Programação Orientada a Objetos II

Aula 05

Prof. Leandro Nogueira Couto
UFU – Monte Carmelo

Padrões de Projeto

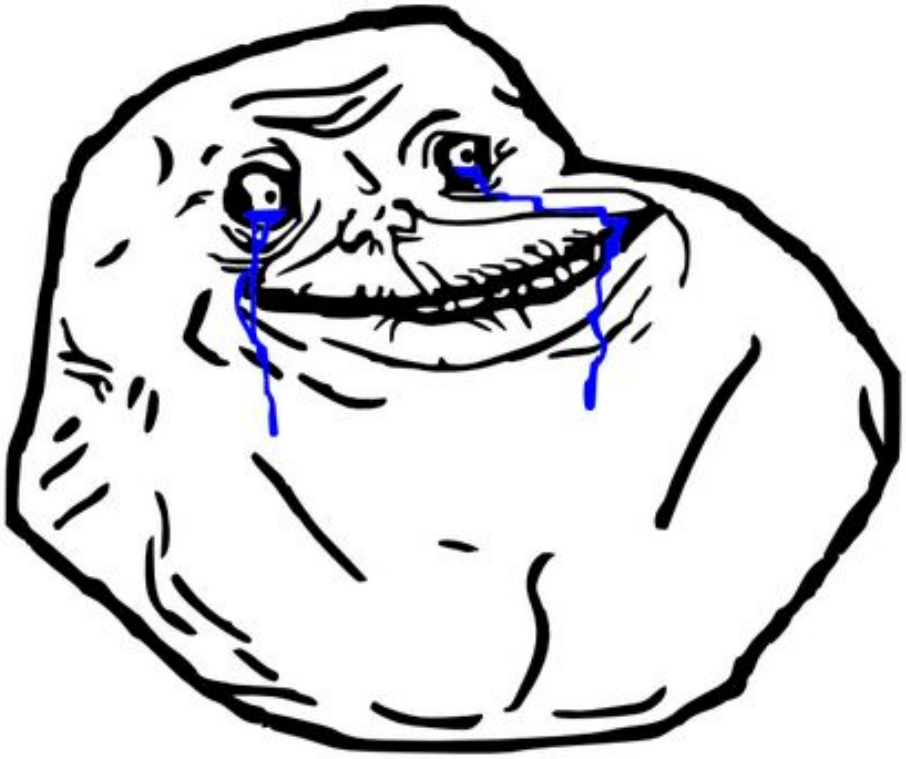
- Na última aula vimos um tipo de padrão de projeto **Creacional**, os padrões **Factory**
- Eles permitem que criemos diversos objetos de uma determinada classe de forma organizada
- Veremos a partir de agora mais alguns **Padrões de Projeto Creacionais**:
  - **Factory (Abstract, Method)**
  - **Singleton**
  - **Builder**
  - **Prototype**
  -

Padrões Creacionais

- Tenha em mente que:
  - Às vezes, padrões creacionais são **competidores**: há casos onde um pode ser usado OU outro, com bons resultados. Há casos também onde são **complementares** e podem ser usados juntos
  - Uma **Abstract Factory** pode ser implementada com **Factory Methods** como visto em aula, ou podem ser implementadas com **Prototype**
  - É comum que o design comece com um **Factory Method** (mais simples) e **evolua para Abstract Factory, Builder ou Prototype**, mais complexos e flexíveis

Singleton

- Objetivo:
  - É um padrão que restringe a instanciação de **uma classe** a apenas **um objeto**
  - Desejamos garantir também um ponto de acesso global à classe
- Pode ser usado na implementação de outros padrões creacionais, como Factory, Builder e Prototype
- Vem da matemática: conjunto com um elemento só.

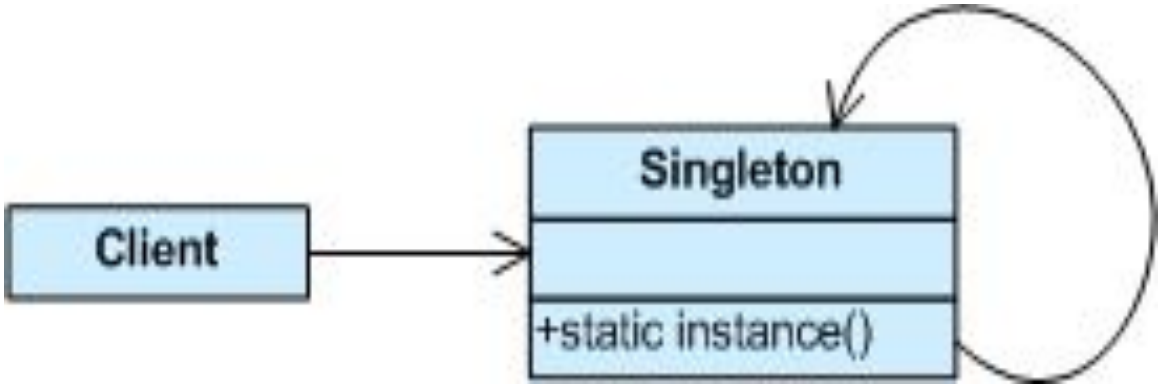


Singleton

- Normalmente usado quando:
  - Inicialização "preguiçosa" (lazy) é desejável
    - Só inicializar quando necessário (questões de performance)
    - Guardar o estado uma vez inicializado para não precisar fazer de novo
  - Não faz sentido atribuir a "ownership" da instância a classe nenhuma
  - Acesso global é desejável (talvez seja inconveniente passar o objeto por referência pra quem precisar)
- Sem alguma essas 3 condições acima, provavelmente é melhor usar apenas métodos **static de uma classe**

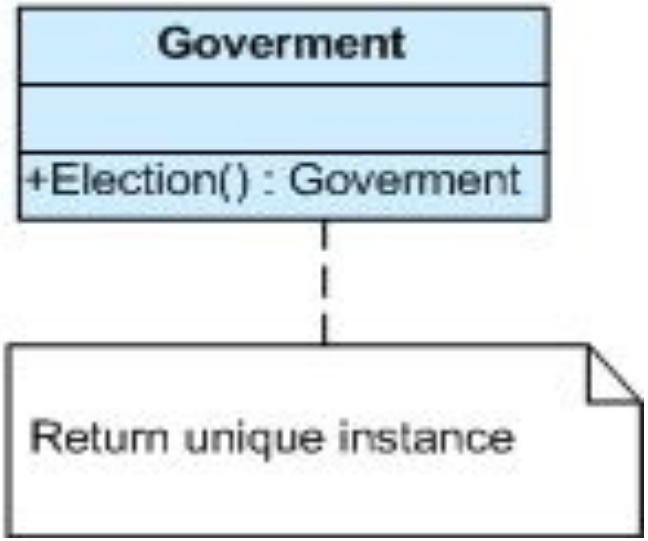
Singleton

- Estrutura



- A classe da única instância deve ser responsável pelo acesso e pela “inicialização ao primeiro uso”. A instância única é um **atributo private static**. A função que o acessa (o accessor, no caso instance()) é um **método public static**.

Singleton


Goverment
+Election() : Goverment
Return unique instance


Singleton
- instance : Singleton = null
+ getInstance() : Singleton
- Singleton() : void

Singleton

- Passos
  - Defina um atributo private static na classe que irá ser o Singleton.
  - Defina um método public static acessor nessa classe.
  - Faça “lazy initialization” (crie o objeto na primeira vez que usar o método) dentro do método acessor.
  - Defina todos os construtores como protected ou private.
  - Clientes só podem usar o método acessor para manipular o Singleton.

Singleton

- **Lazy Initialization:**
  - Se o objeto não existir, instancie
  - Se a instância já existe, apenas retorne uma referência a ela
- Cuidado se usar threads para que duas instâncias não sejam criadas. Lembre-se de garantir a relação **happens-before**
- O Singleton pode ser mais difícil que parece de **deletar**! (To Kill a Singleton)
- O Singleton é global, deve ser usado com cuidado e parcimônia. Se usado erradamente, pode se tornar um "anti-padrão"

Singleton

```
public class Singleton {

  // Private constructor prevents instantiation from other classes
  private Singleton() {}

  /**
   * SingletonHolder is loaded on the first execution of Singleton.getInstance()
   * or the first access to SingletonHolder.INSTANCE, not before.
   */

  private static class SingletonHolder {

    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Builder


Mustard
Sweet Onion
Vinegar
Spicy Brown Mustard
Honey Mustard
Creamy Sriracha
NEW
VINAIGRETTE
ROBERTO

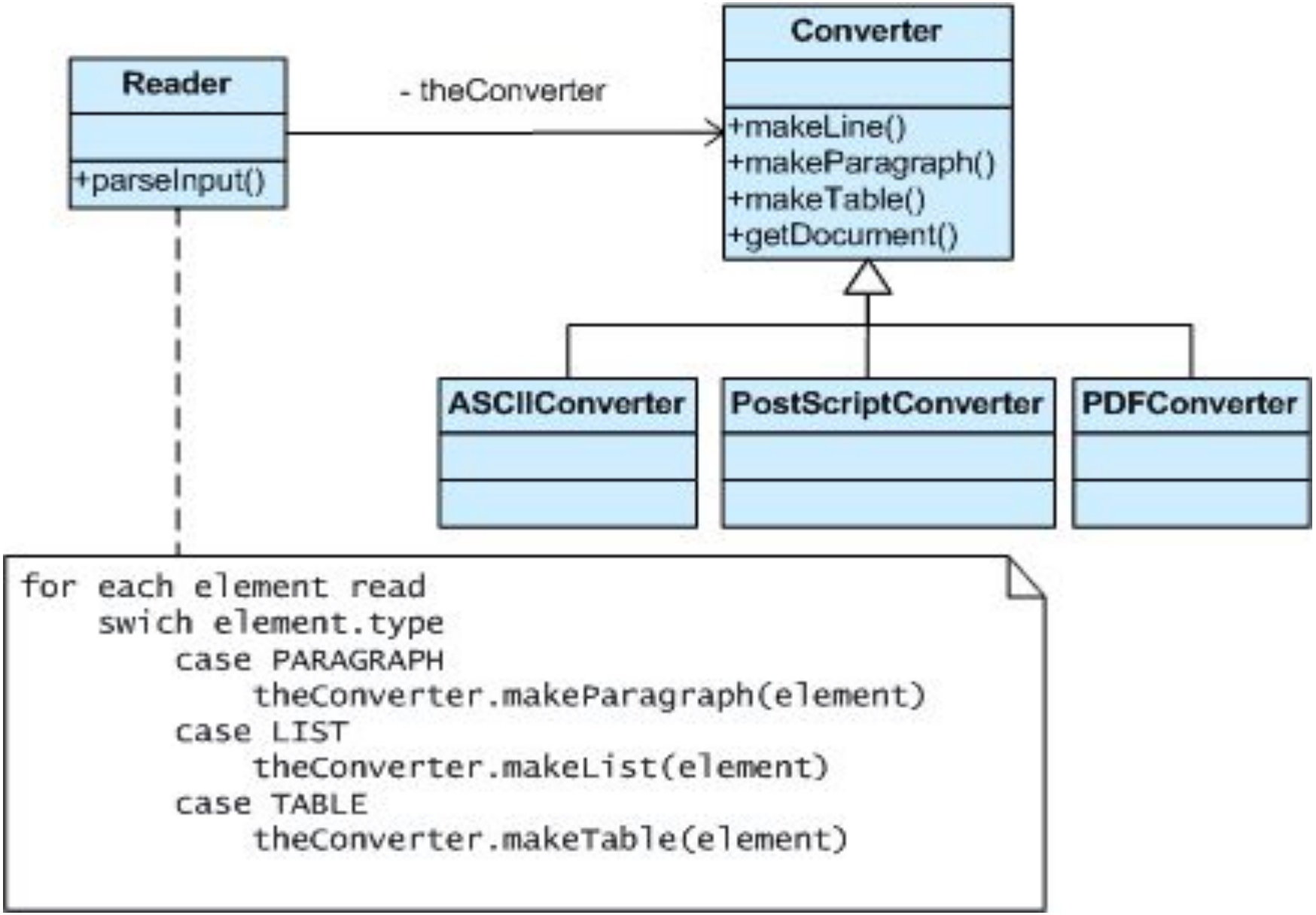
Builder

- Padrão de Projeto Creacional que consiste em construir um **objeto complexo de forma incremental**, ou seja, parte por parte
- Ao final, o Cliente recebe o **objeto completo**
- Pode usar algum dos outros padrões (Factory Method, por exemplo) para implementar que componentes são construídos
- O **objeto complexo** resultante do Builder é comumente um **Composite** (Design Pattern Estrutural)
- Com Builder, não é necessário que o objeto completo seja criado de uma só vez (mas cuidado com o estado temporário do objeto!)

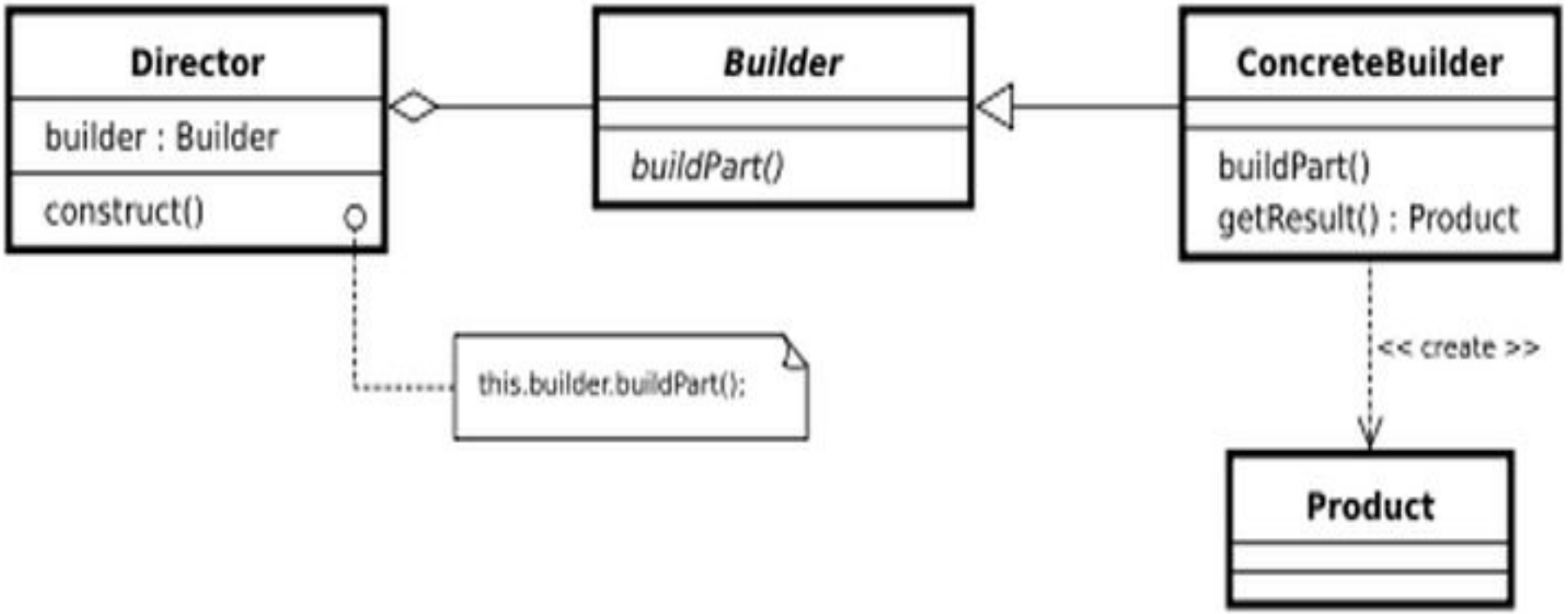
Builder

- Quando usar?
  - Separar a **construção** de um objeto complexo de sua **representação**, permitindo que o mesmo processo de construção gere representações (ou targets) diferentes
  - **Estrutura muito complexa** para ser criada em apenas um método (muito módulos, muitos parâmetros)
  - A **especificação** pode vir de vários lugares (pela rede, *hard-coded*), pode vir de leitura de algum arquivo texto ou de dados.

Builder


Converter
Reader
- theConverter
+makeLine()
+makeParagraph()
+makeTable()
+getDocument()
+parseInput()
ASCIIConverter
PostScriptConverter
PDFConverter
for each element read
swich element.type
case PARAGRAPH
theConverter.makeParagraph(element)
case LIST
theConverter.makeList(element)
case TABLE
theConverter.makeTable(element)

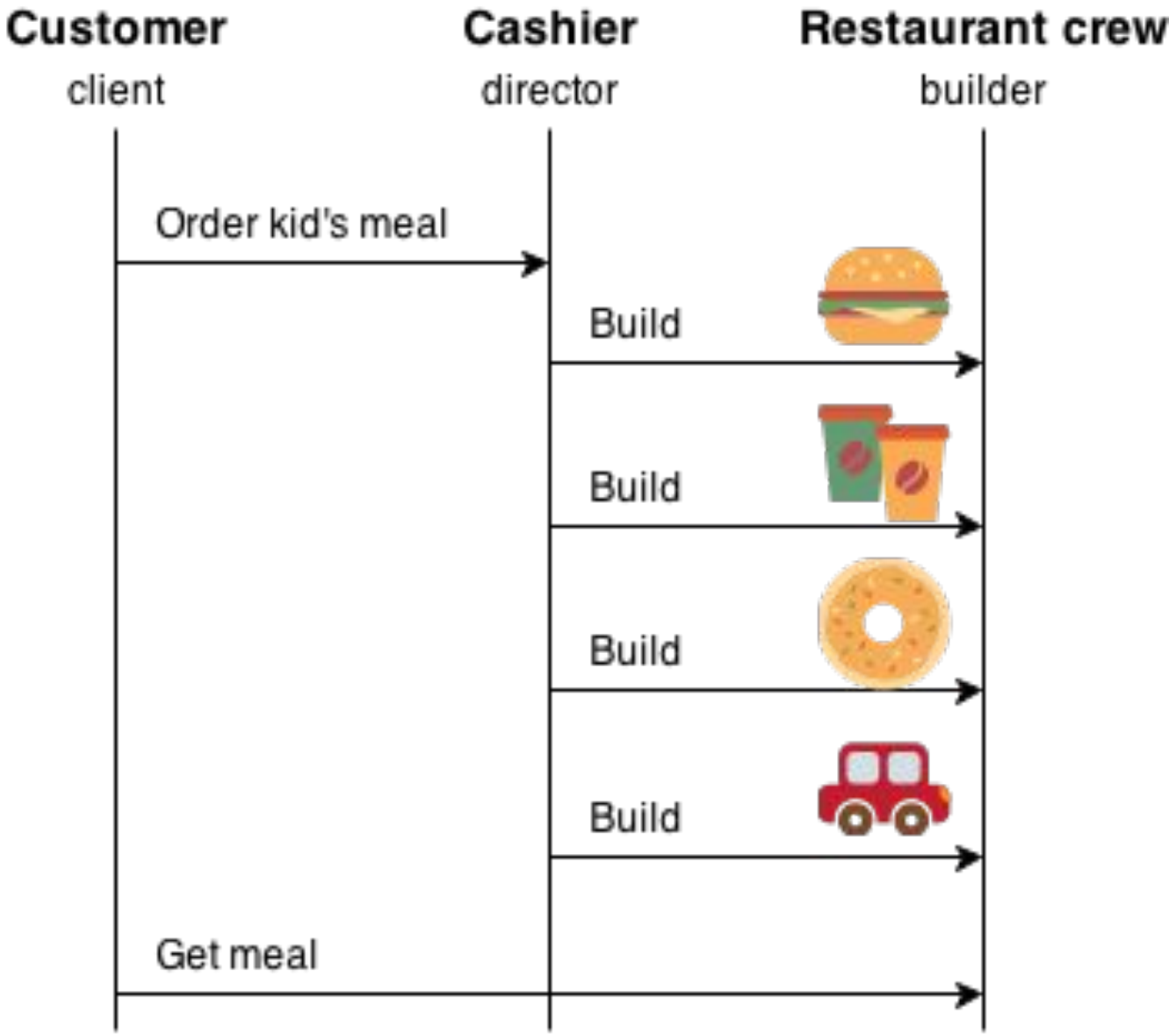
Builder


Director
builder : Builder
construct()
Builder
buildPart()
ConcreteBuilder
buildPart()
getResult() : Product
this.builder.buildPart();
<< create >>
Product

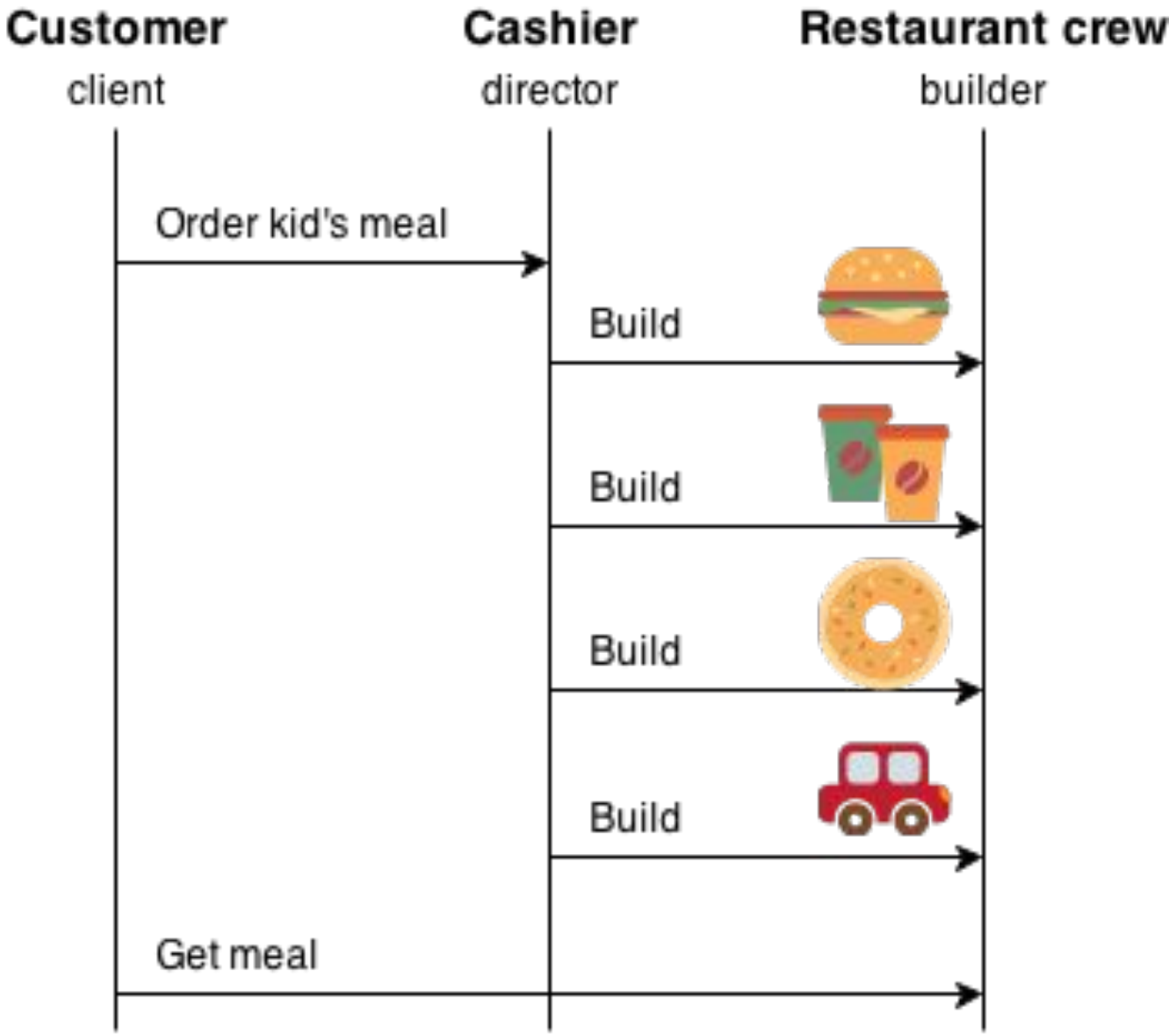
Builder

- Participantes
  - **Director** ou **Reader** é quem oferece a especificação (a "receita") e coordena a construção da estrutura complexa (o "produto")
  - O "gatilho" para a construção pode vir de um **Client**
  - O **Builder** constrói cada módulo até o produto ficar completo

Builder


Customer
client
Cashier
director
Restaurant crew
builder
Order kid's meal
Build
Build
Build
Build
Get meal

Builder


Customer
client
Cashier
director
Restaurant crew
builder
Order kid's meal
Build
Build
Build
Build
Get meal

Builder

- Passos
  - Decida se o construtor é complexo o bastante ou modular o bastante, ou se deseja-se muito parâmetros para possibilitar várias representações
  - Encapsular o parsing (leitura dos parâmetros, da receita) em uma única classe (Reader). Melhor a especificação vir de um só lugar.
  - Projetar um processo/protocolo para criar os componentes e integrá-los e implementar esse processo no Builder.
  - Defina subclasses de Builder para diferentes representações, se necessário.
  - O cliente instancia um Reader e um Builder, e associa os dois.
  - O cliente pede para o Reader/Director "construir" ou "montar".
  - O cliente pede que o Builder retorne o objeto resultante.

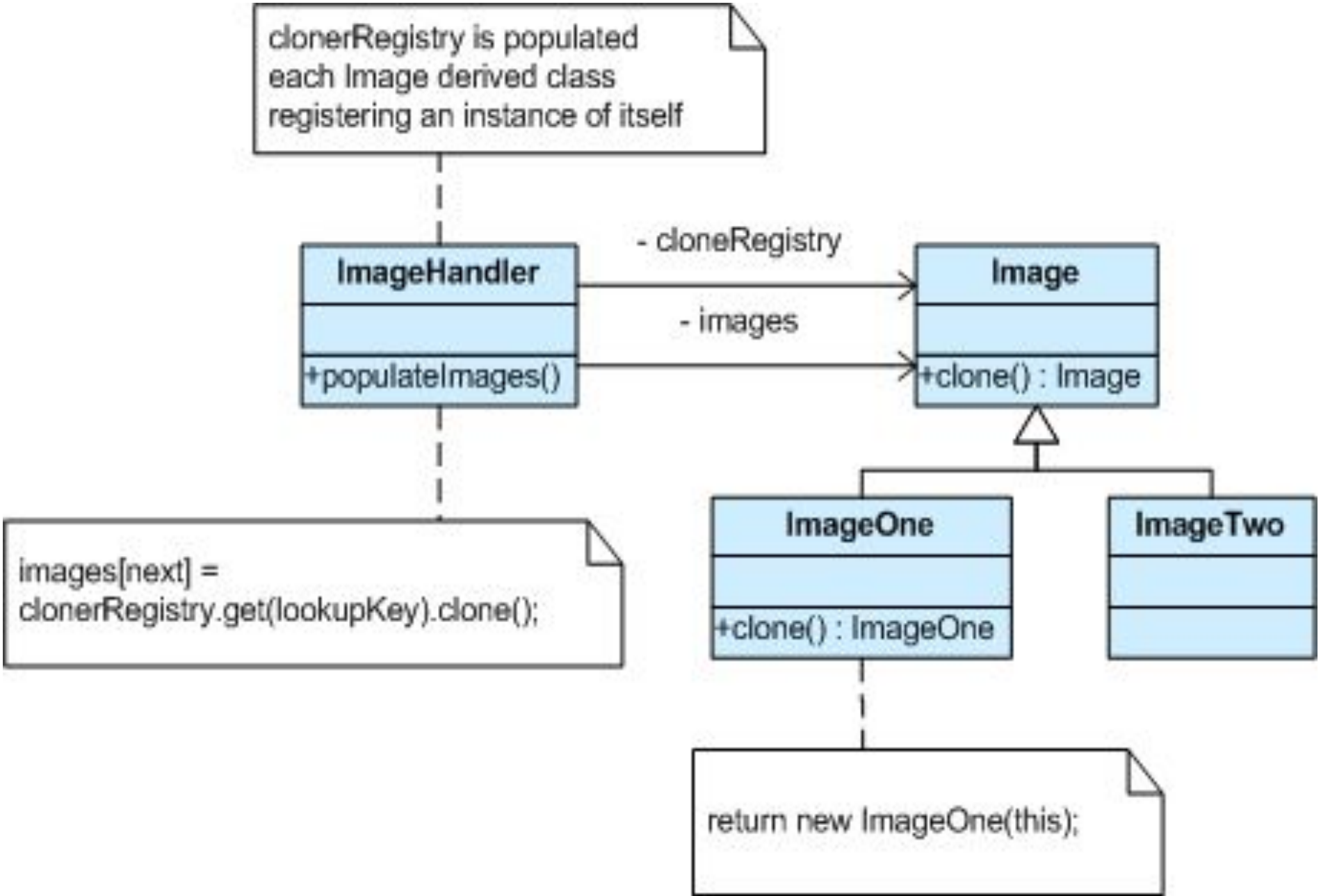
Prototype

- Padrão de Projeto Creacional que se baseia em criar instâncias de uma Classe através de cópias de um objeto da Classe (o **Protótipo**).
- Novas instâncias são clones da instância protótipo.
- Evita os custos inerentes de se criar objetos da forma tradicional (com **new**)

Prototype

- Uma classe abstrata especifica um método **clone()** puramente virtual
- Uma classe pode derivar nessa classe abstrata e implementar o método **clone()**
- O **Cliente**, então, em vez de invocar o new, pode chamar o método clone da superclasse abstrata, oferecendo como parâmetro qual a classe concreta derivada é desejada

Prototype


clonerRegistry is populated
each Image derived class
registering an instance of itself
ImageHandler
+populateImages()
- cloneRegistry
- images
Image
+clone() : Image
images[next] =
clonerRegistry.get(lookupKey).clone();
ImageOne
+clone() : ImageOne
ImageTwo
return new ImageOne(this);

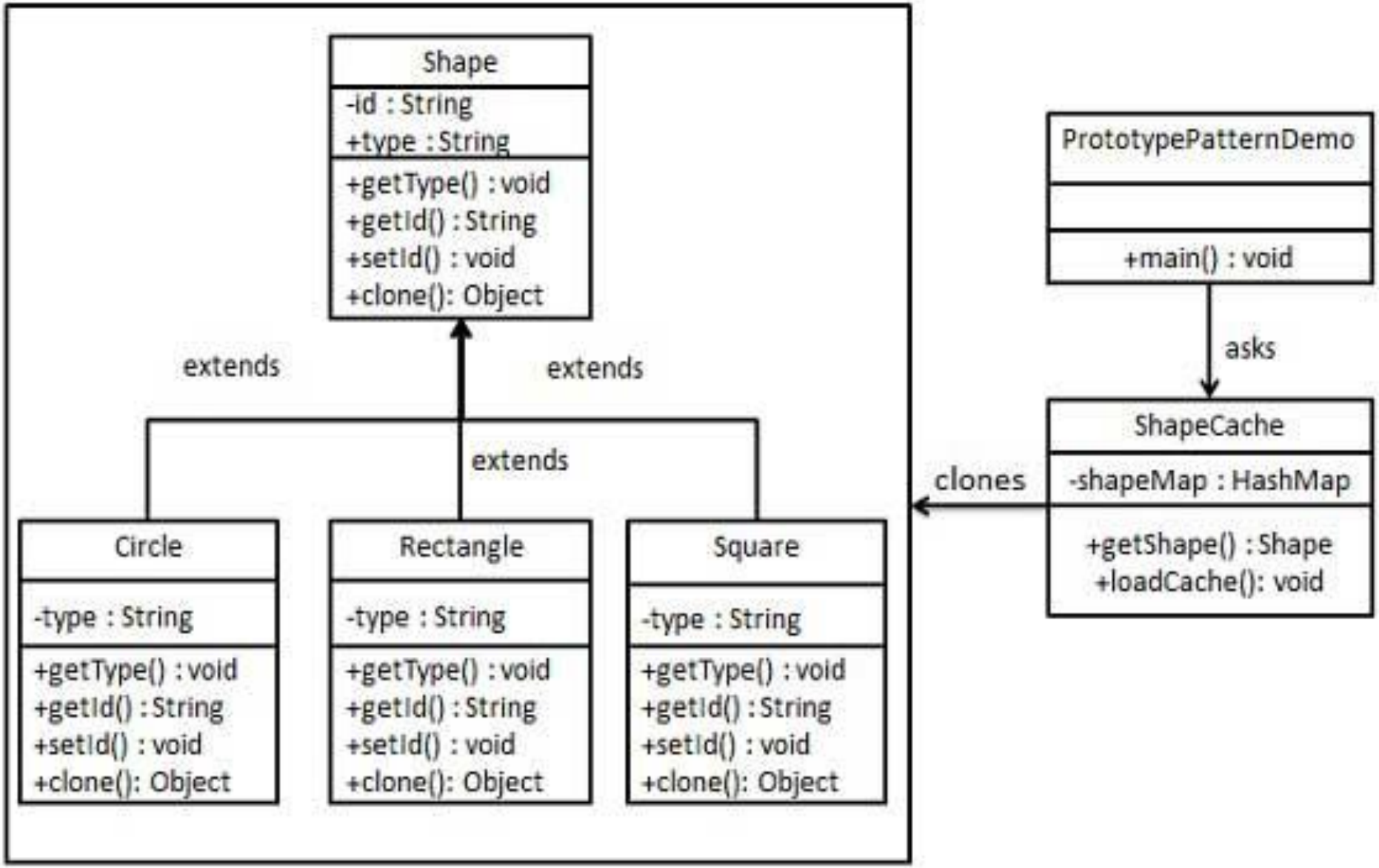
Prototype

- Passos
  - Adicionar método **clone()** à superclasse ou interface. Alternativamente, faça que sua classe implemente a interface **Cloneable**.
  - Projetar um registro (HashMap, Lista) que **mantém um cache de objetos protótipo**.
  - Projetar um método Factory que: **(1)** aceita ou não argumentos, **(2)** encontra o protótipo correto, **(3)** chama clone naquele objeto, **(4)** retorna o objeto resultante.
  - O **Cliente substitui** todas as referências operador **new** com chamadas para o **Factory Method**.

Prototype

- A **Abstract Factory** pode ser construída com **Factory Methods**, mas também com **Prototypes**
- Prototype **não requer subclasses** como outros padrões creacionais, mas **requer um método inicializador**
- Prototype é útil quando se espera **pouca variação de parâmetros de inicialização, ou inicialização é custosa**. Não é preciso fazer do zero se clonar é mais simples e barato.

Prototype


Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
clones
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void

Prototype

- Implementação:
- http://www.javacamp.org/designPattern/prototype.html

Exercício

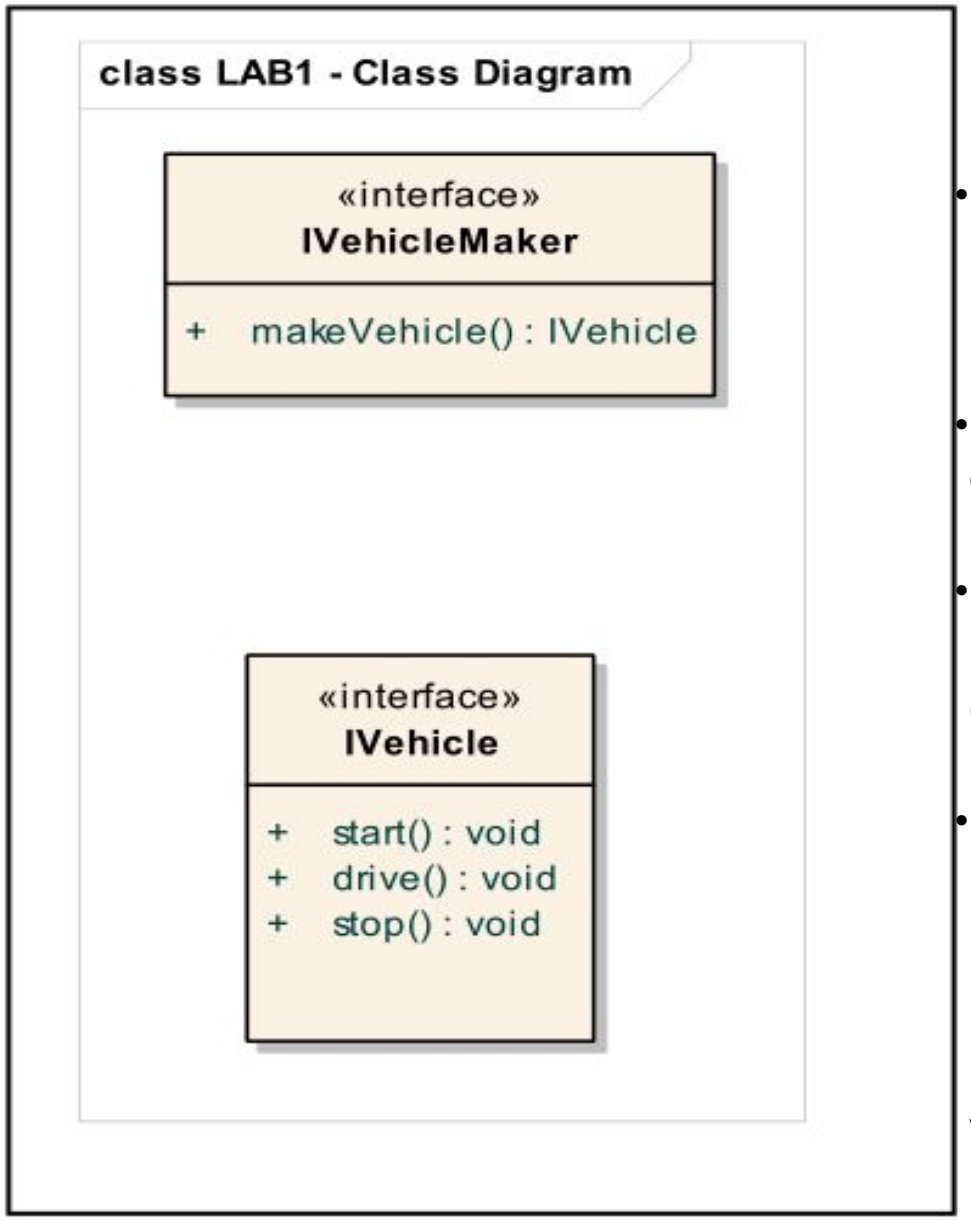


Figura 1. Interfaces

- Considere o domínio das fábricas de veículos. Considere que um veículo possui o seguinte co mportamento padrão: ligar, rodar e parar.
- Considere a interface **IVehicle**, mostrada na Fi gura 1.
- Existem vários fabricantes de veículos (IVehicle Maker) e cada uma delas possui o seu grupo d e modelos.
- Por exemplo, dois frabicantes são a Toytota e a Honda. A Toyota possui o Corolla, a Hilux e o Etios. A Honda possui o City, o Civic e o Fit. Existem no mercado diferetes fabriacntes (**IvehicleMaker**) cujo papel básico é fabricar veículos (makeVehicle).

Exercício

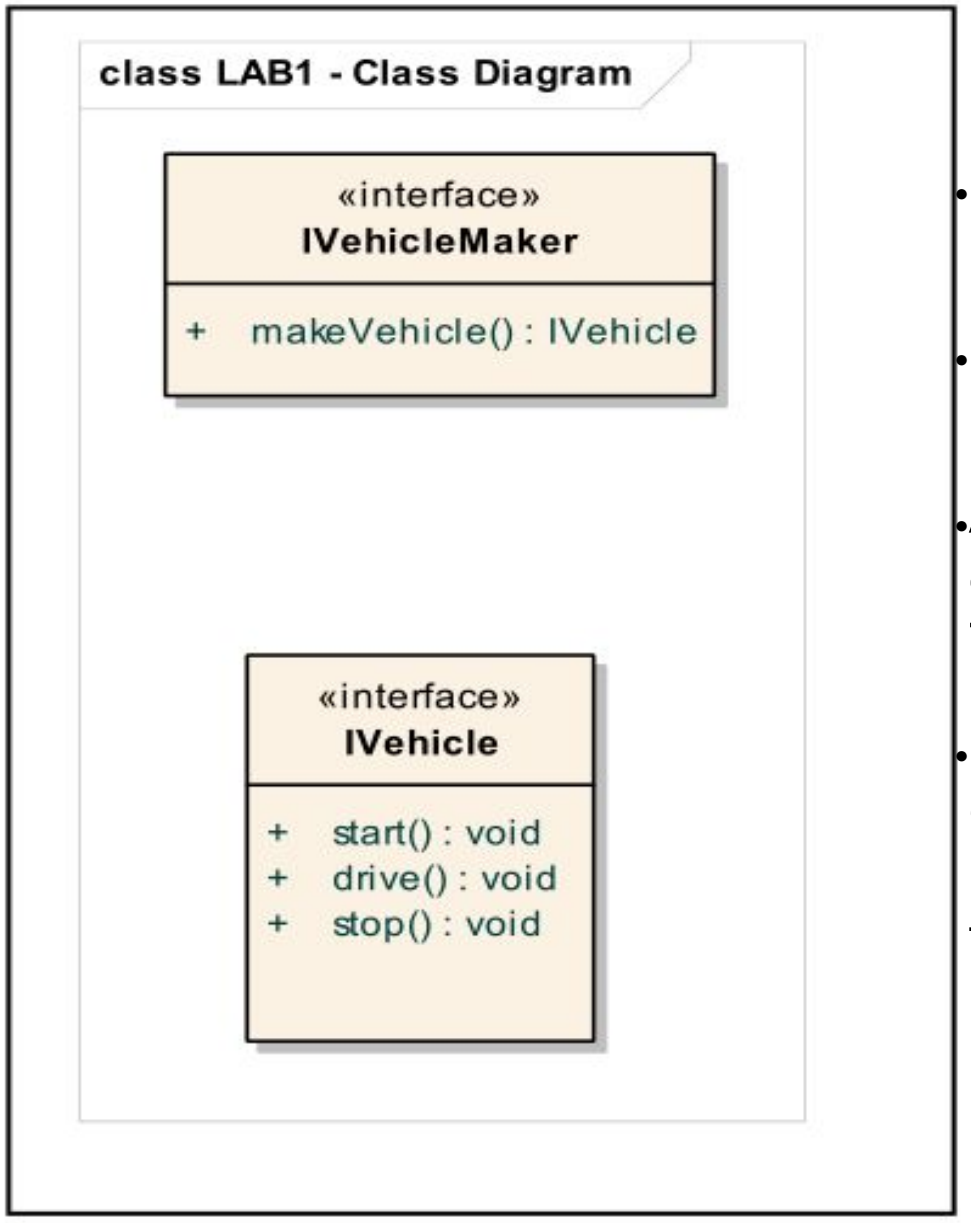


Figura 1. Interfaces

- Considere e aplique algum padrão **Factory Method** no exemplo.
- Considere e aplique o padrão **Abstract Factory** no exemplo.
- Aplique o padrão **Singleton** nas fábricas para que cada haja somente uma instância de Toyota e Honda, por exemplo.
- Crie o código Java mínimo que implementa as classes mencionadas e escreve um método main() simples que demonstre seu funcionamento.