

# PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

---

Daniel Cordeiro

20 de maio de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

## PADRÕES DE CRIAÇÃO

---

## Objetivo

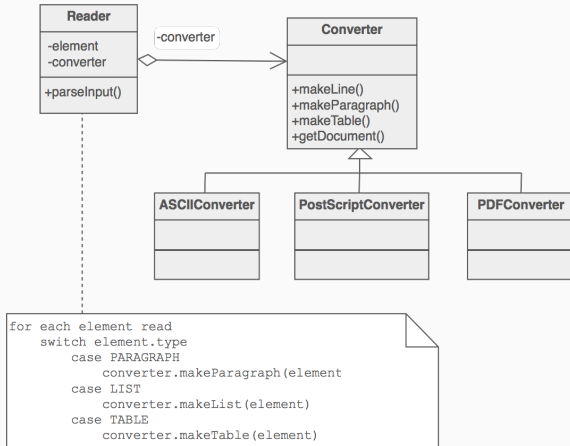
- separar a construção de um objeto complexo da representação desse objeto, de modo que o processo de criação possa criar diferentes representações
- interpretar uma representação complexa e criar uma nova, com várias possibilidades diferentes

## 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 coisas de tipos diferentes.

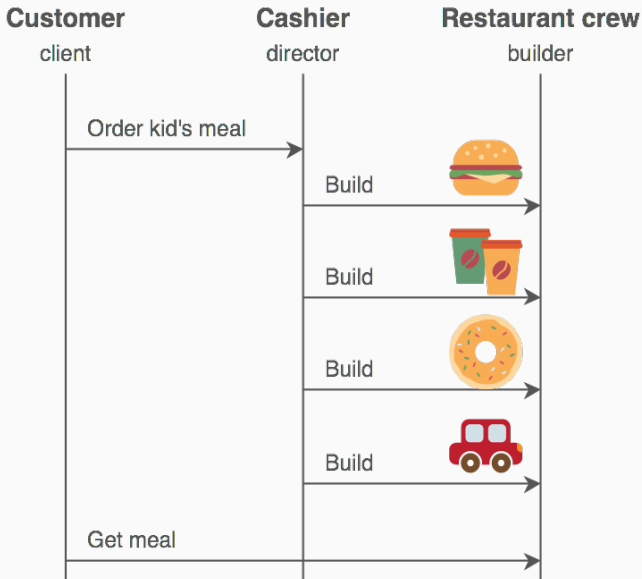
- A ideia é separar o algoritmo de interpretação (leitura e análise) de um mecanismo armazenado em disco (um arquivo RTF, por exemplo) da construção e representação de um ou muitos produtos possíveis (ex: ASCII,  $\text{\TeX}$ , etc.)
- Um “diretor” evoca os serviços do construtor (“builder”) enquanto 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 arquivo de entrada

## BUILDER: EXEMPLO



## BUILDER: IMPLEMENTAÇÃO

```
/* "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

```
/* "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();
    }
}
```

## FACTORY METHOD

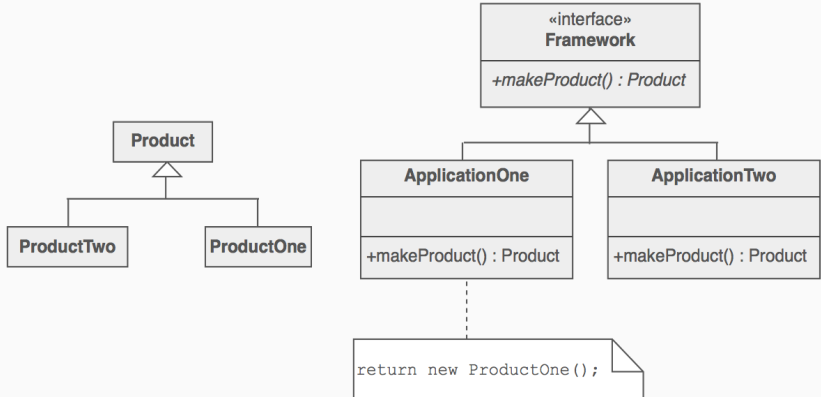
---

- 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

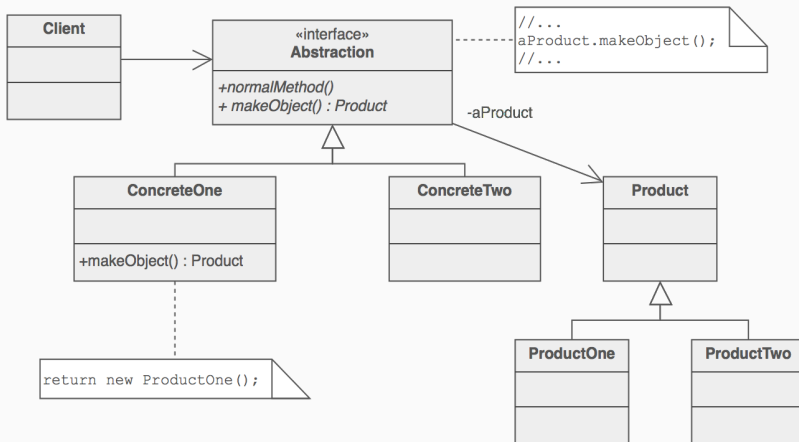


Um modo de definir um factory method é usando um método estático que devolve uma instância daquela classe. Mas ao contrário de um construtor, o objeto devolvido pode ser uma instância de uma subclasse.

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

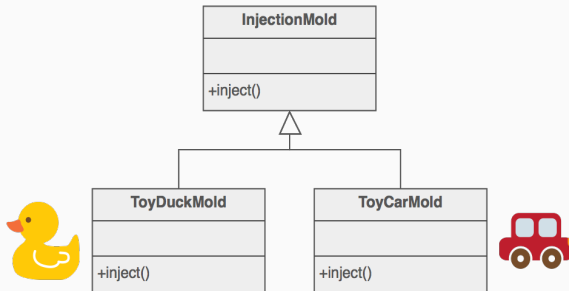
```
Color.make_HSB_color(float hue, float saturation, float bright)
```

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

## OBJECT POOL

---

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.

- 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

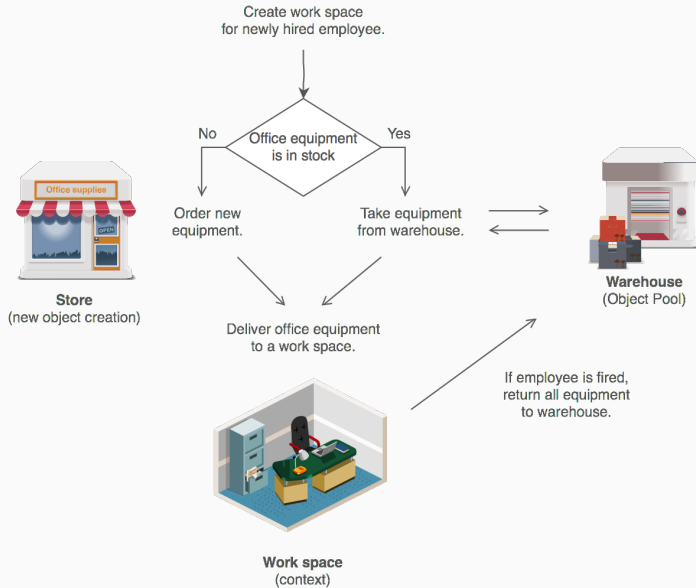


**Reusable** instâncias nesse papel colaborarão com outros objetos por um tempo e depois não serão mais necessários

**Client** instâncias de objetos que usam os objetos Reusables

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

# EXEMPLO



## PROTOTYPE

---

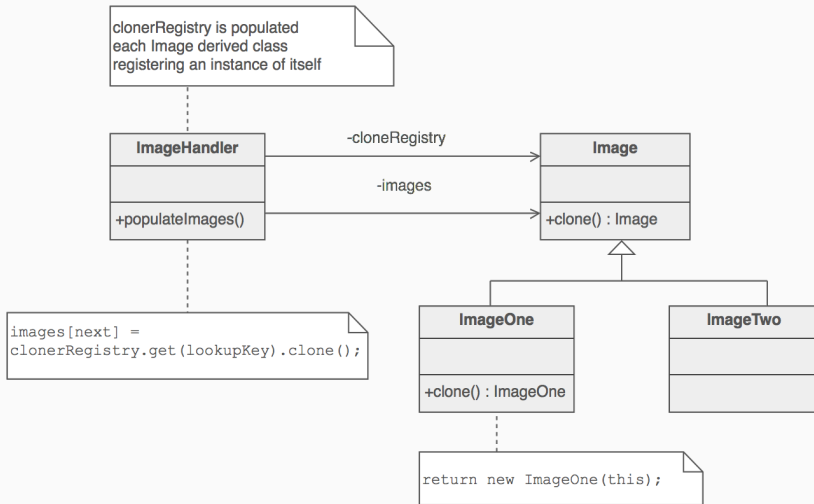
- Especificar o tipo de objeto a ser criado usando para isso uma instância “protótipo”
- Pegar uma instância de uma classe e a usar para criar todas as futuras instâncias
- Operador **new** considerado prejudicial

### Problema

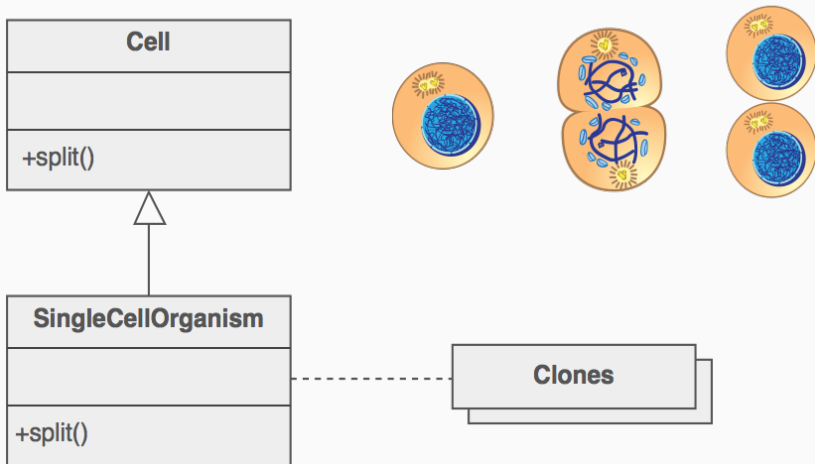
A aplicação “fixa” o tipo de classe de um objeto em cada expressão **new**.



- Declare uma classe base abstrata que especifique um método virtual “clone” e que mantenha um dicionário com todas as classes derivadas “clonáveis”
- Todas as classes que precisarem de um “construtor polimórfico” herdam da classe base, registram suas instâncias protótipos e implementam a operação **clone**
- Os clientes, ao invés de usarem **new**, chamam a operação **clone** na classe base fornecendo algum identificador do tipo concreto que desejam instanciar



## EXEMPLO



1. Adicione um método **clone** a hierarquia existente
2. Defina um “registro” que mantenha um cache de objetos protótipo; o registro pode ser encapsulado em uma nova classe **Factory** ou na classe abstrata base
3. Projete um factory method que: aceite (ou não) argumentos, encontre o protótipo correspondente, chame **clone** nesse protótipo e devolva o resultado
4. O cliente substitui todas as referências ao operador **new** por chamadas ao method factory

- 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/](https://sourcemaking.com/design_patterns/)