Prototype, um Design Patterns de Criação

José Anízio Pantoja Maia

*Este artigo tem como finalidade compreender o funcionamento do padrão de projeto **prototype**, serão abordados os participantes que compõe este padrão, seus objetivos de uso, quando usá-lo, conseqüências de uso, suas desvantagens e um exemplo prático em Java.*

Introdução

Os *Design Patterns* (Padrões de Projeto, em português) são soluções genéricas para problemas que geralmente ocorrem no desenvolvimento de sistemas computacionais. Os mais conhecidos são os 23 *Patterns* catalogados pelos quatro amigos (**Eric Gamma, Richard Helm, Ralph Johnson e John Vlissides**), que ficaram conhecidos como a "**Gang dos Quatro**" (Gang of Four - **GoF**). Depois da publicação deste catalogo de *Patterns*, surgiram outros padrões, mas, a maior referência ainda é o conjunto de *Patterns* publicado pelos quatro amigos.

O **Pattern Prototype** está entre os 23 publicados pelo GoF, este *Pattern* está na categoria dos padrões de criação e nosso objetivo é estudar e compreender seu funcionamento com exemplo prático.

Descrição

O **Pattern Prototype** tem como objetivo criar objetos específicos a partir da instância de um protótipo. Isso permite criar novos objetos através da cópia deste protótipo.

O uso de células representa perfeitamente os objetivos deste *pattern*. Uma célula dividia em duas se tornam duas células idênticas com as mesmas funcionalidades.

Supondo que estamos construindo um *software* que clona células para um grupo de cientistas que já nos ensinaram como é o processo de clonar células. Projetando a definição de **prototype** acima, poderíamos ter a seguinte situação:
Agora vamos colaborar com estes "cientistas malucos", que criam frutas e animais mutantes e construir um *software* para clonar células. A estrutura para clonar células pode ser vista no diagrama de classe (Figura 1).

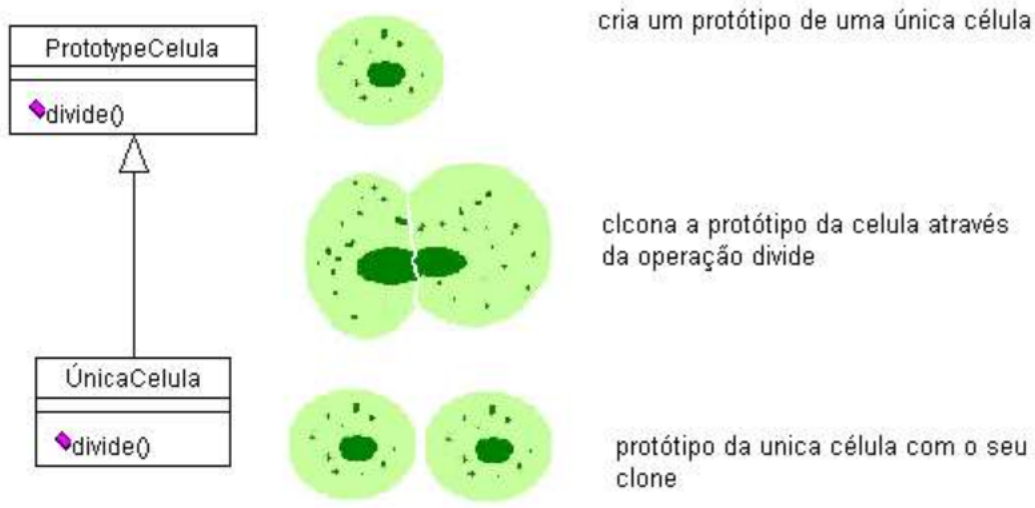

Figura 1: objetivos do pattern Prototype

Estrutura Genérica

A estrutura genérica trata-se de um modelo padrão proposto pelo GoF que mostra através do diagrama de classe (Figura 2) o funcionamento do **Prototype**. É chamada genérica porque a proposta deverá ser adaptada de acordo com o problema a ser aplicado, ou seja, não é obrigatório seguir exatamente o modelo, mas, usar a idéia e aplicar no problema que estamos tentando resolver.

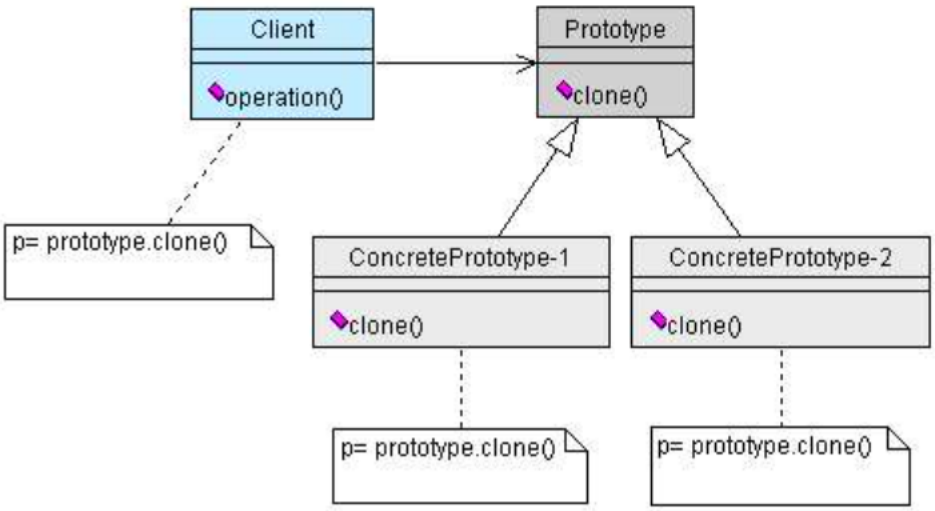

Figura 2: estrutura genérica

Descrição dos Participantes da Estrutura Genérica

1. **Cliente** – solicita a um protótipo que crie uma cópia de si mesmo, gerando outro objeto.
2. **Prototype** – específica uma interface para clonar a si próprio.
3. **Concrete Prototype** – implementa uma operação para clonar a si próprio.

Conseqüências do uso do Prototype

1. **Adição e remoção de produtos em tempo de execução** – incorpora uma nova classe concreta de produto a um sistema simplesmente registrando uma instância protótipo com o cliente.
2. **Especificação de novos objetos pela variação de valores** – define novos comportamentos através da composição de objetos, por exemplo, pela especificação de valores para as variáveis de um objeto e não pela definição de novas classes. Com isso, define-se novos tipos de objetos pela instanciação das classes existentes e registrando as instâncias como protótipo dos objetos-cliente. Um cliente pode exibir um novo comportamento através de delegação de responsabilidades para o protótipo.
3. **Redução do número de subclasses** – permite clonar um protótipo em vez de pedir a um método para construir um novo objeto. Então, não será necessário uma hierarquia de classe.

Quando usar Prototype

Devemos usá-lo quando precisamos criar instâncias de classe e essa tarefa acaba sendo demorada/custosa e complexa de algum modo. Então, no lugar de criar várias instâncias, criamos cópias da instância original e usamos de acordo com as necessidades.

Este *pattern* também pode ser usado sempre que precisamos de classes que só diferem o tipo de processamento que elas oferecem, por exemplo, no caso de um grande banco de dados onde precisamos fazer um número de questões para construir uma resposta. De posse desta resposta em uma tabela, poderíamos querer manipular esta tabela para produzir outras respostas sem emitir questões adicionais.

Outro caso como um banco de dados contendo um número grande de nadadores de uma liga ou organização qualquer. Cada nadador, nada vários metros de distâncias ao longo de seus períodos de treino, os melhores tempos dos nadadores são agrupado por idade, e muitos nadadores vão fazer aniversários em seus respectivos grupos de idade e período de treino. Assim, determinar quais nadadores fizeram o melhor tempo no grupo de idade e em que período de treino se encontra cada nadador aniversariante. O cálculo computacional para juntar esta tabela de tempos deverá ser bastante considerável.

Tendo uma classe que contenha esta tabela de tempos e ordenada por sexo, poderíamos imaginar tentar fazer uma buscar desta informação ordenando antes por tempo, ou por idade atual no lugar de grupo de idade. Não seria sensato re-computar estes dados, não queremos destruir os dados originais, mas, obter algum tipo de cópia do objeto de dados seria desejável.

Desvantagem

Cada subclasse de **Prototype** deve implementar a operação clone, o que pode ser difícil. Por exemplo, acrescentar clone é difícil quando as classes consideradas já existem. A implementação de clone pode ser complicada quando uma estrutura interna da classe inclui objetos que não suportam operação de cópia ou têm referencias circulares.

Exemplo Prático

Clonar uma célula foi somente um treino. Agora os cientistas malucos querem clonar um animal. Então vamos clonar um animal ovelha que se chamará "Dolly", eu acho que esse nome é familiar!

Aplicando a estrutura genérica do **Prototype** no problema, implementamos as seguintes classes:

Classe cliente – será responsável por fazer as solicitações de clones de ovelhas.

```
/**
 * @author anizio maia
 * Está é a classe cliente que solicita o clone de um objeto ovelha
 * Aqui é solicitado o clone de um objeto ovelha chamada Dolly
 */
class Cliente {
 public static void main(String[] args) {
        PrototypeFactory prototype = new PrototypeFactory(new Ovelha("Dolly"));
        //solicita o clone de uma ovelha
        Animal animal = prototype.criaClone();
        System.out.println("Nome do animal: " + animal.getNameAnimal());
   }
}
```

Classe PrototypeFactory – retorna o clone de objetos quando solicitado pela classe cliente.

```
/**
 * @author anizio maia
 * Está classe é responsável por criar e retornar clone de objetos
 */
public class PrototypeFactory {
	Animal prototypeAnimal;
	//construtor
	public PrototypeFactory(Animal animal) {
		prototypeAnimal = animal;
	}
	/**
	 * retorna um clone de um objeto
	 * @return clone - clone de um objeto
	 */
	public Animal criaClone() {
		return (Animal) prototypeAnimal.clone();
	}
}
```

Classe Ovelha – é o objeto que está sendo clonado. Ela implementa os métodos da classe abstrata Animal.

```
/**
 * @author anizio maia
 * Está classe é uma classe Animal, mas agora especializada
 * É um animal específico, um animal ovelha.
 */
public class Ovelha extends Animal {
	// construtor
	public Ovelha(String nameOvelha) {
		this.animalNome = nameOvelha;
		setNomeAnimal(animalNome);
		andar();
		comer();
	}
}
```

Classe Animal – É uma classe abstrata que implementa a interface **Cloneable** do Java que possui um método clone responsável por retornar clone de objetos quando implementado corretamente.

```
/**
 * @author anizio maia
 * Está classe implementa a classe Cloneable que fornece o método clone
 * responsável por clonar objetos
 */
public abstract class Animal implements Cloneable {
	String animalNome;
	public void setNomeAnimal(String animalNome) {
		this.animalNome = animalNome;
	}
	public String getNameAnimal() {
		return this.animalNome;
	}
	public void comer() {
		System.out.println(animalNome + " está comendo...");
	}
	public void andar() {
		System.out.println(animalNome + " está andando...");
	}

	/**
	 * método responsável por clonar objetos
	 */
	public Object clone() {
		Object object = null;
		try {
			object = super.clone();
		} catch (CloneNotSupportedException exception) {
			System.err.println("A Ovelha não foi clonada");
		}
		return object;
	}
}
```

Saída ao Compilar

Ao compilar este programa a saída deverá ser algo do tipo:

```
Dolly está andando...
Dolly está comendo...
Nome do animal: Dolly
```

Estrutura do Programa Exemplo

O diagrama de classe abaixo (Figura 3) mostra a estrutura do programa acima usado como exemplo.

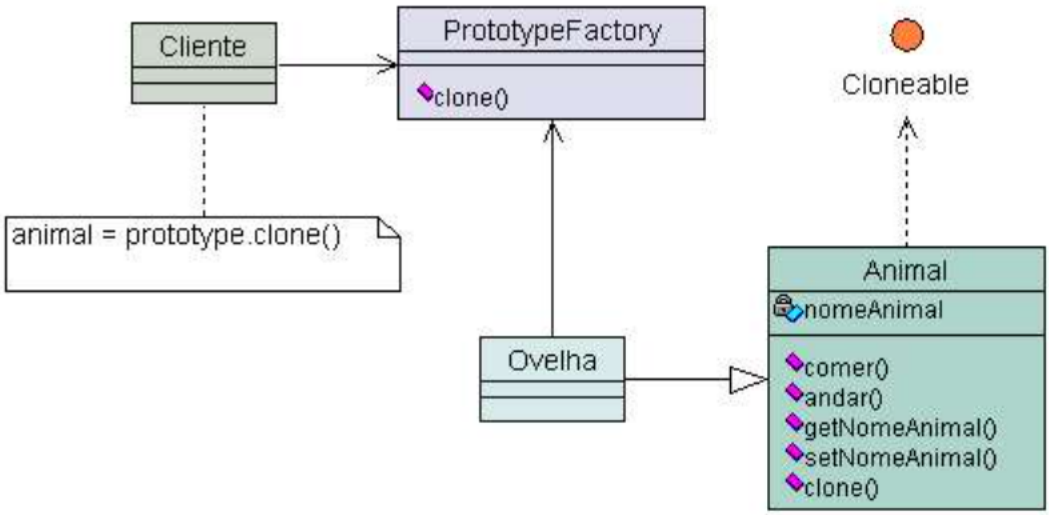

Figura 3: diagrama de classe do exemplo proposto

Sugestão de Treino para Fixar Conhecimento

Criar novos animais estendendo a classe abstrata **Animal**, depois tentar clonar estes objetos usando a fábrica de protótipos **PrototypeFactory**, não esquecendo de fazer solicitações de clone usando a classe **Cliente.**

Conclusão

Desde a publicação do primeiro livro sobre padrões de projeto, cada vez mais se tem falado sobre tais padrões, muitos projetistas e programadores se interessam pelo assunto, mas muitos ignoram por não saberem os benefícios que os patterns podem trazer, ou talvez porque gostem de reinventar a roda.

Prototype assim como os outros *Patterns* podem ser causadores de confiabilidade, reusabilidade e manutenibilidade de código, também podem ser capazes de economizar tempo e custo. Podemos perceber que as vantagens são maiores que as desvantagens, então, porque não usar?

José Anízio Pantoja Maia (gorran4@hotmail.com / anizio_maia@yahoo.com.br) graduado em Sistemas de Informação, participou a implantação e operação do SIPAM (Sistema de proteção da Amazônia), certificado em Java atualmente trabalha com desenvolvimento de dispositivos móveis usando a Plataforma Java Micro Edition e é um dos fundadores do JugManaus (http://www.jugmanaus.com).

Referencias:

GAMA, Erich, HELM, Richard, JOHNSON, Ralph, VLISSIDES, John: Padrões de Projeto Soluções Reutilizáveis de Software Orientado a Objetos. Ed. Bookman, Port Alegre, 2000.

http://www.fluffycat.com/java/JavaNotes-GoFPrototype.html