# Types of Patterns

**Creational** patterns are for creating objects

**Structural** patterns are for object connections

**Behavioral** patterns are for object behavior

# Design Patterns

- The Strategy Pattern
- **The Factory Method**
- Generics
- The Abstract Factory Pattern
- The State Pattern
- The Observer Pattern
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- The Builder Pattern
- Fallen Patterns
  - The Singleton Pattern
  - The Visitor Pattern

## Factories

- A simple creational pattern
- A Factory creates objects to be handed off to other classes for their use
- A factory is just a class with "create" methods
- Example: Traffic Model
  - A car factory creates cars and places them on the roadway
- Example: Retail website
  - Creates a webpage based on a user's profile

# Factory Example: Coffee Machine

```
class CoffeeMachine {
    CoffeeMachine(
        int nCoffeeBeans,
        int nEspressoBeans ) {…}

    Coffee cafeAmericano() {
         n_coffee_beans -= 20;

         if( n_coffee_beans < 0 )
             throw new InsufficientBeans();

         return new Coffee( 20, 0 );
    }

    Coffee cafeMocha() { … }
}
```

## Coffee Machine Factory

Each coffee uses beans.

The factory makes sure we have enough.

Avoids having to check bean count each time coffee created

By centralizing this behavior, we have only one point of failure.

## Guideline

Don’t use a factory unless it actually simplifies your code (YAGNI)

Too many factories is a *smell*

## Code Smells

- Generic term for “code that is probably bad somewhere”
- A smell in general is an aspect of software that suggests something is wrong elsewhere, but you can’t put your finger on why
- Examples: http://wiki.c2.com/?CodeSmell
- Too many factories smells like cargo cult programming

## Cargo Cult Programming

- Copying code that you don’t understand because you’ve seen it before
- Characteristic of inexperienced programmers
- What’s the difference between:
  - Widget widget = factory.makeWidget();
  - Widget widget = new Widget();

## New Pattern: Factory Method

- Extends the flexibility of factories
  - An interface/class with an abstract method (the FactoryMethod)
  - The method creates a new object
  - Type determined by the implementer
  - Usually a non-abstract method calls the abstract method.
  - Similar to the strategy pattern, but for data.

## Example: Factory method

```
class Bev {
   float caffeine;
   float darkness;
   String name;

   Coffee( … ){ … }
}

class BevMachine {
   //state
   int beanCount;

   //public interface
   public Bev makeCoffee(int beans)   {
      beanCount -= beans;
      if( beanCount < 0 ) throw …

      return brewCoffee( beans );
   }

   //factory method
   private abstract Bev brewCoffee(int beans);
}
```

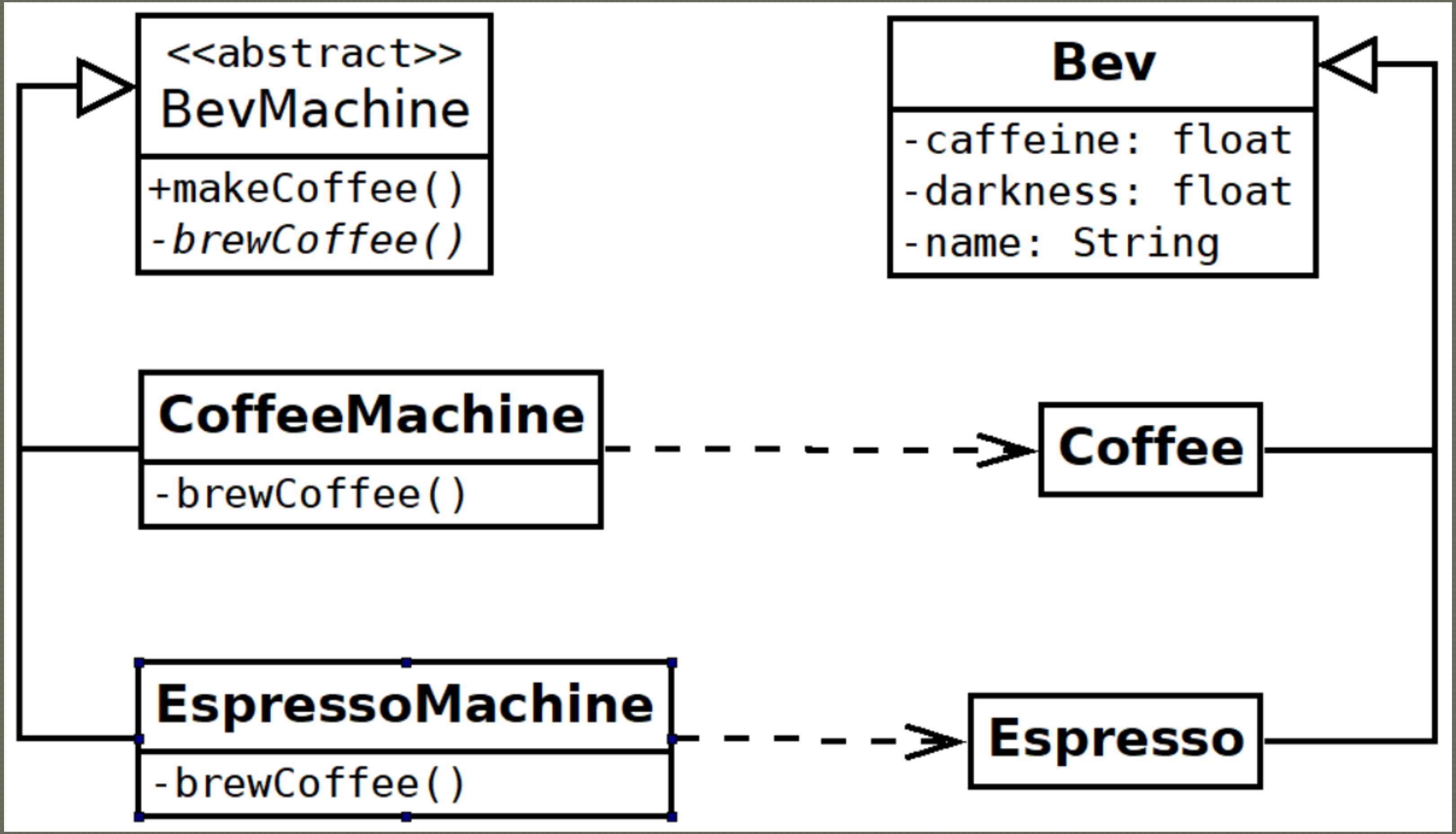
## Example: Concrete Factories

```
class CoffeeMachine extends BevMachine {
  //factory method
  private Bev brewCoffee(int beans) {
    return new Coffee(beans);
  }
}

class EspressoMachine extends BevMachine {
  //factory method
  private Bev brewCoffee(int beans) {
    return new Espresso(beans);
  }
}
```

## Example: Individual Beverage Types

```
class Coffee extends Bev {
    Coffee(int beans) {
        super( (float)beans / 20.0,
               (float)beans / 30.0,
               “Coffee” );
    }
}

class Espresso extends Bev {
    Espresso(int beans) {
        super( (float)beans / 5.0,
               (float)beans / 10.0,
               “Espresso” );
    }
}
```

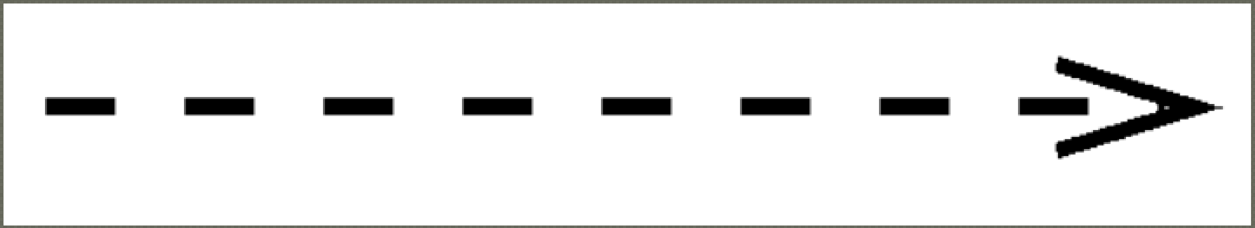
# UML Diagram



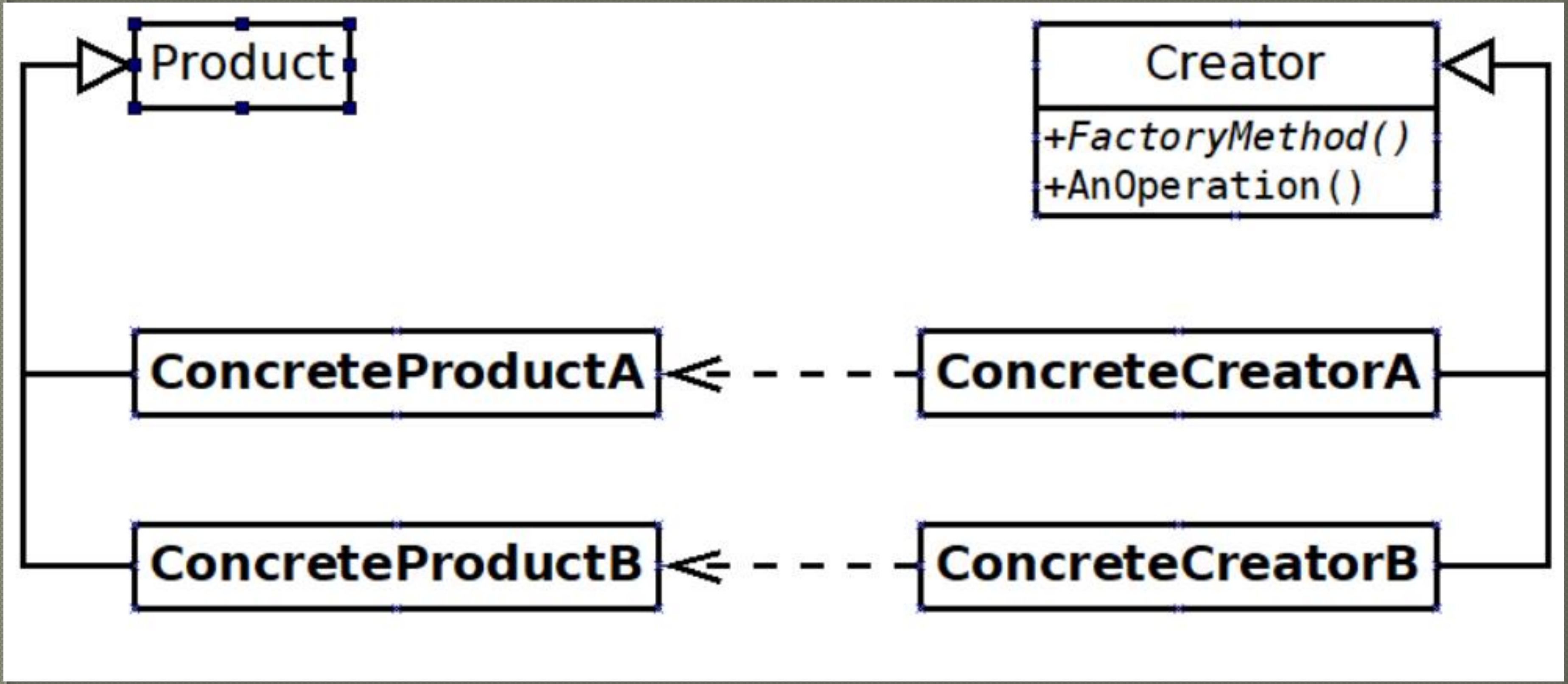
<<abstract>>
BevMachine
+makeCoffee()
-brewCoffee()
Bev
-caffeine: float
-darkness: float
-name: String
CoffeeMachine
-brewCoffee()
Coffee
EspressoMachine
-brewCoffee()
Espresso

## Dependency

- The dependency connection
- Calls methods or creates objects



## Generic Diagram



## Nomenclature

- The creator is the **factory**
- FactoryMethod is the actual **factory method**
- The whole pattern is the **factory method pattern**

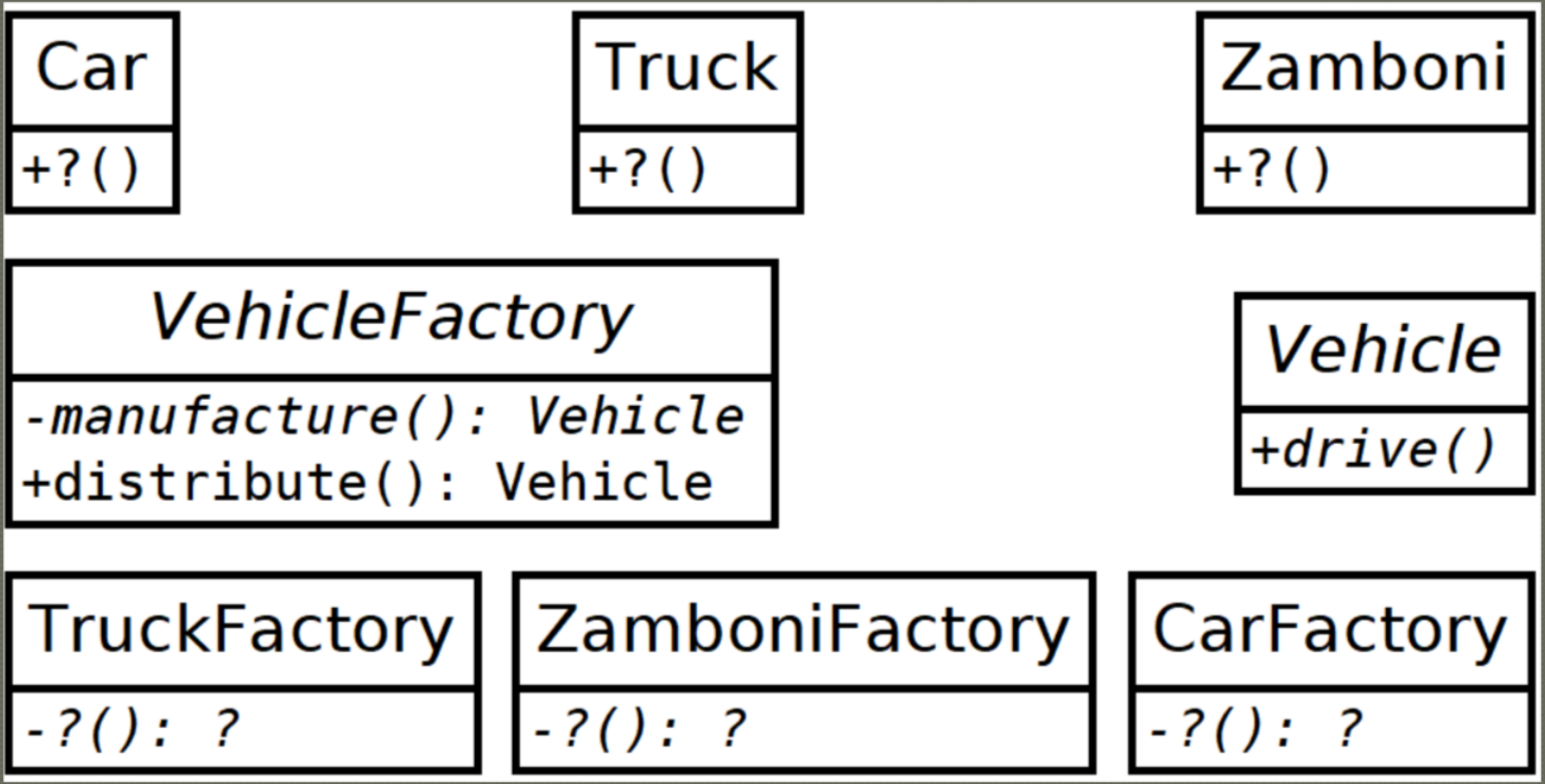
## Regarding the Private Method

The actual factory method can be private

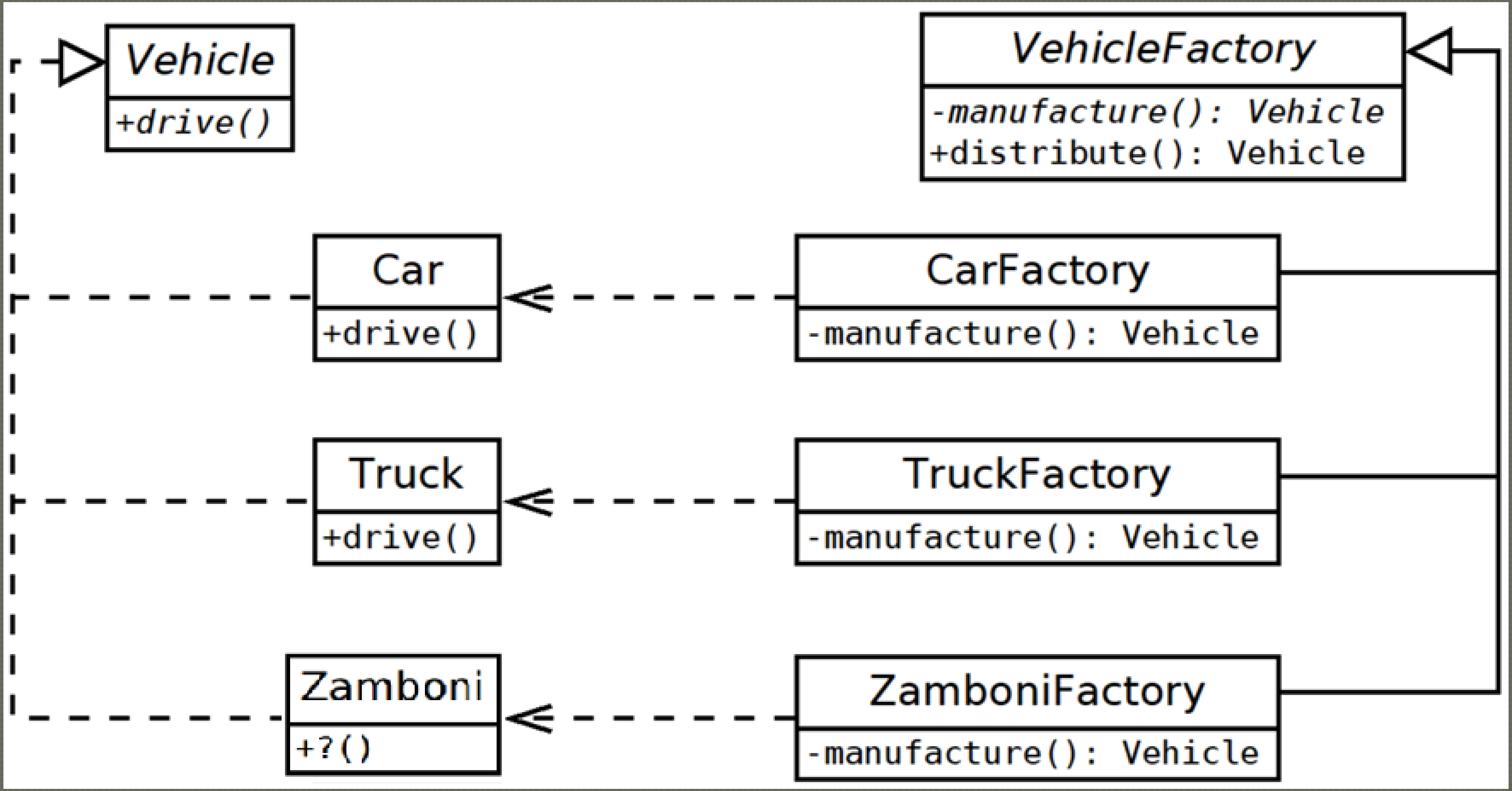
We can thus isolate the actual creation from other required operations that take place before and after.

If this is unnecessary, and there is no state, make the factory an interface.

# Example Puzzle



# Puzzle Answer



## YAGNI

- Often times factories and the factory method patterns can be replaced with static methods
- If your factory does not need state or access control, it should not be a class
- If a constructor would work fine, just use that
- Don’t make a factory just to hide a case statement (this is common online)

## Factory Methods vs Static Constructors

Static constructors are not to be inherited.

Mainly useful when multiple ways to instantiate a class with the same kind of input:

Example: make a document from a data string or a filename string

## Static Constructor Example

```
public class Document {
    private Document( … ) {…}

    public static FromFile( String file );
    public static FromString( String str);
}
```

## Mission

- Apply the Factory Method to your Project