PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

13 de maio de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

PADRÕES DE CRIAÇÃO

Objetivo

- separar a construção de um objeto complexo da representação desse objeto, de modo que o processo de criação possa criar diferentes representações
- interpretar uma representação complexa e criar uma nova, com várias possibilidades diferentes

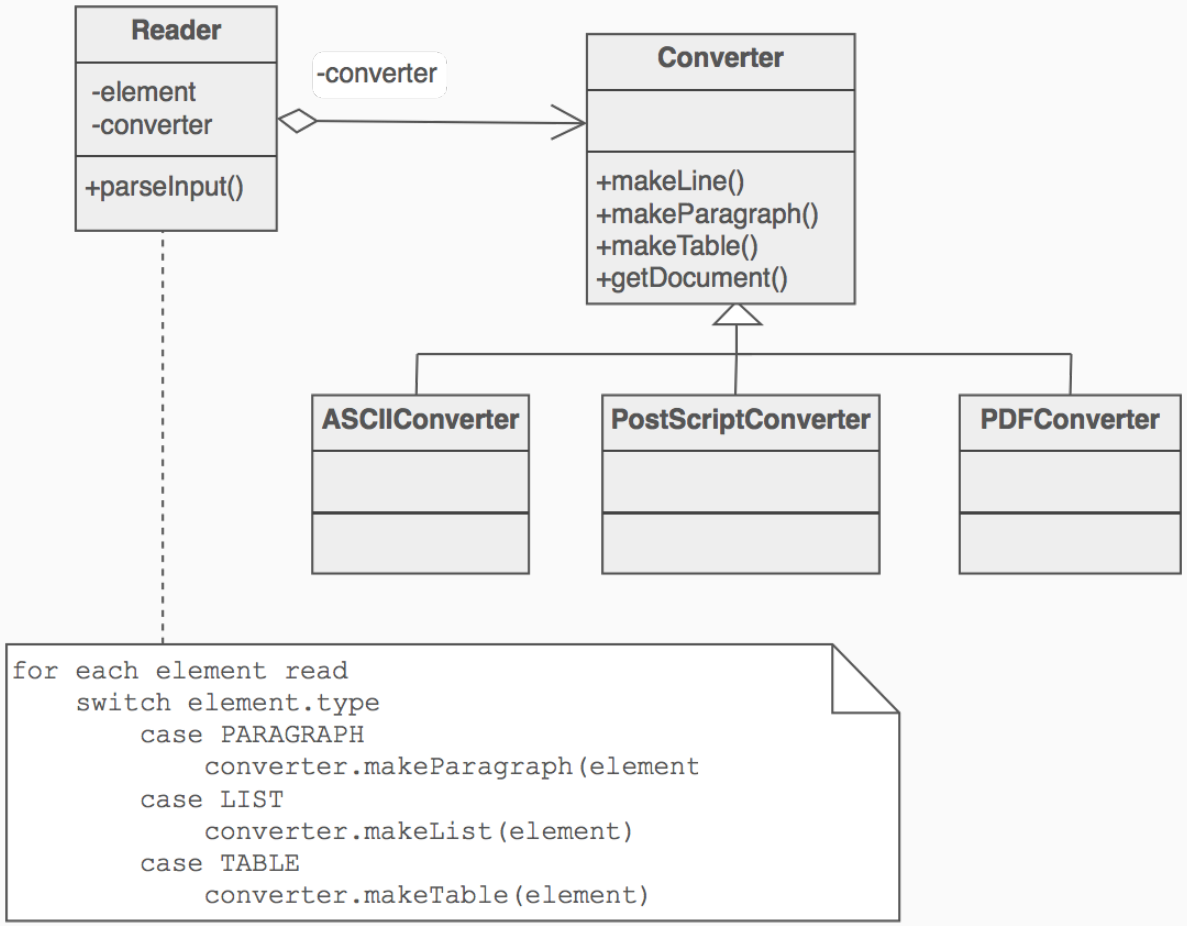
Problema

Uma aplicação precisa criar os vários elementos que compõem um tipo complexo. O processo de criação é complicado e exige vários passos. A especificação do processo de criação está armazenado em algum lugar e queremos usar esse mesmo processo para criar/construir várias cosias de tipos diferentes.

BUILDER: DISCUSSÃO

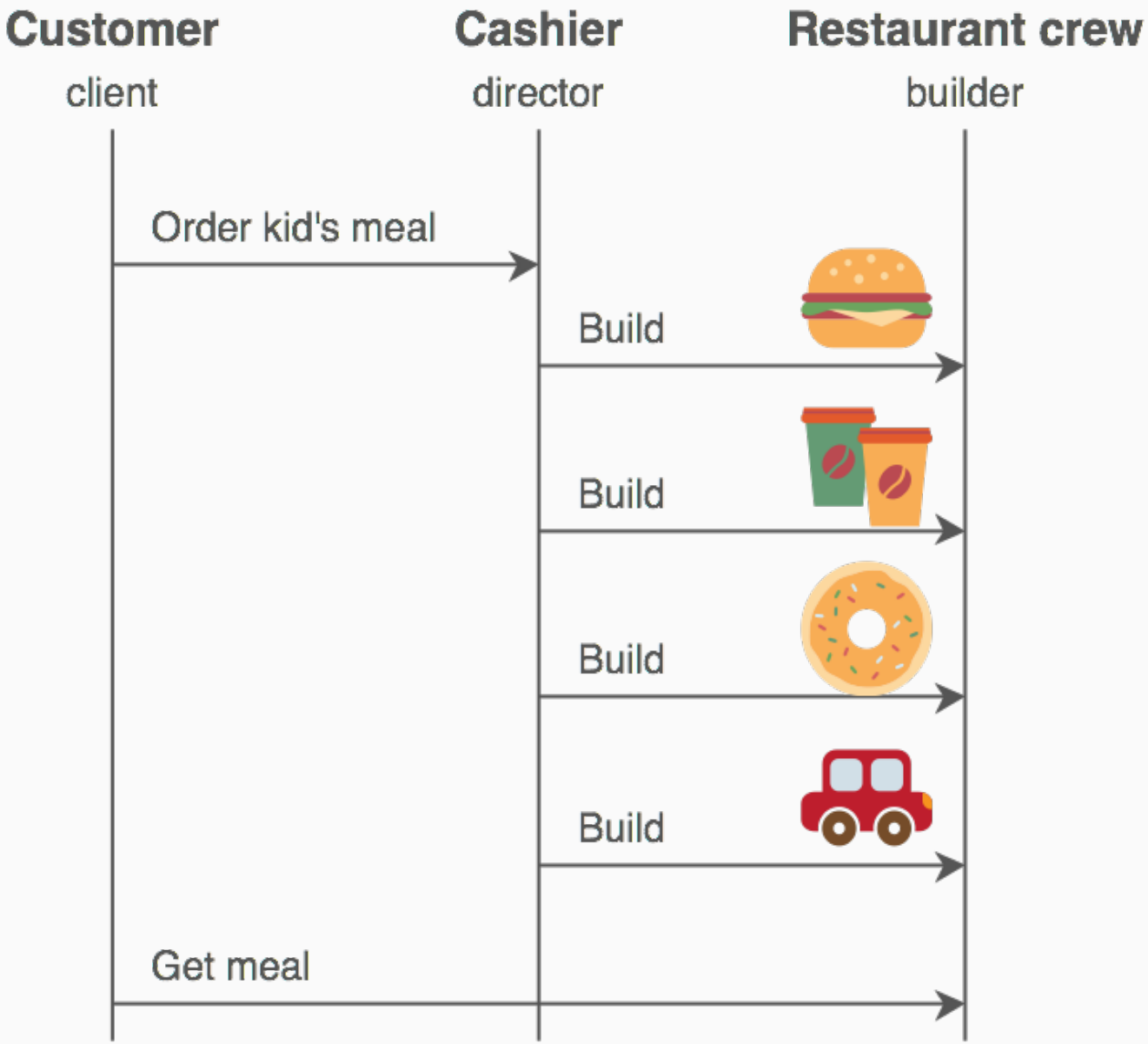
- A ideia é separar o algoritmo de interpretação (leitura e análise) de um mecanismo armazenado em disco (um arquivo RTF, por exemplo) da construção e representação de um ou muitos produtos possíveis (ex: ASCII, TEX, etc.)
- Um “diretor” evoca os serviços do construtor (“builder”) enquanto ele vai interpretando o formato externo
- O “builder” cria as partes de um objeto complexo a cada vez que for chamado e mantém todo o estado intermediário
- Quando o produto for terminado, o cliente recupera o resultado do “builder”

BUILDER: ESTRUTURA



- o `Reader` encapsula a análise e interpretação do aquivo de entrada

BUILDER: EXEMPLO



BUILDER: IMPLEMENTAÇÃO

```java
/* "Produto" */
class Pizza {
  private String dough = "";   // massa
  private String sauce = "";   // molho
  private String topping = ""; // cobertura

  public void setDough(String dough)     { this.dough = dough; }
  public void setSauce(String sauce)     { this.sauce = sauce; }
  public void setTopping(String topping) { this.topping = topping; }
}
```

BUILDER: IMPLEMENTAÇÃO

```java
/* "Produto" */
class Pizza {
  private String dough = "";   // massa
  private String sauce = "";   // molho
  private String topping = ""; // cobertura

  public void setDough(String dough)     { this.dough = dough; }
  public void setSauce(String sauce)     { this.sauce = sauce; }
  public void setTopping(String topping) { this.topping = topping; }
}

/* "Abstract Builder" */
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```

BUILDER: IMPLEMENTAÇÃO

```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("pan baked"); }
  public void buildSauce()   { pizza.setSauce("hot"); }
  public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}
```

BUILDER: IMPLEMENTAÇÃO

```
class Waiter { /* "Diretor" */
  private PizzaBuilder pizzaBuilder;

  public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
  public Pizza getPizza() { return pizzaBuilder.getPizza(); }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}

class BuilderExample { /* Um cliente pedindo uma pizza. */
  public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
    PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();

    waiter.setPizzaBuilder( hawaiian_pizzabuilder );
    waiter.constructPizza();

    Pizza pizza = waiter.getPizza();
  }
}
```

BUILDER: LISTA DE VERIFICAÇÃO

1. Decida se o problema a ser resolvido é o fato de haver uma entrada comum e muitas representações possíveis
2. Encapsule a leitura e análise da entrada comum na classe `Reader`
3. Projete um protocolo para a criação de todas as possíveis saídas. Capture os passos envolvidos na interface de `Builder`
4. Defina uma classe derivada de `Builder` para cada possível representação
5. o cliente cria um objeto `Reader` e o associa a um novo objeto `Builder`
6. O cliente pede pro `Reader` para “construir”
7. O cliente pede pro `Builder` devolver o resultado

FACTORY METHOD

OBJETIVO

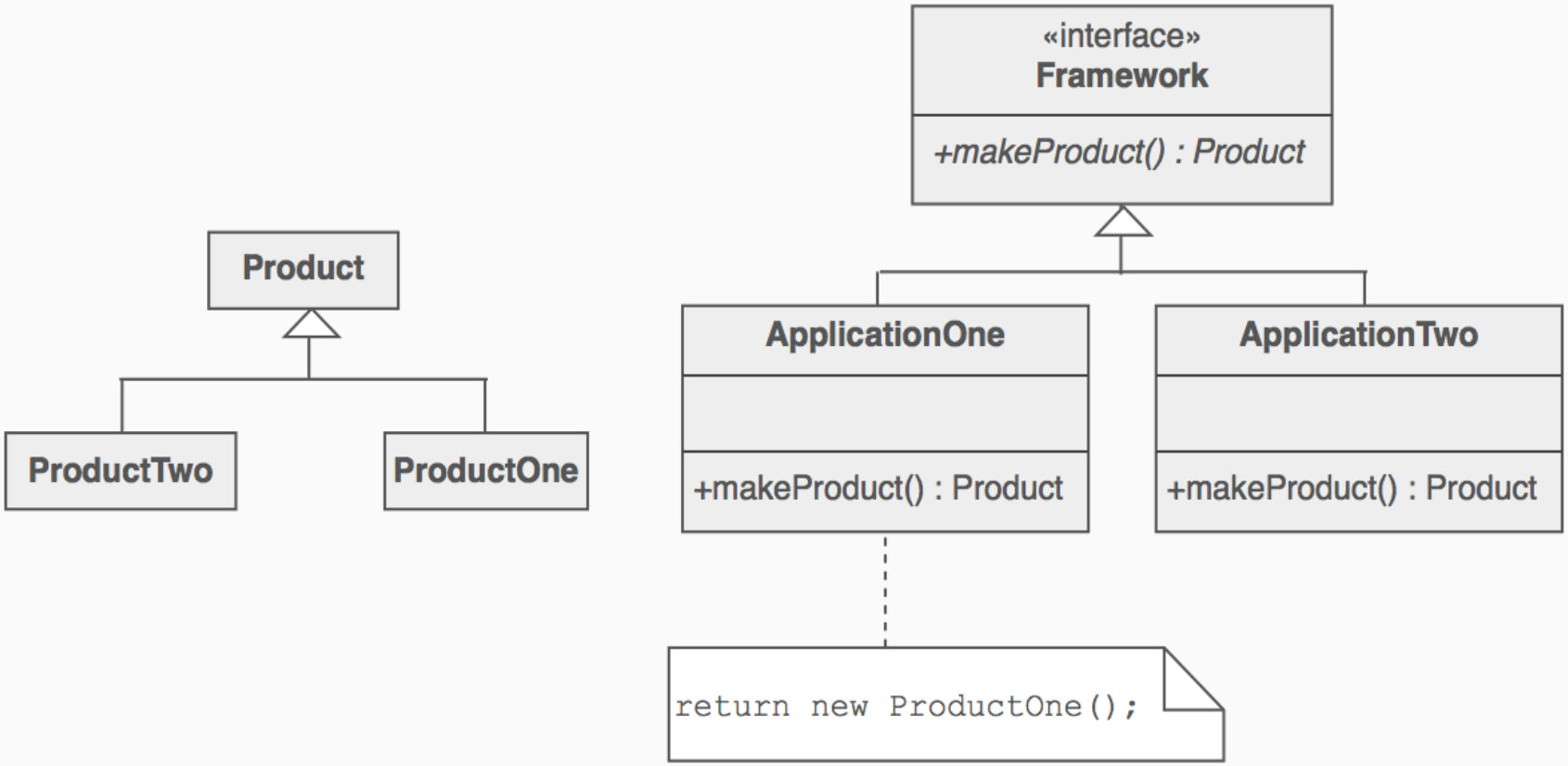
- Definir uma interface para a criação de um objeto, mas permitir que subclasses decidam qual classe instanciar
- Definir um construtor “virtual”
- Operador **new** considerado prejudicial

Um arcabouço precisa padronizar o modelo arquitetural de um conjunto de aplicações, mas precisa permitir que aplicações individuais definam e instanciem seus próprios objetos de negócio.

Discussão

- Uma superclasse especifica todo o comportamento genérico (usando alguns “marcadores” para os passos de criação) e delega os detalhes da criação para subclasses fornecidas pelos clientes
- O Factory Method torna o projeto mais fácil de ser adaptado pelo cliente; basta que ele forneça as subclasses
- Factory Method é parecido com Abstract Factory, mas sem a ênfase nas plataformas

ESTRUTURA

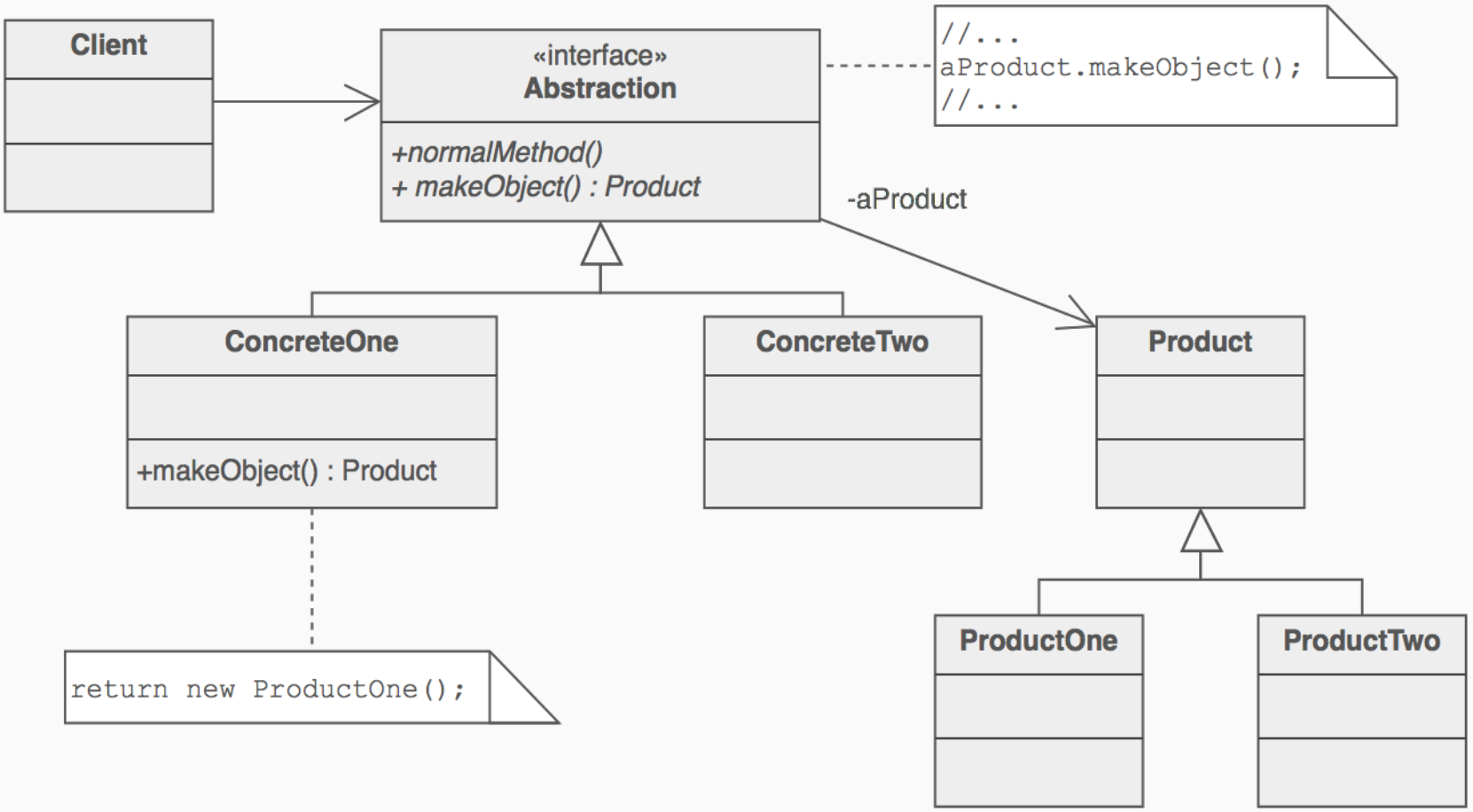

«interface»
Framework
+makeProduct() : Product
Product
ProductTwo
ProductOne
ApplicationOne
+makeProduct() : Product
ApplicationTwo
+makeProduct() : Product
return new ProductOne();

Um modo de definir um factory method é usando um método estático que devolve uma instância daquela classe. Mas ao contrário de um construtor, o objeto devolvido pode ser uma instância de uma subclasse.

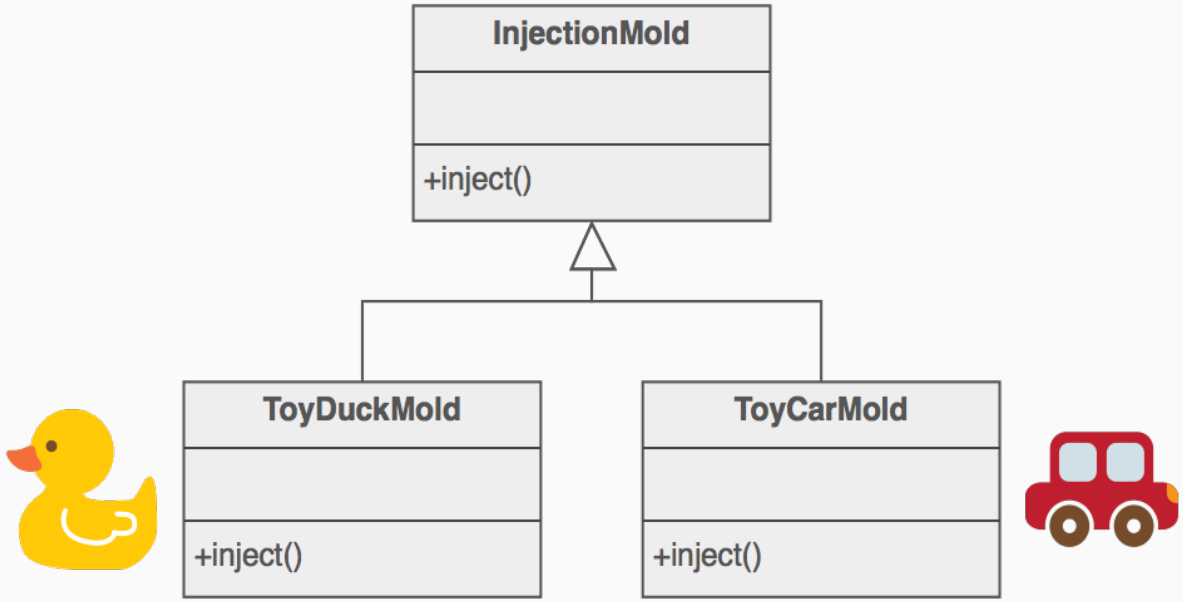
```
Color.make_RGB_color(float red, float green, float blue)
```

```
Color.make_HSB_color(float hue, float saturation, float bright)
```

O cliente fica totalmente desacoplado dos detalhes de implementação das classes derivadas.



EXEMPLO



- Exemplo com moldagem por injeção
- Uma máquina injeta termoplástico em uma forma. A classe do brinquedo é definido pela forma

LISTA DE CHECAGEM

1. Se você usa polimorfismo com uma hierarquia de heranças, considere adicionar um método de criação polimórfico usando um factory method estático na classe base
2. Defina os argumentos do factory method com cuidado. Quais qualidades ou características são suficientes e necessários para identificar qual classe derivada instanciar?
3. Considere implementar um “object pool” que permita que objetos possam ser reutilizados ao invés de criados do zero
4. Considere definir os construtores como `private` ou `protected`

OBJECT POOL

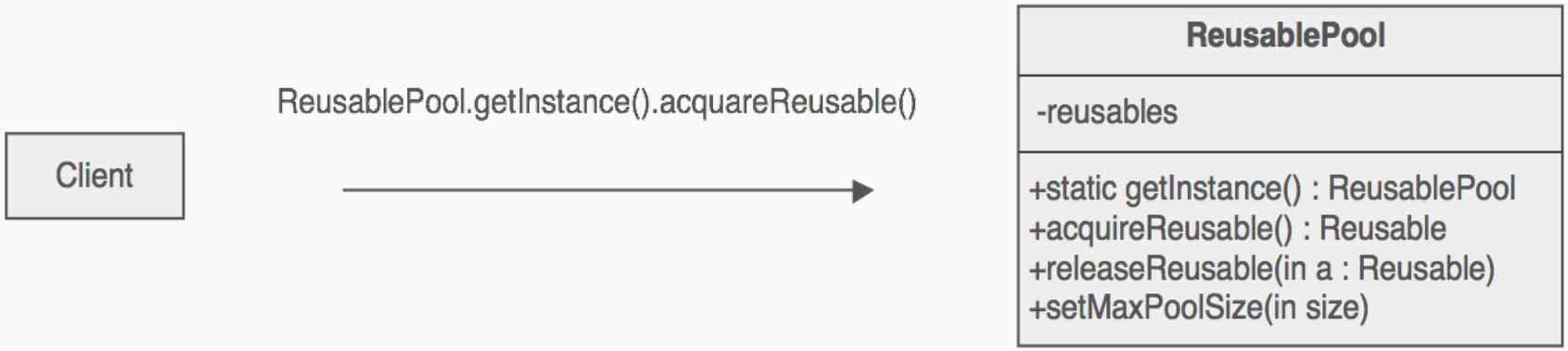
OBJETIVO

Um “pool de objetos” pode oferecer um ganho de desempenho importante. É muito eficaz em situações onde inicializar uma instância é muito custoso, onde a taxa de criação de novas instâncias é alta e o número de instância em uso em um dado momento é baixo.

PROBLEMA

- Object Pools (também conhecido como *resource pools*) são usados para gerenciar caches de objetos
- Um cliente com acesso ao *pool* evita a instanciação de novos objetos vendo se existe algum já instanciado no *pool*
- É recomendável que todos os objetos reutilizáveis que não estão em uso fiquem no mesmo *pool*, para que possam ser gerenciados usando a mesma política
- A classe com esses objetos deve ser projetada para ser um Singleton

ESTRUTURA

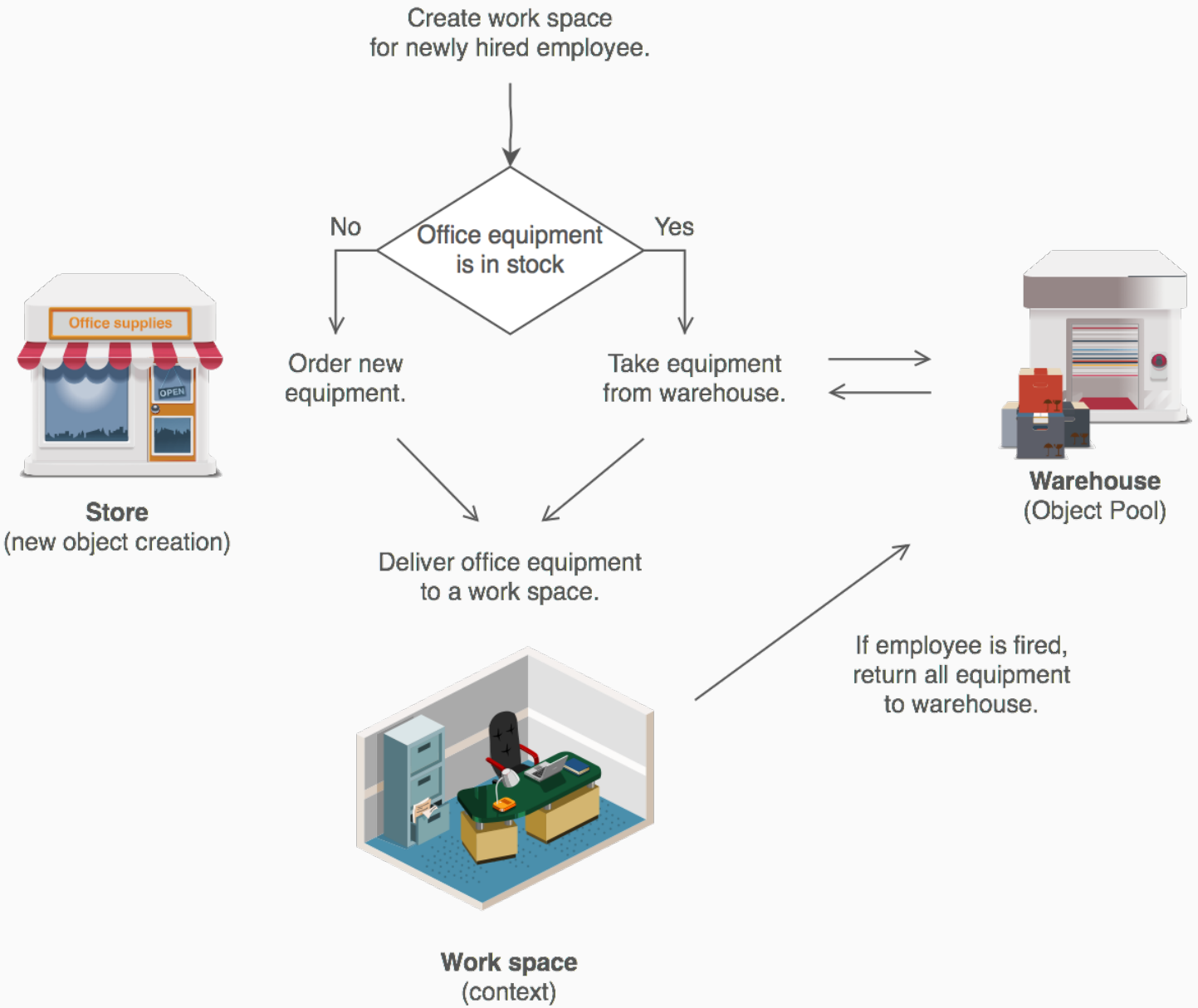


Reusable instâncias nesse papel colaborarão com outros objetos por um tempo e depois não serão mais necessários

Client instâncias de objetos que usam os objetos Reusables

ReusablePool instâncias dessa classe gerenciam os objetos Reusable que serão utilizados pelos clientes

EXEMPLO


Create work space
for newly hired employee.
Office equipment
is in stock
No
Yes
Order new
equipment.
Take equipment
from warehouse.
Office supplies
OPEN
Store
(new object creation)
Warehouse
(Object Pool)
Deliver office equipment
to a work space.
If employee is fired,
return all equipment
to warehouse.
Work space
(context)

LISTA DE CHECAGEM

1. Crie a classe `ObjectPool` com um vetor `private` de objetos
2. Crie métodos `acquire` e `release` para recuperar e devolver os objetos do *pool*
3. Garanta que seu `ObjectPool` seja um Singleton

BIBLIOGRAFIA

- The Gang of Four Book, ou GoF: E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns — Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Alexander Shvets. Design patterns explained simply. `https://sourcemaking.com/design_patterns/`