Design Patterns

Padrão

- *Maneira testada ou documentada de alcançar um objetivo qualquer*
  - *Padrões são comuns em várias áreas da engenharia*
- *Design Patterns, ou Padrões de Design**
  - *Padrões para alcançar objetivos na engenharia de software usando classes e métodos em linguagens orientadas a objeto*
  - *Inspirado em "A Pattern Language" de Christopher Alexander, sobre padrões de arquitetura de cidades, casas e prédios*

* *Ou ainda Padrões de Projeto, embora algo se perca nesta tradução!!*

Responsabilidade

- *Booch e Rumbaugh "Responsabilidade é um contrato ou obrigação de um tipo ou classe."*
- *Dois tipos de responsabilidades dos objetos:*
  - *De conhecimento (knowing): sobre dados privativos e encapsulados; sobre objetos relacionados; sobre coisas que pode calcular ou derivar.*
  - *De realização (doing): fazer alguma coisa em si mesmo; iniciar uma ação em outro objeto; controlar e coordenar atividades em outros objetos.*
- *Responsabilidades são atribuídas aos objetos durante o design*

Responsabilidades e Métodos

- *A tradução de responsabilidades em classes e métodos depende da granularidade da responsabilidade*
- *Métodos são implementados para cumprir responsabilidades*
  - *Uma responsabilidade pode ser cumprida por um único método ou uma coleção de métodos trabalhando em conjunto*
- *Responsabilidades do tipo knowing geralmente são inferidas a partir do modelo conceitual (são os atributos e relacionamentos)*

Responsabilidades e Diagramas

- *Diagramas de interação mostram escolhas ao atribuir responsabilidades a objetos*
  - *No diagrama de colaboração ao lado objetos Order têm a responsabilidade de se prepararem: método prepare()*
  - *O cumprimento dessa responsabilidade requer colaboração com objetos Order Line e Stock Item*

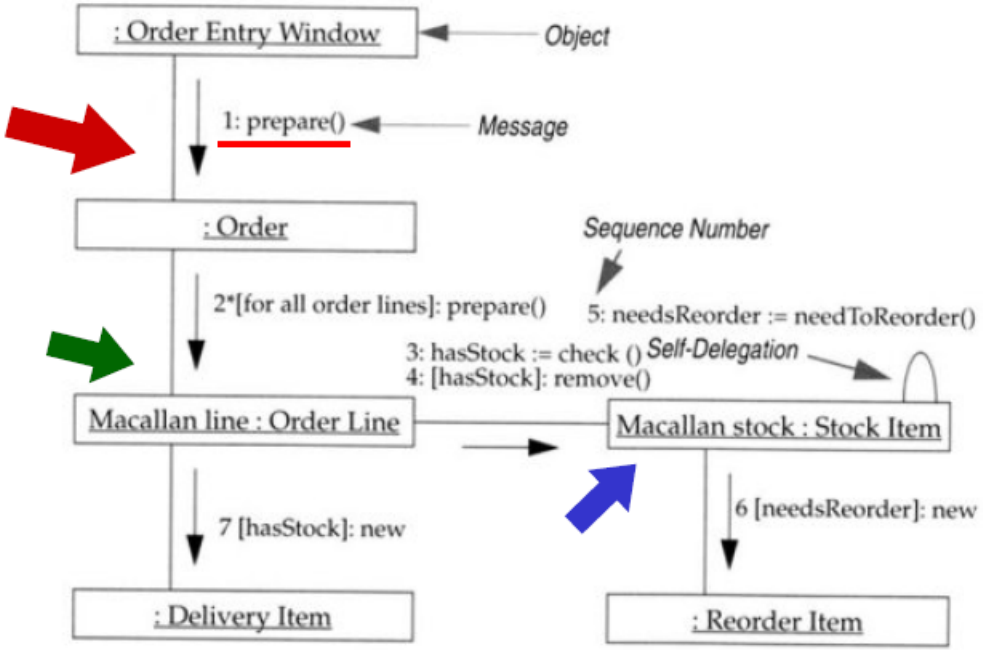


Figure 5-4: *Collaboration Diagram with Simple Numbering*

Padrões

- *Padrões são um repertório de soluções e princípios que ajudam os desenvolvedores a criar software e que são codificados em um formato estruturado consistindo de*
  - *Nome*
  - *Problema que soluciona*
  - *Solução do problema*
- *O objetivo dos padrões é codificar conhecimento (knowing) existente de uma forma que possa ser reaplicado em contextos diferentes*

GRASP

- *Introduzidos por Craig Larman em seu livro "Applying UML and Patterns" [4]*
- *GRASP: **G**eneral **R**esponsibility and **A**ssignment **S**oftware **P**atterns*
  - *Os padrões GRASP descrevem os princípios fundamentais da atribuição de responsabilidades a objetos, expressas na forma de padrões*
- *Esses padrões exploram os princípios fundamentais de sistemas OO*
  - *5 padrões fundamentais*
  - *4 padrões avançados*
- *Se você conhece os padrões GRASP, pode dizer que compreende o paradigma orientado a objetos*

Padrões Clássicos (GoF)

- *O livro "Design Patterns" (1994) de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, descreve 23 padrões de design [2]*
  - *São soluções genéricas para os problemas mais comuns do desenvolvimeto de software orientado a objetos*
  - *O livro tornou-se um clássico na literatura orientada a objeto e continua atual*
  - *Não são invenções. São documentação de soluções obtidas através da experiência. Foram coletados de experiências de sucesso na indústria de software, principalmente de projetos em C++ e SmallTalk*
  - *Os quatro autores, são conhecidos como "The Gang of Four", ou GoF*

Para que ?

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

Para que ?

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

Elementos

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

Classificação

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

Padrões GRASP focam neste objetivo

Classificação (GoF)

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

Classificação (Metsker)

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

Interface (Revisão)

- *Interface: coleção de métodos e dados que uma classe permite que objetos de outras classes acessem*
- *Implementação: código dentro dos métodos*
- *Interface Java: componente da linguagem que representa apenas a interface de um objeto*
  - *Exigem que classe que implementa a interface ofereça implementação para seus métodos*
  - *Não garante que métodos terão implementação que faça efetivamente alguma coisa (chaves vazias): stubs.*

Interfaces e Classes Abstratas (Revisão)

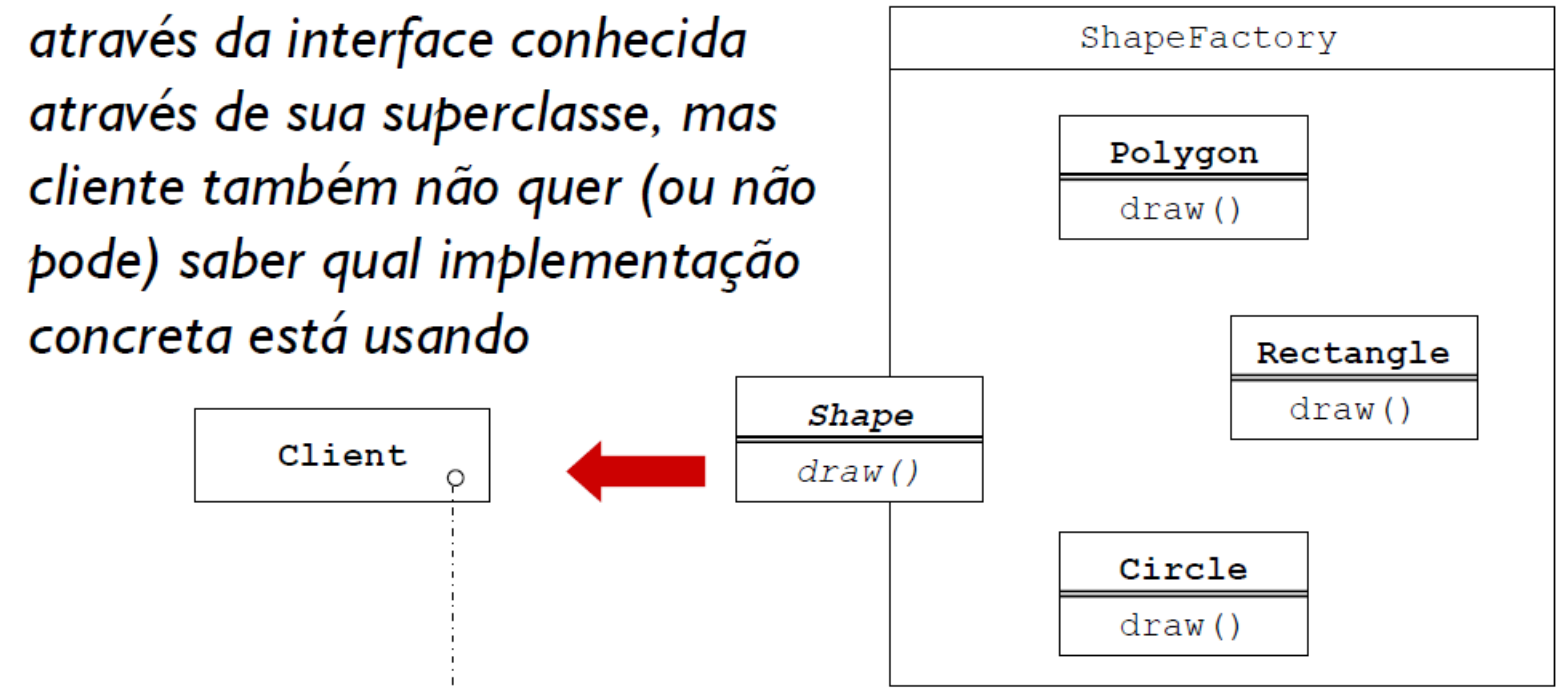
Interfaces x Classes Abstratas		
Característica	**Interface**	**Classe Abstrata**
Herança múltipla	Uma classe pode implementar diversas interfaces	Uma classe pode herdar somente uma classe
Implementação Padrão	Uma interface não pode conter qualquer tipo de código, muito menos código padrão.	Uma classe abstrata pode fornecer código completo , código padrão ou ter apenas a declaração de seu esqueleto para ser posteriormente sobrescrita.
Constantes	Suporte somente constantes do tipo estática.	Pode conter constantes estáticas e de instância.
Componentes de terceiros	Uma implementação de uma interface pode ser incluída a qualquer classe de terceiros.	Uma classe de terceiros precisa ser reescrita para estender somente a partir da classe abstrata.
Homogeneidade	Se todas as diversas implementações compartilham a assinatura do método então a interface funciona melhor.	Se as várias implementações são todas do tipo e compartilham um comportamento e status comum , então a classe abstrata funciona melhor.
Manutenção	Se o código do seu cliente conversa somente em termos de uma interface, você pode facilmente alterar a implementação concreta usando um método factory.	idêntico
Velocidade	Lento , requer trabalho extra para encontrar o método correspondente na classe atual.	Rápido
Clareza	Todas as declarações de constantes em uma interface são presumidamente publicas ou estáticas.	Você pode por código compartilhado em uma classe abstrata. Você pode usar código para computar o valor inicial de suas constantes e variáveis de instância ou estáticas.
Funcionalidades Adicionais	Se você incluir um novo método em uma interface você precisa ajustar todas as implementações da interface.	Se você incluir um novo método em uma classe abstrata você tem a opção de fornecer uma implementação padrão para ele.

Factory Method

"Definir uma interface para criar um objeto mas deixar que subclasses decidam que classe instanciar. FactoryMethod permite que uma classe delegue a responsabilidade de instanciamento às subclasses." [GoF]

Problema

O acesso a um objeto concreto será através da interface conhecida através de sua superclasse, mas cliente também não quer (ou não pode) saber qual implementação concreta está usando




```
~~Shape shape = new Rectangle();~~
Shape shape = ShapeFactory.getShape("rect");
shape.draw();
```

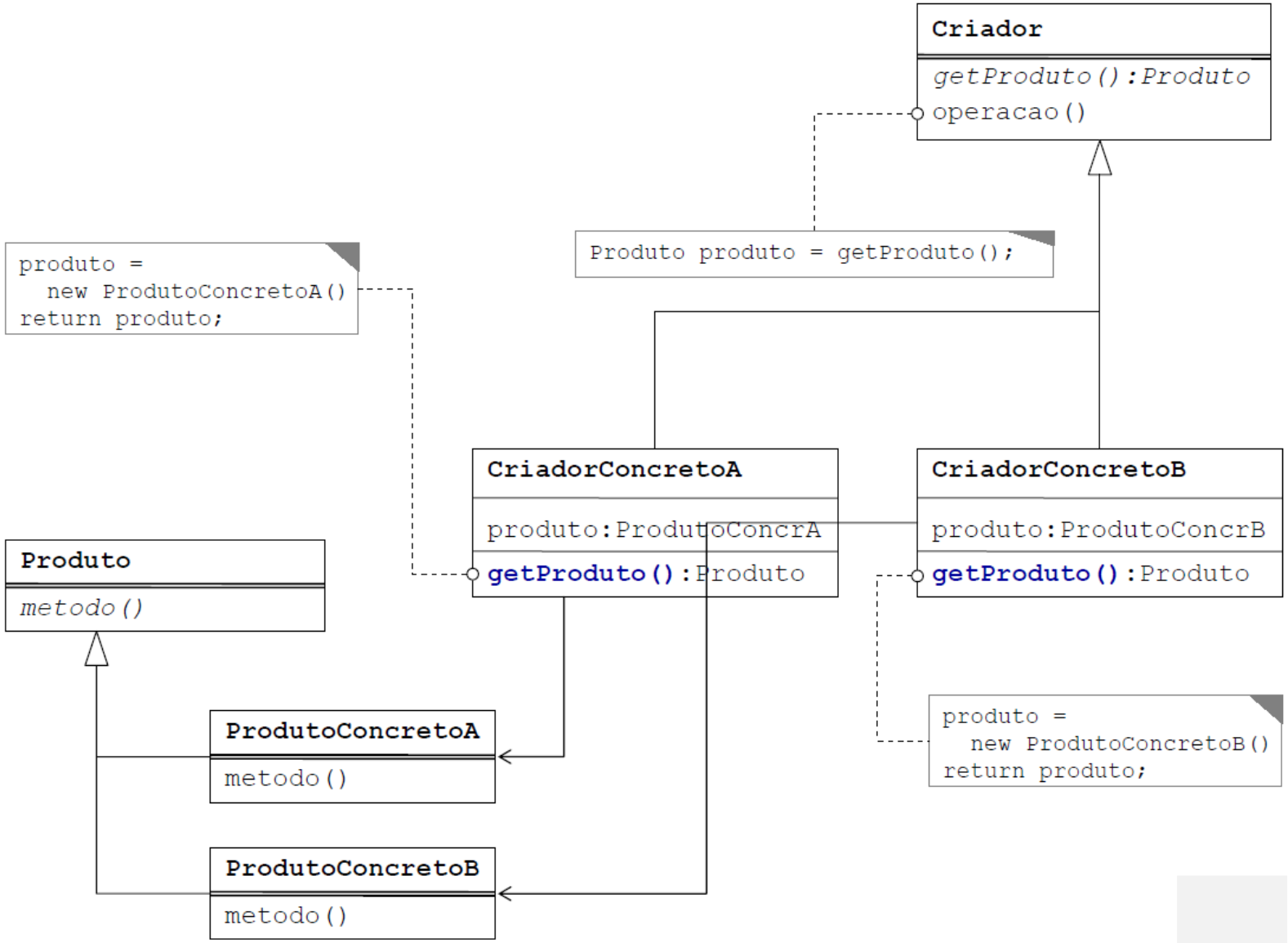
return new Rectangle()
neste contexto

```
public static Shape getShape(String type) {
   ShapeFactory factory = (ShapeFactory)typeMap.get(type);
   return factory.getShape(); // non-static Factory Method
}
```

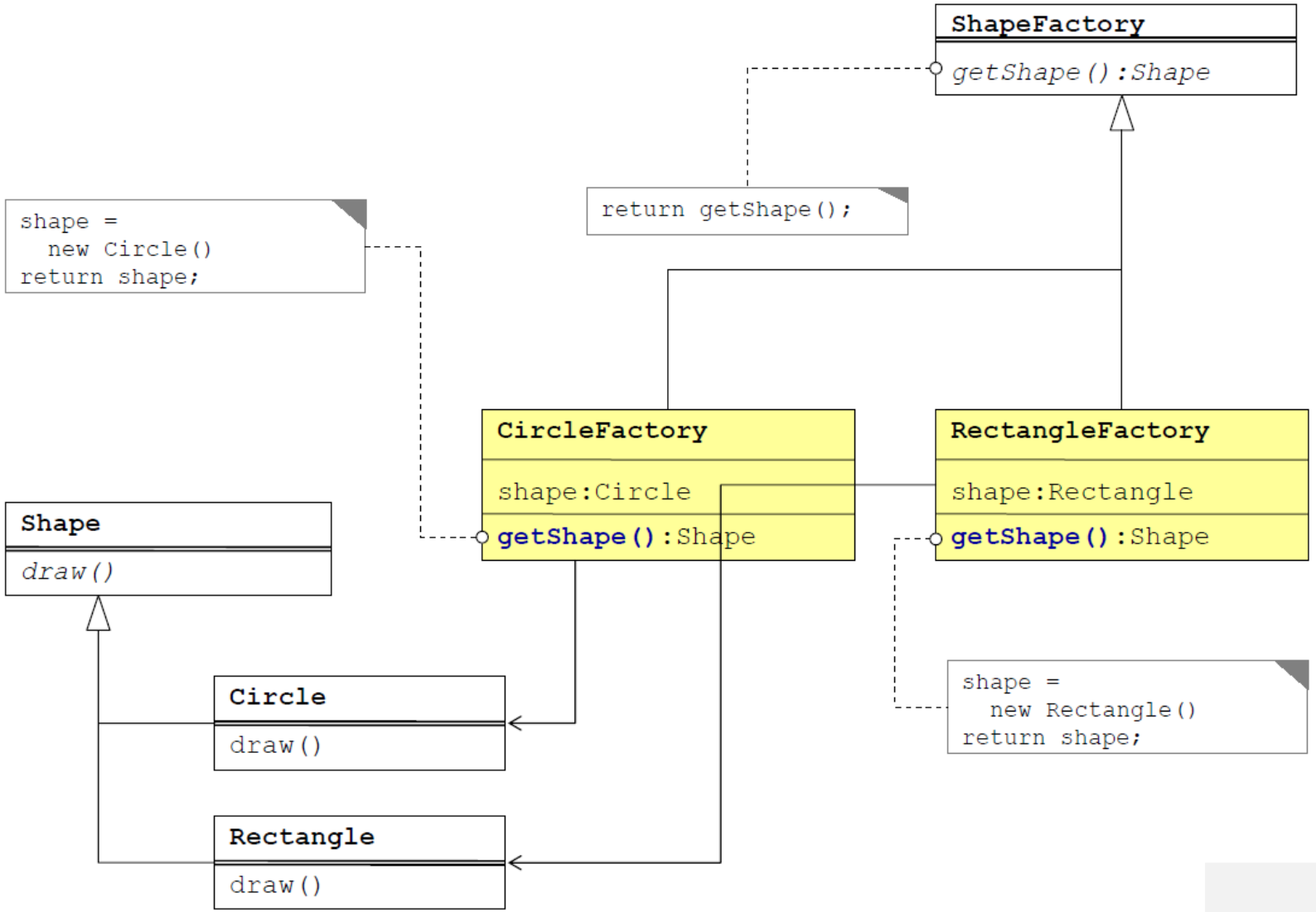
Como fazer ?

- *É possível criar um objeto sem ter conhecimento algum de sua classe concreta?*
  - *Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente*
  - *FactoryMethod define uma interface comum para criar objetos*
  - *O objeto específico é determinado nas diferentes implementações dessa interface*
  - *O cliente do FactoryMethod precisa saber sobre implementações concretas do objeto criador do produto desejado*

Estrutura


Criador
getProduto():Produto
operacao()
Produto produto = getProduto();
produto =
new ProdutoConcretoA()
return produto;
CriadorConcretoA
produto:ProdutoConcrA
getProduto():Produto
CriadorConcretoB
produto:ProdutoConcrB
getProduto():Produto
Produto
metodo()
ProdutoConcretoA
metodo()
ProdutoConcretoB
metodo()
produto =
new ProdutoConcretoB()
return produto;

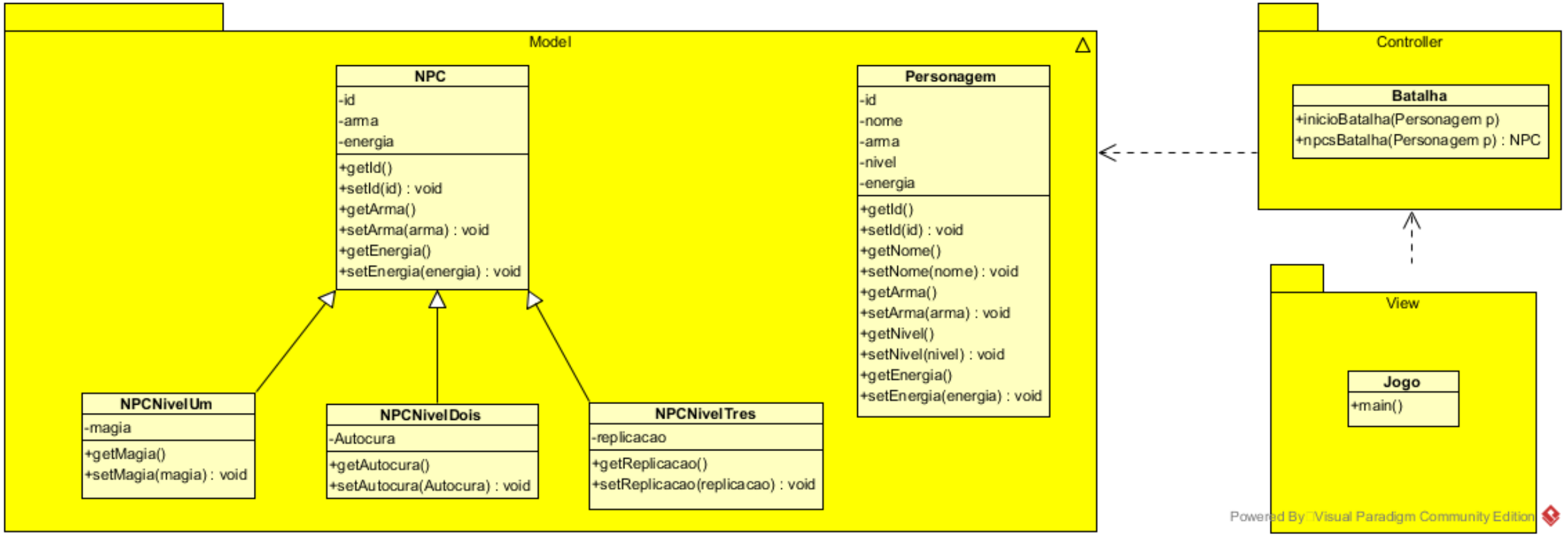
Estrutura


ShapeFactory
getShape():Shape
return getShape();
shape =
new Circle()
return shape;
CircleFactory
shape:Circle
getShape():Shape
RectangleFactory
shape:Rectangle
getShape():Shape
Shape
draw()
Circle
draw()
Rectangle
draw()
shape =
new Rectangle()
return shape;

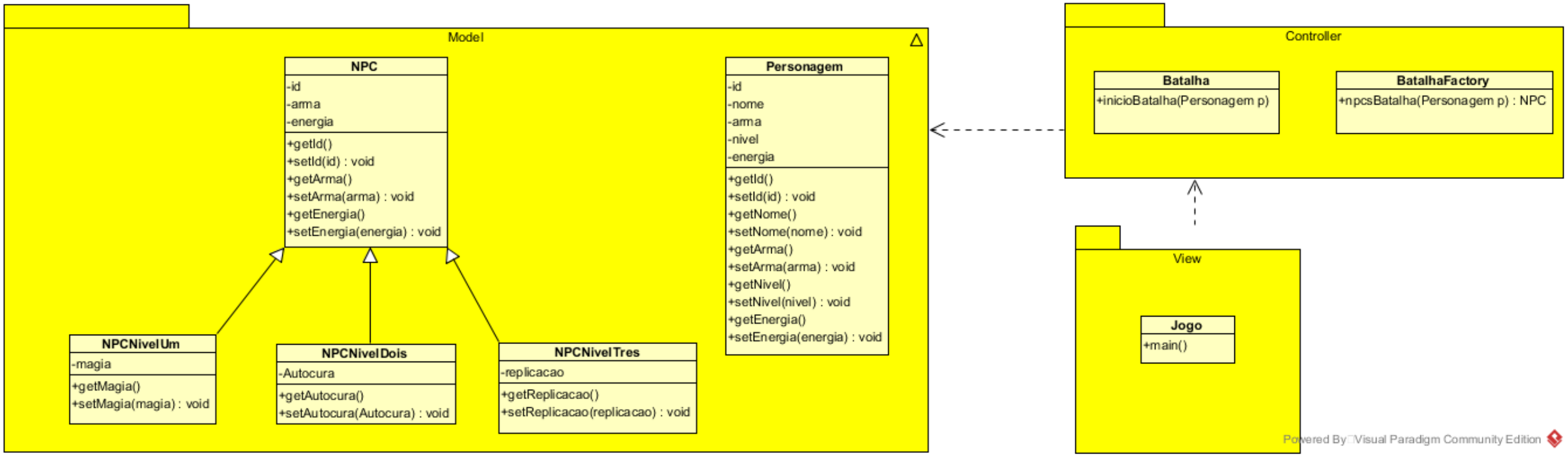
Prática e aplicação em Java

- Considere um jogo que um personagem vai enfrentar NPCs de acordo com suas características e seu nível de evolução. Todo NPC tem um **id**, uma **arma** e uma quantidade de **energia**.
- Os NPCs nível 1 tem alguma quantidade de **Magia**, os nível 2 tem nível de **Auto Cura** e os nível 3 tem nível de **Replicação** de si próprio.
- O personagem tem **id, nome, arma, nível e energia**.
- A batalha, inicialmente 1 x 1, deve ser iniciada com uma mensagem e carregar o NPC equivalente ao nível do personagem (Limitados a 1, 2 ou 3).
  - Se for carregar o NPC nível 1, o id dele vai de **1000 a 2000** (Random), a arma é uma **espada**, a energia é **100** e a magia é **10**;
  - Se for carregar o NPC nível 2, o id dele vai de **10000 a 20000** (Random), a arma é um **machado**, a energia é **100** e a auto cura é **20**;
  - Se for carregar o NPC nível 3, o id dele vai de **100000 a 200000** (Random), a arma é uma **adaga**, a energia é **100** e a replicação é **5**;
- Na main, um personagem deve ser iniciado, a batalha deve ser iniciada com a mensagem e o NPC equivalente ao nível do personagem.

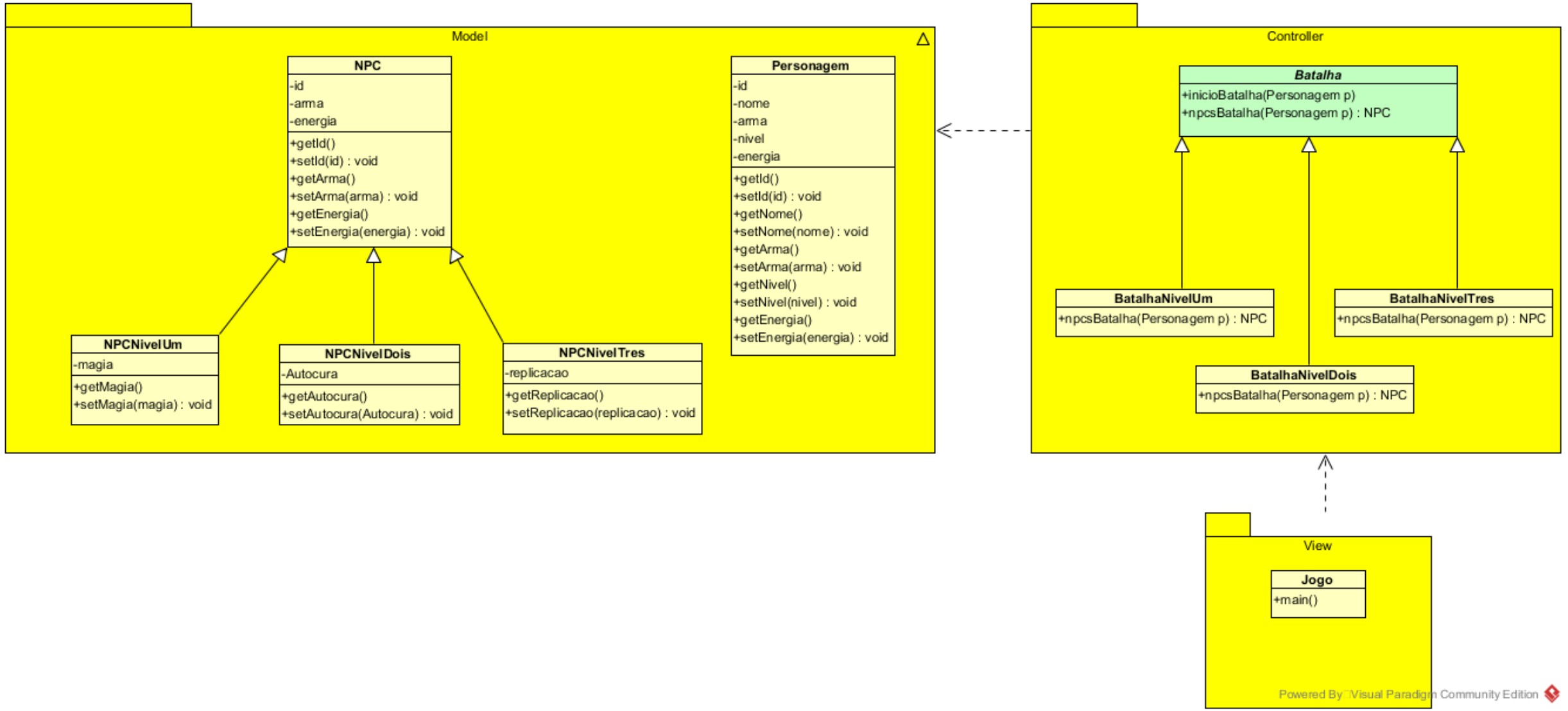
Implementar



Simple Factory



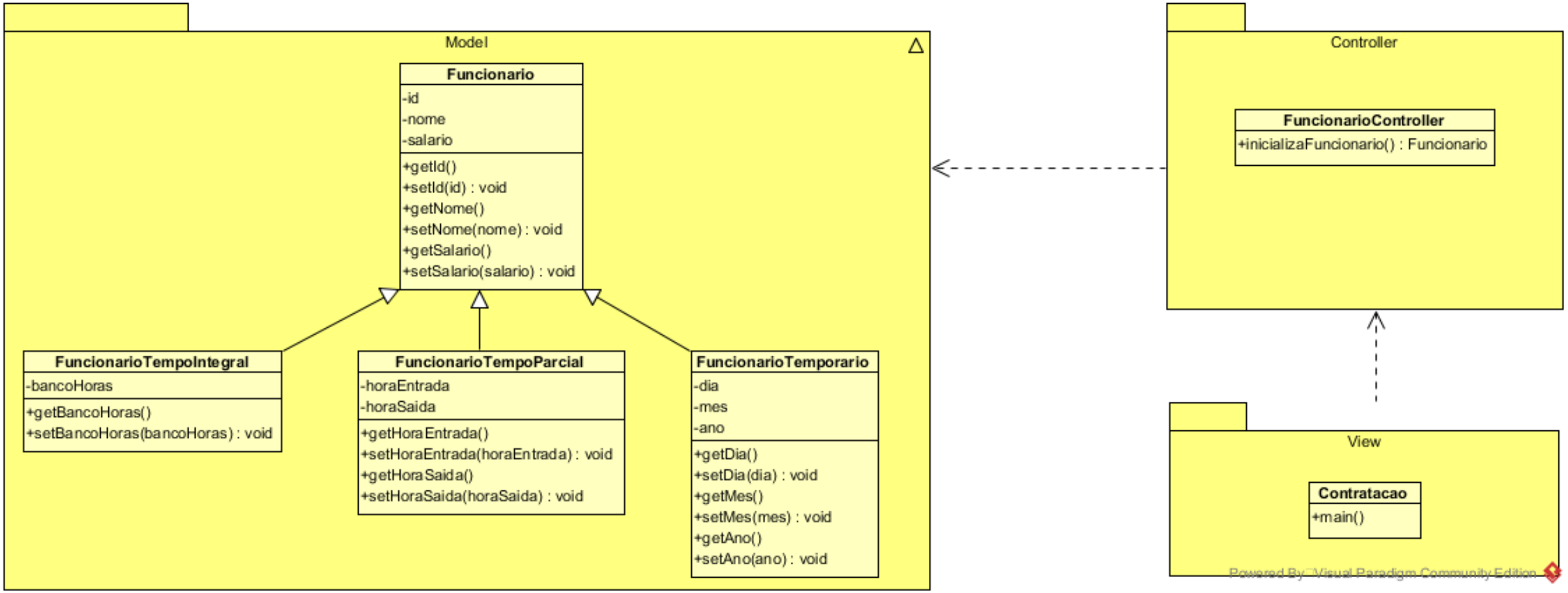
Factory Method



Exercício

- Considere uma aplicação que gerencie a contratação de funcionários. A empresa tem 3 tipos de funcionários: Funcionários de Tempo Integral, de Tempo Parcial e Temporários.
- Todos os funcionários tem id, nome e salário
  - Funcionários de Tempo integral tem banco de horas;
  - Funcionários de Tempo Parcial tem hora de entrada e hora de saída;
  - Funcionários Temporários tem dia, mês e ano para o fim do contrato;
- A classe de controle tem a operação de contratação, que, inicializa o funcionário com seus atributos e retorna o funcionário
- A classe main chama o novo funcionário e imprime seus atributos, simulando a gravação em um BD.
- Aplicar o Factory Method no diagrama, implementar em Java
  - Considere que pode-se, agora, contratar um estagiário que tem como atributo, a instituição parceira

Exercício


Model
Funcionario
-id
-nome
-salario
+getId()
+setId(id) : void
+getNome()
+setNome(nome) : void
+getSalario()
+setSalario(salario) : void
FuncionarioTempoIntegral
-bancoHoras
+getBancoHoras()
+setBancoHoras(bancoHoras) : void
FuncionarioTempoParcial
-horaEntrada
-horaSaida
+getHoraEntrada()
+setHoraEntrada(horaEntrada) : void
+getHoraSaida()
+setHoraSaida(horaSaida) : void
FuncionarioTemporario
-dia
-mes
-ano
+getDia()
+setDia(dia) : void
+getMes()
+setMes(mes) : void
+getAno()
+setAno(ano) : void
Controller
FuncionarioController
+inicializaFuncionario() : Funcionario
View
Contratacao
+main()
Powered By Visual Paradigm Community Edition