assim, tem-se a impressão que o objeto muda sua classe.

Aplicabilidade

O uso deste padrão de projeto dá-se quando o comportamento de um objeto depende de seu estado e este comportamento deve se modificar em tempo de execução (*runtime*) de acordo com este estado e, também, quando muitas declarações condicionais (**if-else-if-else**) dependem do estado do objeto. Neste caso, cada operação, que é realizada de acordo com um valor constante, deve ser transferida para uma classe própria. Esta transferência permite tratar o estado do objeto como um objeto por si só.

Estrutura



Descrição

`Context` pode ser qualquer classe na aplicação que contém vários estados internos. Quando o método `request()` é invocado sobre o objeto contexto, o processamento é delegado ao objeto estado (`State`). Assim, os estados concretos (`ConcreteStateA`, `ConcreteStateB`) tratam (manipulam) as requisições provenientes do contexto e os estados concretos têm suas próprias implementações para o tratamento da requisição. Desta forma,

quando `Context` muda seu estado (o(s) valor(es) de seu(s) atributo(s) é(são) alterado(s)), seu comportamento também muda.

Exemplo de Código

```
// Context
public class Encadeamento {
    private State atual;

    public Encadeamento() { atual = new DesligadoState(); }
    public void setState(State s) { atual = s; }
    public void trocar() { atual.trocar(this); }
} // public class Encadeamento
```

```
// State
public abstract class State {
   public void trocar(Encadeamento en) {
      en.setState(new DesligadoState());
      System.out.println("desligando");
   } // public void trocar(Encadeamento)
} // public abstract class State
```

```
// ConcreteState
public class BaixoState extends State {
   @Override
    public void trocar(Encadeamento en) {
       en.setState(new MedioState());
       System.out.println("velocidade media");
   } // public void trocar(Encadeamento)
} // public class BaixoState extends State
```

Exemplo na API Java

Observações

Padrões Relacionados

Flyweight.

Anti-Pattern

Classes com extensos códigos condicionais para tratarem seus diversos estados são o exemplo de *anti-pattern* apresentado para este padrão de projeto.

Strategy – Comportamento

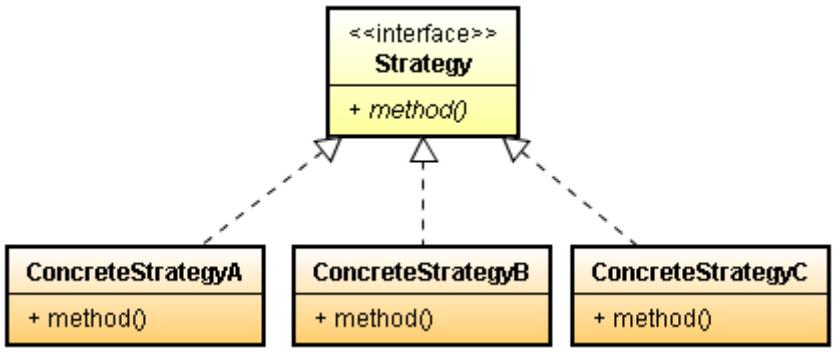
Apresentação

Este padrão de projeto define uma família de algoritmos, os encapsula e os torna intercambiáveis. Estes algoritmos podem variar independente da aplicação cliente que os usa.

Aplicabilidade

Este padrão de projeto provê uma forma de configurar uma classe com um de vários comportamentos ou algoritmos. Além disto, é possível esconder da aplicação as estruturas de dados, possivelmente complexas, utilizadas pelos diversos algoritmos. Isto permite que diferentes regras de negócios possam ser utilizadas dependendo do contexto no qual elas ocorram, significando que este padrão de projeto fornece um meio de configurar uma classe com um entre vários comportamentos.

Estrutura



Descrição

Encapsula algoritmos em classes, permitindo que eles possam variar independente da aplicação cliente que os usa. Isto possibilita a aplicação do princípio denominado **Princípio Aberto-Fechado** (***Open-Closed Principle***), no qual uma classe está aberta para extensões enquanto mantém-se fechado para alterações. Tal princípio é alcançado pelo uso deste padrão de projeto, pois a aplicação cliente faz uso de uma classe base (classe abstrata ou interface) e os detalhes de implementação estão presentes em classes

derivadas. A aplicação cliente não sofre impacto no caso do aumento de classes derivadas, nem tampouco com eventuais mudanças que essas classes venham a sofrer. Com isto, a aplicação cliente minimiza o acoplamento, pois mantém um vínculo com a abstração (classe abstrata ou interface) e não com uma implementação específica.

Exemplo de Código

```
 7 // Strategy
 8 public abstract class DatabaseStrategy {
 9    protected String server;
10    protected String databaseURL;
11    protected int    port;
12    protected String user;
13    protected String password;
14
15    public DatabaseStrategy(String server, String databaseURL, int port,
16                            String user, String password) {
17       this.server      = server;
18       this.databaseURL = databaseURL;
19       this.port        = port;
20       this.user        = user;
21       this.password    = password;
22    } // public DatabaseStrategy(String, String, int, String, String)
23
24    public abstract void   loadDriver() throws ClassNotFoundException;
25    public abstract String generateURL();
26    public abstract String format(Object obj);
27 } // public abstract class DatabaseStrategy
```

```
10 // ConcreteStrategy
11 public class SQLServerStrategy extends DatabaseStrategy {
12    private static final String JDBC_DRIVER =
13                                  "com.microsoft.jdbc.sqlserver.SQLServerDriver";
14    private static final String JDBC_URL    = "jdbc:microsoft:sqlserver://";
15    private static final int    JDBC_PORT   = 1433;
16
17    private static final SimpleDateFormat dateFormat =
18                                new SimpleDateFormat("''MM-dd-yyyy HH:mm:ss''");
19
20    public SQLServerStrategy(String server, String databaseURL, int port,
21                             String user, String password) {
22       super(server,databaseURL,port,user,password);
23    } // public SQLServerStrategy(String, String, int, String, String)
24
25    public SQLServerStrategy(String server, String databaseURL, String user,
26                             String password) {
27       super(server,databaseURL,JDBC_PORT,user,password);
28    } // public SQLServerStrategy(String, String, int, String, String)
29
```

Exemplo na API Java

As classes `CheckInputStream` e `CheckOutputStream`, ambas presentes no pacote `java.util.zip`, utilizam este padrão de projeto para calcular *checksums*[4] no fluxo de bytes lidos e gravados.

Os componentes presentes na AWT (*Abstract Window Toolkit*), tais como botões, menus e listas, entre outros, são posicionados em componentes do tipo contêiner, como painéis, janelas de diálogo e janelas, por exemplo. Este posicionamento não é realizado pelos contêineres, mas sim delegado aos gerenciadores de *layout*. Tal delegação é um exemplo do padrão de projeto *Strategy*.

Observações

Padrões Relacionados

Flyweight.

Anti-Pattern

A utilização de estruturas de seleção do tipo **if-else-if-else-if-else...** torna a aplicação altamente acoplada ao conjunto de algoritmos que necessita utilizar, além de não ser possível aplicar o **Princípio Aberto-Fechado**.

[4] *Checksum*s são valores produzidos por algoritmos matemáticos cujo objetivo é o de realizar verificação de dados.

Proxy – Estrutura

Apresentação

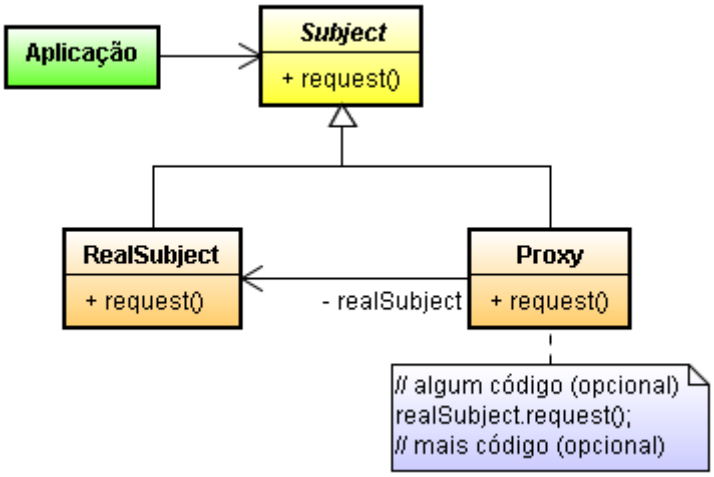
Este padrão de projeto provê acesso controlado a outros objetos por meio de um objeto substituto (ou procurador), denominado *proxy*.

Aplicabilidade

Normalmente, quando uma simples referência a um objeto não é suficiente necessitando-se de uma que seja mais sofisticada, pois o acesso ao objeto precisa ser controlado, possivelmente, verificando se os direitos de acesso o permitem.

Alguns tipos de *proxies* que podem, dependendo da situação, ser utilizados são: ***Proxy* Remoto**, ***Proxy* Virtual**, ***Proxy* de Proteção** e ***Proxy* de Referência Inteligente**.

Estrutura



Descrição

Tanto `RealSubject` quanto `Proxy` implementam a mesma interface, `Subject`, o que permite que a aplicação possa tratar objetos `Proxy` como se fossem objetos `RealSubject`. `RealSubject` realiza o trabalho verdadeiro, enquanto `Proxy` simplesmente controla o acesso a ele. O objeto `Proxy` mantém uma referência a `RealSubject`, instanciando-o ou cuidando de sua

criação. Desta forma, as requisições feitas ao objeto `Proxy` são redirecionadas ao objeto `RealSubject` de forma controlada.

Exemplo de Código

```
// Subject
public interface FileOperations {
   public boolean deleteFile(String fileName);
} // public interface FileOperations
```

```
// RealSubject
public class FileOperationsImpl implements FileOperations {
   @Override
   public boolean deleteFile(String fileName) {
      // EXCLUIR O ARQUIVO RETORNANDO true NO CASO DE SUCESSO OU false EM CASO
      // DE ERRO
      return false;
   } // public boolean deleteFile(String)
} // public class FileOperationsImpl implements FileOperations
```

```
// Proxy
public class FileOperationsProxy implements FileOperations {
   private FileOperations fo;

   public FileOperationsProxy() { this.fo = new FileOperationsImpl(); }

   @Override
   public boolean deleteFile(String fileName) {
      boolean canDelete = false;

      // se há permissão para excluir o arquivo
      //    entao
      canDelete = fo.deleteFile(fileName);
      // fim-se

      return canDelete;
   } // public boolean deleteFile(String)
} // public class FileOperationsProxy implements FileOperations
```

Exemplo na API Java

A classe `java.lang.reflect.Proxy` fornece métodos para a criação de classes e instâncias de *proxies* dinâmicos, sendo a super-classe para todos os *proxies* dinâmicos criados por estes métodos.

Observações

Um ***Proxy* Remoto** fornece uma referência a um objeto localizado em outro espaço de endereçamento, no mesmo computador ou em um computador conectado à rede. Normalmente, um *proxy* remoto é utilizado em

tecnologias de chamadas remotas a procedimentos (ou métodos), como acontece com RMI (*Remote Method Invocation*) em Java. O ***Proxy* Virtual** possibilita a um objeto que tenha um processo de criação custoso à aplicação ser criado somente quando for necessário. No ***Proxy* de Proteção** o acesso ao objeto real é controlado e cada cliente tem um nível diferenciado de acesso. O ***Proxy* de Referência Inteligente** permite controlar, por exemplo, a quantidade de referências feitas ao objeto real.

Padrões Relacionados

Adapter e *Decorator*.

Anti-Pattern

Quando a invocação de um método necessita que outras ações sejam tomadas antes que este possa efetivamente ser executado, o padrão de projeto *Proxy* é uma boa solução. Caso a aplicação tenha que tomar ciência (algum processamento, como, por exemplo, verificar direitos do usuário) das ações necessárias antes ou depois da invocação de um determinado método, isto a torna complexa, de difícil manutenção e extensibilidade se tal processamento for implementado na aplicação cliente.

Visitor – Comportamento

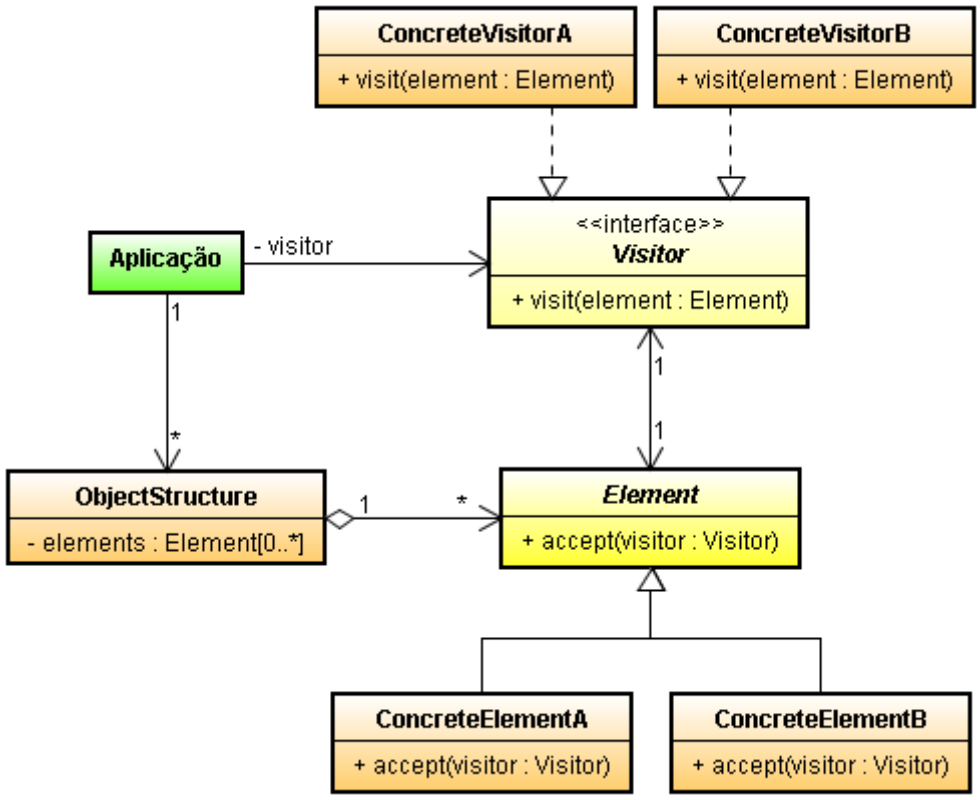
Descrição

Representa uma operação que é executada em uma estrutura de objetos. Novas operações são definidas sem, no entanto, conhecer as classes dos objetos sobre as quais elas serão executadas.

Aplicabilidade

Este padrão de projeto pode ser utilizado quando uma estrutura de objetos contém muitas classes com interfaces diferentes e se deseja executar operações distintas ou sem relação entre si. Isto evita a “poluição” de código e mantém códigos relacionados em uma única classe. Além disto, normalmente a freqüência de mudanças nas estruturas de objetos é pequena, enquanto que novas operações podem ser acrescentadas à medida que os requisitos se modificam.

Estrutura



Descrição

Este padrão de projeto é aplicado quando há necessidade de se executar operações semelhantes em objetos de diferentes tipos agrupados em uma estrutura, possivelmente uma coleção. Ele também fornece um meio prático para o acréscimo de novos *visitors* que estendem funcionalidades pré-existentes sem que o código seja modificado.

Exemplo de Código

```
12 // ObjectStructure
13 public class Empregados {
14    private List<Empregado> empregados = new ArrayList<Empregado>();
15
16    public void addEmpregado(Empregado emp) { empregados.add(emp); }
17    public void removeEmpregado(Empregado emp) { empregados.remove(emp); }
18
19    public void accept(Visitor visitor) {
20       for (Empregado empregado : empregados)
21          empregado.accept(visitor);
22    } // public void accept(Visitor)
23 } // public class Empregados
```

```
 8 // Visitor
 9 public interface Visitor {
10    public void visit(Elemento elemento);
11 } // public interface Visitor
```

```
11 // ConcreteVisitor
12 public class AssalariadoVisitor implements Visitor {
13    public void visit(Elemento elemento) {
14       Empregado empregado = (Empregado) elemento;
15
16       double salario = empregado.getSalario();
17       empregado.setSalario(empregado.getSalario() * 1.15);
18       System.out.println(empregado.getNome() + " (" +
19                          empregado.getClass().getSimpleName() + ") ganhando " +
20                          salario + " teve um reajuste de 15% e passou a " +
21                          "receber: " + empregado.getSalario());
22    } // public void visit(Elemento)
23 } // public class AssalariadoVisitor implements Visitor
```

```
 7 // Element
 8 public abstract class Elemento {
 9    public abstract void accept(Visitor visitor);
10 } // public abstract class Elemento
```

```
 7 // ConcretElement
 8 public class Diretor extends Empregado {
 9    public Diretor() { super("Pato Donald",1010.5,25); }
10 } // public class Diretor extends Empregado
```

Exemplo na API Java

Observações

O padrão de projeto *Visitor* possibilita acrescentar funcionalidades a bibliotecas de classes para as quais não há possibilidade de alteração no código fonte; obter dados de uma coleção de objetos não relacionados e apresentar resultados de um processamento global.

Padrões Relacionados

Composite e *Interpreter*.

Anti-Pattern

O *anti-pattern* deste padrão pode levar a um código rebuscado, confuso e de difícil manutenção e extensibilidade. Como exemplo, tem-se um sistema de vendas cujos produtos devem ser enviados aos clientes. Para que os envios sejam processados faz-se necessário à aplicação verificar, por exemplo, por meio do operador `instanceof`, qual o objeto venda sendo processado para que o envio adequado de seus produtos seja realizado. Se novas classes representando vendas e/ou novas classes representando envios forem acrescentadas ao sistema, a verificação apontada tornará o código maior e mais complexo.

Memento – Comportamento

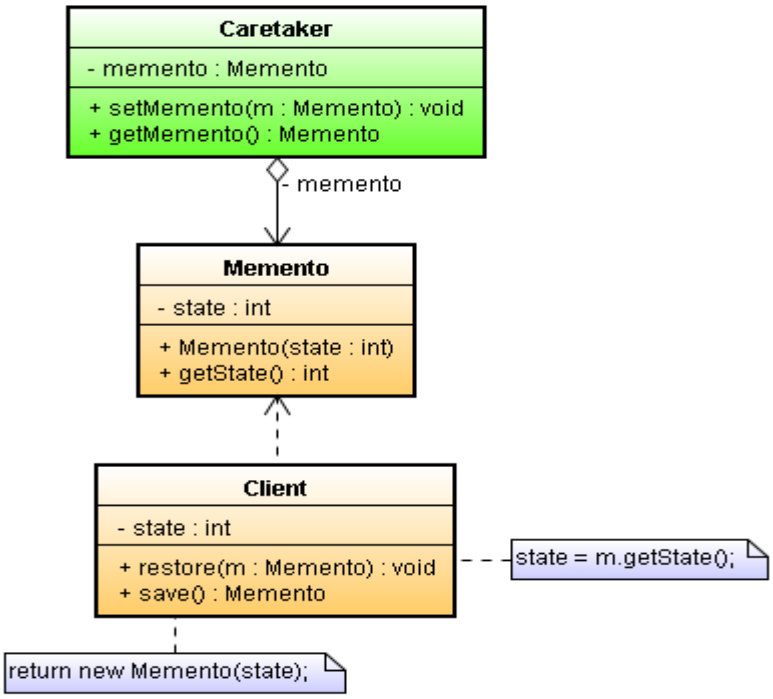
Apresentação

Este padrão de projeto baseia-se em dois objetos: o objeto `Originator` e o objeto `Caretaker`. O primeiro é qualquer objeto da aplicação, enquanto o segundo necessita realizar algum processamento no primeiro, mas precisa que o estado inicial possa ser restaurado. Desta forma, o conceito de encapsulamento não é violado, o que, de outra forma, exporia à aplicação os detalhes de implementação do objeto que sofrerá o processamento.

Aplicabilidade

Quando se deseja realizar algum processamento temporário em determinado objeto da aplicação, o padrão de projeto *Memento* é aplicado de tal forma que um "instantâneo" (*snapshot*) do estado (ou de parte deste) do objeto em questão seja salvo para ser restaurado posteriormente ao processamento realizado.

Estrutura



Descrição

O objeto `Caretaker` deve realizar algum processamento com o objeto `Originator`, possivelmente mudando seu estado. Antes que isto aconteça, o objeto `Caretaker` pede ao objeto `Originator` por uma referência a `Memento`, realiza o processamento desejado e devolve a referência `Memento` ao objeto `Originator` para que seu estado seja restaurado.

Exemplo de Código

```
 9 // Caretaker
10 public class Caretaker {
11    private Memento memento;
12
13    public void    setMemento(Memento memento) { this.memento = memento; }
14    public Memento getMemento() { return memento; }
15 } // public class Caretaker
```

```
 7 // Memento
 8 public class Memento {
 9    private String nome;
10    private String cidade;
11
12    public Memento(String nome, String cidade) {
13       this.nome   = nome;
14       this.cidade = cidade;
15    } // public Memento(String, String)
16    // getters
17    public String getNome() { return nome; }
18    public String getCidade() { return cidade; }
19 } // public class Memento
```

```
 9 // Client
10 public class Cliente {
11    private String nome;
12    private String cidade;
13
14    public Cliente(String nome, String cidade) {
15       setNome(nome);
16       setCidade(cidade);
17    } // public Cliente(String, String)
18    // setters
19    public void setNome(String nome) { this.nome = nome; }
20    public void setCidade(String cidade) { this.cidade = cidade; }
21    // getters
22    public String getNome() { return nome; }
23    public String getCidade() { return cidade; }
24
25    public Memento save() { return new Memento(nome,cidade); }
26
27    public void restore(Memento memento) {
28       setNome(memento.getNome());
29       setCidade(memento.getCidade());
30    } // public void restore(Memento)
31
32    @Override
33    public String toString() {
34       return getClass().getSimpleName() + " [" + nome + ", " + cidade + "]";
35    } // public String toString()
36 } // public class Cliente
```

Exemplo na API Java

Observações

A implementação deste padrão de projeto deve ser realizada com cautela, pois o objeto `Originator` pode modificar outros objetos e este padrão de projeto age sobre um único objeto.

Padrões Relacionados

Command e *Iterator*.

Anti-Pattern

Para este padrão de projeto o seu *anti-pattern* é a falta da possibilidade de restauração do estado do objeto com o qual se deseja realizar algum processamento (`Originator`). Mesmo que a aplicação acrescente a possibilidade de restauração de estado de um dado objeto sem a aplicação deste padrão de projeto, ainda seria o uso do *anti-pattern*, pois a aplicação precisaria expor os detalhes de implementação do objeto em questão.

Mediator – Comportamento

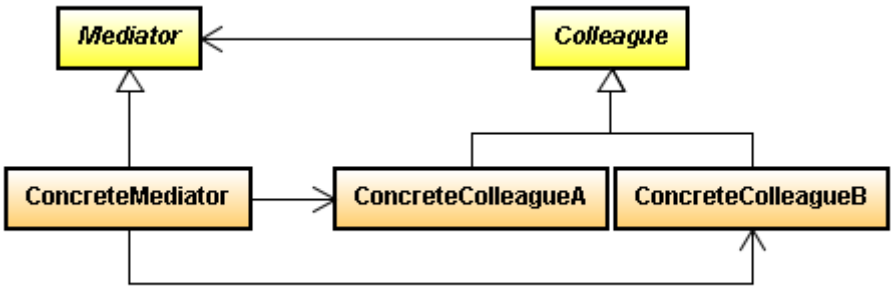
Apresentação

Este padrão de projeto provê uma interface unificada a um conjunto de interfaces em um subsistema, fazendo com que a comunicação entre os vários objetos do subsistema seja realizada por um objeto. Desta forma, os objetos do subsistema deixam de manter relações entre si, reduzindo, assim, o acoplamento.

Aplicabilidade

A aplicabilidade para este padrão de projeto dá-se quando se quer separar a comunicação que vários objetos têm entre si através de uma interface bem definida, diminuindo, assim, suas interdependências.

Estrutura



Descrição

Objetos `Colleague` são desacoplados uns dos outros. As classes concretas de `Colleague` “conversam” com a implementação concreta de `Mediator`, que, por sua vez, conduz a troca de mensagens entre objetos `Colleague`.

Exemplo de Código

```
 7 // Mediator
 8 public class Mediator {
 9    private boolean slotFull = false;
10    private int     number;
11
12    public synchronized void storeMessage(int num) {
13       while (slotFull == true)
14          try { wait(); }
15          catch (InterruptedException ex) {}
16       slotFull = true;
17       number   = num;
18       notifyAll();
19    } // public synchronized void storeMessage(int)
20
21    public synchronized int retrieveMessage() {
22       while (slotFull == false)
23          try { wait(); }
24          catch (InterruptedException ex) {}
25       slotFull = false;
26       notifyAll();
27
28       return number;
29    } // public synchronized int retrieveMessage()
30 } //public class Mediator
```

```
 9 // ConcreteColleague
10 public class Producer extends Thread {
11    private Mediator mediator;
12    private int      id;
13
14    private static int num = 1;
15
16    public Producer(Mediator mediator) {
17       this.mediator = mediator;
18       this.id       = num++;
19    } // public Producer(Mediator)
20
21    @Override
22    public void run() {
23       int number = 0;
24
25       while (true) {
26          mediator.storeMessage(number = (int) (Math.random() * 100));
27          System.out.println("Produtor[" + id + "] " + number);
28       } // while (true)
29    } // public void run()
30 } // public class Producer extends Thread
```

```
 9 // ConcreteColleague
10 public class Consumer extends Thread {
11    private Mediator mediator;
12    private int      id;
13
14    private static int num = 1;
15
16    public Consumer(Mediator mediator) {
17       this.mediator = mediator;
18       this.id = num++;
19    } // public Consumer(Mediator)
20
21    @Override
22    public void run() {
23       while (true)
24          System.out.println("Consumidor[" + id + "] " +
25                             mediator.retrieveMessage());
26    } // public void run()
27 } // public class Consumer extends Thread
```

Exemplo na API Java

Observações

Este padrão de projeto, bem como os padrões *Chain of Responsibility*, *Command* e *Observer* realizam o desacoplamento entre objetos **remetentes** e **destinatários**, mas com diferenças. O primeiro passa o remetente ao longo de uma cadeia de destinatários em potencial. O segundo descreve uma conexão entre remetente e destinatário mediante o uso de uma subclasse. E o último especifica uma interface que realiza um alto desacoplamento entre vários destinatários, que serão notificados de alguma mudança.

Padrões Relacionados

Façade e *Observer*.

Anti-Pattern

O *anti-pattern* para este padrão de projeto trata do alto acoplamento entre classes que necessitam se relacionar entre si. A evolução e manutenção deste tipo de código trazem uma série de dificuldades, entre elas a leitura impraticável do código para reconhecer quem se relaciona com quem; o acréscimo de uma nova classe que necessita manter um relacionamento com outras classes que já se relacionam.

Bridge – Estrutura

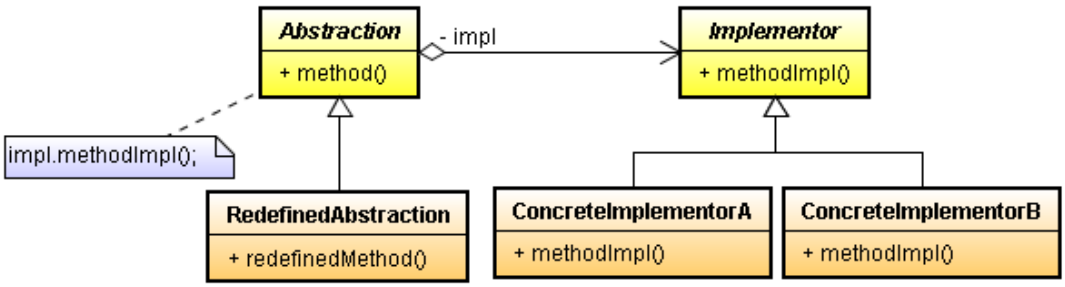
Apresentação

Este padrão de projeto possibilita que **abstrações** sejam desacopladas de suas **implementações**, permitindo que ambas possam variar independentemente umas das outras. Este padrão é útil não só quando uma dada classe varia, mas quando o que ela faz também varia. Neste sentindo, a classe em si é vista como a **implementação** e o que ela faz como a **abstração**.

Aplicabilidade

Em casos quando se deseja que a implementação seja selecionada em tempo de execução evitando o seu vínculo com a abstração. Quando abstração e implementação devem ser estendidas por meio de classes derivadas permitindo combinar diferentes abstrações e implementações.

Estrutura



Descrição

A relação entre `Abstraction` e `Implementor` é conhecida como *bridge* e todos os métodos na abstração são implementados em função da implementação, pois `Abstraction` possui uma referência para `Implementor`, e todas as classes concretas de `Abstraction` são declaradas em termos de `Abstraction` e não de `Implementor`.

Exemplo de Código

```
10  // Abstraction
11 public class ClienteNavigator {
12    private DataNavigator dn;
13
14    public DataNavigator getClienteNavigator() { return dn; }
15    public void          setClienteNavigator(DataNavigator dn) { this.dn = dn; }
16
17    public void insert(Object c) { dn.insert(c); }
18    public void update(Object c) { dn.update(c); }
19    public void delete(Object c) { dn.delete(c); }
20    public List select() { return dn.select(); }
21 } // public class ClienteNavigator
```

```
14 // RefinedAbstraction
15 public class Clientes extends ClienteNavigator {
16    public void mostrar() {
17       List clientes = select();
18
19       ListIterator li = clientes.listIterator();
20       while (li.hasNext()) {
21          Cliente c = (Cliente) li.next();
22          System.out.println(c);
23       } // while (li.hasNext())
24    } // public void mostrar()
25 } // public class Clientes extends ClienteNavigator
```

```
10 // Implementor
11 public abstract class DataNavigator {
12    public abstract void insert(Object t);
13    public abstract void update(Object t);
14    public abstract void delete(Object t);
15    public abstract List select();
16 } // public abstract class DataNavigator
```

```
13 // ConcreteImplementor
14 public class ClienteDataNavigator extends DataNavigator {
15
16    private List clientes = new ArrayList();
17
18    @Override
19    public void insert(Object c) { clientes.add(c); }
20
21    @Override
22    public void update(Object c) {
23       int index = findById(((Cliente) c).getId());
24
25       if (index > -1) clientes.set(index,c);
26    } // public void update(Cliente)
27
28    @Override
29    public void delete(Object c) {
30       int index = findById(((Cliente) c).getId());
31
32       if (index > -1) clientes.remove(index);
33    } // public void delete(Cliente)
34
35    @Override
36    public List select() { return new ArrayList(clientes); }
37
38    private int findById(int id) {
39       for (int i = 0; i < clientes.size(); i++) {
40          Cliente c = (Cliente) clientes.get(i);
41          if (c.getId() == id) return i;
42       } // for (int i = 0; i < clientes.size(); i++)
43
44       return -1; // nao encontrado
45    } // private int findById(int)
46 } // public class ClienteDataNavigator extends DataNavigator
```

Exemplo na API Java

Observações

Apesar de os padrões de projeto *State*, *Strategy*, *Bridge* e, em menor grau, *Adapter* terem estruturas semelhantes, eles são aplicados para a solução de diferentes problemas.

Padrões Relacionados

Abstract Factory e *Adapter*.

Anti-Pattern

Quando uma implementação está diretamente vinculada à sua abstração tem-se o *anti-pattern* para o padrão de projeto *Bridge*, pois se novos requisitos surgirem, tanto para a abstração quanto para a implementação, a proliferação de classes aumentará muito, tornado a evolução da aplicação difícil e trabalhosa.

Flyweight – Estrutura

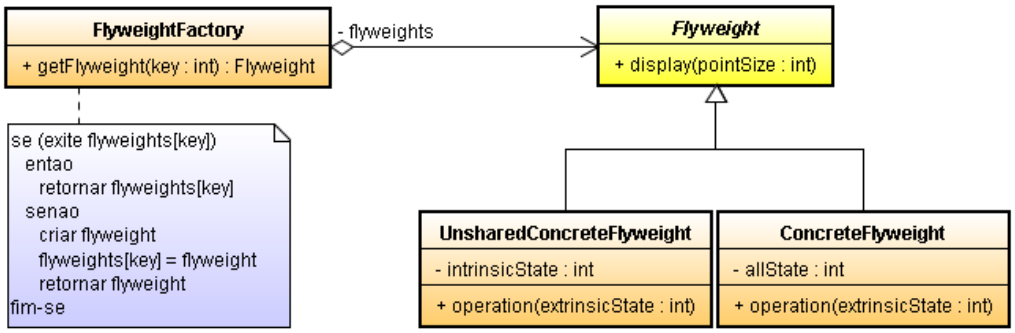
Apresentação

Este padrão de projeto se propõe às aplicações que necessitam de grandes quantidades de dados, porém usando o compartilhamento de objetos, evitando, assim, o consumo (muitas alocações) de memória.

Aplicabilidade

Aplica-se este padrão de projeto quando há a necessidade de se compartilhar um grande volume de dados. A aplicação cria os dados uma vez e os compartilha através de um ou mais objetos.

Estrutura



Descrição

`Flyweight` declara uma interface comum a partir da qual classes concretas receberão e atuarão sobre estado extrínseco (aquele que é dependente do contexto no qual o objeto está atuando). A classe (fábrica) `FlyweightFactory` cria e cuida de objetos `Flyweight`, que são compartilhados. Quando a aplicação requisita um objeto `Flyweight`, a fábrica devolve a instância existente ou cria uma. `ConcreteFlyweight` implementa `Flyweight` para que o estado intrínseco (aquele que é independente do estado no qual o objeto está atuando) possa ser armazenado, pois este objeto deve ser compartilhado. Às vezes, é possível que uma subclasse de `Flyweight` não precise ser compartilhada. Neste caso,

`UnsharedConcreteFlyweight` faz este papel e é comum que esta implementação tenha objetos `ConcreteFlyweight` como seus filhos.

Exemplo de Código

```
14 // FlyweightFactory
15 public class CharacterFactory {
16    private Map<Character, AbstractCharacter> characters =
17                                    new HashMap<Character, AbstractCharacter>();
18
19    public AbstractCharacter getCharacter(char key) {
20       AbstractCharacter character = characters.get(key);
21
22       if (character == null) {
23          switch (key) {
24             case 'A' : character = new CharacterA(); break;
25             case 'B' : character = new CharacterB(); break;
26             case 'Z' : character = new CharacterZ(); break;
27          } // switch (key)
28          characters.put(key,character);
29       } // if (character == null)
30
31       return character;
32    } // public AbstractCharacter getCharacter(char)
33 } // public class CharacterFactory
```

```
 7 // Flyweight
 8 public abstract class AbstractCharacter {
 9    protected char symbol;
10    protected int  width;
11    protected int  height;
12    protected int  ascent;
13    protected int  descent;
14    protected int  pointSize;
15
16    public abstract void display(int pointSize);
17 } // public abstract class AbstractCharacter
```

```
 9 // ConcreteFlyweight
10 public class CharacterA extends AbstractCharacter {
11    public CharacterA() {
12       symbol  = 'A';
13       width   = 120;
14       height  = 100;
15       ascent  = 70;
16       descent = 0;
17    } // public CharacterA()
18
19    @Override
20    public void display(int pointSize) {
21       this.pointSize = pointSize;
22       System.out.println(symbol + " (pointSize " + pointSize + ")");
23    } // public void display(int)
24 } // public class CharacterA extends AbstractCharacter
```

Exemplo na API Java

A classe `java.lang.String` é um exemplo da aplicação deste padrão de projeto, pois a Máquina Virtual Java (JVM – *Java Virtual Machine*) faz uso do mesmo objeto `String` para representar cadeias de caracteres que são compostas pela mesma seqüência de caracteres, desde que não tenham sido passadas ao construtor de `java.lang.String`.

Observações

Em comparação com o padrão de projeto *Façade*, que permite que um único objeto represente um subsistema completo, o padrão de projeto *Flyweight* permite criar uma grande quantidade de pequenos objetos.

Padrões Relacionados

Composite, *State* e *Strategy*.

Anti-Pattern

A cada processamento uma quantidade pequena de informações é necessária à sua realização. Se a cada realização deste processamento novos objetos para estas informações forem criados, a aplicação terá seu desempenho comprometido e haverá uma alta redundância de objetos contendo o mesmo estado.

Interpreter – Comportamento

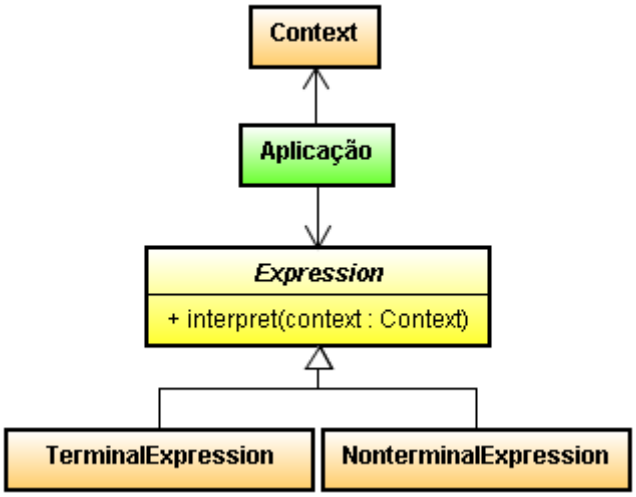
Apresentação

Padrão de projeto utilizado para modelar a gramática para uma linguagem específica a um domínio de problemas.

Aplicabilidade

Analisadores de expressões regulares, expressões algébricas e partituras musicais (um dado som e sua duração), além de linguagens de consulta (*query language*) e protocolos de comunicação são exemplos da aplicabilidade deste padrão de projeto.

Estrutura



Descrição

`AbstractExpression` declara a operação comum a todos os elementos na árvore de sintaxe abstrata, enquanto `TerminalExpression` implementa tal operação para um determinado elemento terminal (que não pode ser definido em termos de outros elementos). `NonterminalExpression` implementa a mesma operação de forma que esta possa ser chamada recursivamente para cada símbolo na gramática e `Context` possui a informação que é global ao interpretador. Como exemplo, tem-se, a seguir, a gramática que define expressões regulares:

```
expressão ::= literal | alternativa | seqüência |
              repetição | ‘(’ expressão ‘)’
alternativa ::= expressão ‘|’ expressão
seqüência ::= expressão ‘&’ expressão
repetição ::= expressão ‘*’
literal ::= ‘a’ | ‘b’ | ‘c’ | ... { ‘a’ | ‘b’ | ‘c’ | ... }*
```

Exemplo de Código

```
 7 // Context
 8 public class Context {
 9    private String in;
10    private int    out;
11
12    public Context(String in) { setIn(in); }
13    // setters
14    public void setIn(String in) { this.in = in; }
15    public void setOut(int out) { this.out = out; }
16    // getters
17    public String getIn() { return in; }
18    public int    getOut() { return out; }
19 } // public class Context
```

```
 7 // Expression
 8 public abstract class Expression {
 9    public void interpret(Context context) {
10       if (context.getIn().isEmpty()) return;
11
12       if (context.getIn().startsWith(nine())) {
13         context.setOut(context.getOut() + (9 * multiplier()));
14         context.setIn(context.getIn().substring(2));
15       } // if (context.getIn().startsWith(nine()))
16       else
17          if (context.getIn().startsWith(four())) {
18             context.setOut(context.getOut() + (4 * multiplier()));
19             context.setIn(context.getIn().substring(2));
20          } // if (context.getIn().startsWith(four()))
21          else
22             if (context.getIn().startsWith(five())) {
23                context.setOut(context.getOut() + (5 * multiplier()));
24                context.setIn(context.getIn().substring(1));
25             } // if (context.getIn().startsWith(five()))
26
27       while (context.getIn().startsWith(one())) {
28          context.setOut(context.getOut() + (1 * multiplier()));
29          context.setIn(context.getIn().substring(1));
30       } // while (context.getIn().startsWith(one()))
31    } // public void interpret(Context)
32
33    public abstract String one();
34    public abstract String four();
35    public abstract String five();
36    public abstract String nine();
37    public abstract int    multiplier();
38 } // public abstract class Expression
```

```
 9 // TerminalExpression
10 public class OneExpression extends Expression {
11    @Override
12    public String one() { return "I"; }
13    @Override
14    public String four(){ return "IV"; }
15    @Override
16    public String five(){ return "V"; }
17    @Override
18    public String nine(){ return "IX"; }
19    @Override
20    public int multiplier() { return 1; }
21 } // public class OneExpression extends Expression
```

Exemplo na API Java

Observações

Este padrão de projeto é bem pouco utilizado em aplicações de um modo geral, sendo mais usado em aplicações específicas, nas quais se faz necessário a criação de uma linguagem para um determinado domínio de aplicação.

Padrões Relacionados

Composite, *Flyweight*, *Iterator* e *Visitor*.

Anti-Pattern

A definição de uma linguagem para um domínio é um problema de solução complexa. Assim, torna-se difícil o desenvolvimento de uma aplicação que não use este padrão de projeto, visto que ele possui uma aplicação bem específica.

REFERÊNCIAS

AntiPatterns. <http://www.antipatterns.com/> Acesso 27/04/2008.

FREEMAN, Eric; FREEMAN, Elisabeth; BATES, Bert; SIERRA, Kathy. ***Head First Design Patterns*** O'Reilly: 2004.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. ***Design Patterns: elements of reusable object-oriented software*** Addison-Wesley: 1994.

HOUSTON, Vince. ***Design Patterns*** <http://www.vincehuston.org/dp/> Acesso 27/04/2008.

LARMAN, Craig. ***Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and Iterative Development***. 3 ed. Addison Wesley Professional: 2004.

Net Objectives ***The Net Objectives Pattern Repository*** <http://www.netobjectivesrepository.com/> Acesso 27/04/2008.

PENDER, Tom. ***UML Bible***. Indianapolis, Indiana, EUA John Wiley & Sons: 2003.

SHALLOWAY, Alan; TROTT, James R.. ***Design Patterns Explained: a new perspective on object-oriented design*** Addison-Wesley: 2004.

SourceMaking. ***AntiPatterns*** <http://sourcemaking.com/antipatterns/> Acesso 27/04/2008.

SourceMaking. ***Design Patterns*** <http://sourcemaking.com/design_patterns/> Acesso 27/04/2008.

Wikipedia: The Free Encyclopedia. ***Design pattern (computer science)*** <http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29> Acesso 27/04/2008.

Wikipedia: The Free Encyclopedia. ***Anti-pattern*** <http://en.wikipedia.org/wiki/Anti-pattern> Acesso 27/04/2008.

Wikipedia: The Free Encyclopedia. ***Padrões de projeto de software*** <http://pt.wikipedia.org/wiki/Padr%C3%B5es_de_projeto_de_software> Acesso 27/04/2008.