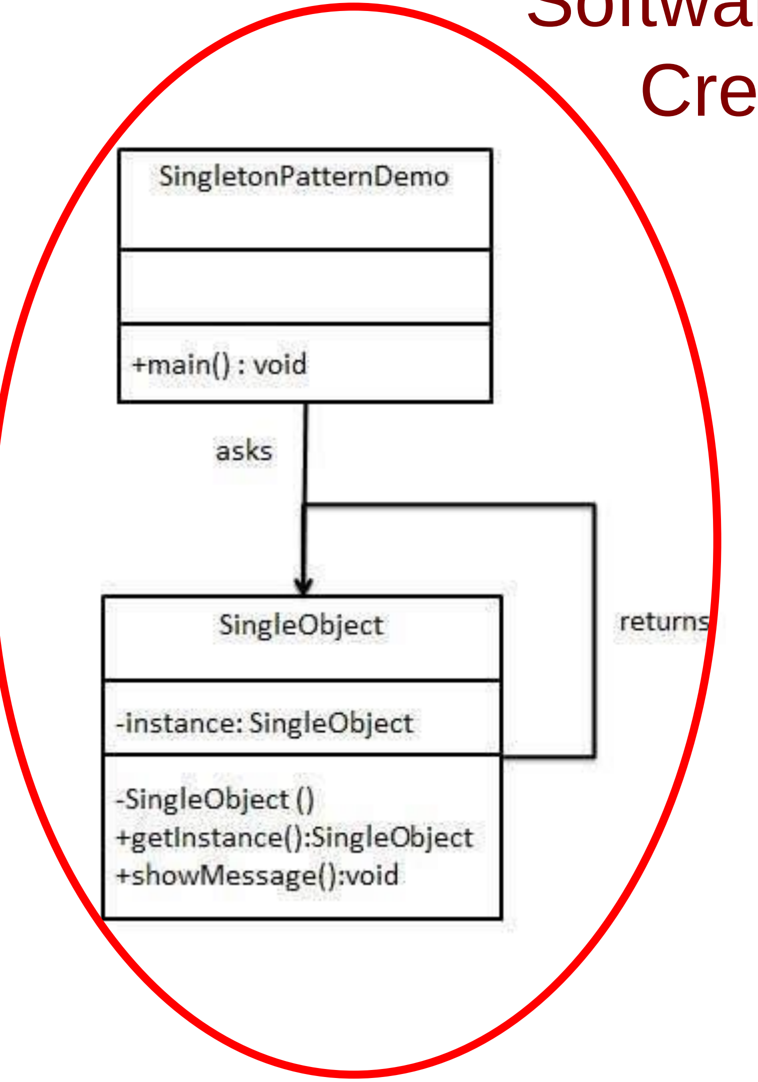
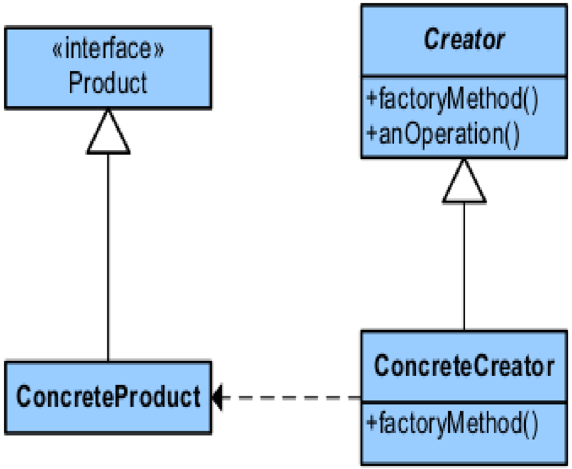
Software Design Patterns: Creational Patterns

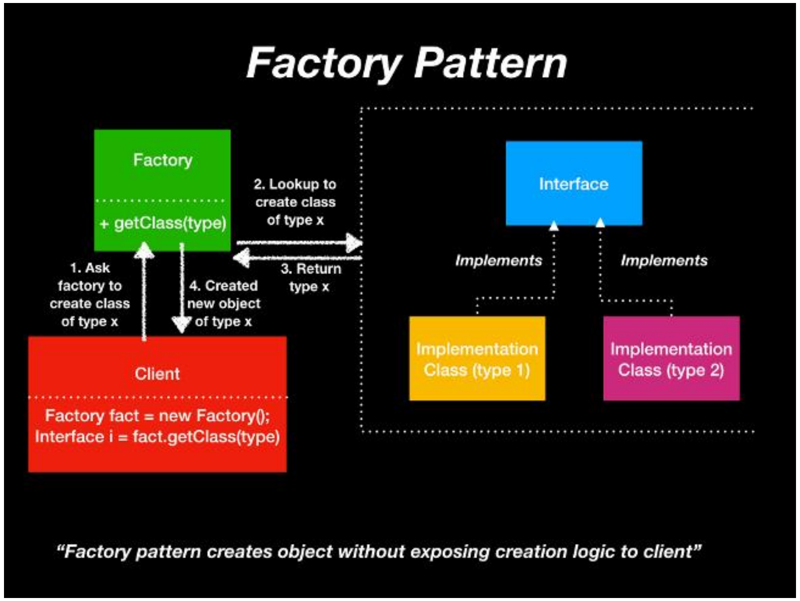


Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.




Factory Pattern
Factory
+ getClass(type)
2. Lookup to create class of type x
3. Return type x
1. Ask factory to create class of type x
4. Created new object of type x
Client
Factory fact = new Factory();
Interface i = fact.getClass(type)
Interface
Implements
Implements
Implementation Class (type 1)
Implementation Class (type 2)
"Factory pattern creates object without exposing creation logic to client"

Creational Design Patterns:

as defined in Elements of Reusable OO software

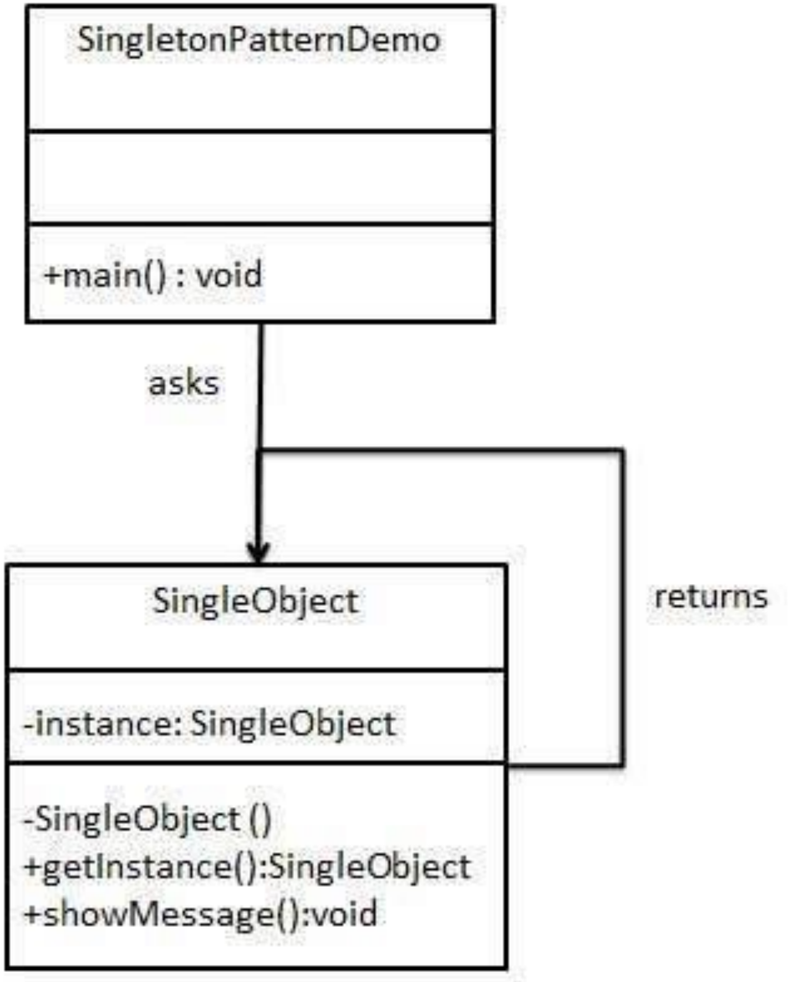
- Creational design patterns *abstract the instantiation process*.
  - Abstracting the instantiation process allows us to build systems that are independent of *how objects are created*, *composed*, and *represented*.

- These patterns are more important as systems evolve to depend more on *object composition* than class inheritance.
  - Creating objects with particular behavior requires a more specialized instantiation process.

- There are two important themes with Creational patterns:
  - They *encapsulate* knowledge about which concrete classes the system uses.
  - They *hide* how instances of these classes are created.

Singleton Pattern

Intent: Ensures that a class has only *one instance* of a specific class, and provides a *global* point of access to it.



Tutorials Point

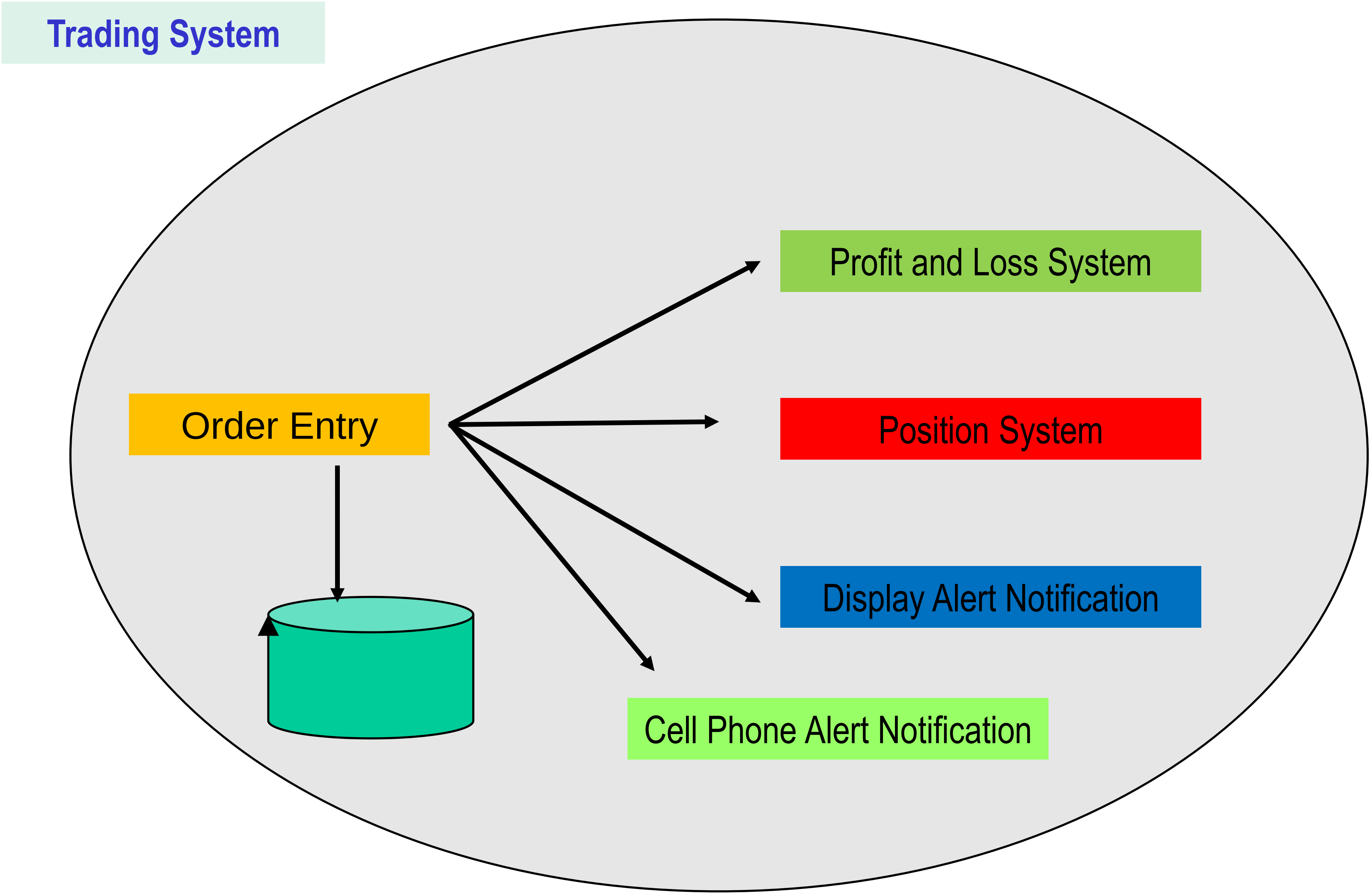
Singleton Pattern:

As defined in Elements of Reusable OO Software

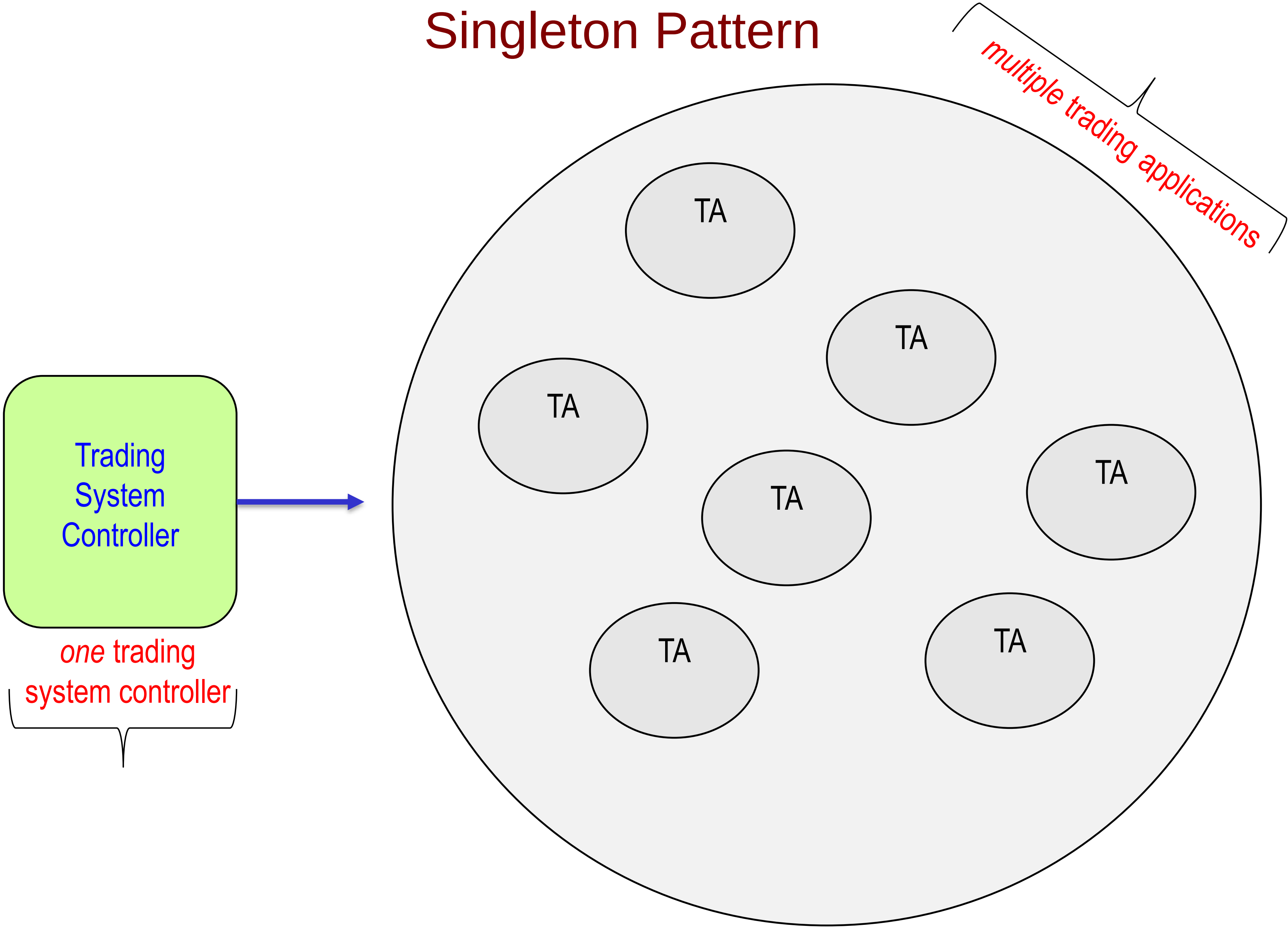
- Motivation **and Applicability**: It is important for some systems that there exist only one instance of a class (i.e. spooler for a printer, memory resource allocator, run-time stack, window manager, etc.)
  - How do we ensur ance is easily accessib
  - **There must be e e accessible to cli**
  - When the sole ins clients should be able to use an extended instance without modifying their code.

What if you wanted to ensure that there existed only ONE instance of a specific object?

Trading System Example



Singleton Pattern



multiple trading applications

TA

TA

TA

TA

TA

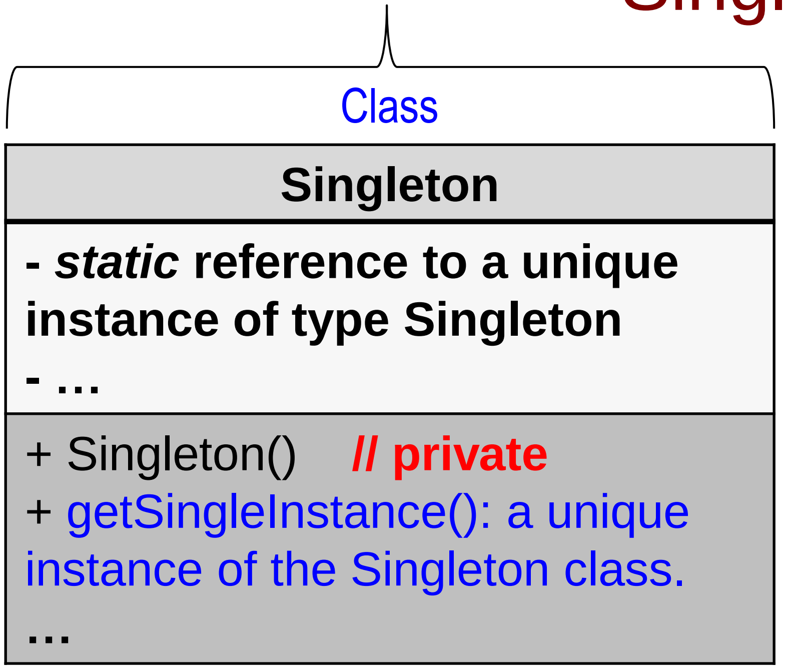
TA

TA

Trading System Controller

one trading system controller

Singleton Pattern



Class

Singleton

- *static* reference to a unique instance of type Singleton

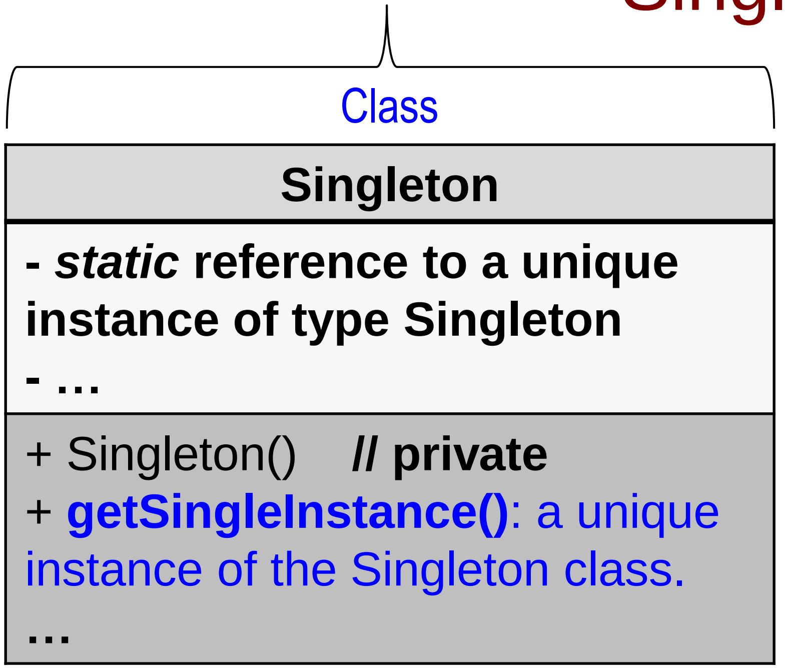
- …

+ Singleton() ***// private***

+ getSingleInstance(): a unique instance of the Singleton class.

…

Singleton Pattern



Class

Singleton

- *static* reference to a unique instance of type Singleton

- …

+ Singleton() ***//* private**

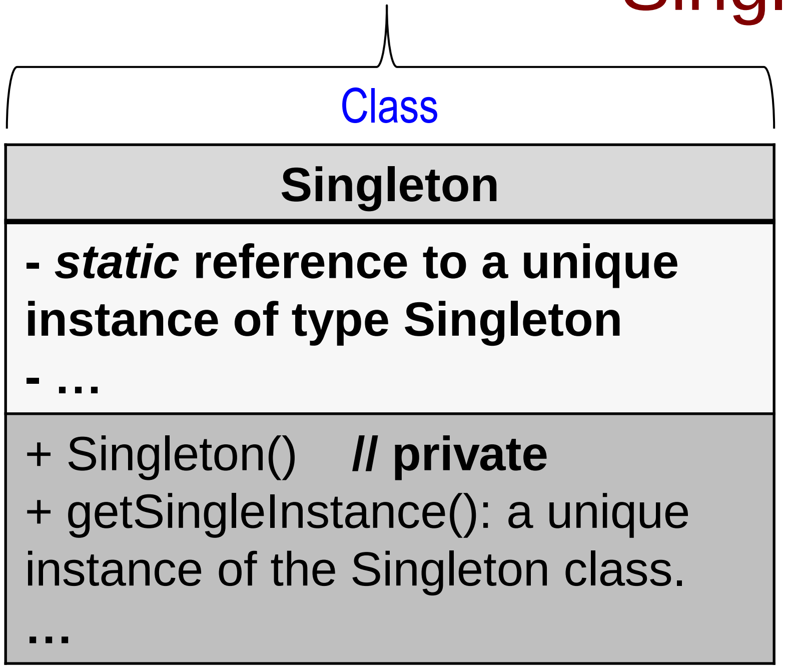
+ **getSingleInstance()**: a unique instance of the Singleton class.

…

`getSingleInstance` must be a

static method!

Singleton Pattern



Class

Singleton

- *static* reference to a unique instance of type Singleton

- …

+ Singleton() ***// private***

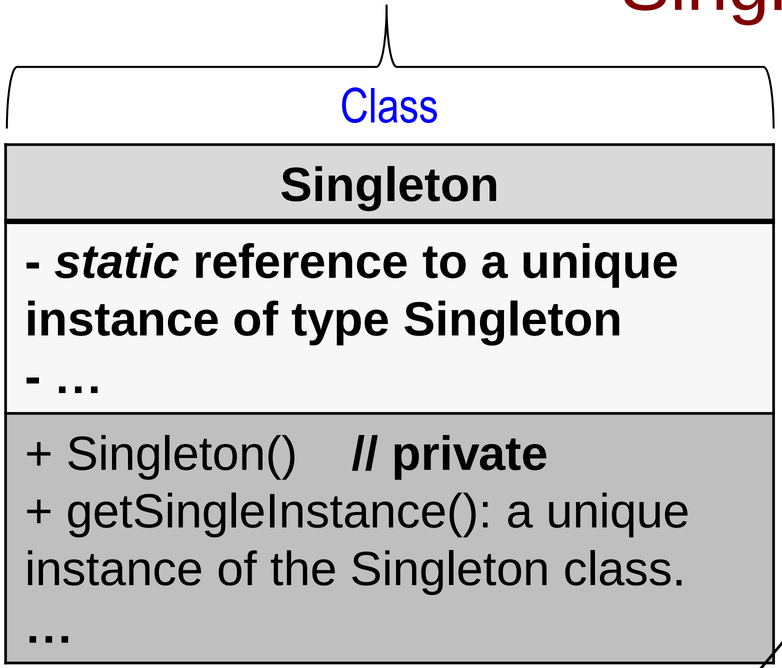
+ getSingleInstance(): a unique instance of the Singleton class.

…

Declares a single reference to the instance of the class we are interested in.

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

    private Singleton() { ... }

    public static Singleton getSingleInstance() {
       if (singleInstance == null)
           singleInstance = new Singleton();

       return singleInstance;
    }
} // class
```

Singleton Pattern



Class

Singleton

- *static* reference to a unique instance of type Singleton

- …

+ Singleton()   ***// private***

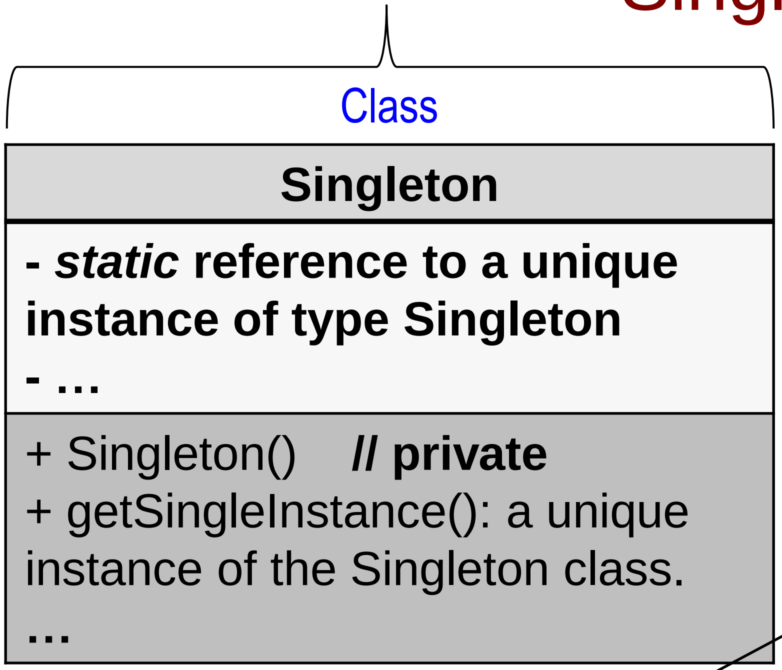
+ getSingleInstance(): a unique instance of the Singleton class.

…

Ensures that there is no way to construct an instance of this class, from outside of this class!

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

    private Singleton() { ... }

    public static Singleton getSingleInstance() {
       if (singleInstance == null)
           singleInstance = new Singleton();

       return singleInstance;
    }
} // class
```

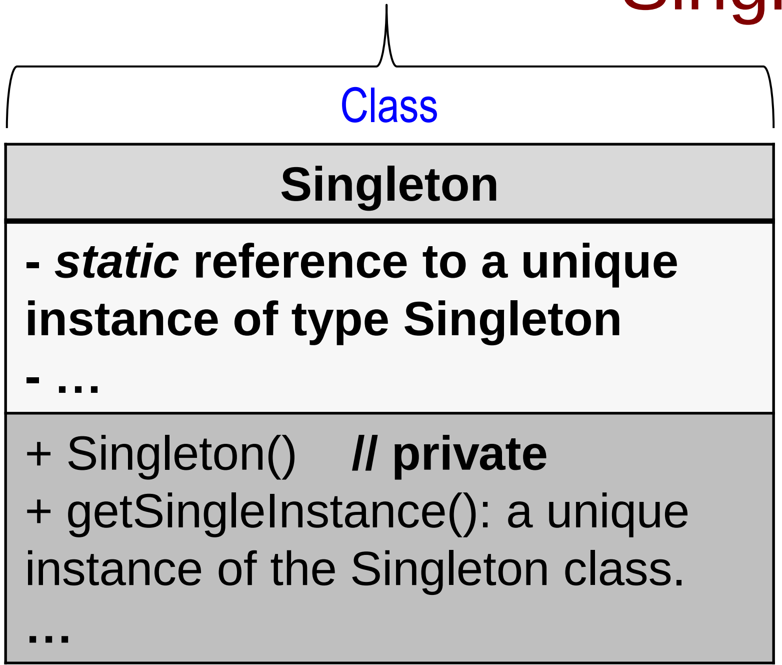
Singleton Pattern


Class
Singleton
- *static* reference to a unique instance of type Singleton
- ...
+ Singleton()   *// private*
+ getSingleInstance(): a unique instance of the Singleton class.
...

Allows us to invoke this method from outside of this class!

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

    private Singleton() { ... }

    public static Singleton getSingleInstance() {
       if (singleInstance == null)
           singleInstance = new Singleton();

       return singleInstance;
    }
} // class
```

Singleton Pattern



Class

Singleton

- *static* reference to a unique instance of type Singleton

- …

+ Singleton()   ***// private***

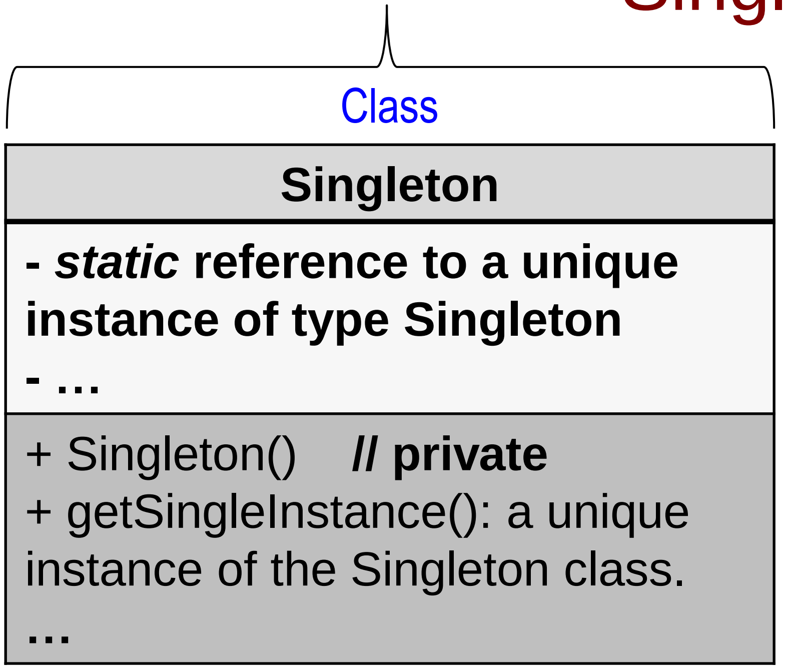
+ getSingleInstance(): a unique instance of the Singleton class.

…

Allows us to invoke this method without first creating an instance of this class.

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

    private Singleton() { ... }

    public static Singleton getSingleInstance() {
       if (singleInstance == null)
           singleInstance = new Singleton();

       return singleInstance;
    }
} // class
```

Singleton Pattern



- *static* reference to a unique instance of type Singleton

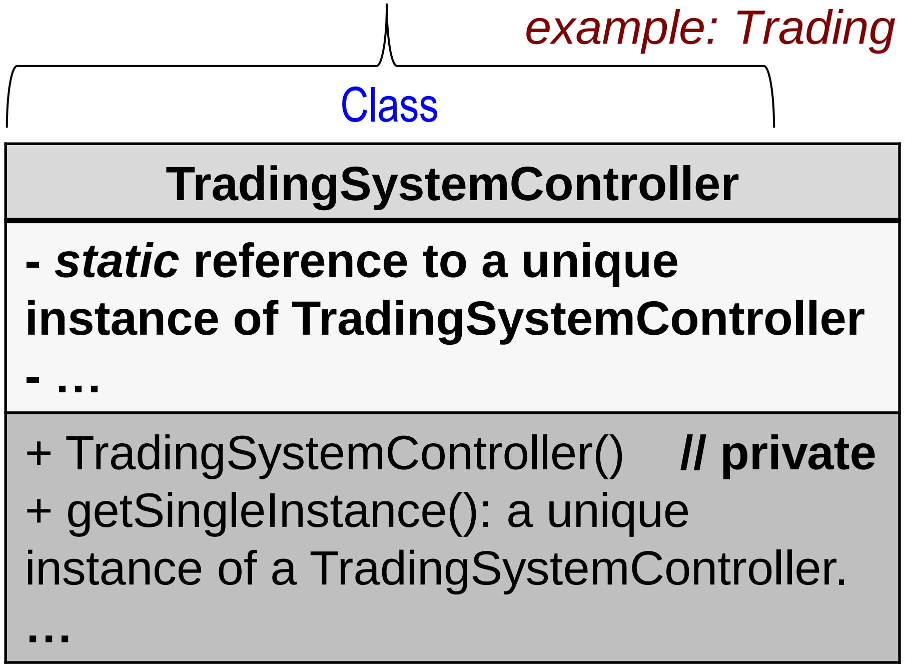
+ Singleton() ***// private***
+ getSingleInstance(): a unique instance of the Singleton class.

As this call is being made from within a method of the class, we are allowed to call the constructor – even though it is declared to be *private*.

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

    private Singleton() { ... }

    public static Singleton getSingleInstance() {
       if (singleInstance == null)
           singleInstance = new Singleton();

       return singleInstance;
    }
} // class
```

Singleton Pattern:

example: Trading System Controller



TradingSystemController

- *static* reference to a unique instance of TradingSystemController
- ...

+ TradingSystemController() ***// private***
+ getSingleInstance(): a unique instance of a TradingSystemController.
...

```
public class LaunchTradingSystem
{
  public static void main( ... )
  {

    TradingSystemController tsc =
      TradingSystemController.
         getSingleInstance();
      tsc.createTS(trader)
  }
}
```

```
public class TradingSystemController {
    private static TradingSystemController singleInstance;

    private TradingSystemController() { ... }

    public static TradingSystemController getSingleInstance() {
       if (singleInstance == null)
           singleInstance = new TradingSystemController();

       return singleInstance;
    }
} // class
```

Singleton Pattern:

As defined in Elements of Reusable OO Software

- **Consequences**: **(Advantages/Disadvantages)**
  - Controlled access to one instance of a class.
  - Avoids the alternative of creating a global reference to the specified instance.
  - Allows you to c … instance of the … rn is appropriate h … e Factory pattern
  - **Even though w … program's nar … l like object.**
  - In a distributed client server model, need to synchronize the creation of the instance.

Ensures that you only have one single instance of a class… but it ensures that you only have one single instance of a class!

Singleton Pattern:

Elements of Reusable OO Software

- **Implementation Issues**: Some argue that you should never use the Singleton pattern. Why?
  - Providing global access: This is contrary to a core principle to software programming which is to avoid the use of global variables.
  - Single instance: Some argue that this is not a reasonable assumption for any Class. You can really never be certain that at some future point you will never-ever need more than one instance.
  - “One man’s constant is another man’s variable” Allan Perlis
    - This is applicable here, as we should never make assumptions about future growth. Although it is perfectly fine for an application to have only one instance of something, but it may not be perfectly fine to force this without any flexibility.

Factories:
*the industrialization of **object creation***

Program to an *interface* and not an *implementation*.

Coding to an *interface*, insulates our code from future changes.

Allows us to encapsulate the instantiation process of **concrete** types.

- Allows client code to depend only on the interface and not on the concrete classes required to instantiate them.
- Avoids code duplication of code at the client level.

Code written to an interface will work with any new classes of that interface through *polymorphism*. **Open Close Principle**.

Factories:
the industrialization of ***object creation***

Program to an *interface* and not an *implementation*.

```
{

    Animal dog = new Dog();
    Animal cat = new Cat();
    Animal mouse = new Mouse();
```

What if the application itself does not know exactly what objects it needs or exactly how to create them?

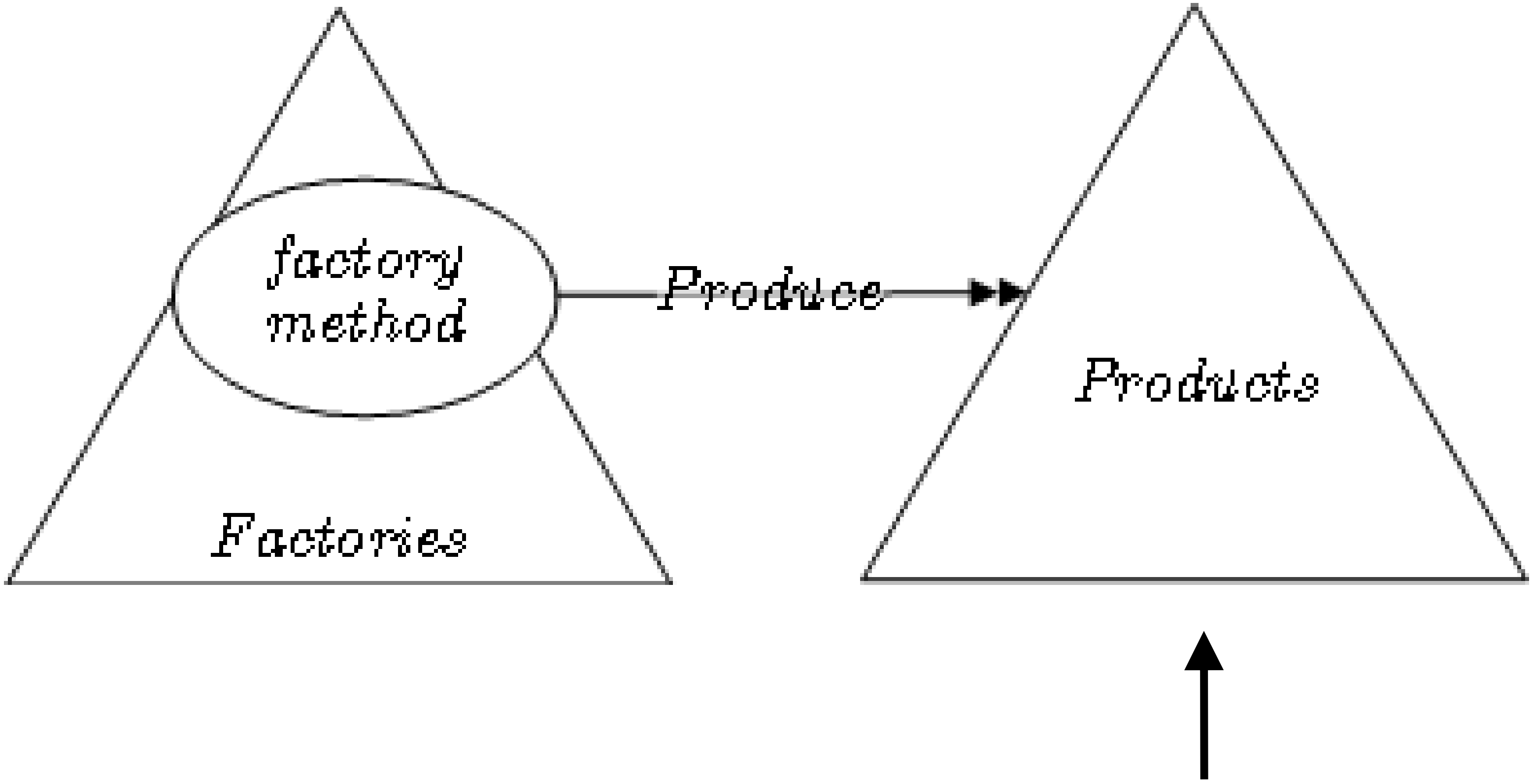
Factories:
*the industrialization of **object creation***

Program to an *interface* and not an *implementation*.

```
{

   Animal dog = new Dog();
   Animal cat = new Cat();
   Animal mouse = new Mouse();

}
```

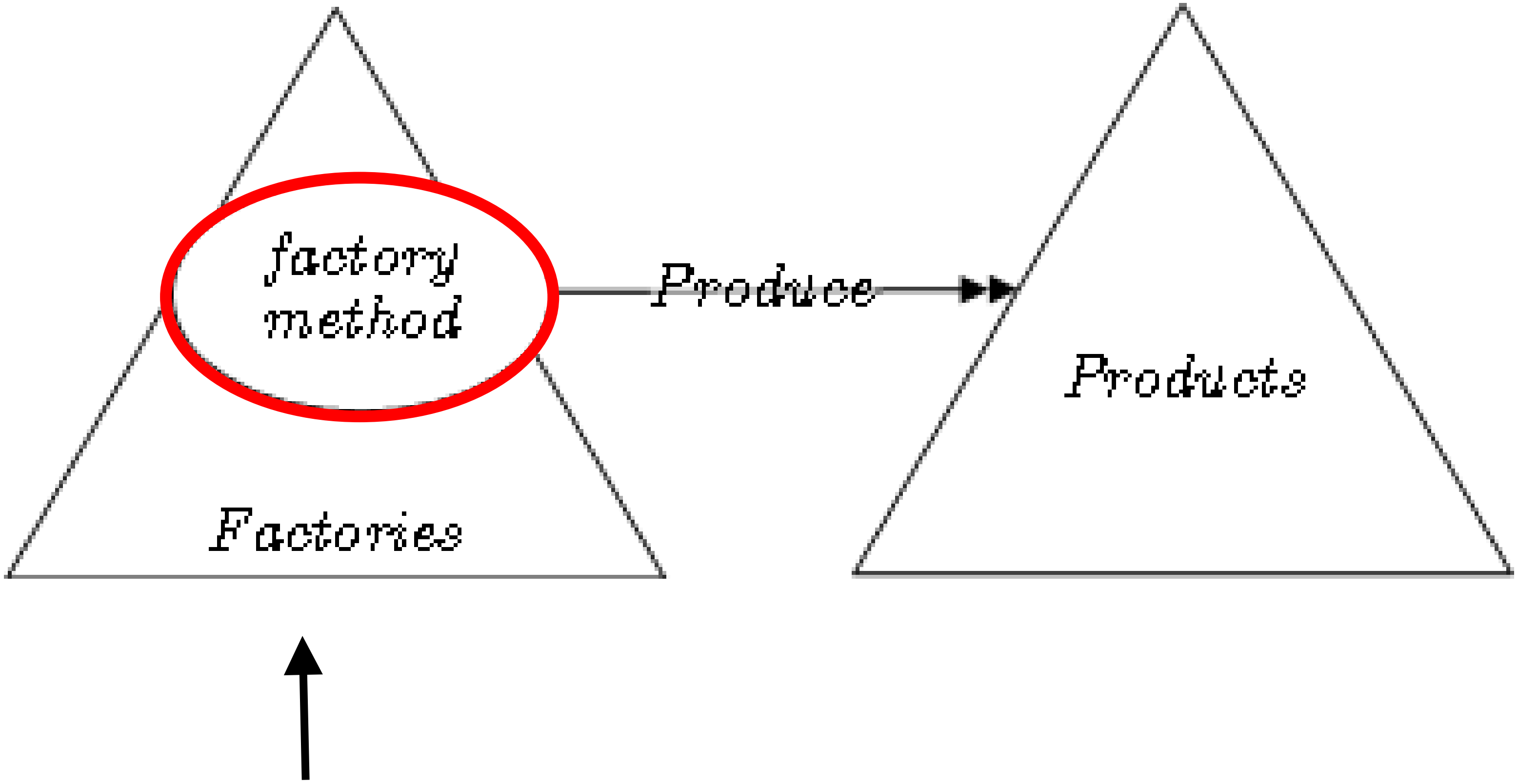
Tying our application to a Concrete class.

Factories:
*the industrialization of **object creation***



Rather than your program creating a new instance of a specific product, request the factory to give you an instance of the product you want. This allows you to encapsulate the knowledge of object creation in the factory!

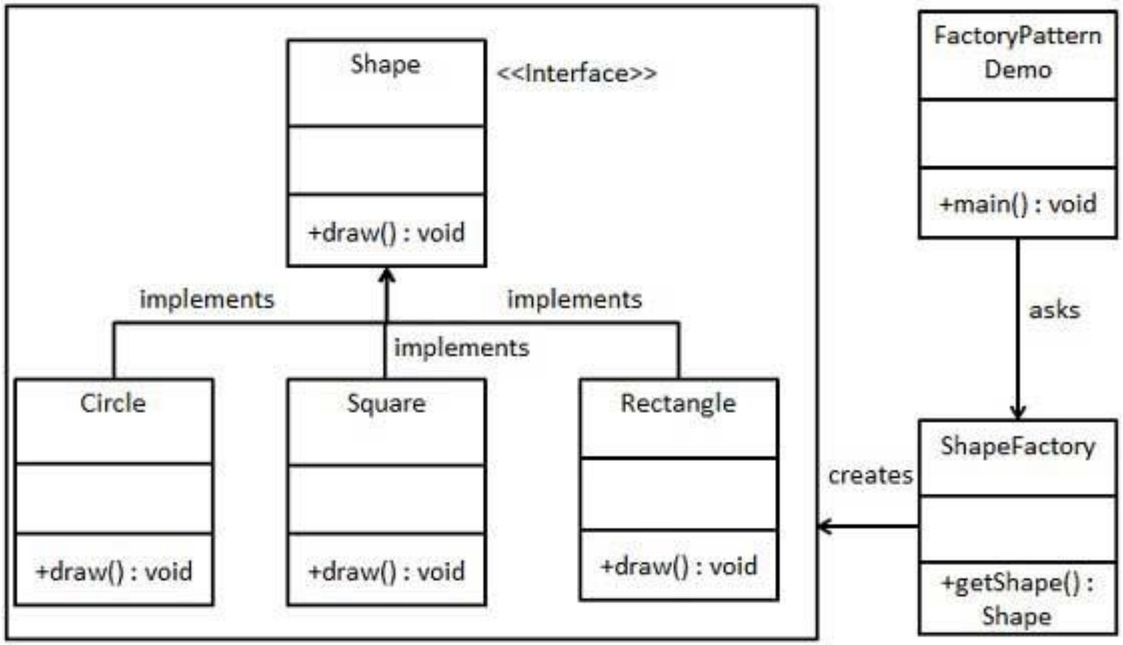
Factories:
*the industrialization of **object creation***



Rather than your program creating a new instance of a specific product, *request the factory to give you an instance of the product you want*. This allows you to encapsulate the knowledge of object creation in the factory!

Factory Method Pattern

Intent: Define an interface for creating an object, and allow subclasses to decide which class to instantiate. ***The factory method allows a class to defer instantiation to subclasses***.



Tutorials Point

Factory Method Pattern:

Elements of Reusable OO Software

- **Motivation** and Applicability: Frameworks use abstract classes to define and m a framework th (i.e. new docume
  - When you ing objects to
  - An applic t create.
  - A class w es.
  - Classes subclasse ich helper su

Instantiating complex objects requires each application to know a lot of information about how to create the object. Should that knowledge be part of every application?

Why not encapsulate that knowledge in one place? **A factory to build objects!**

Factory Method Pattern:

As defined in Elements of Reusable OO Software

- Motivation **and Applicability**: Frameworks use abstract classes to define and maintain relationships between objects. Consider a framework that
new document
  - When you w
object to an
  - **An applica** **t**
create.
  - A class wan
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

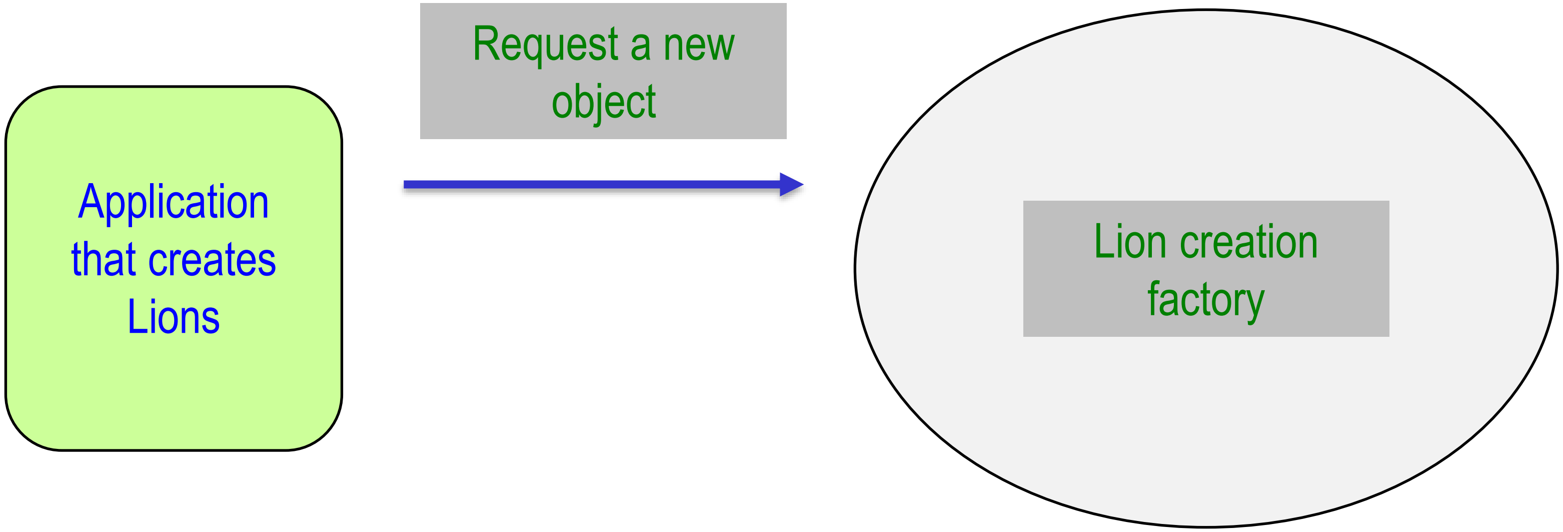
Example: *An application which builds a Zoo.* You may not know in advance what animals you want to create and/or how many animals of each type. It is possible that you may want to balance the animals in the Zoo or create new animals in some random fashion.

Factory Method Pattern:
Elements of Reusable OO Software

- Motivation **and Applicability**: Frameworks use abstract classes to define and maintain relationships between objects. Consider a framework that will need to create different instances of object (i.e. new docume
  - When you
    object to
  - An applic
    create.
  - A class w
  - **Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.**

If creating an instance of a class (e.g. an Animal) requires specialized knowledge, then you can encapsulate that knowledge in another class, rather then require every application to know exactly how to create an instance of that class

Factory Method Pattern

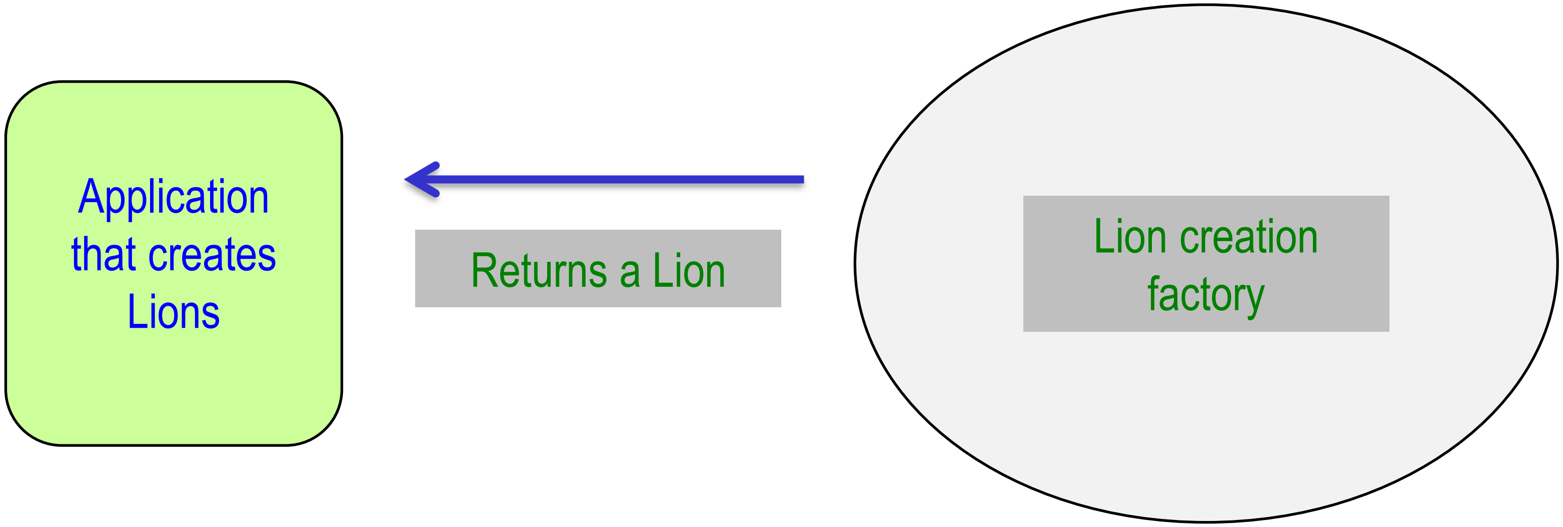


Application that creates Lions

Request a new object

Lion creation factory

Factory Method Pattern

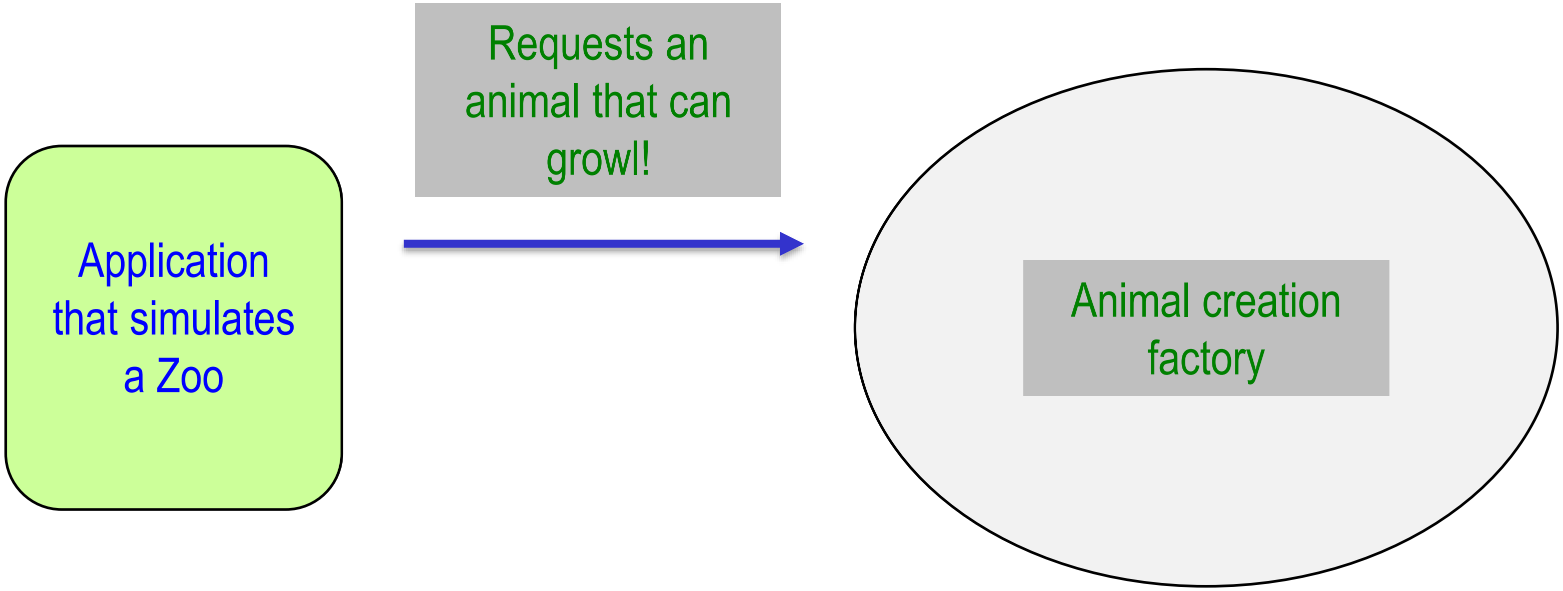


Application that creates Lions

Returns a Lion

Lion creation factory

Factory Method Pattern

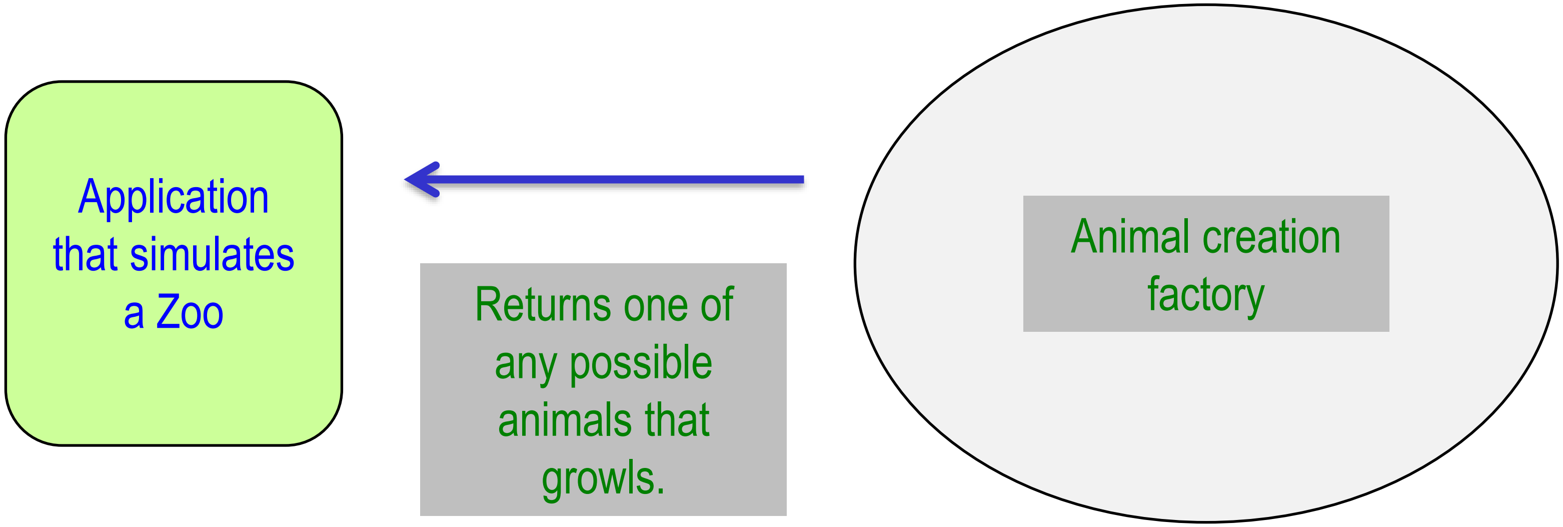


Application that simulates a Zoo

Requests an animal that can growl!

Animal creation factory

Factory Method Pattern

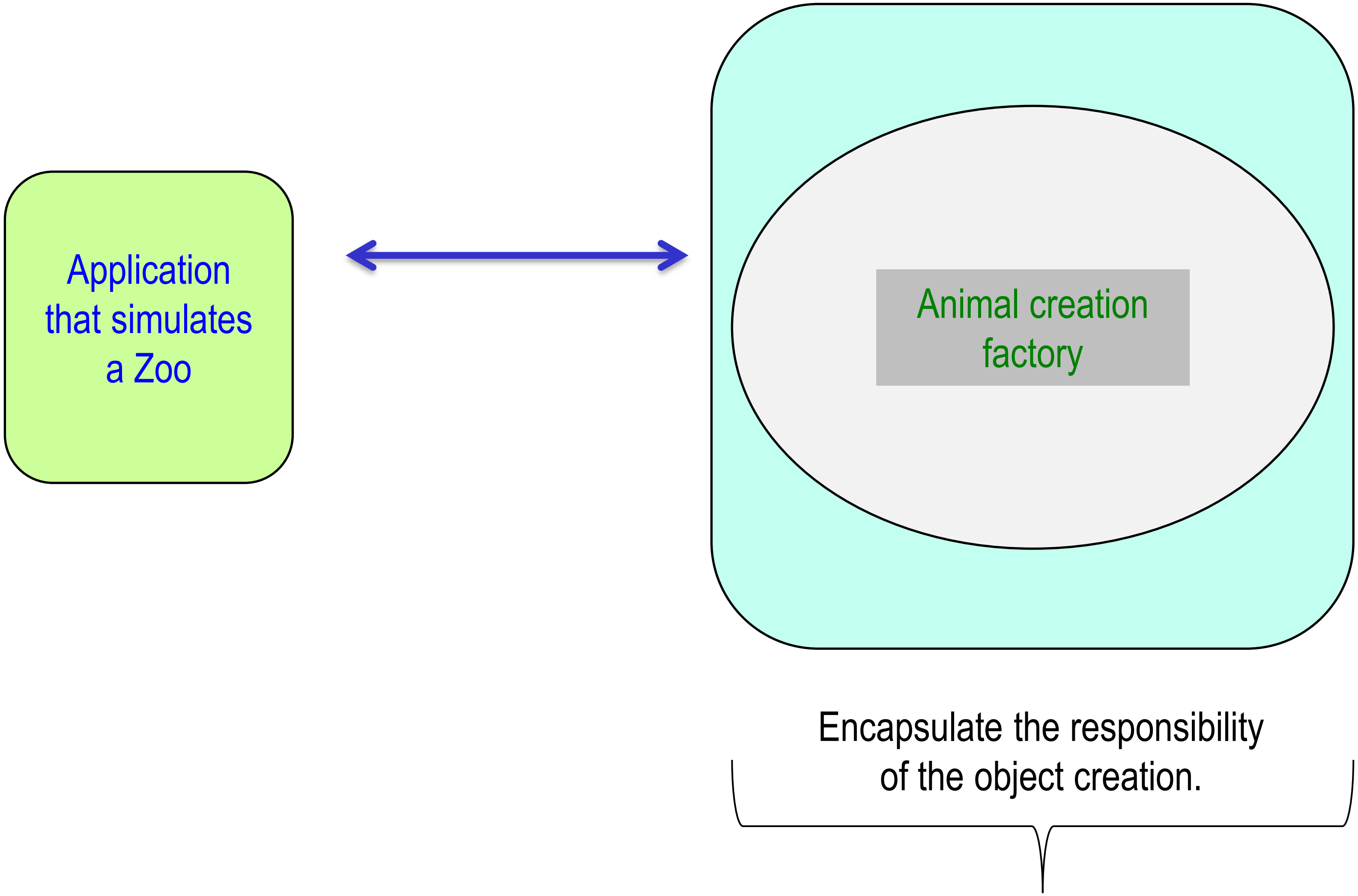


Application that simulates a Zoo

Returns one of any possible animals that growls.

Animal creation factory

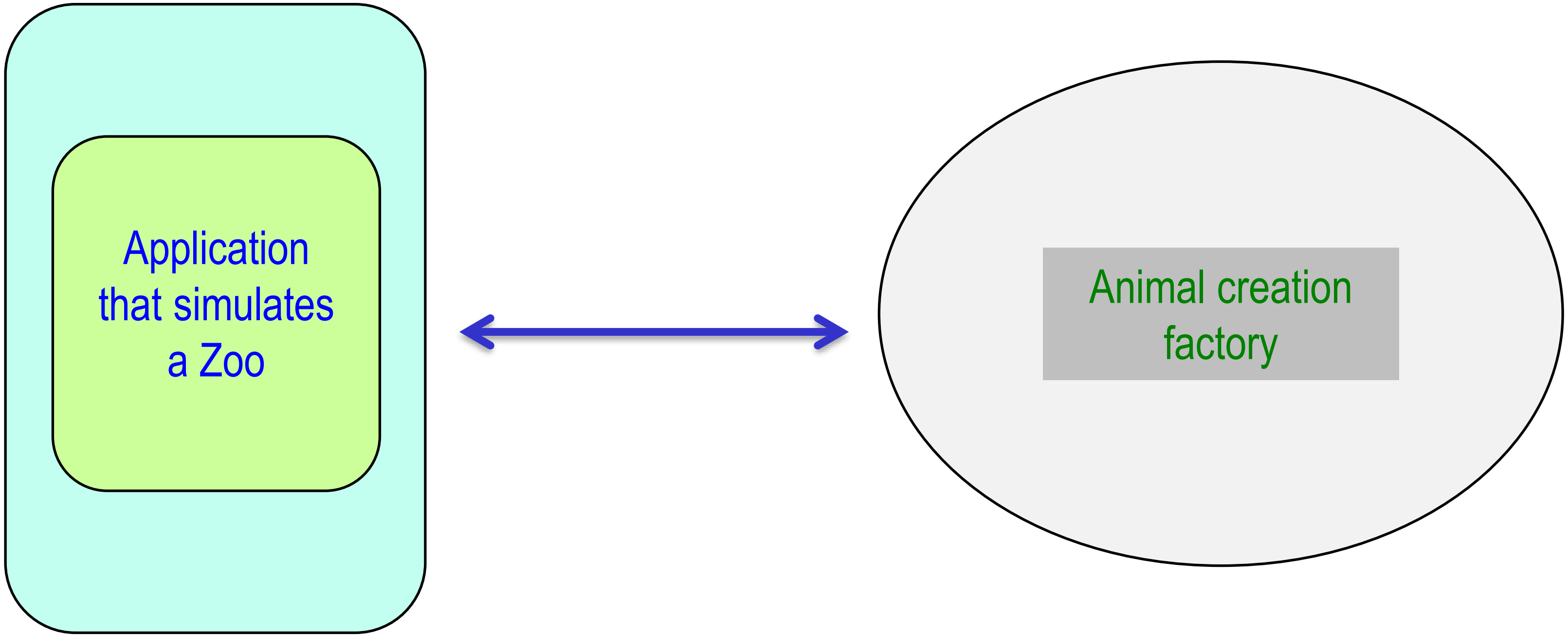
Factory Method Pattern



Encapsulate the responsibility of the object creation.

Factory Method Pattern

Simple Creator



Application that simulates a Zoo

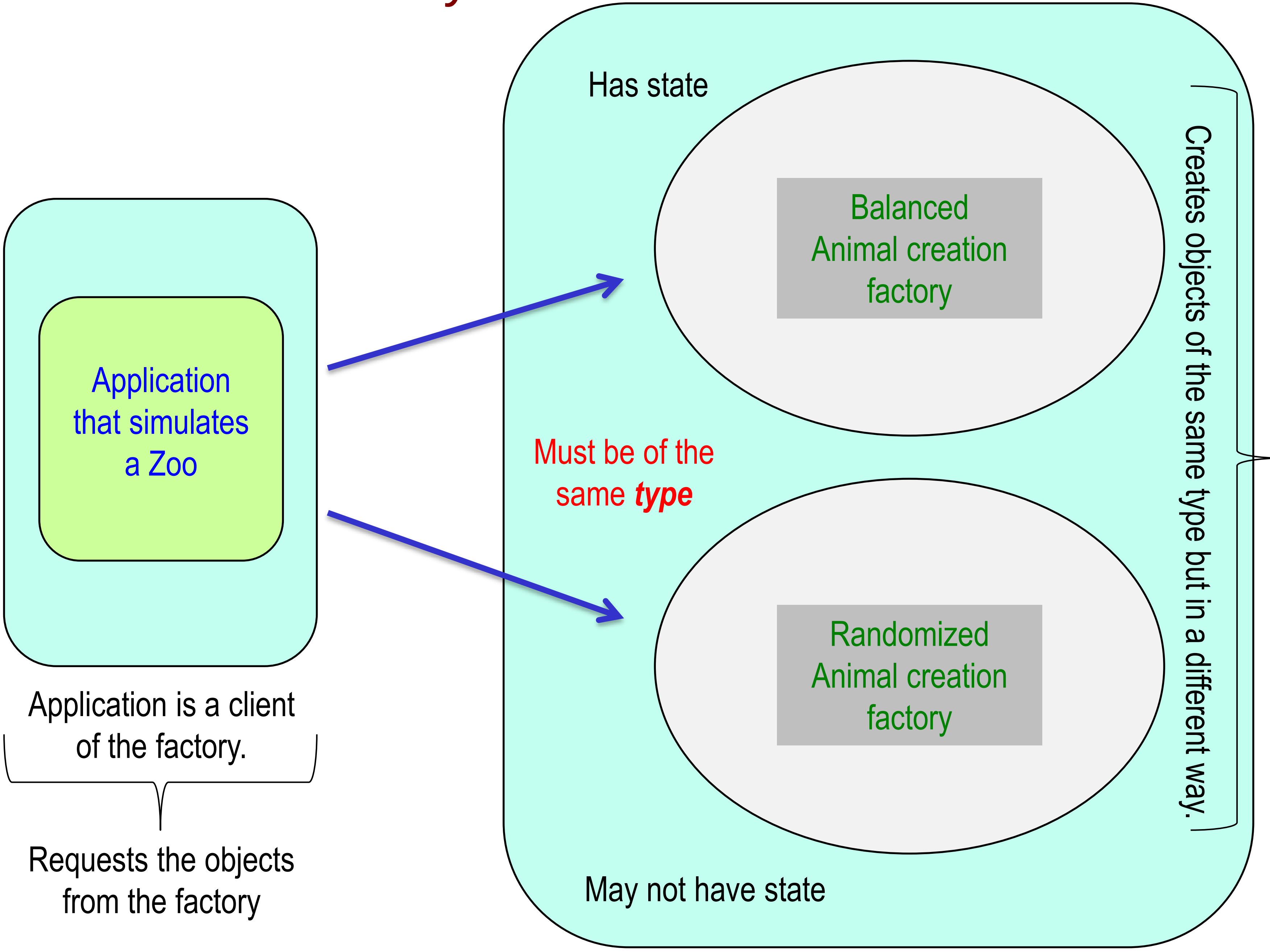
Animal creation factory

Application is a client of the factory.

Requests the objects from the factory

A concrete class which allows applications to create objects of type Animal.

Factory Method Pattern



Application that simulates a Zoo

Application is a client of the factory.

Requests the objects from the factory

Has state

Balanced Animal creation factory

Must be of the same ***type***

Randomized Animal creation factory

May not have state

Creates objects of the same type but in a different way.

Factory Method Pattern:

as stated in Head First Design Patterns

Factory method pattern defines an *interface* for creating an object, but lets concrete classes decide which class to instantiate.

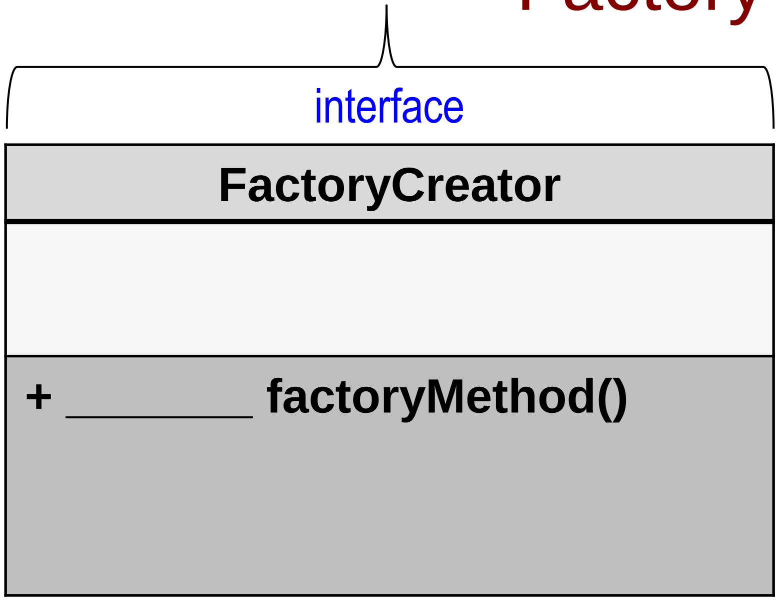
Factory method allows a class to *defer* instantiation to the concrete classes of the Factories.

The rational is that applications may wants to create an object of some type but, the application many not necessarily know:

- **how to create the object.**
- **why it is creating the object.**
- **what parameters to pass when constructing the object.**

unknowns

Factory Method Pattern

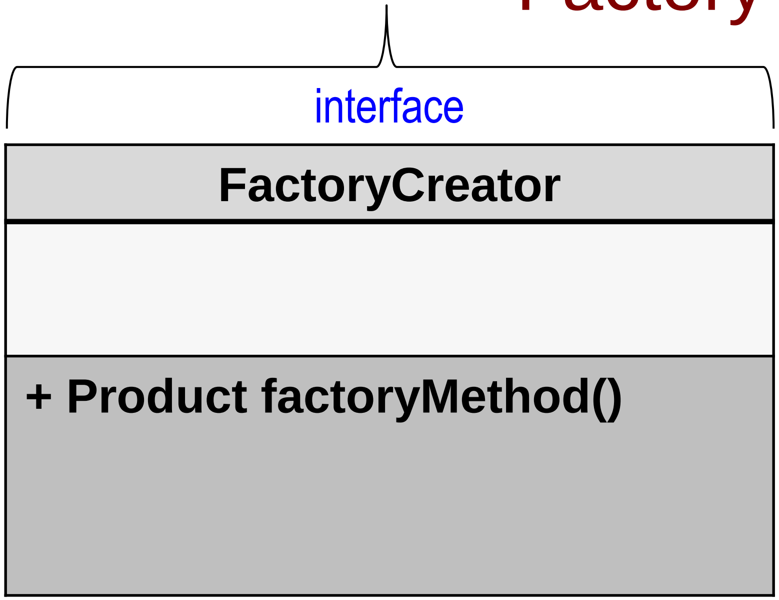


interface

FactoryCreator

+ ________ factoryMethod()

Factory Method Pattern

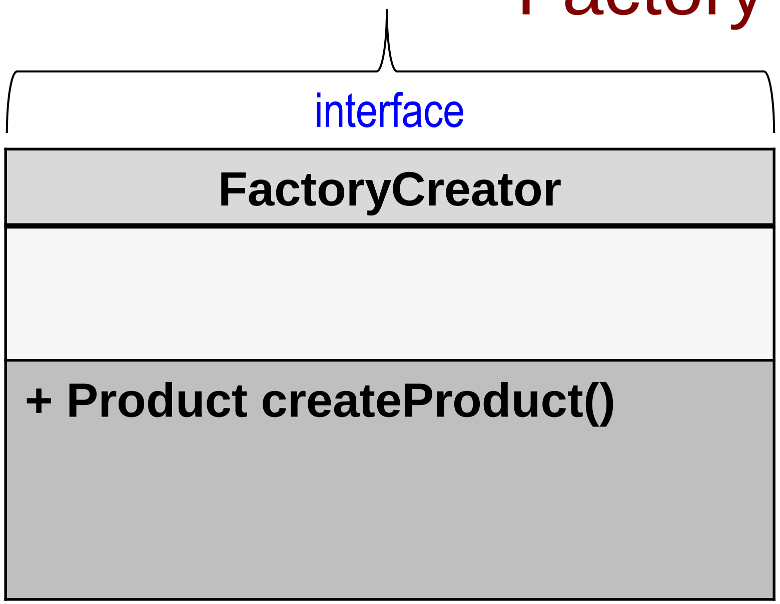


interface

FactoryCreator

+ Product factoryMethod()

Factory Method Pattern

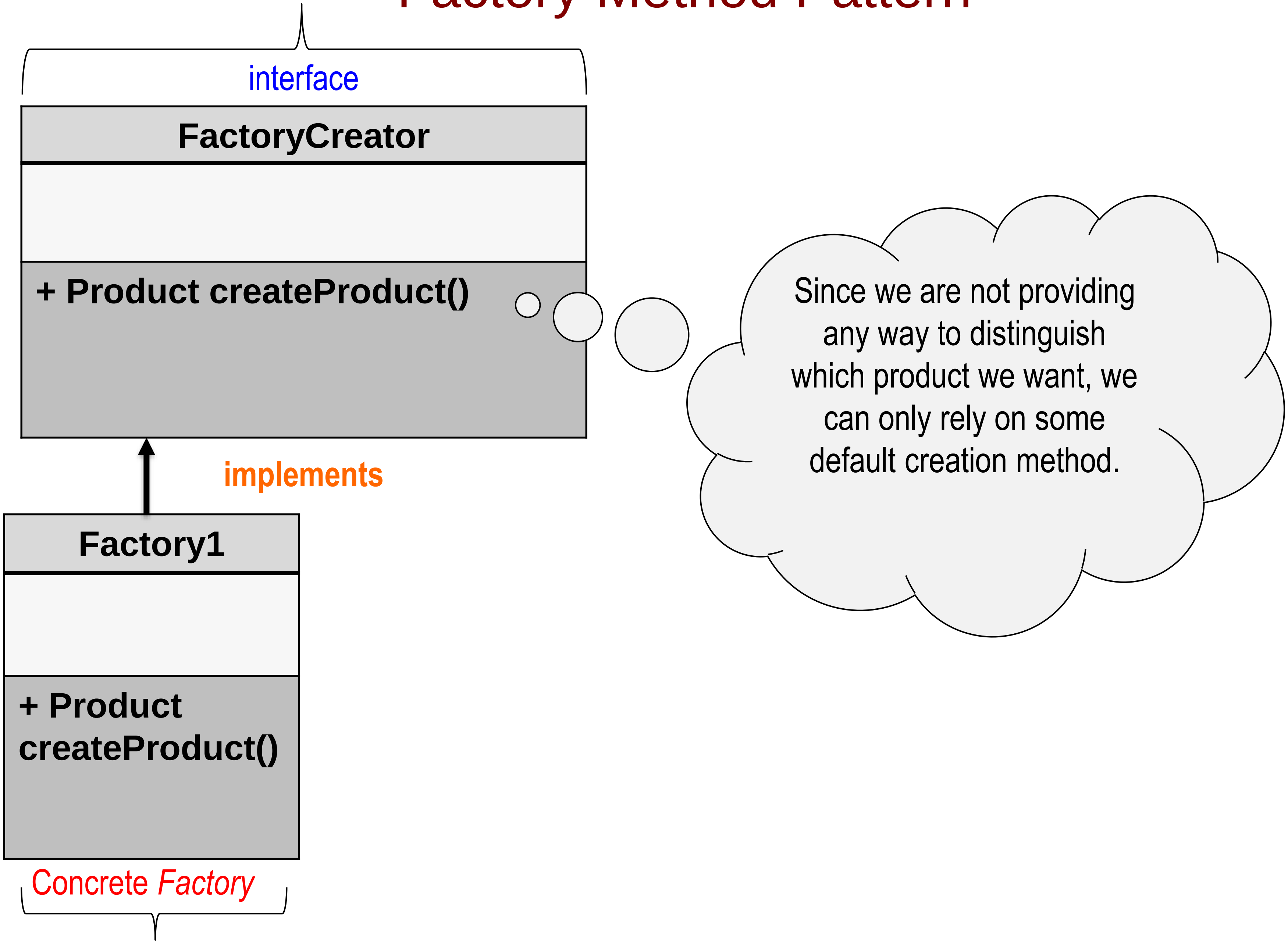


interface

FactoryCreator

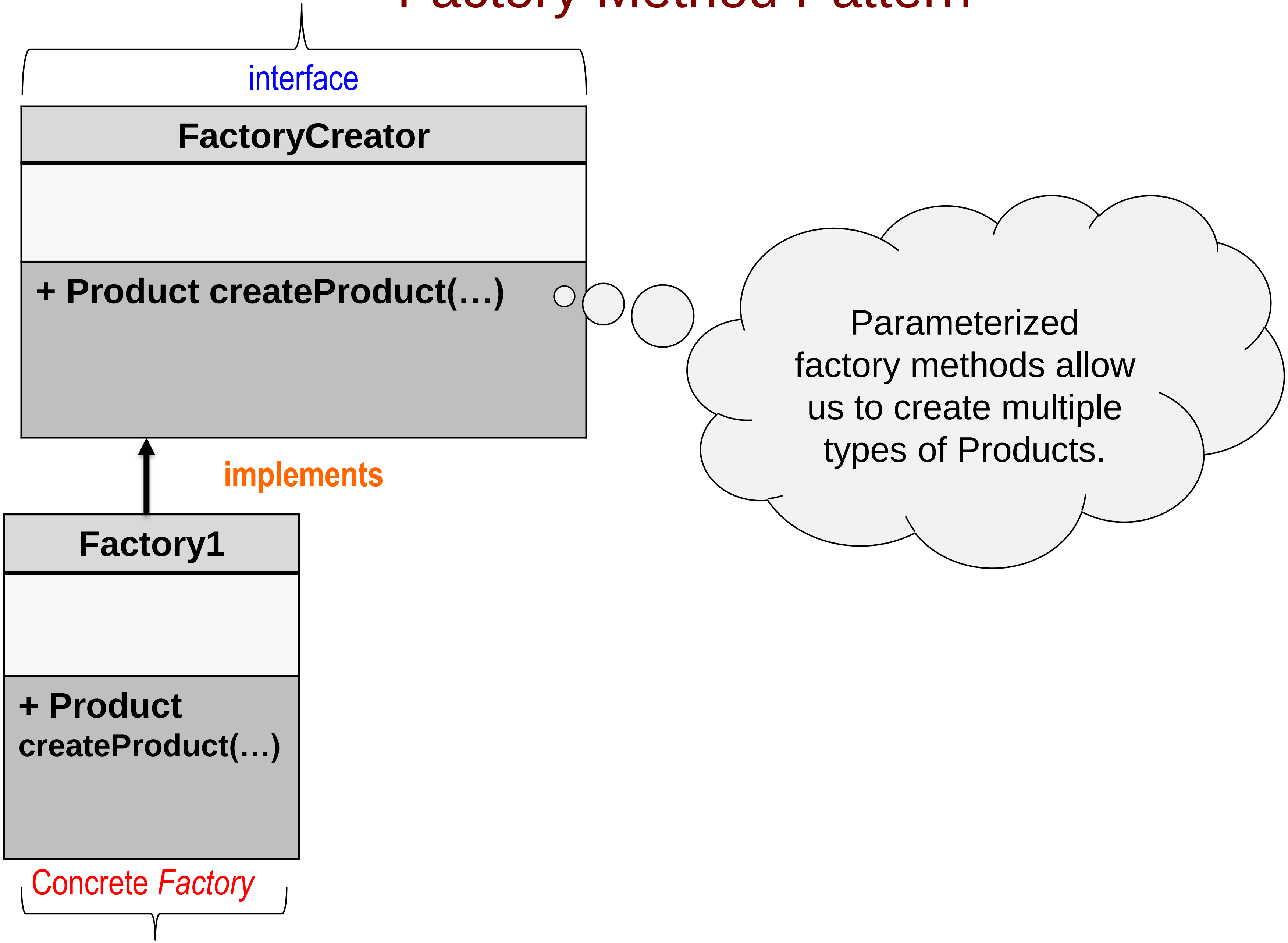
+ Product createProduct()

Factory Method Pattern



Since we are not providing any way to distinguish which product we want, we can only rely on some default creation method.

Factory Method Pattern



interface

FactoryCreator

+ Product createProduct(…)

implements

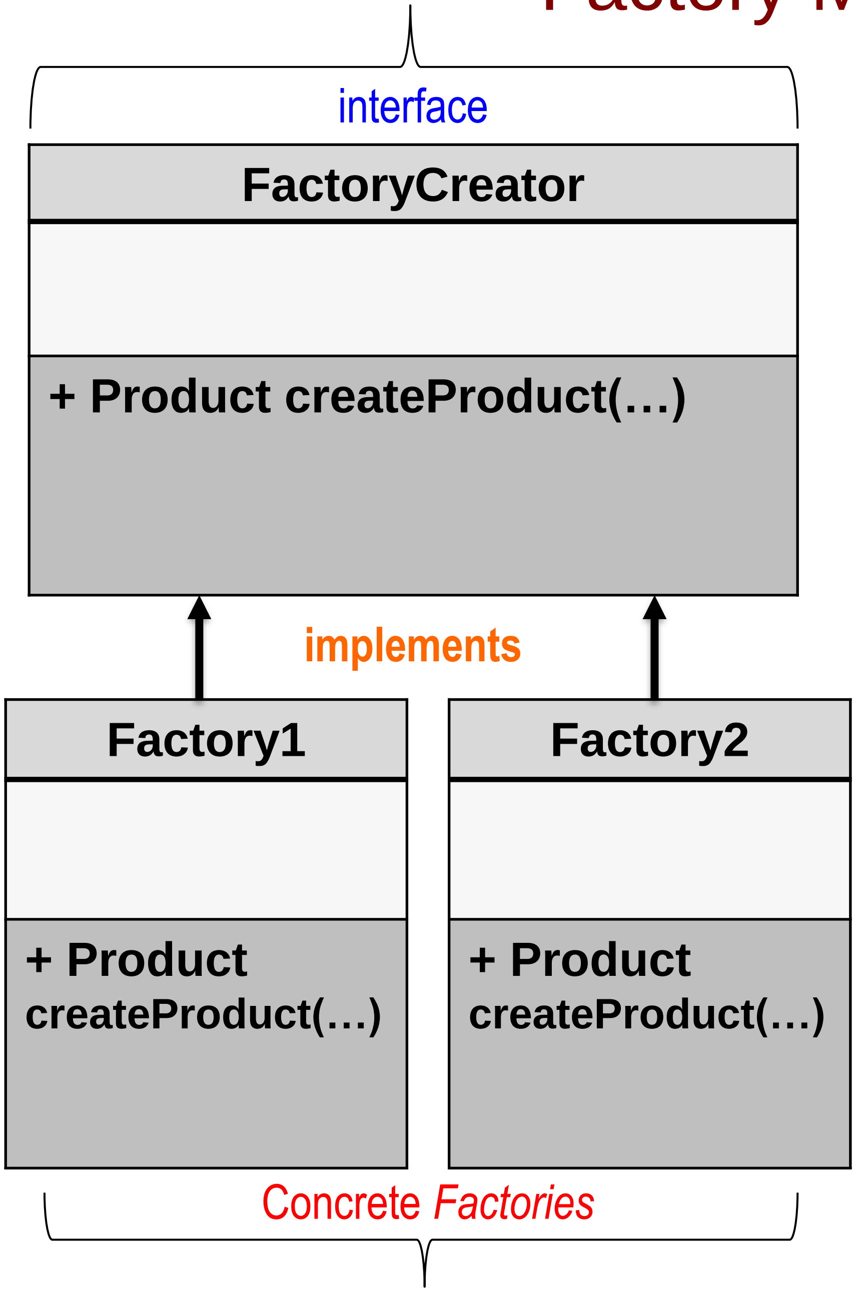
Factory1

+ Product createProduct(…)

Concrete *Factory*

Parameterized factory methods allow us to create multiple types of Products.

Factory Method Pattern



interface

FactoryCreator

+ Product createProduct(…)

implements

Factory1

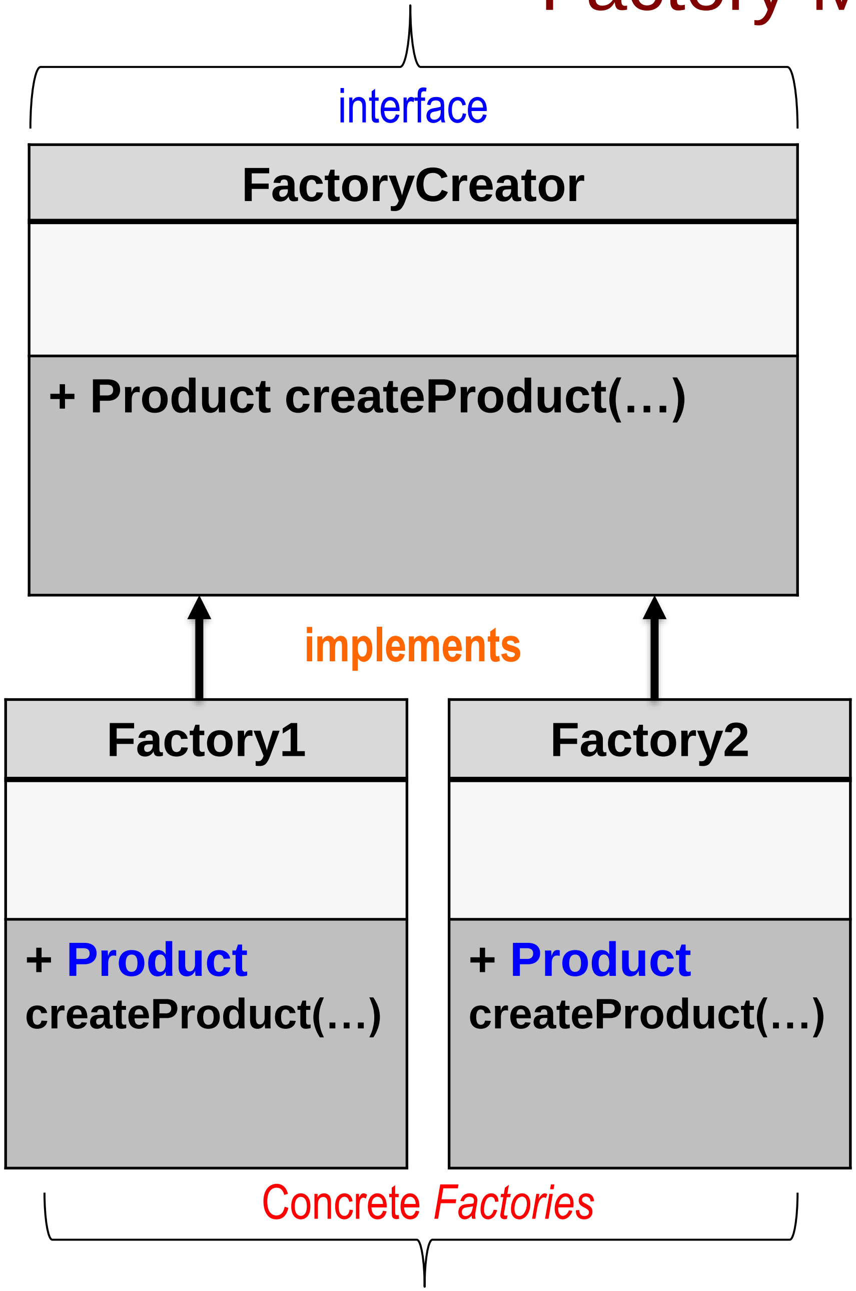
+ Product createProduct(…)

Factory2

+ Product createProduct(…)

Concrete *Factories*

Factory Method Pattern



interface

FactoryCreator

+ Product createProduct(…)

implements

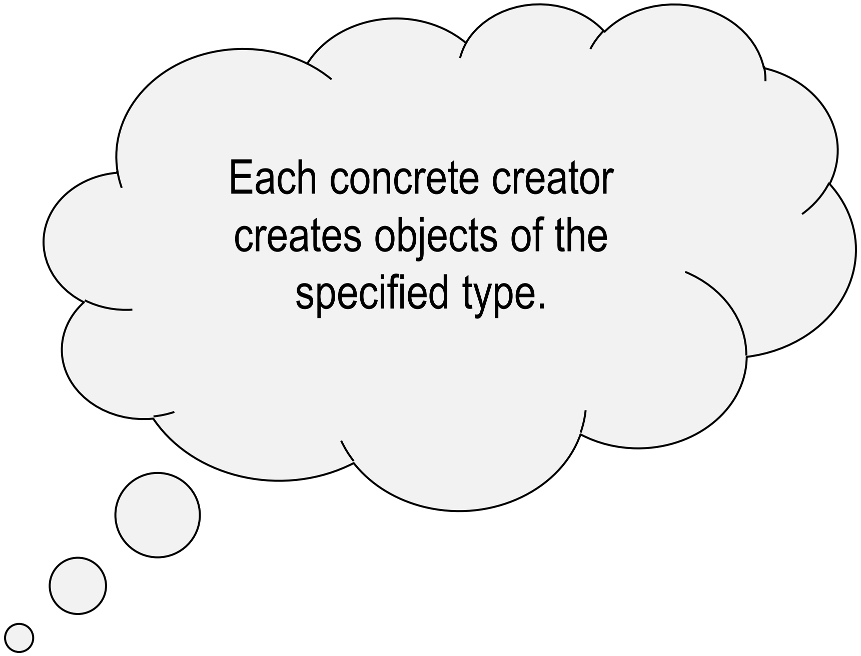
Factory1

+ Product createProduct(…)

Factory2

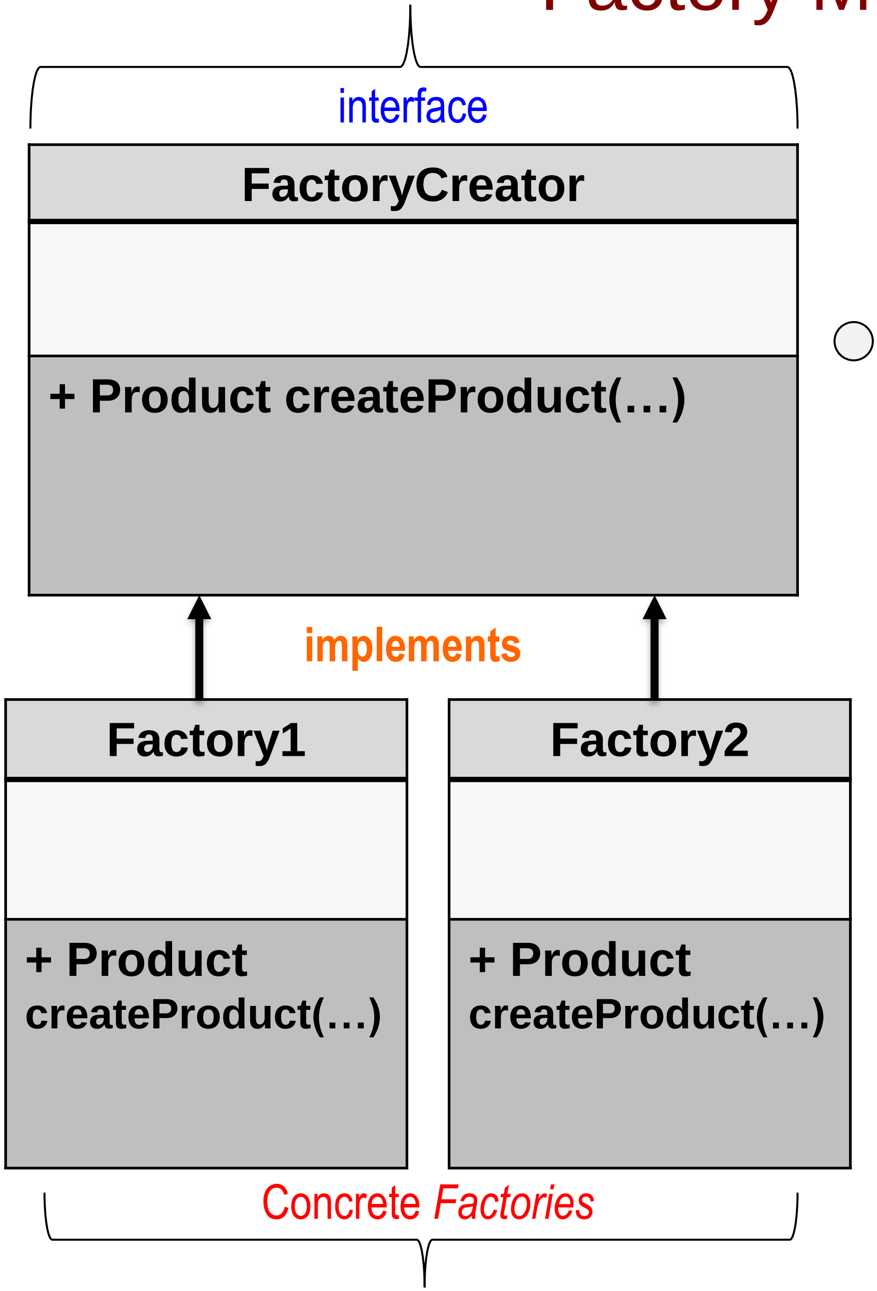
+ Product createProduct(…)

Concrete *Factories*



Each concrete creator creates objects of the specified type.

Factory Method Pattern



interface

FactoryCreator

+ Product createProduct(…)

implements

Factory1

+ Product createProduct(…)

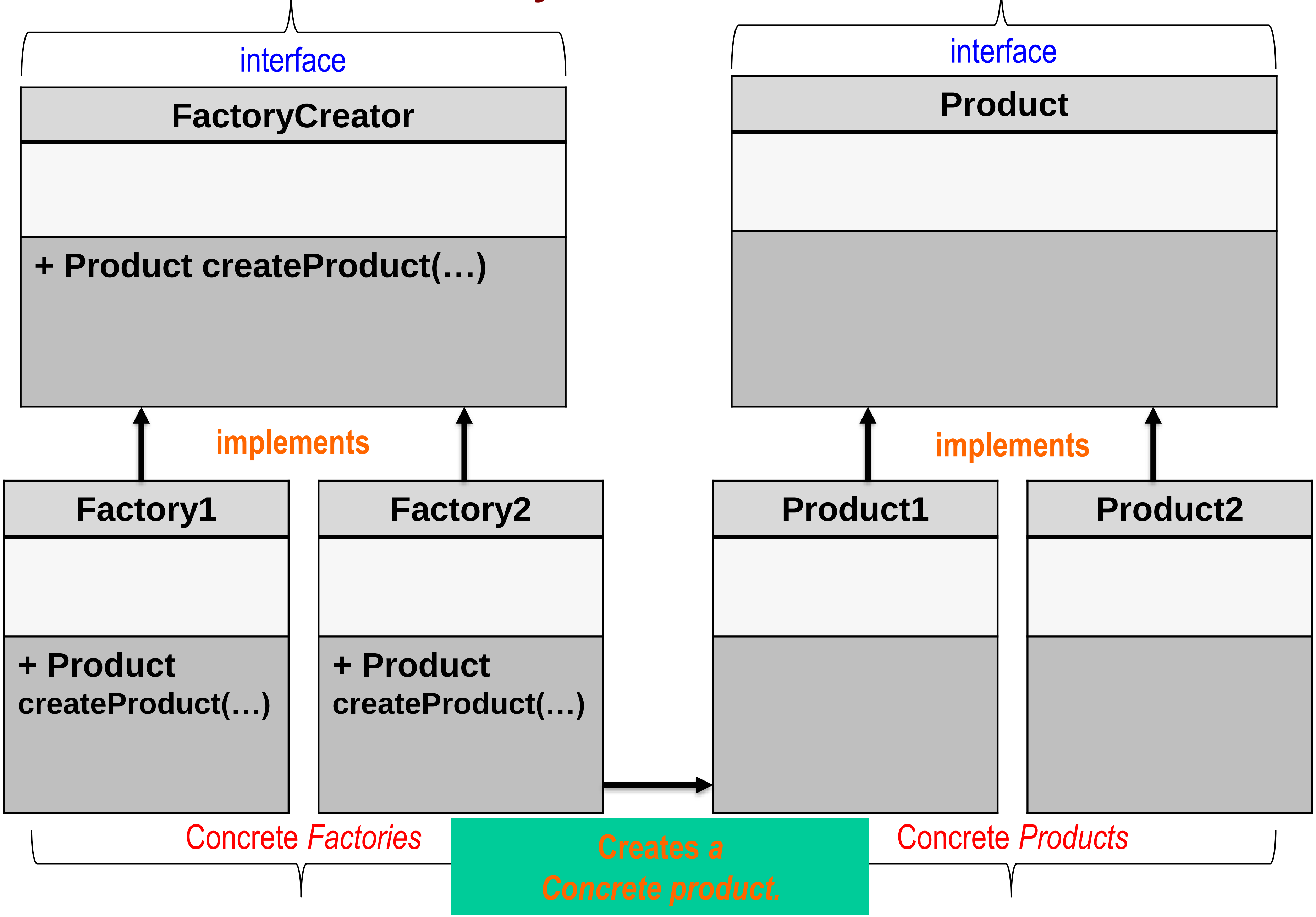
Factory2

+ Product createProduct(…)

Concrete *Factories*

Creating a `Factory Type` allows us to take advantage of the power of Polymorphism and *inject* into our application the specific factory to use!

Factory Method Pattern



interface

FactoryCreator

+ Product createProduct(…)

interface

Product

implements

implements

Factory1

+ Product createProduct(…)

Factory2

+ Product createProduct(…)

Product1

Product2

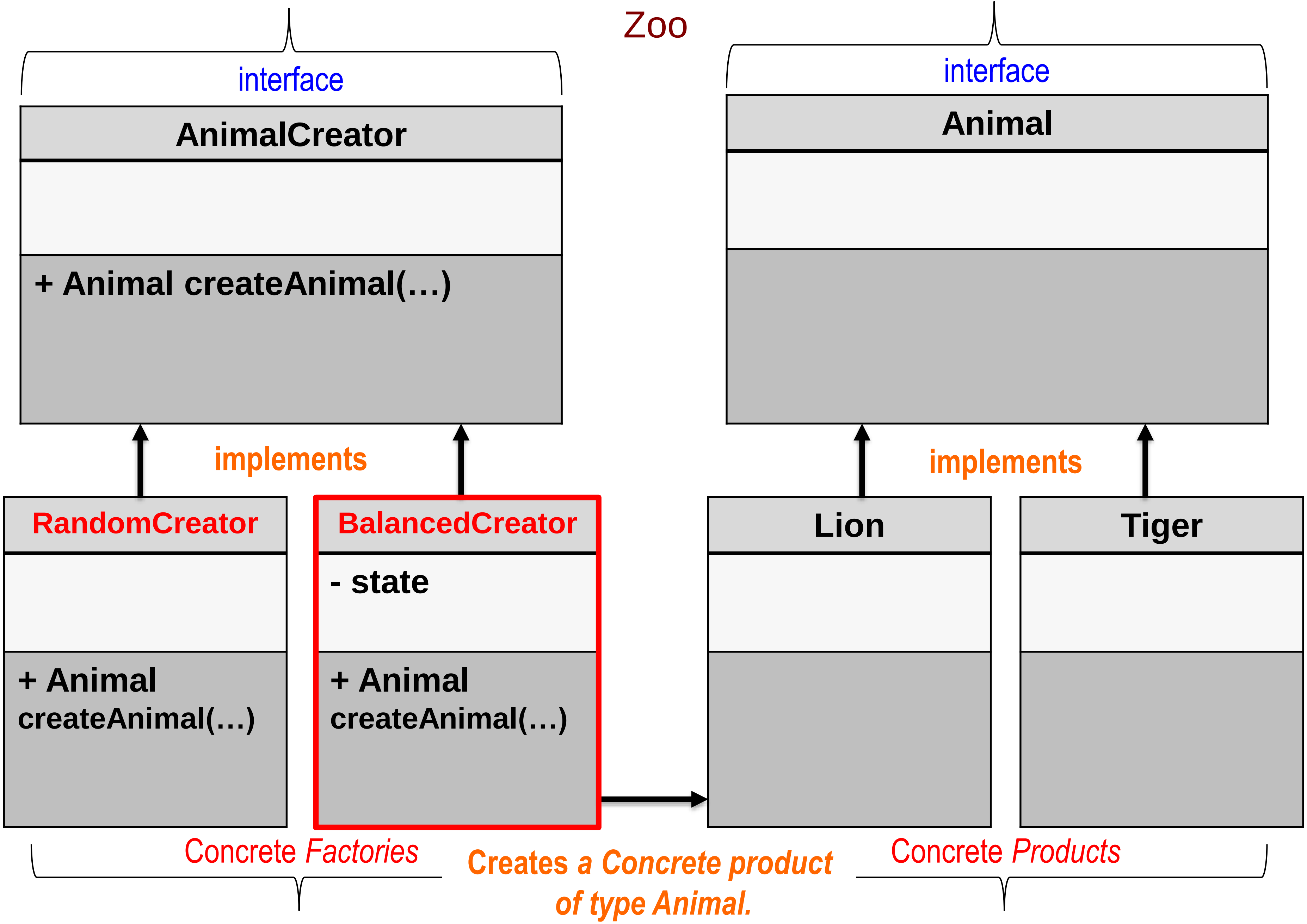
Concrete *Factories*

Creates a *Concrete product.*

Concrete *Products*

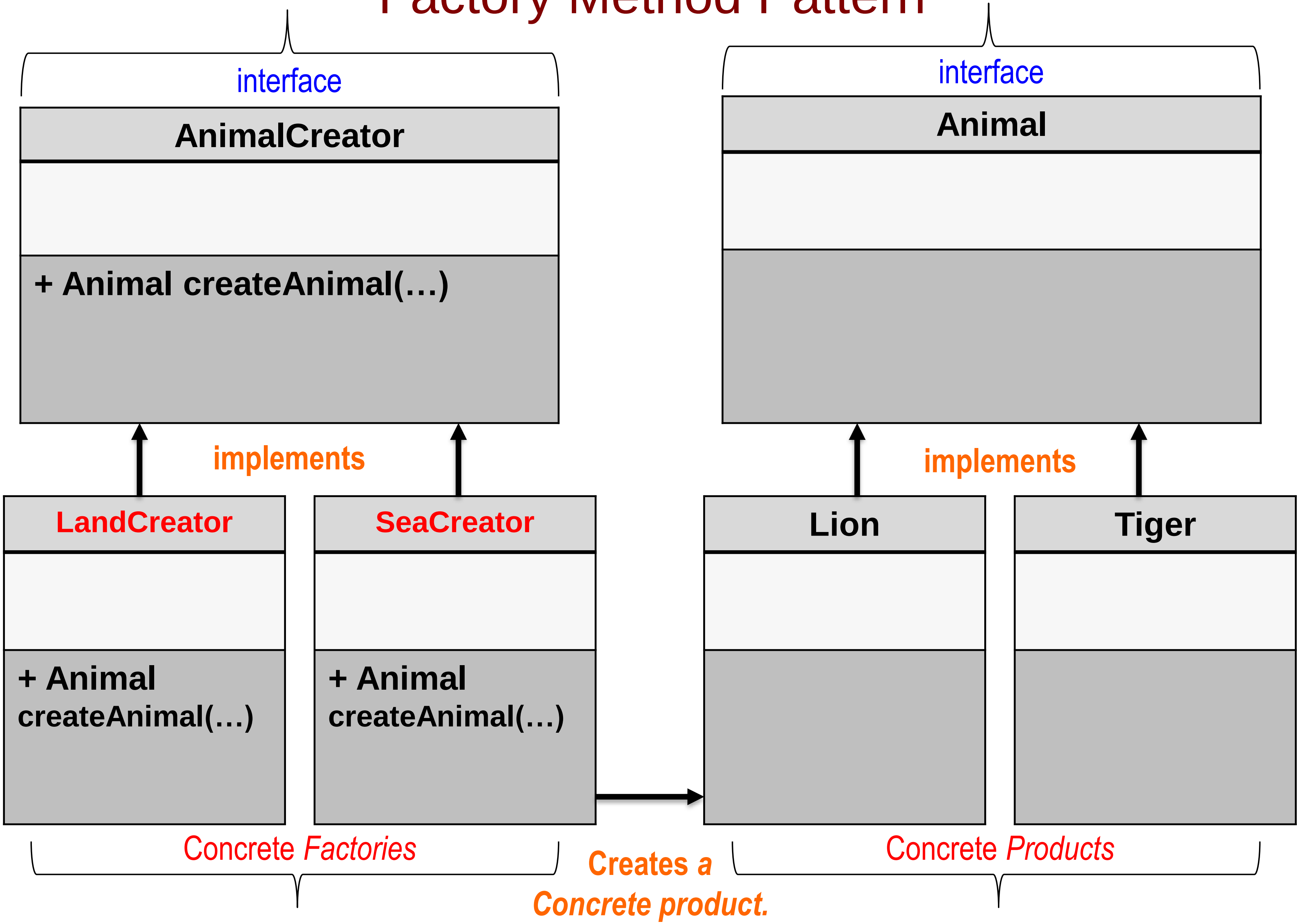
Factory Method Pattern:

Zoo



Creates a *Concrete product of type Animal.*

Factory Method Pattern



Implementation

```
public class SampleFactoryMethod {

  public static void main(String args[]) {

    Zoo zoo_R = createZoo( new RandomCreator() );
    Zoo zoo_B = createZoo( new BalancedCreator() );
  }

  public static Zoo createZoo( Factory factory ) {
    Zoo zoo = new Zoo();      // create a new Zoo
    zoo.add( factory.createAnimal("growls") );
    zoo.add( factory.createAnimal("barks") );
    zoo.add( factory.createAnimal("growls") );
    return zoo;
  }
} // class
```

Implementation

```
public class SampleFactoryMethod {

  public static void main(String args[]) {

    Zoo zoo_R = createZoo( new RandomCreator() );
    Zoo zoo_B = createZoo( new BalancedCreator() );
  }

  public static Zoo createZoo( Factory factory ) {
    Zoo zoo = new Zoo();      // create a new Zoo
    zoo.add( factory.createAnimal("growls") );
    zoo.add( factory.createAnimal("barks") );
    zoo.add( factory.createAnimal("growls") );
    return zoo;
  }
} // class
```

Implementation

```
public class SampleFactoryMethod {

  public static void main(String args[]) {

    Zoo zoo_R = createZoo( new RandomCreator() );
    Zoo zoo_B = createZoo( new BalancedCreator() );
  }

  public static Zoo createZoo( Factory factory ) {
    Zoo zoo = new Zoo(factory); // create a new Zoo
    zoo.add( "growls" );
    zoo.add( "fur" );
    zoo.add( "growls" );
    return zoo;
  }
} // class
```

Implementation

```
public class SampleFactoryMethod {

  public static void main(String args[]) {

    Zoo zoo_R = createZoo( new RandomCreator() );
    Zoo zoo_B = createZoo( new BalancedCreator() );
  }

  public static Zoo createZoo( Factory factory ) {
    Zoo zoo = new Zoo(factory); // create a new Zoo
    zoo.add( "growls" );
    zoo.add( "fur" );
    zoo.add( "growls" );
    return zoo;
  }
} // class
```

Dependency Injection
(Constructor)

Factory Method Pattern:
Elements of Reusable OO Software

- **Consequences (Advantages/Disadvantages)**: Factory methods eliminate the need to bind concrete classes into your application.
  - Facilitates programming to a type and not an implementation.
  - Allows ap er class and instantiati
  - An applic ory and know

Allows applications to delegate away the responsibility of object creation. And it also allows applications to be concerned with the type of object being created and not the class it is being created from.

Factory Method Pattern:

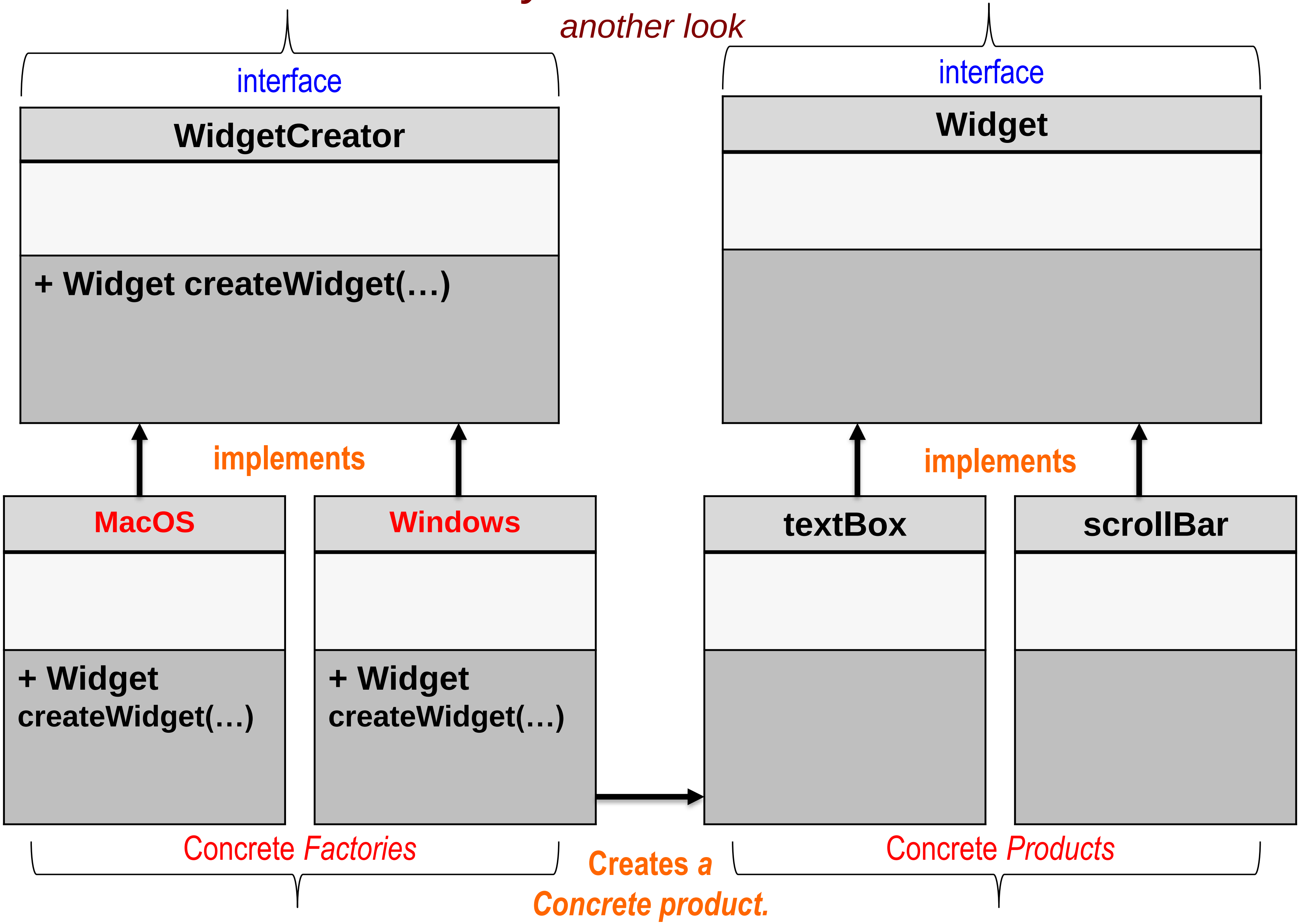
Elements of Reusable OO Software

- **Consequences (Advantages/Disadvantages)**: Factory methods eliminate the need to bind concrete classes into your application.
  - Facilitates programming to a type and not an implementation.
  - Allows ap r
    class and
    instantiati
  - An applic ory
    and know

But now the application needs to know about the factory! Thus far these Factories have not been unionized!

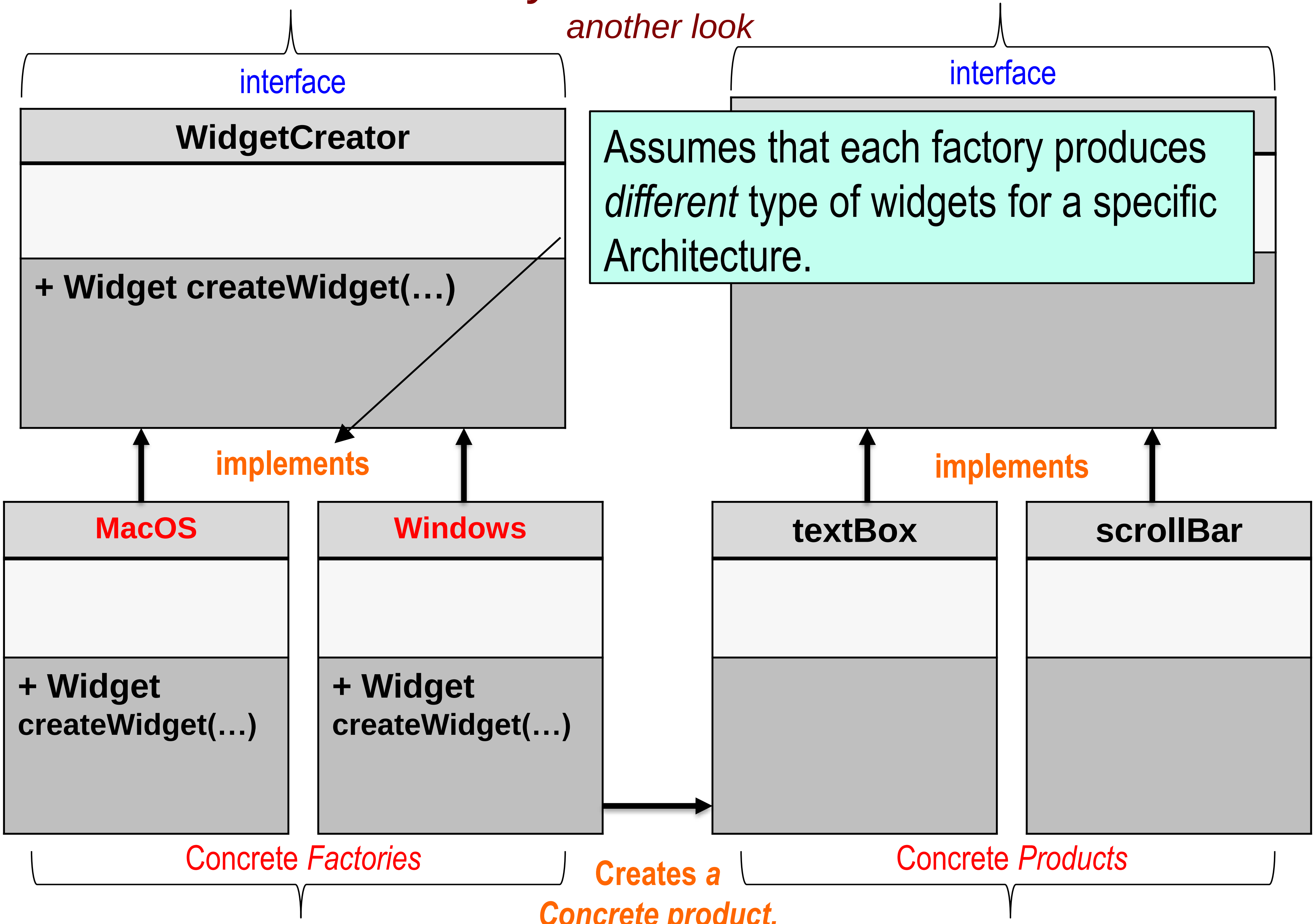
Factory Method Pattern:

another look



Factory Method Pattern:

another look



interface

WidgetCreator

+ Widget createWidget(…)

interface

Assumes that each factory produces *different* type of widgets for a specific Architecture.

implements

MacOS

+ Widget createWidget(…)

Windows

+ Widget createWidget(…)

implements

textBox

scrollBar

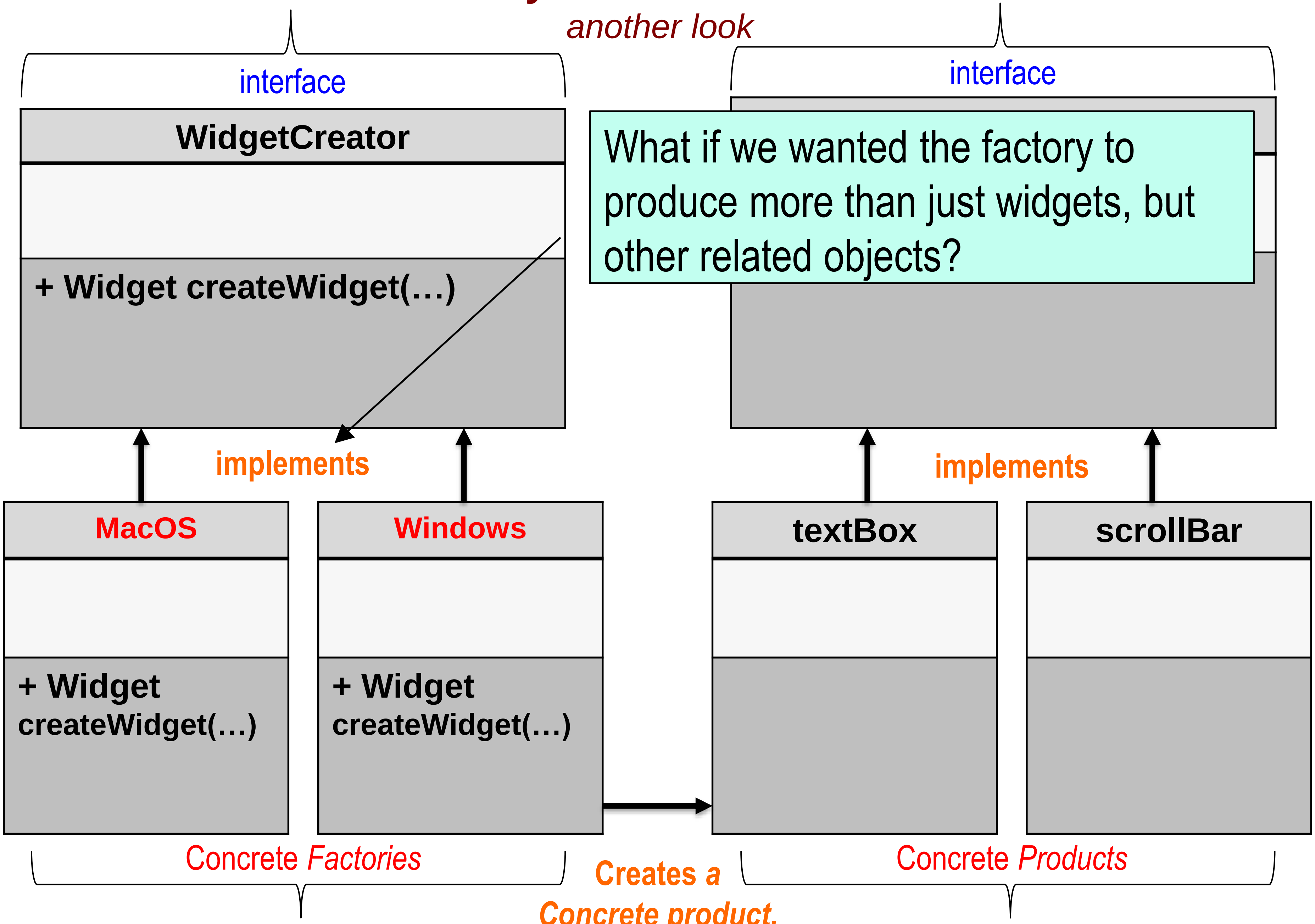
Concrete *Factories*

Creates a Concrete product.

Concrete *Products*

Factory Method Pattern:

another look



interface

WidgetCreator

+ Widget createWidget(…)

interface

implements

MacOS

+ Widget createWidget(…)

Windows

+ Widget createWidget(…)

implements

textBox

scrollBar

Concrete *Factories*

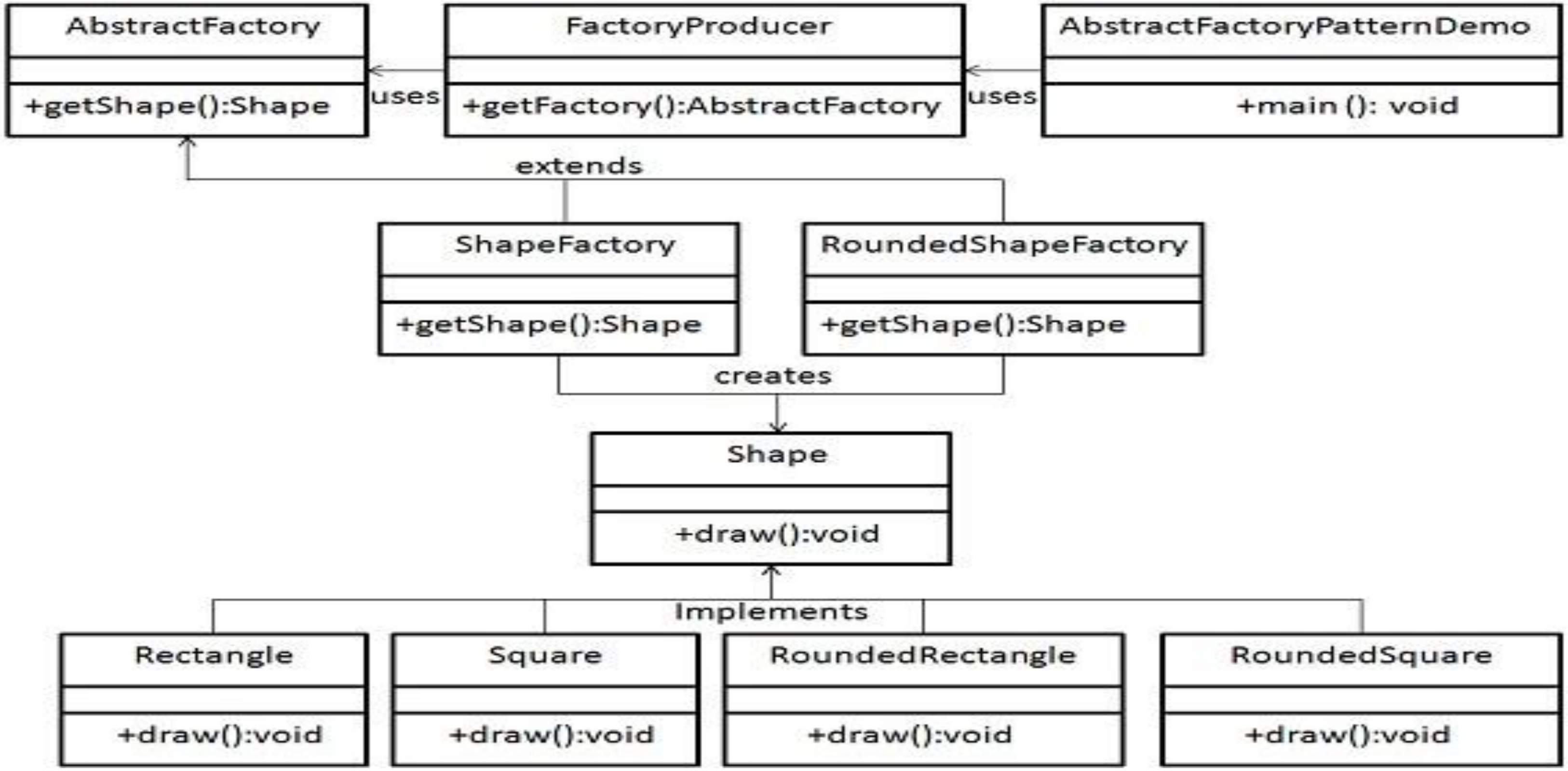
Creates a *Concrete product.*

Concrete *Products*

What if we wanted the factory to produce more than just widgets, but other related objects?

Abstract Factory Pattern

Intent: Provide an interface for creating *families* of *related* or *dependent* objects without specifying their concrete classes.



Tutorials Point

Abstract Factory Pattern

Factory Method pattern is concerned with constructing a single object type.

vs.

Abstract Factory pattern is responsible *for constructing multiple types of objects such that some relationship or dependency exists amongst them*.

Abstract Factory Pattern

Factory Method pattern is concerned with constructing a single object type.

Powerful pattern because it allows us to create Factories that implement multiple `factoryMethods`

vs.

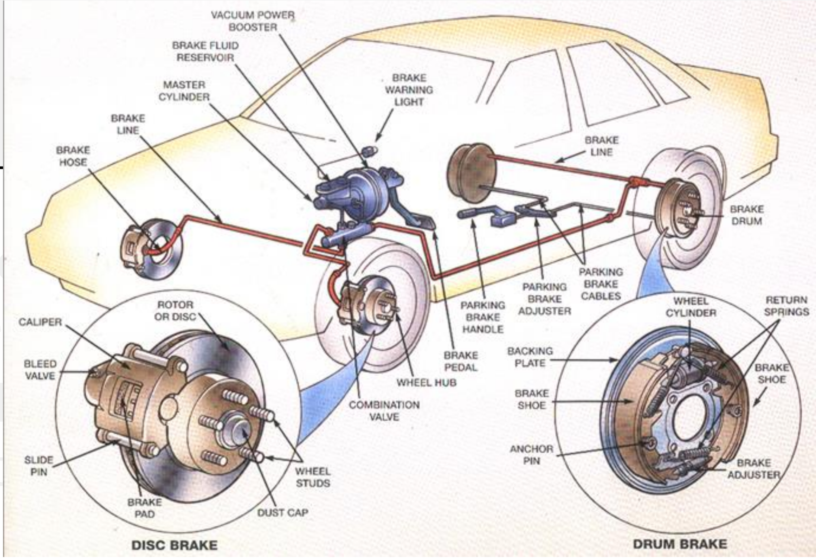
Abstract Factory pattern is responsible for constructing multiple types of objects such that some relationship or dependency exists amongst them.

Abstract Factory Pattern:

As defined in Elements of Reusable OO Software

- **Motivation** and Applicability: Consider a User Interface that needs to support multiple platforms. It is not enough to be able to create widgets, but you may need to create those widgets for different GUI platforms.
  - When you
    object to a
  - A system s
    composed
  - **A system**
    products.
  - **A family of related product**
    together, and you need to e
  - You want to provide a class li   reveal
    only their interfaces, and not t

This pattern is used when you need a factory to create a family of *related* products! Example:



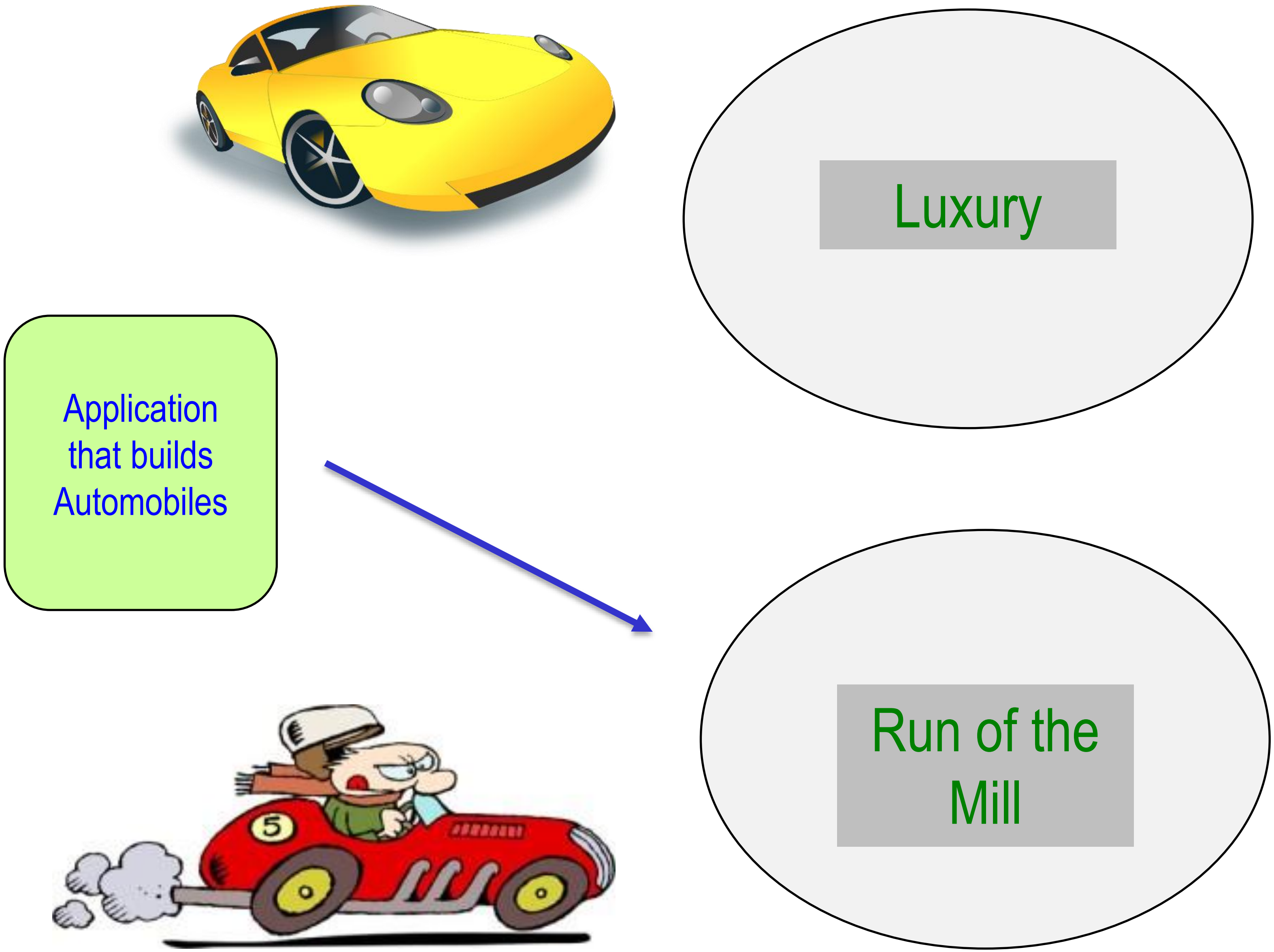
Abstract Factory Pattern

Application that builds Automobiles

Luxury

Run of the Mill

Abstract Factory Pattern

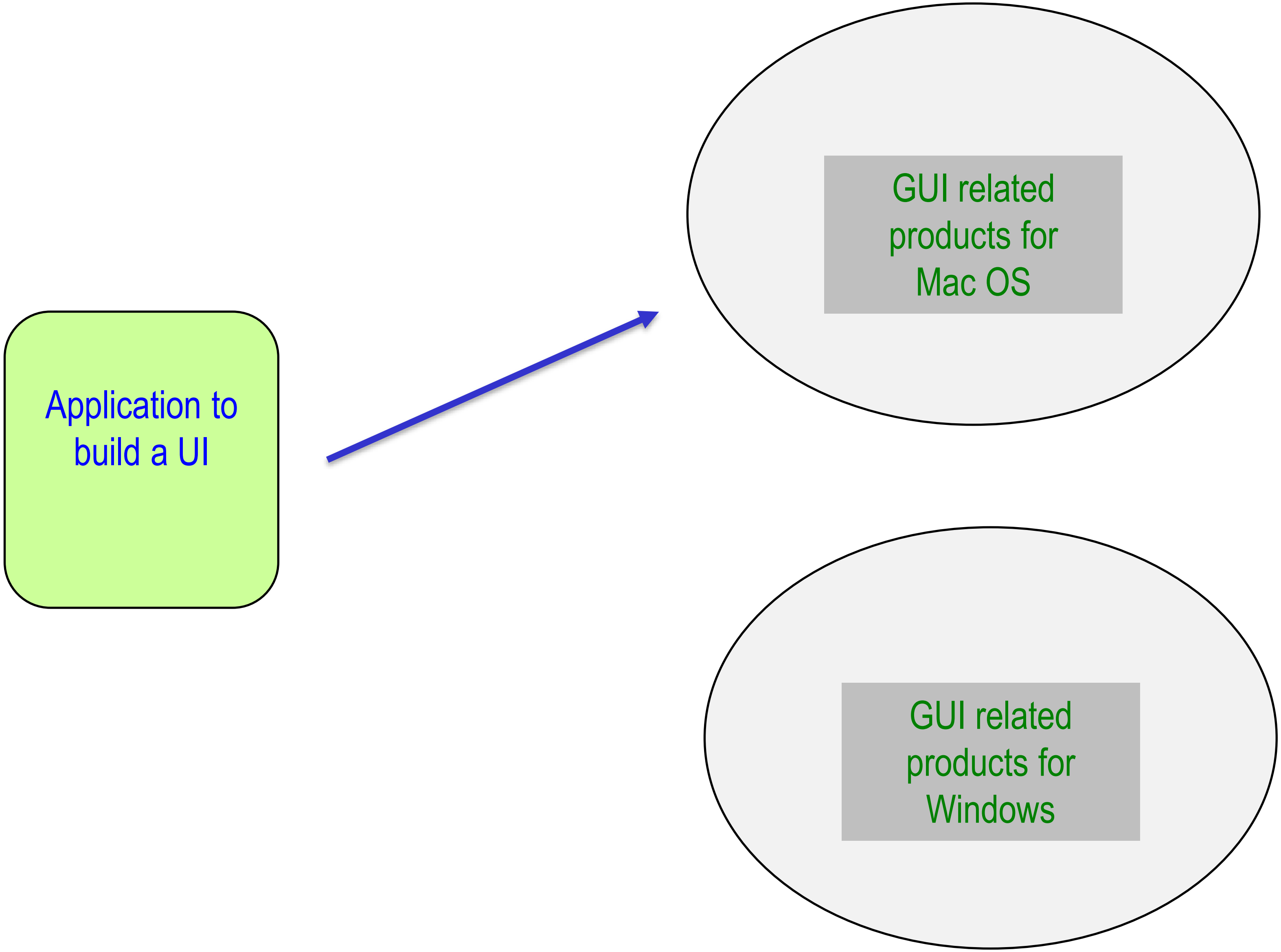


Application that builds Automobiles

Luxury

Run of the Mill

Abstract Factory Pattern

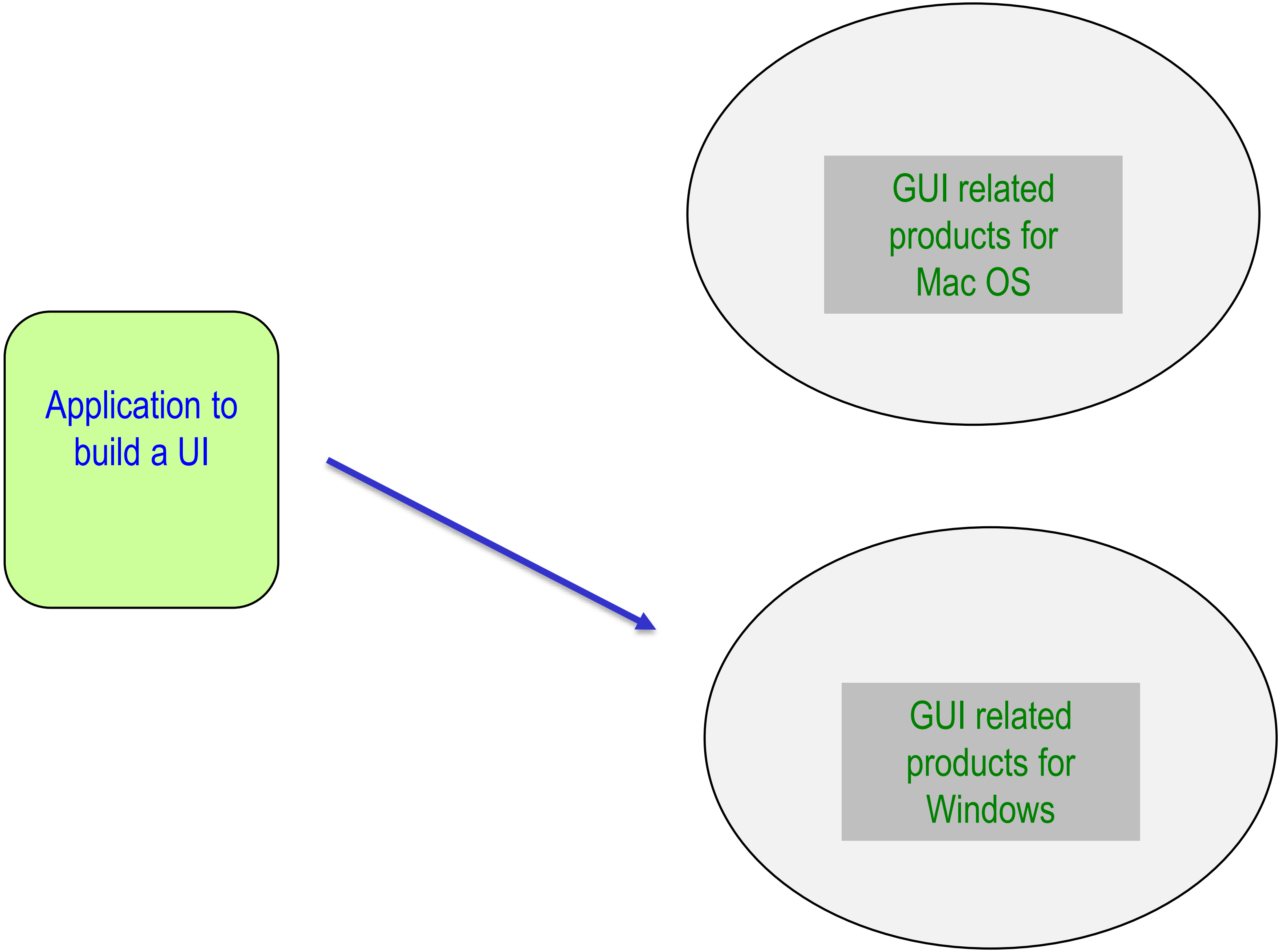


Application to build a UI

GUI related products for Mac OS

GUI related products for Windows

Abstract Factory Pattern

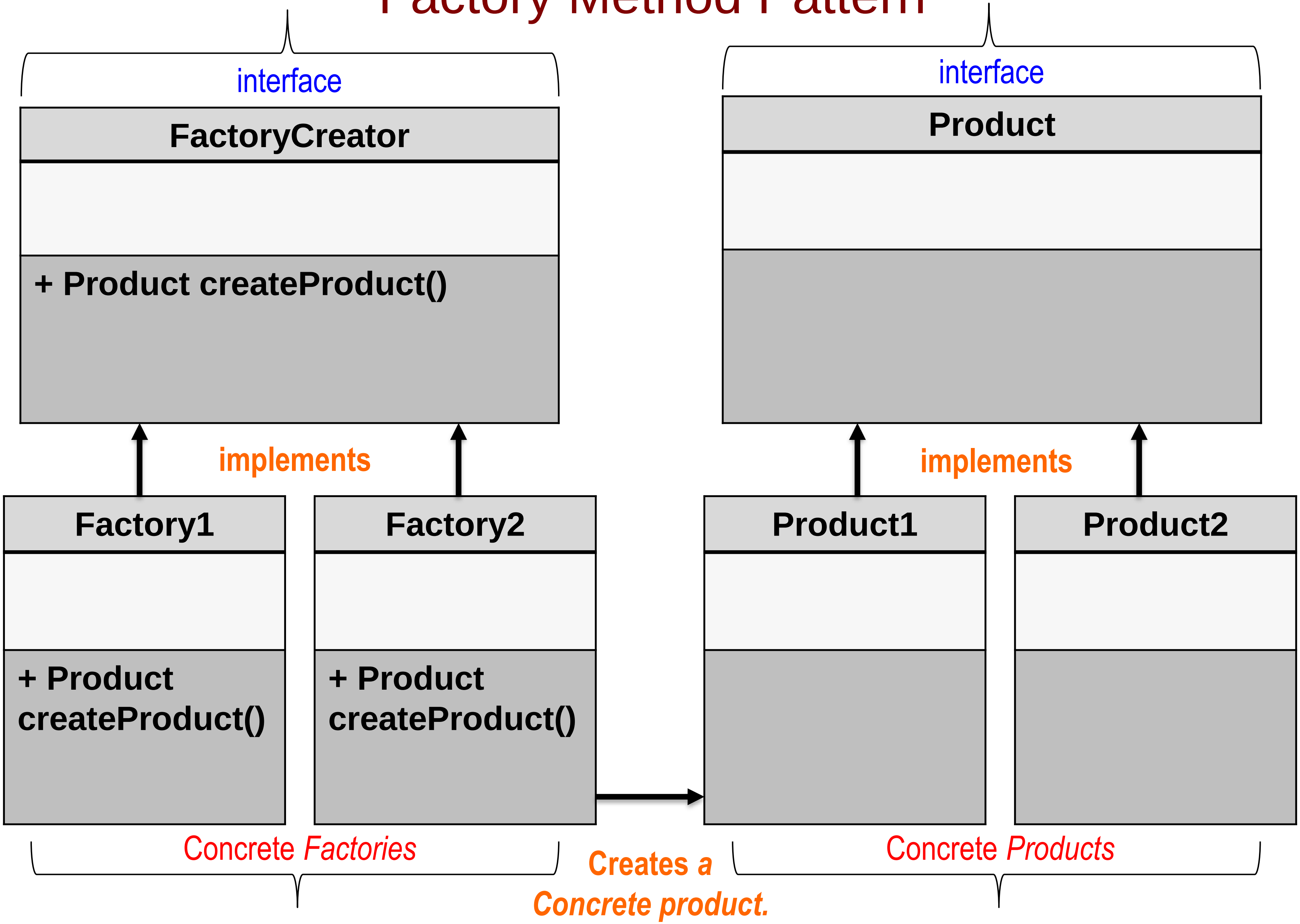


Application to build a UI

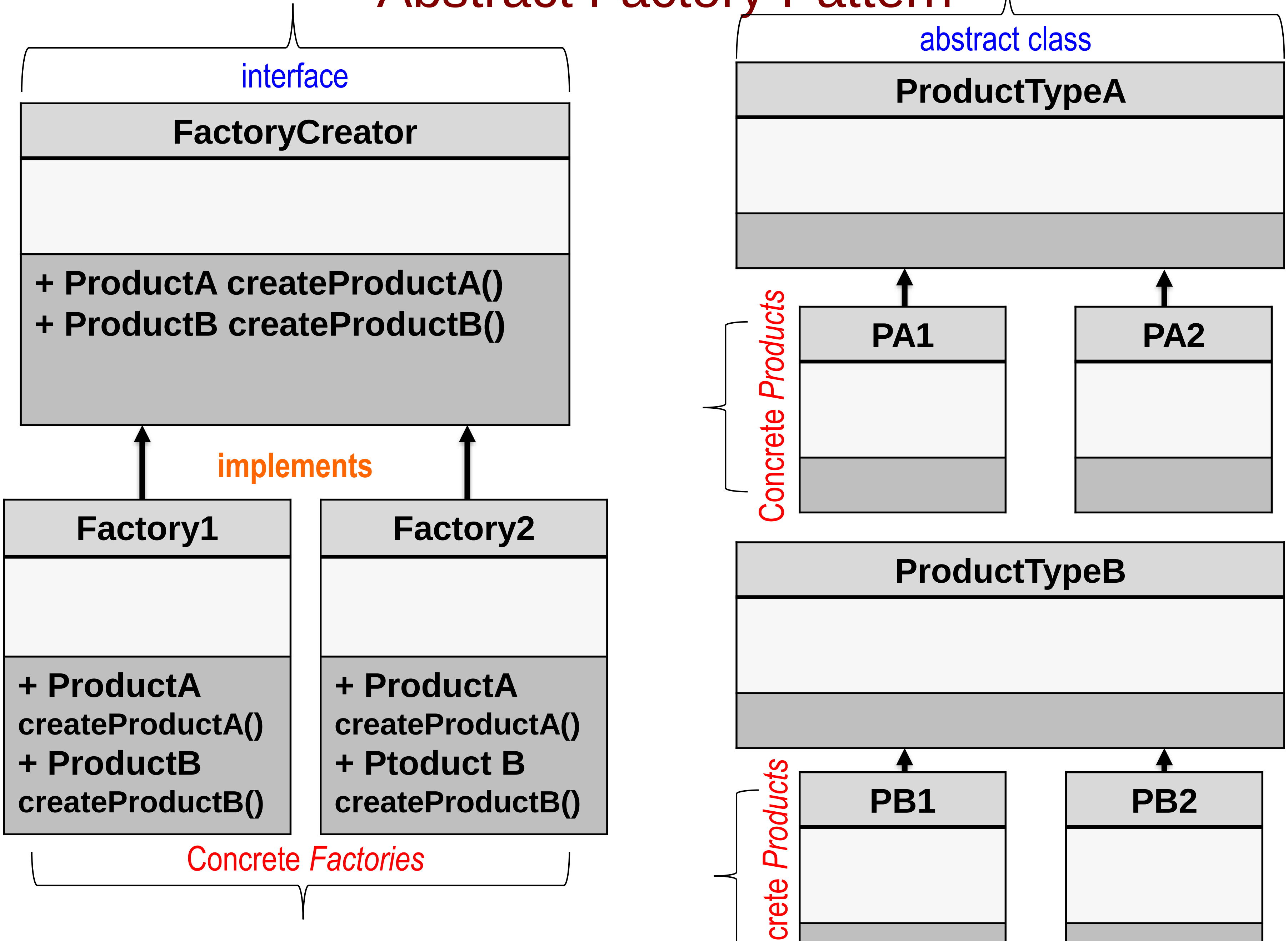
GUI related products for Mac OS

GUI related products for Windows

Factory Method Pattern



Abstract Factory Pattern



interface

FactoryCreator

+ ProductA createProductA()
+ ProductB createProductB()

implements

Factory1

+ ProductA createProductA()
+ ProductB createProductB()

Factory2

+ ProductA createProductA()
+ Ptoduct B createProductB()

Concrete *Factories*

abstract class

ProductTypeA

Concrete *Products*

PA1

PA2

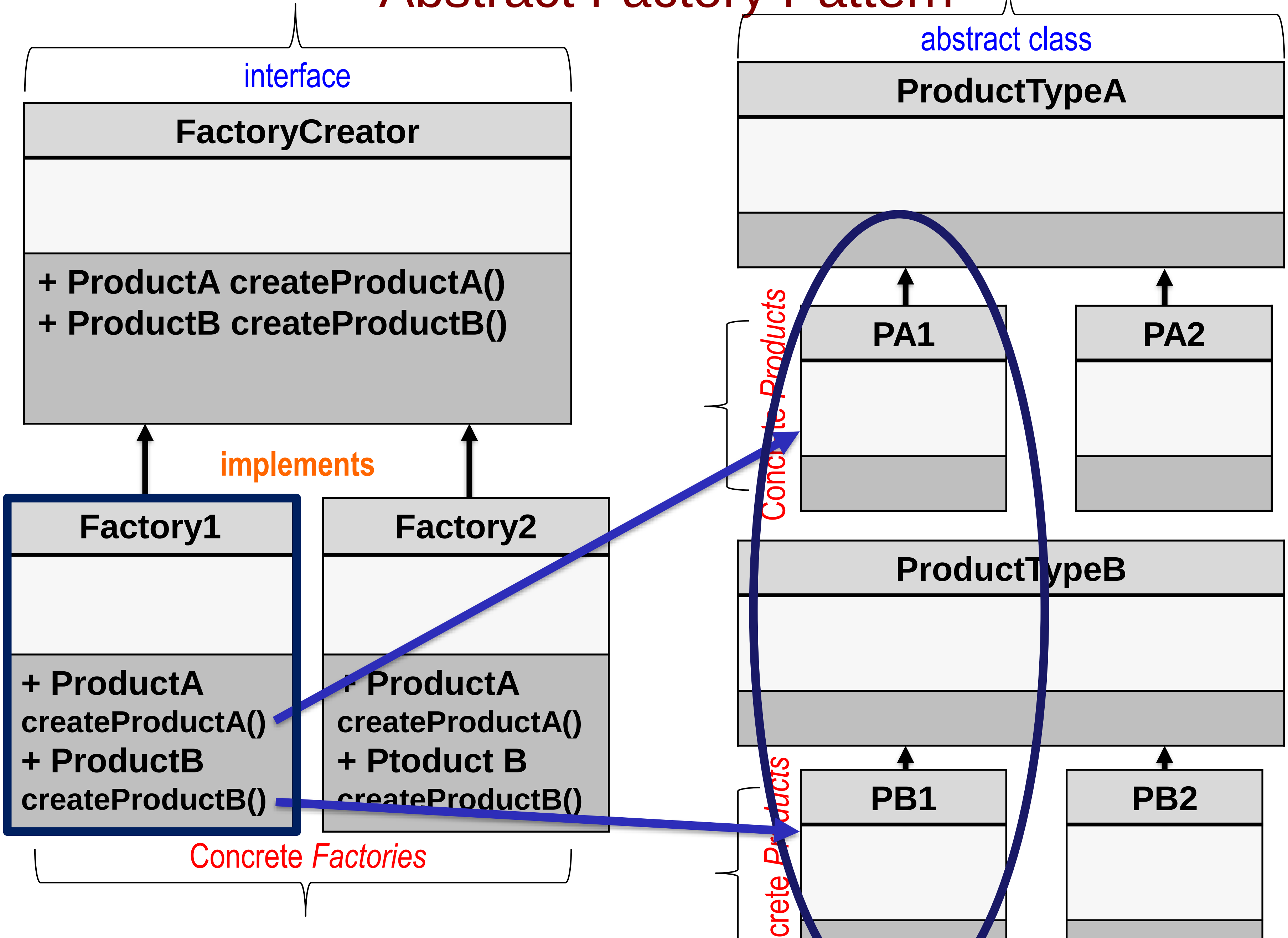
ProductTypeB

crete *Products*

PB1

PB2

Abstract Factory Pattern



interface

FactoryCreator

+ ProductA createProductA()
+ ProductB createProductB()

implements

Factory1

+ ProductA createProductA()
+ ProductB createProductB()

Factory2

+ ProductA createProductA()
+ Ptoduct B createProductB()

Concrete Factories

abstract class

ProductTypeA

Concrete Products

PA1

PA2

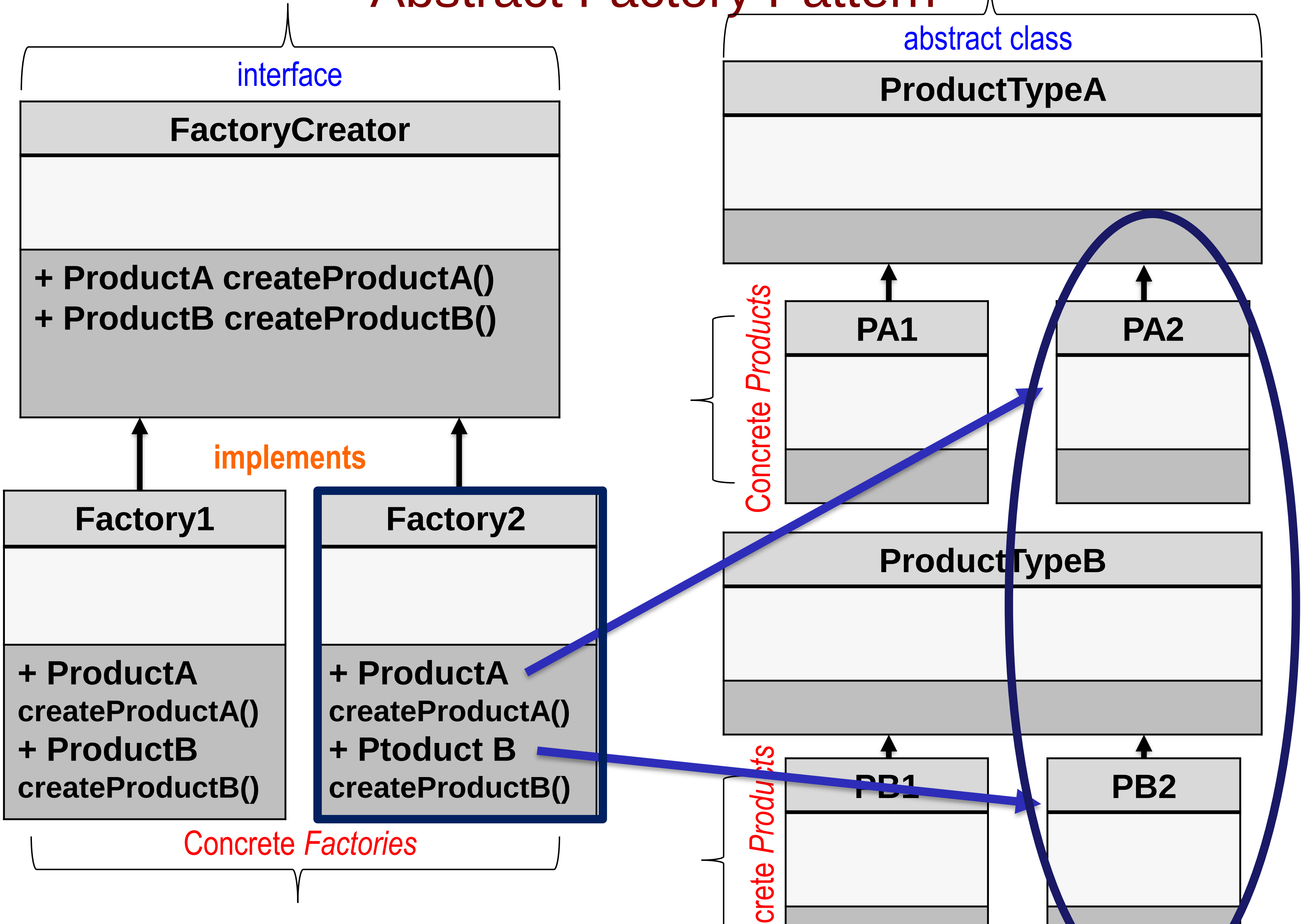
ProductTypeB

crete Products

PB1

PB2

Abstract Factory Pattern



interface

FactoryCreator

+ ProductA createProductA()
+ ProductB createProductB()

implements

Factory1

+ ProductA createProductA()
+ ProductB createProductB()

Factory2

+ ProductA createProductA()
+ Ptoduct B createProductB()

Concrete *Factories*

abstract class

ProductTypeA

Concrete *Products*

PA1

PA2

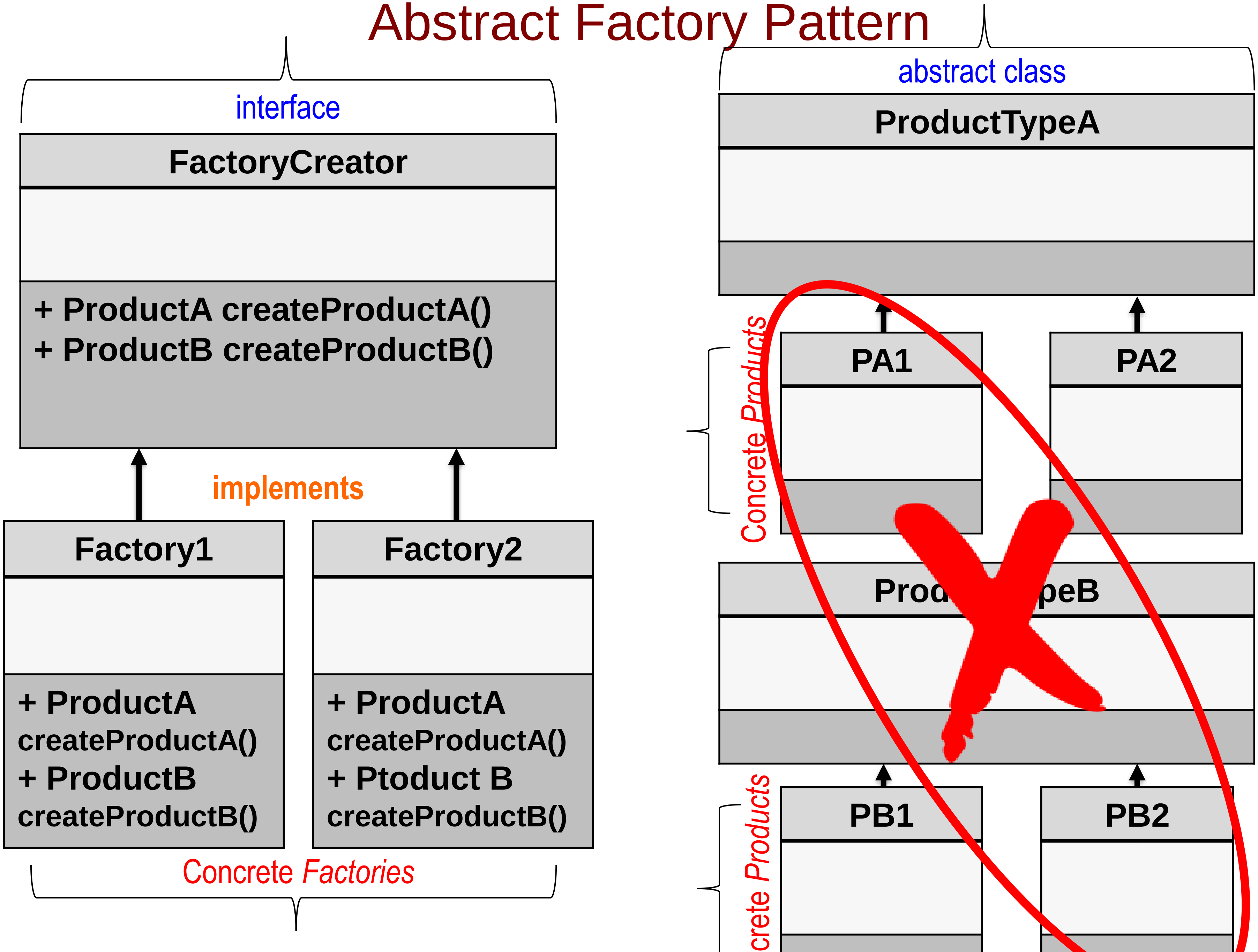
ProductTypeB

crete *Products*

PB1

PB2

Abstract Factory Pattern



interface

FactoryCreator

+ ProductA createProductA()
+ ProductB createProductB()

implements

Factory1

+ ProductA createProductA()
+ ProductB createProductB()

Factory2

+ ProductA createProductA()
+ Ptoduct B createProductB()

Concrete *Factories*

abstract class

ProductTypeA

Concrete *Products*

PA1

PA2

Prod peB

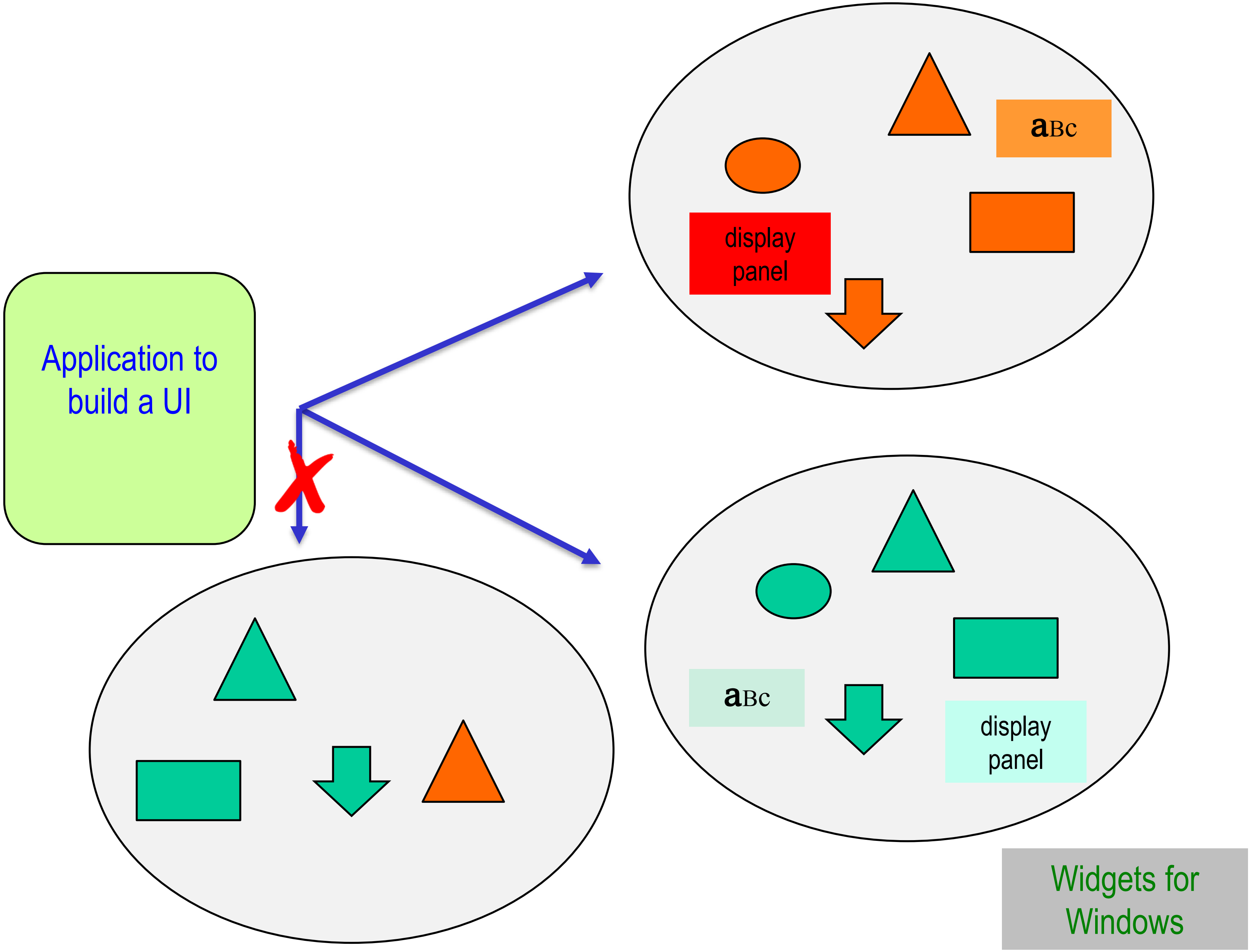
crete *Products*

PB1

PB2

Abstract Factory Pattern

Widgets for Mac OS



Application to build a UI

aBc

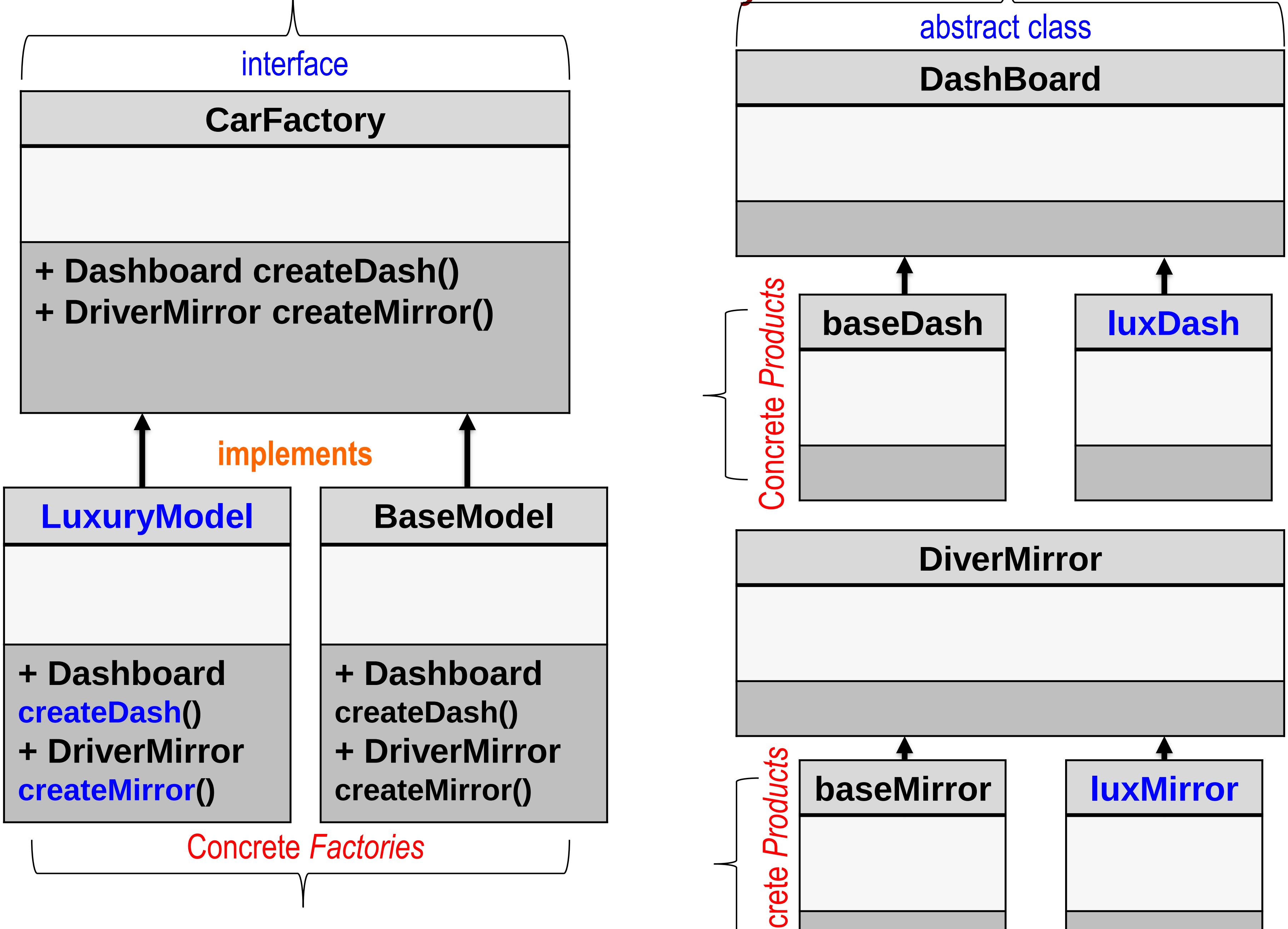
display panel

aBc

display panel

Widgets for Windows

Abstract Factory Pattern



interface

CarFactory

+ Dashboard createDash()
+ DriverMirror createMirror()

implements

LuxuryModel

+ Dashboard createDash()
+ DriverMirror createMirror()

BaseModel

+ Dashboard createDash()
+ DriverMirror createMirror()

Concrete *Factories*

abstract class

DashBoard

Concrete *Products*

baseDash

luxDash

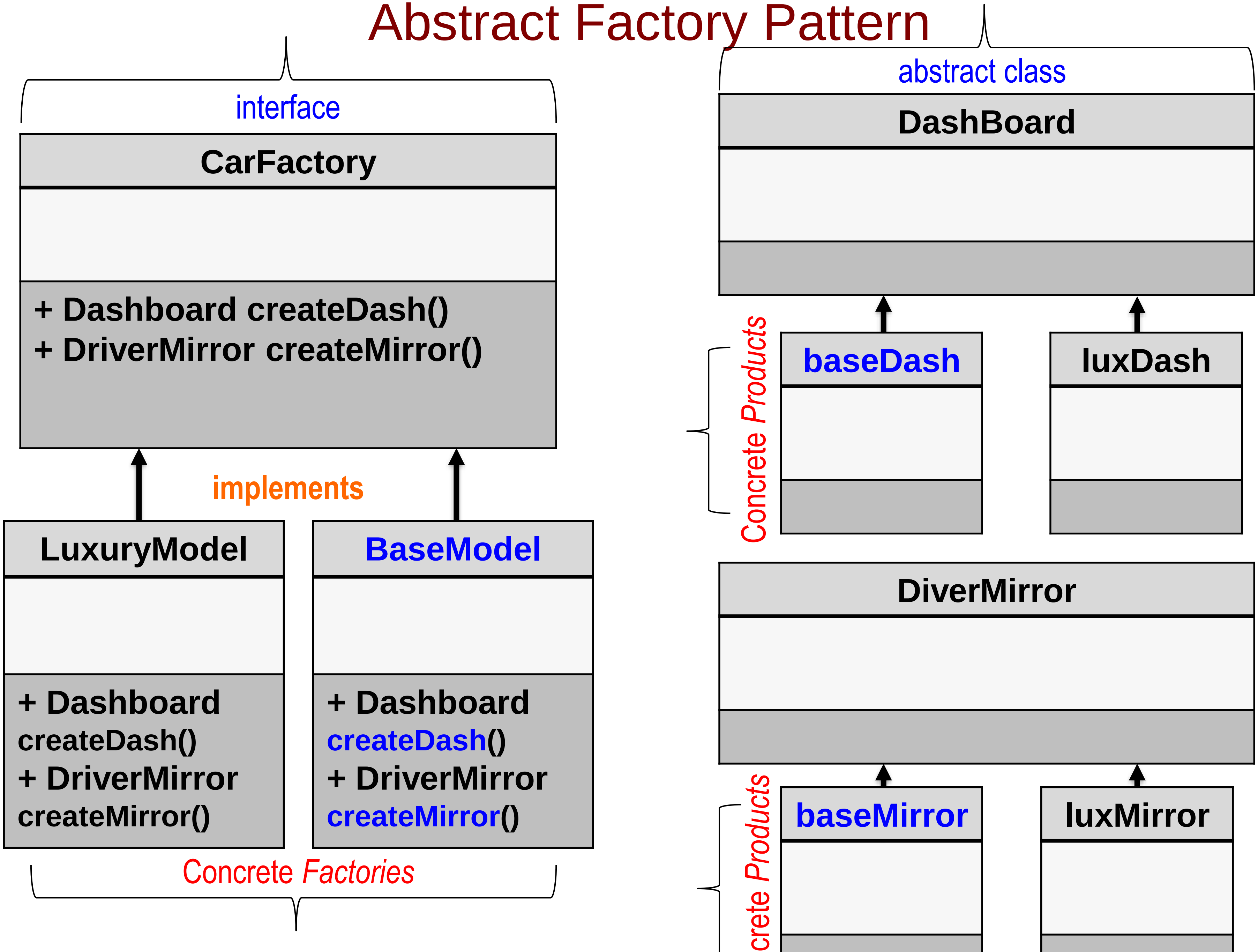
DiverMirror

crete *Products*

baseMirror

luxMirror

Abstract Factory Pattern



interface

CarFactory

+ Dashboard createDash()
+ DriverMirror createMirror()

implements

LuxuryModel

+ Dashboard createDash()
+ DriverMirror createMirror()

BaseModel

+ Dashboard createDash()
+ DriverMirror createMirror()

Concrete *Factories*

abstract class

DashBoard

Concrete *Products*

baseDash

luxDash

DiverMirror

crete *Products*

baseMirror

luxMirror

Abstract Factory Pattern:

As defined in Elements of Reusable OO Software

- Consequences (**Advantages**/Disadvantages): The Factory Pattern helps you control the classes of objects that an application creates.
  - As fac s of
    creatin ation
    classe
  - Allows sing to
    create **s a**
    compl anges
    at onc
  - It prom
  - Supporting new type of products requires changing the interface and concrete implementations of the interface.
  - Onerous when families of products differ slightly.

An abstract factory creates a complete family of products, therefore once you change which factory your application uses, you are guaranteed to create products of the same family. Facilitates portability and promotes consistency!

Abstract Factory Pattern:

As defined in Elements of Reusable OO Software

- Consequences (**Advantages**/Disadvantages): The Factory Pattern helps you control the classes of objects that an application creates.
  - As factories encapsulate the responsibility and the process of creating p tation classes.
  - Allows an using to create its s a **complete anges at once.**
  - It promotes consistency among products.
  - Supporting new type of products requires changing the interface and concrete implementations of the interface.
  - Onerous when families of products differ slightly.

Not ideal in the situation where there is product overlap or when the family of products only differ slightly!

Abstract Factory Pattern:

Elements of Reusable OO Software

- Consequences (Advantages/Disadvantages): The Factory Pattern helps you control the classes of objects that an application creates.
  - As factories encapsulate the responsibility and the process of creating product objects, it isolates clients from implementation classes.
  - Allows an application to easily change which factory it is using to create its products. Because an abstract factory creates a complete family of products, whole product family changes at once.
  - It promotes consistency among products.
  - Supporting new type of products requires changing the interface and concrete implementations of the interface.
  - Onerous when families of products differ slightly.
- If there should only be one specific concrete factory, can also implement the concrete factory as a *Singleton*.