PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

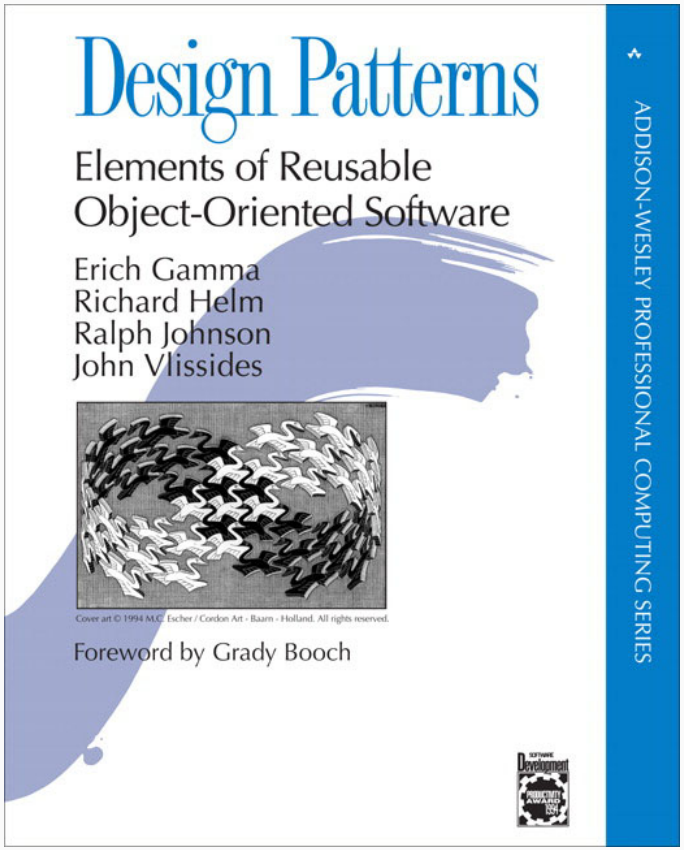
Daniel Cordeiro

11 de maio de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

GANG OF FOUR (GOF)

E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

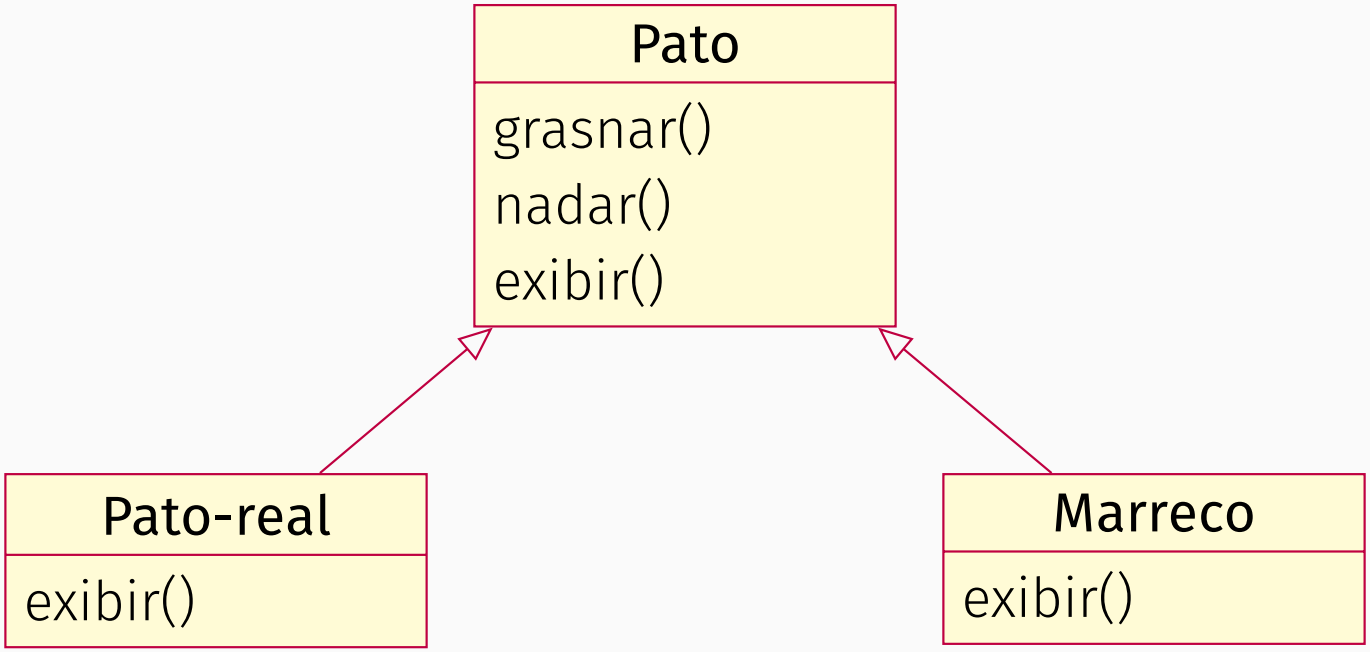


I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior. Patterns go beyond applying objects to some graphical shape example, with a shape class hierarchy and some polymorphic draw method. You really learn about polymorphism when you've understood the patterns. So patterns are good for learning OO and design in general.

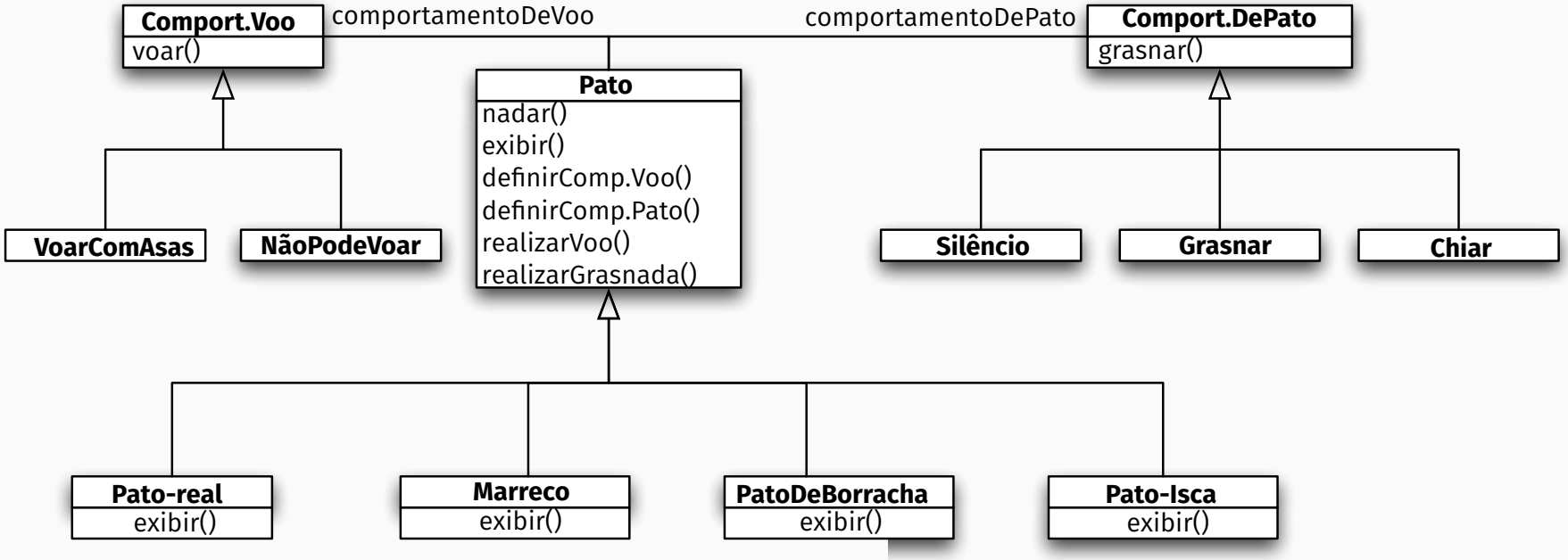
Erich Gamma

Exemplo:

Imagine um “simulador de patos em uma lagoa”, que é capaz de exibir uma grande variedade de espécies de patos nadando e grasnando.



NOVO DIAGRAMA DE CLASSE



VISÃO GERAL

- Um *padrão de projeto* é uma solução geral, repetível, para um problema recorrente em projetos de software
- Um padrão não é algo que pode ser transformado diretamente em código, mas sim uma descrição ou modelo de como resolver um problema em diferentes situações
- Um projeto de software eficiente por vezes envolve prevenir problemas sutis (veja exemplo do Pato); reutilizar padrões conhecidos pode evitá-los
- Além disso, padrões auxiliam na comunicação

TIPOS DE PADRÕES NO GOF

- Padrões de Criação
- Padrões Estruturais
- Padrões Comportamentais

PADRÕES DE CRIAÇÃO

- São padrões relacionados à instanciação de classes
- Podem ser divididos em:
  - padrões de criação de classes (usam herança)
  - padrões de criação de objetos (usam delegação)

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton

PADRÕES ESTRUTURAIS

São padrões relacionados à composição de classes (com herança) e objetos.

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Private Class Data
- Proxy

PADRÕES COMPORTAMENTAIS

São padrões tratam problemas relacionados à comunicação entre objetos.

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template method
- Visitor

PADRÕES DE CRIAÇÃO

Objetivo

- Garantir que uma classe tenha apenas uma única instância e prover um ponteiro global para acessá-la
- Encapsular o conceito de inicialização “just-in-time” ou inicialização tardia

Problema

A aplicação precisa de uma, e apenas uma, instância de um objeto. Além disso, a aplicação precisa conseguir acessar essa instância globalmente e precisa garantir que a instanciação do objeto ocorrerá tardiamente (só quando necessário)

COMO IMPLEMENTAR ALGO ASSIM?

SINGLETON: DISCUSSÃO

- Faça a classe dessa instância única ser responsável pela criação, inicialização, acesso e pelo cumprimento da restrição
- Declare a instância como um membro estático privado
- Forneça um método estático público que encapsule todo o código de inicialização e que dê acesso a essa instância única

SINGLETON: DISCUSSÃO

Quando usar Singleton?

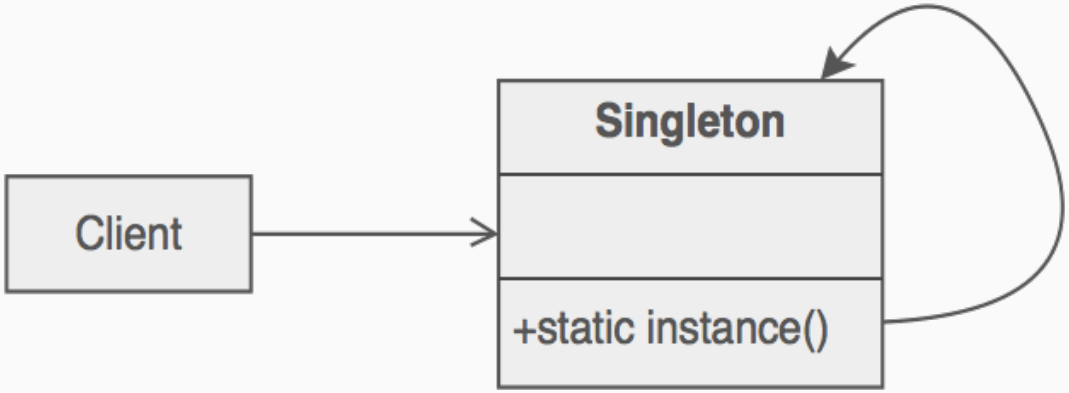
Deve-se considerar usar um Singleton quando:

- não for possível determinar qual deveria ser o “dono” dessa única instância
- precisarmos de inicialização tardia
- acesso global não puder ser provido de outra forma

Observações

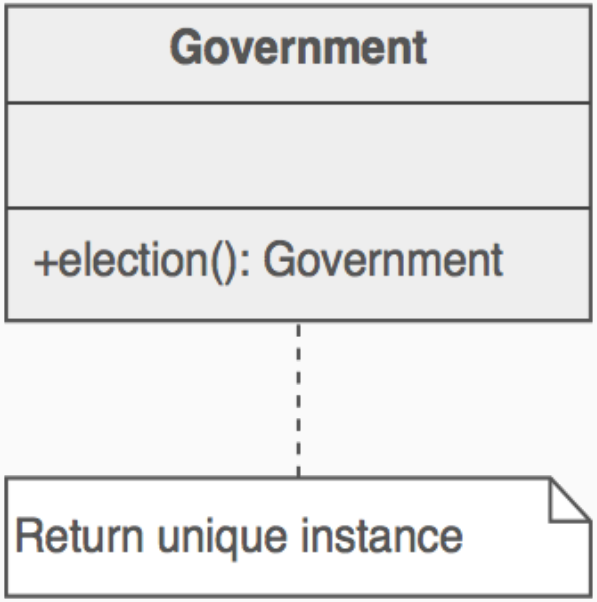
- Se os critérios não forem satisfeitos, então Singleton não é uma solução interessante
- O padrão pode ser estendido para permitir um número fixo de instâncias (ao invés de apenas uma)
- Não é possível criar subclasses de uma classe implementada com Singleton

SINGLETON: ESTRUTURA



Faça a classe ser responsável pelo acesso e pela inicialização no primeiro uso. A instância é um atributo estático privado. O método assessor é estático e público.

SINGLETON: EXEMPLO



O poder executivo de um país como o Brasil ou EUA é um Singleton. A constituição especifica que um presidente eleito tem um mandato por tempo determinado determina as regras de sucessão. Só é possível ter um presidente ativo por vez. Independentemente da identidade do eleito, o título *President[ea] da república federativa do Brasil* é um ponto global de acesso que identifica o eleito.

SINGLETON: IMPLEMENTAÇÃO

```
public class Singleton {
   private static Singleton instance;

   // Construtor privado, impede o uso do construtor padrão
   private Singleton() {
   }

   public static Singleton getInstance() {
     if(instance == null) {
       instance = new Singleton();
     }
   }

   return instance;
}
```

SINGLETON: IMPLEMENTAÇÃO

```
public class Singleton {
   private static Singleton instance;

   private Singleton() {
   }

   public static Singleton getInstance() {
     if(instance == null) {
        synchronized (Singleton.class) {
           if(instance == null) {
              instance = new Singleton();
           }
        }
     }
     return instance;
   }
}
```

SINGLETON: REGRAS GERAIS

- A vantagem de Singleton sobre variáveis globais é que você tem certeza absoluta do número de instâncias ativas e é fácil mudar de ideia de usar o número de instâncias que quiser
- Singleton é talvez o padrão mais usado incorretamente. Projetistas tendem a usá-lo sempre que precisam de variáveis globais (argh!)
- Singleton raramente é necessário. Se for mais fácil passar a instância como parâmetro (ao invés de recuperá-la de uma entidade global), não use Singleton!

ABSTRACT FACTORY: OBJETIVO

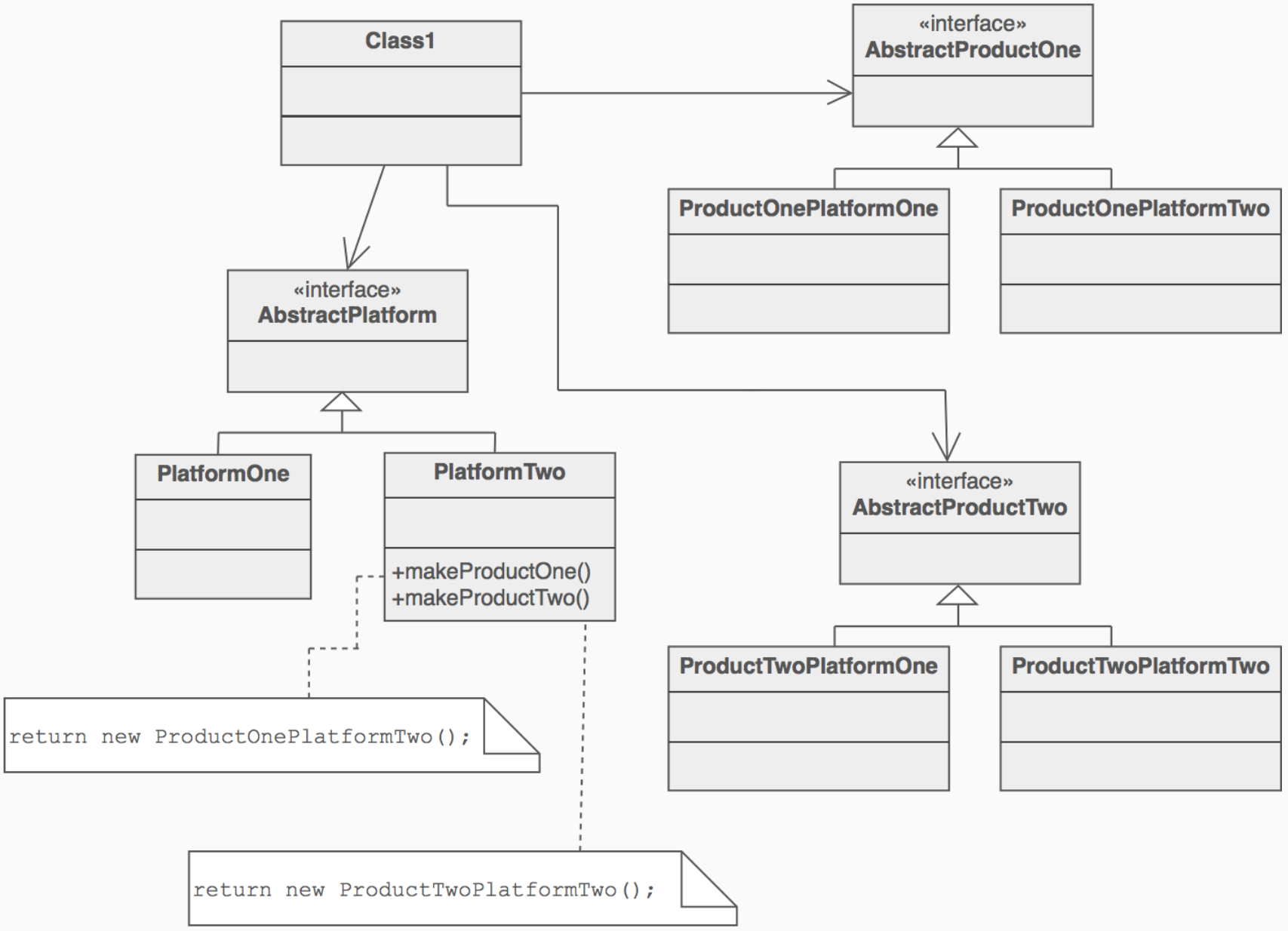
- Fornecer uma interface para a criação de famílias de objetos relacionados sem especificar suas classes concretas
- Criar uma hierarquia que encapsula diversas “plataformas” possíveis e a construção de um conjunto de produtos
- “Operador `new` considerado prejudicial”

ABSTRACT FACTORY: PROBLEMA

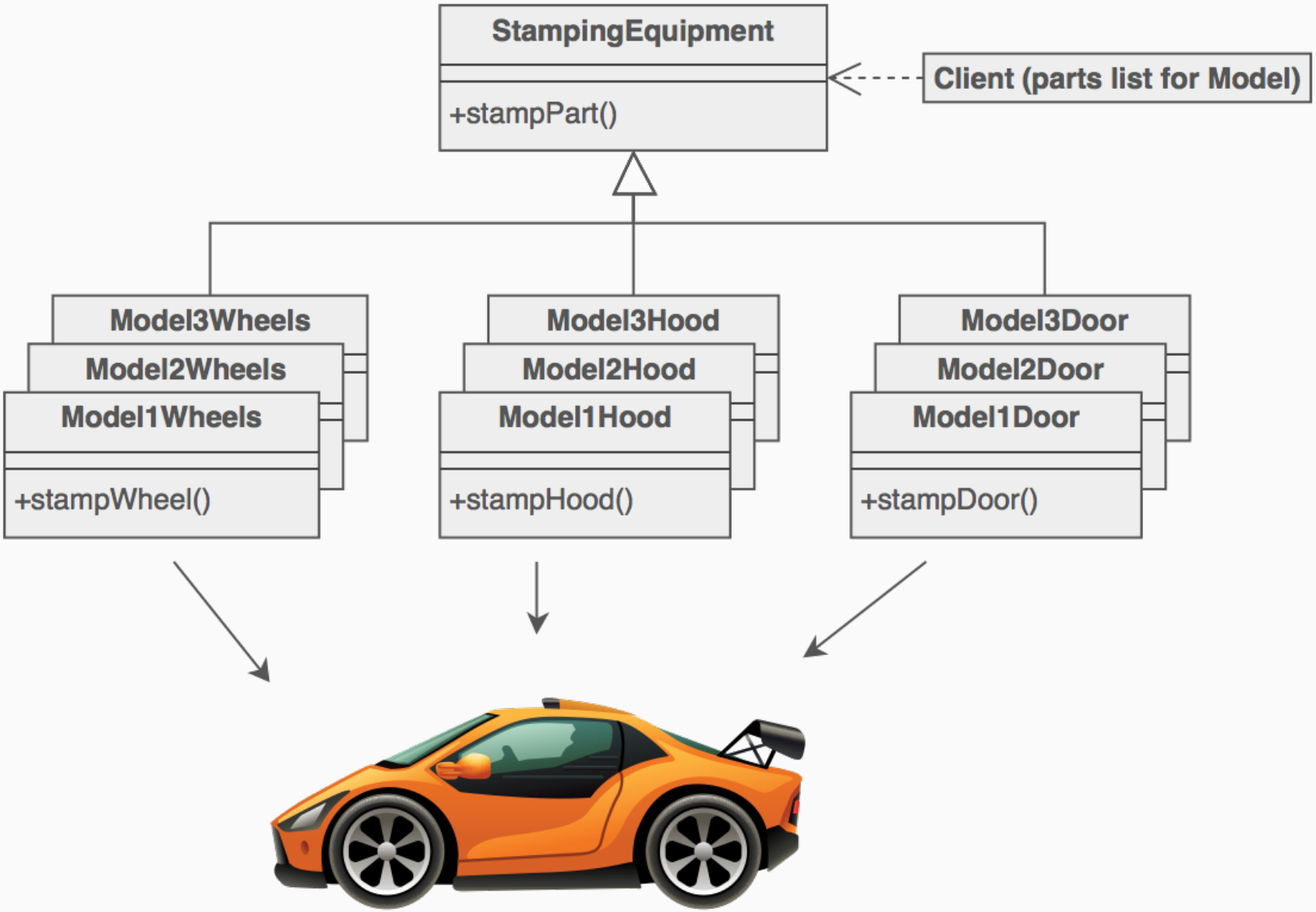
Se uma aplicação precisa ser portátil, ela precisa encapsular dependências da plataforma. Essas “plataformas” podem incluir: sistemas (gráficos) de gerenciamento de janelas, sistemas operacionais, bancos de dados, etc.

Acontece com frequência (mais do que gostaríamos) dessas dependências não serem projetadas antecipadamente, o que leva a códigos cheios de `#ifdef`.

ABSTRACT FACTORY: ESTRUTURA



ABSTRACT FACTORY: EXEMPLO



ABSTRACT FACTORY: IMPLEMENTAÇÃO

Suponha que queiramos criar interfaces gráficas para dois sistemas operacionais diferentes: Windows e MacOS

```
// AbstractProduct
public interface Window{
  public void setTitle(String text);
  public void repaint();
}
```

ABSTRACT FACTORY: IMPLEMENTAÇÃO

Suponha que queiramos criar interfaces gráficas para dois sistemas operacionais diferentes: Windows e MacOS

```java
// AbstractProduct
public interface Window{
  public void setTitle(String text);
  public void repaint();
}
```

```java
// ConcreteProductA1
public class MSWindow implements Window {
  public void setTitle(){
    // Comportamento específico do Windows
  }

  public void repaint(){
    // Comportamento específico do Windows
  }
}
```

```java
//ConcreteProductA2
public class MacOSXWindow implements Window {
  public void setTitle(){
    // Comportamento específico do MacOS
  }

  public void repaint(){
    // Comportamento específico do MacOS
  }
}
```

ABSTRACT FACTORY: IMPLEMENTAÇÃO

Agora precisamos fornecer uma fábrica abstrata:

```
//AbstractFactory
public interface AbstractWidgetFactory{
  public Window createWindow();
}
```

```
//ConcreteFactory1
public class MsWindowsWidgetFactory{
  // cria uma MSWindow
  public Window createWindow(){
    MSWindow window = new MSWindow();
    return window;
  }
}
```

```
//ConcreteFactory2
public class MacOSXWidgetFactory{
  // cria uma MacOSXWindow
  public Window createWindow(){
    MacOSXWindow window = new MacOSXWindow();
    return window;
  }
}
```

ABSTRACT FACTORY: IMPLEMENTAÇÃO

```
// Cliente
public class GUIBuilder{
  public void buildWindow(AbstractWidgetFactory widgetFactory){
    Window window = widgetFactory.createWindow();
    window.setTitle("New Window");
  }
}
```

ABSTRACT FACTORY: IMPLEMENTAÇÃO

```
// Cliente
public class GUIBuilder{
  public void buildWindow(AbstractWidgetFactory widgetFactory){
    Window window = widgetFactory.createWindow();
    window.setTitle("New Window");
  }
}

public class Main{
  public static void main(String[] args){
    GUIBuilder builder = new GUIBuilder();
    AbstractWidgetFactory widgetFactory = null;

    // verifica a plataforma atual
    if(Platform.currentPlatform()=="MACOSX"){
      widgetFactory  = new MacOSXWidgetFactory();
    } else {
      widgetFactory  = new MsWindowsWidgetFactory();
    }

    builder.buildWindow(widgetFactory);
  }
}
```

ABSTRACT FACTORY: LISTA DE VERIFICAÇÃO

- Decida se *independência de plataforma* e serviços de criação são uma fonte de dor de cabeça
- Mapeie a matriz de “plataformas” vs. “produtos”
- Defina uma classe derivada como a fábrica para cada plataforma que encapsule todas as referências ao operador `new`
- O código cliente deve aposentar o uso de `new` e começar a usar apenas os métodos da fábrica para a criação de instâncias do objeto produto

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/`