Factory Method Pattern

CSCI-4448 - Boese

University of Colorado Boulder

Objectives

- Problem
- Definition
- Why
- Examples
- How
- Comparisons
- Summary

Problem

Factory Method Design Pattern

Problem:

- A class can't anticipate the class of objects it must create.
  - Client code dependencies

```
if (type.equals("cheese")) {
   pizza = new CheesePizza();
} else if (type.equals("greek")) {
   pizza = new GreekPizza();
} else if (type.equals("pepperoni")) {
   pizza = new PepperoniPizza();
}
```

Needs to be recompiled each time

- A dep. changes,
- Add new classes,
- Change this code
- Remove existing classes

This means that this code violates the **open-closed principle** and the "encapsulate what varies" design principle

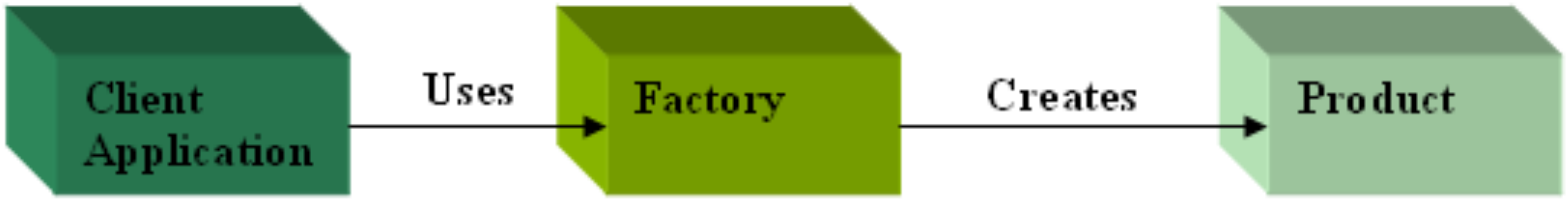
Definition

Definition

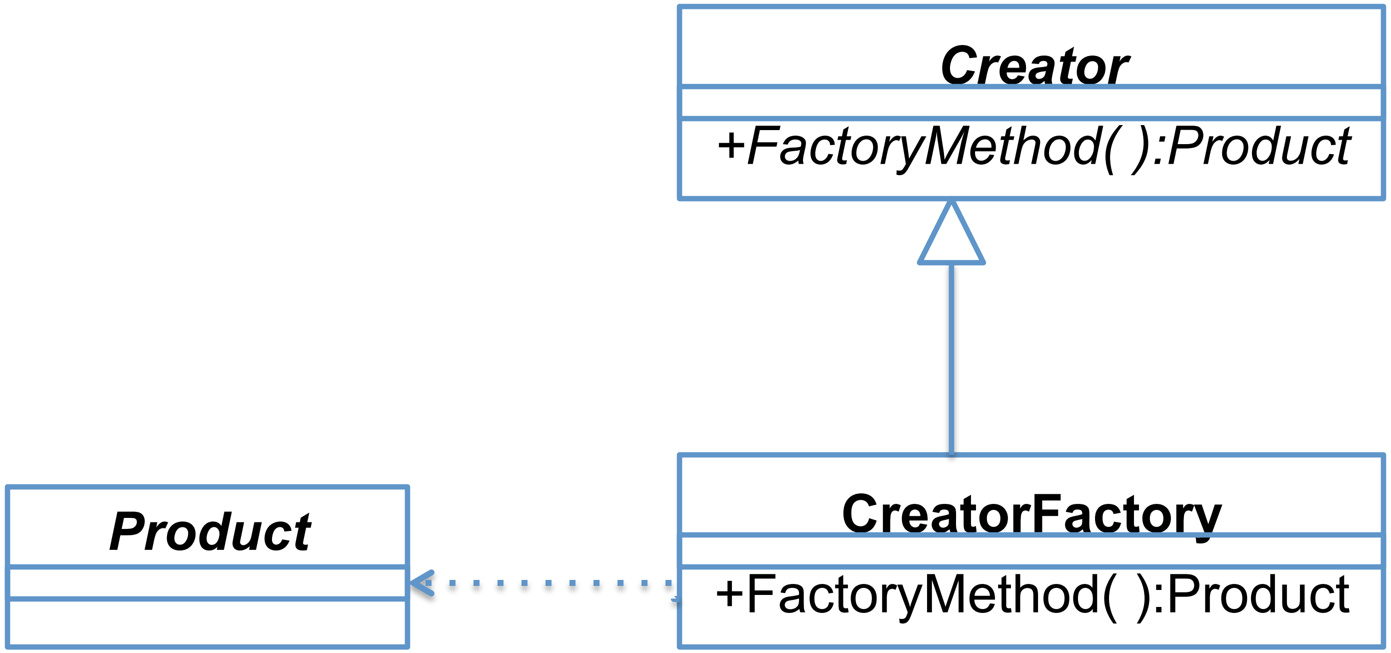
Used to replace class constructors,
by creating an abstraction through which one of several classes is returned determined at run-time.

High Level

Overview

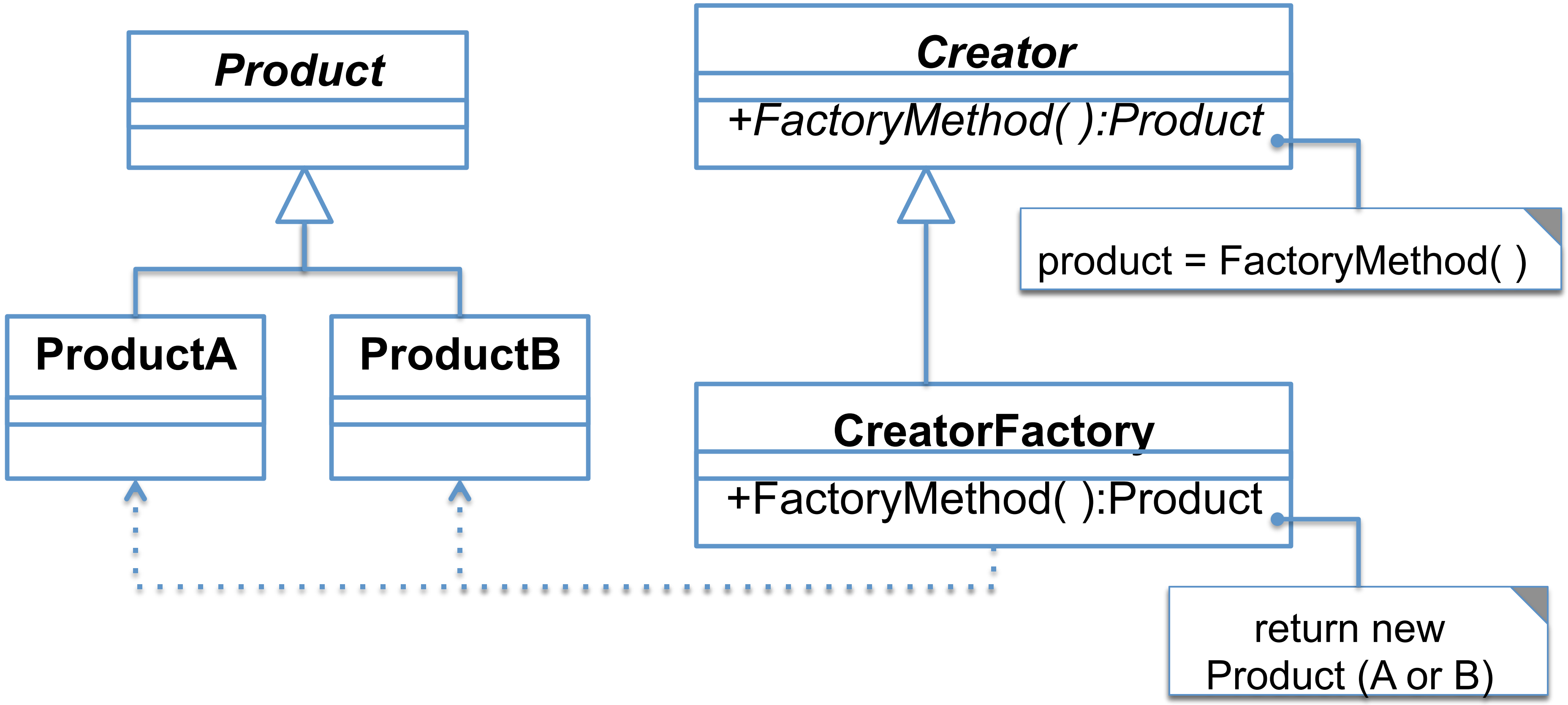


Structure


Creator
+FactoryMethod( ):Product
CreatorFactory
+FactoryMethod( ):Product
Product

Structure

See the need for polymorphic reference to Product?



Product
Creator
+FactoryMethod( ):Product
product = FactoryMethod( )
ProductA
ProductB
CreatorFactory
+FactoryMethod( ):Product
return new Product (A or B)

Definition – in more depth

- The **Factory Method** returns an instance of a class
  - Defined through an **interface** or **abstract parent class**.
  - Client is decoupled from the actual instantiated classes.
    - *Subclasses/classes that implement the interface can return self or another class type*
    - *Lets a class defer instantiation to subclasses*
  - All the returned classes through the factory methods have the same type via polymorphism:
    - *Interface or*
    - *Abstract parent class.*

Definition

Name "Factory Method"

- A 'factory' is a method (static or otherwise), an object, or anything else that is used to instantiate other objects.

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The new operator considered harmful.

Type

- Creational
- Class

*Makes use of **inheritance** or **interface** to decide what object to instantiate*

Factory Method Design Pattern

Problem:

- A class can't anticipate the class of objects it must create.
  - Client code dependencies

Needs to be recompiled each time

- A dep. changes,
- Add new classes,
- Change this code
- Remove existing classes

```
if (type.equals("cheese")) {
   pizza = new CheesePizza();
} else if (type.equals("greek")) {
   pizza = new GreekPizza();
} else if (type.equals("pepperoni")) {
   pizza = new PepperoniPizza();
}
```

This means that this code violates the **open-closed principle** and the "encapsulate what varies" design principle

Factory Method Design Pattern

Problem:

- Creating an object often requires complex processes not appropriate to include within a constructor.
  - The object's creation may lead to a significant duplication of code,
  - May require information not accessible to the composing object,
  - May not provide a sufficient level of abstraction,
  - May otherwise not be part of the composing object's concerns.

Why

Why use Factory Method Pattern?

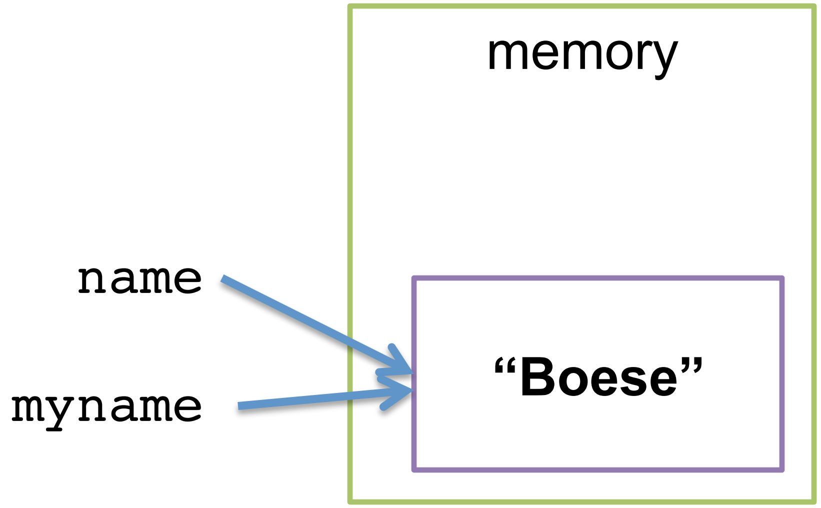
- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates
- Code is made more flexible and reusable by the elimination of instantiation of application-specific classes
- Introduces loose coupling between classes.
- Good approach to encapsulation.
- Commonly used in logging frameworks

Example Scenarios

- Optimize with an “object pool” to allow re-using of objects instead of creating from scratch.

```
String name = new String(“Boese”);
String myname = new String(“Boese”);
```

Strings are immutable. Therefore, if program needs another “Boese” String, can re-use the same one already in memory.



Examples

Pizza

```java
public class PizzaStore {
  public Pizza orderPizza(String type) {
    Pizza pizza;

      if (type.equals("cheese")) {
            pizza = new CheesePizza();
      } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
      } else if (type.equals("clam")) {
            pizza = new ClamPizza();
      } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
      }

         pizza.prepare();
         pizza.bake();
         pizza.cut();
         pizza.box();

         return pizza;
    }
}
```

Problem:
Creating objects

Excellent example of
"Coding to an interface"

Initial fix: encapsulate creation code in a separate class

```
public class PizzaStore
{
   private SimplePizzaFactory factory;
   public PizzaStore(SimplePizzaFactory factory)
   {
       this.factory = factory;
   }
  public Pizza orderPizza(String type) {
     Pizza pizza = factory.createPizza(type);

       pizza.prepare();
       pizza.bake();
       pizza.cut();
       pizza.box();

       return pizza;
   }
}
```

```
public class SimplePizzaFactory {
 public Pizza createPizza(String type) {
  Pizza pizza = null;

  if (type.equals("cheese")) {
     pizza = new CheesePizza();
  } else if (type.equals("pepperoni")) {
     pizza = new PepperoniPizza();
  } else if (type.equals("clam")) {
     pizza = new ClamPizza();
  } else if (type.equals("veggie")) {
     pizza = new VeggiePizza();
  }
  return pizza;
 }
}
```

This class no longer needs to change with new/deleted pizza types

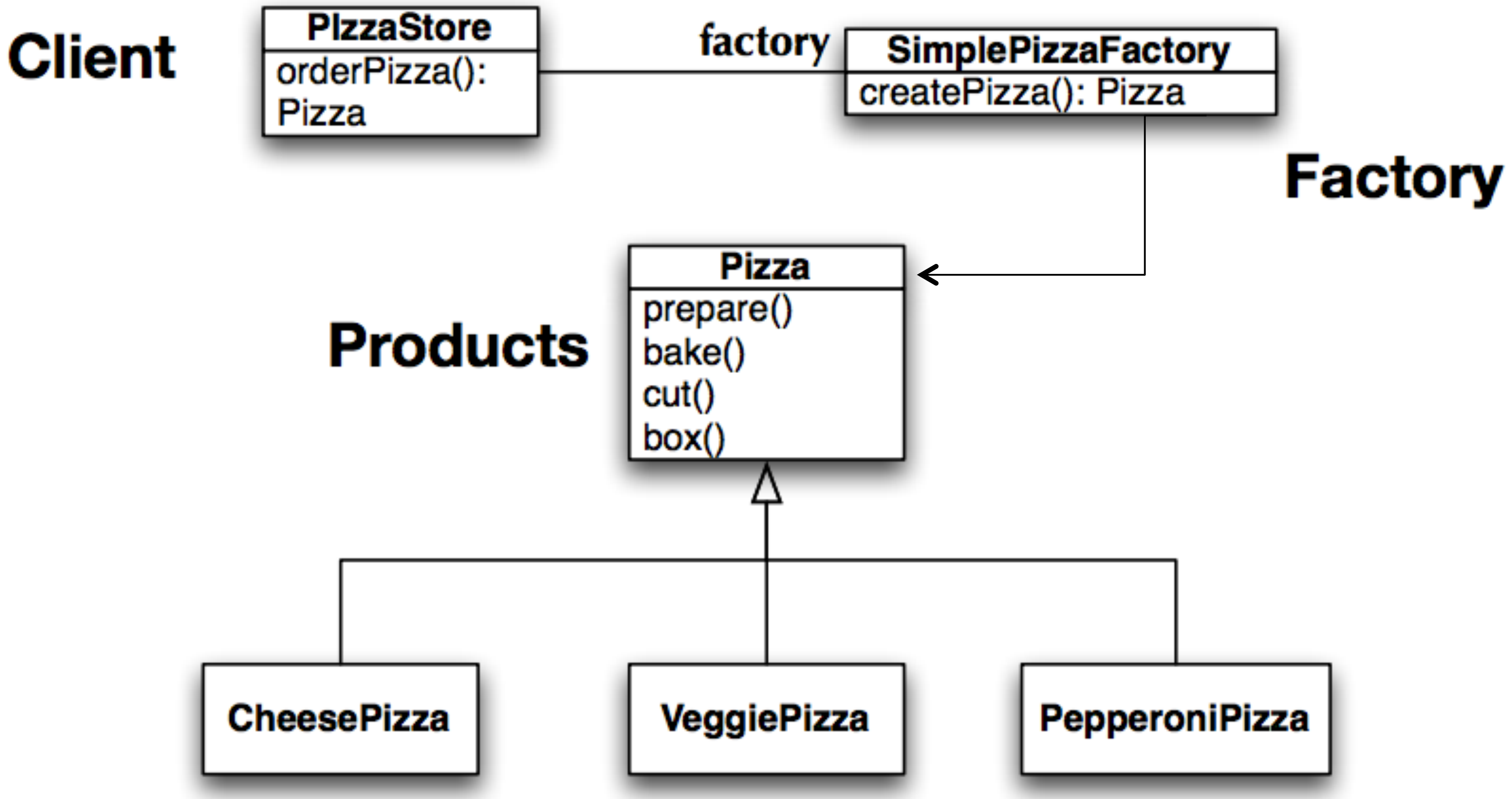
Encapsulate what varies!

This class now depends on concrete classes…

University of Colorado Boulder

Example: Pizza

- New Diagram, based on Factory Pattern.



Example: Pizza

- Move towards Factory Method Pattern
  - Abstract base class

```
public abstract class PizzaStore {
   abstract Pizza createPizza(String item);
    public Pizza orderPizza(String type)
    {
       Pizza pizza = createPizza(type);

       pizza.prepare();
       pizza.bake();
       pizza.cut();
       pizza.box();

       return pizza;
    }
}
```

createPizza() = factory method

Dependencies on concrete “product” classes are encapsulated in subclass

Example: Pizza

- Subclass ensures the correct pizza is created.

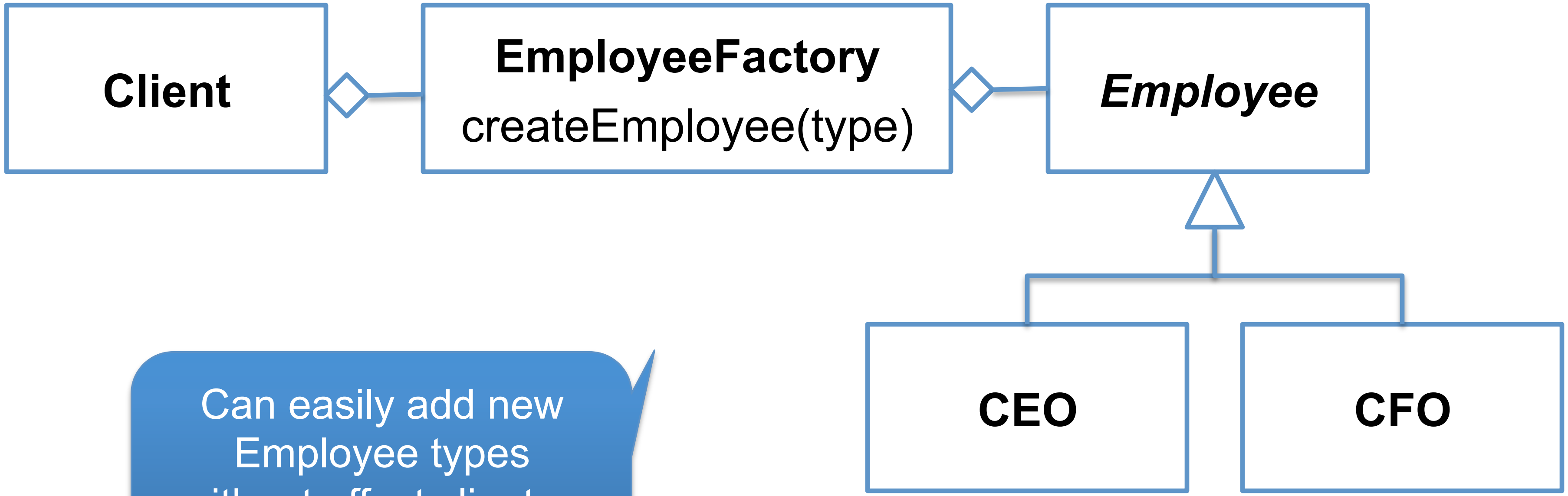
```
public class NYPizzaStore extends PizzaStore
{
    public static Pizza createPizza(String item)
    {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

Examples

Employees

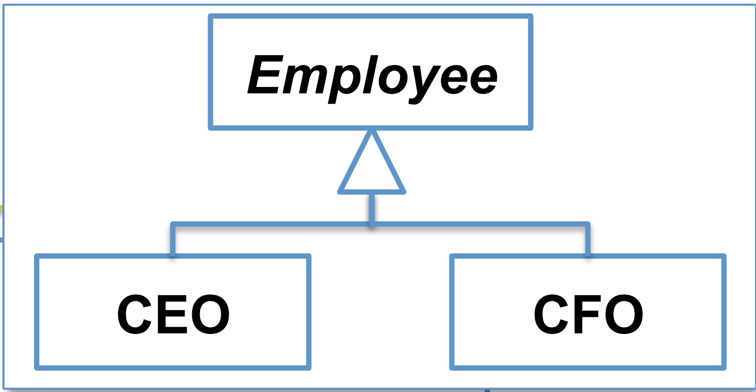
Example: Employees

Keep track of employees and their roles in the company.



Can easily add new Employee types without affect client or Factory

Example: Employees



```
public abstract class Employee {
   public abstract String getRole();
}
```

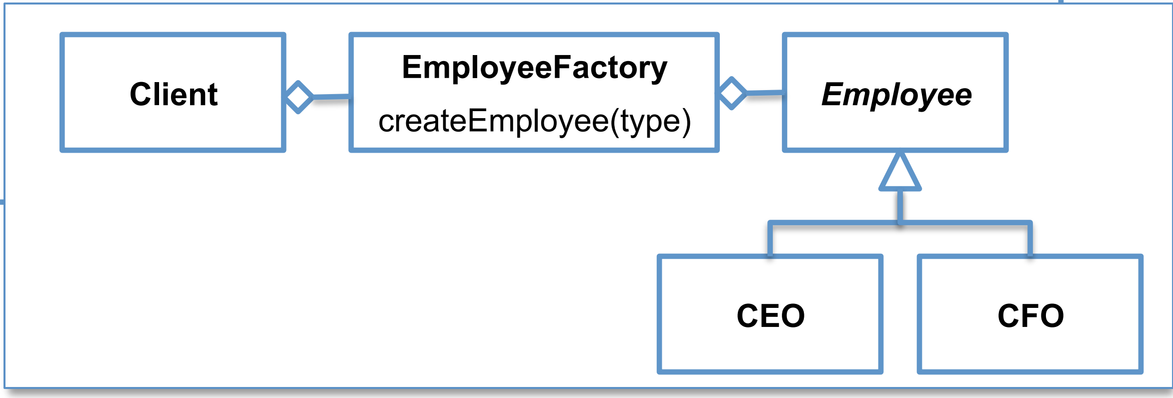
```
public class CEO extends Employee {
   public String getRole() {
      return (“CEO is the head of the company.”);
   }
}
```

```
public class CFO extends Employee {
   public String getRole() {
      return (”CFO is head of finance.”);
   }
}
```

Example: Employees

Factory class

```
public class EmployeeFactory
{
   public static void getEmployee(String type)
   {
      Employee ee = null;
      if (type.equals(“CEO”))
         ee = new CEO();
      else if (type.equals(“CFO”))
         ee = new CFO();
      return ee;
   }
```



University of Colorado Boulder

Example: Person

- Client only needs to call the generic function in the factory.
- If new roles are added, doesn't change the client

```
public class Client
{
   public static void main(String args[])
   {
      Employee liz = EmployeeFactory.getEmployee("CEO");
   }
}
```

Examples

Java API

Example: Java API

- Factory design pattern used in Java standard library

- Based on different parameter, `getInstance()` returns a different instance of `Calendar`.

- `java.util.Calendar – getInstance()`
- `java.util.Calendar – getInstance(TimeZone zone)`
- `java.util.Calendar – getInstance(Locale aLocale)`
- `java.util.Calendar – getInstance(TimeZone zone, Locale aLocale)`

- `java.text.NumberFormat – getInstance()`
- `java.text.NumberFormat – getInstance(Locale inLocale)`
- `java.util.ResourceBundle - getBundle()`
- `java.net.URLStreamHandlerFactory - createURLStreamHandler(String)`
  `(Returns singleton object per protocol)`

Examples

Image Reader

Example: Image Reader

- Encapsulate creation of objects.
- Ex: Readd image files and make thumbnails.

```java
public interface ImageReader {
   public DecodedImage getDecodedImage();
}
```

```java
public class GifReader implements ImageReader {
   public DecodedImage getDecodedImage() {
   // ...
   return decodedImage;
   }
}
```

```java
public class JpegReader implements ImageReader {
   // ...
   return decodedImage;
}
```

Example: Image Reader

Factory

```
public class ImageReaderFactory {
  public static ImageReader getImageReader(InputStream is)
  {
     int imageType = determineImageType(is);
     switch(imageType) {
        case ImageReaderFactory.GIF:
        return new GifReader(is);
     case ImageReaderFactory.JPEG:
        return new JpegReader(is);
     // etc.
     }
  }
}
```

Examples

Descriptive Instantiation Names

Example: Descriptive Instantiation Names

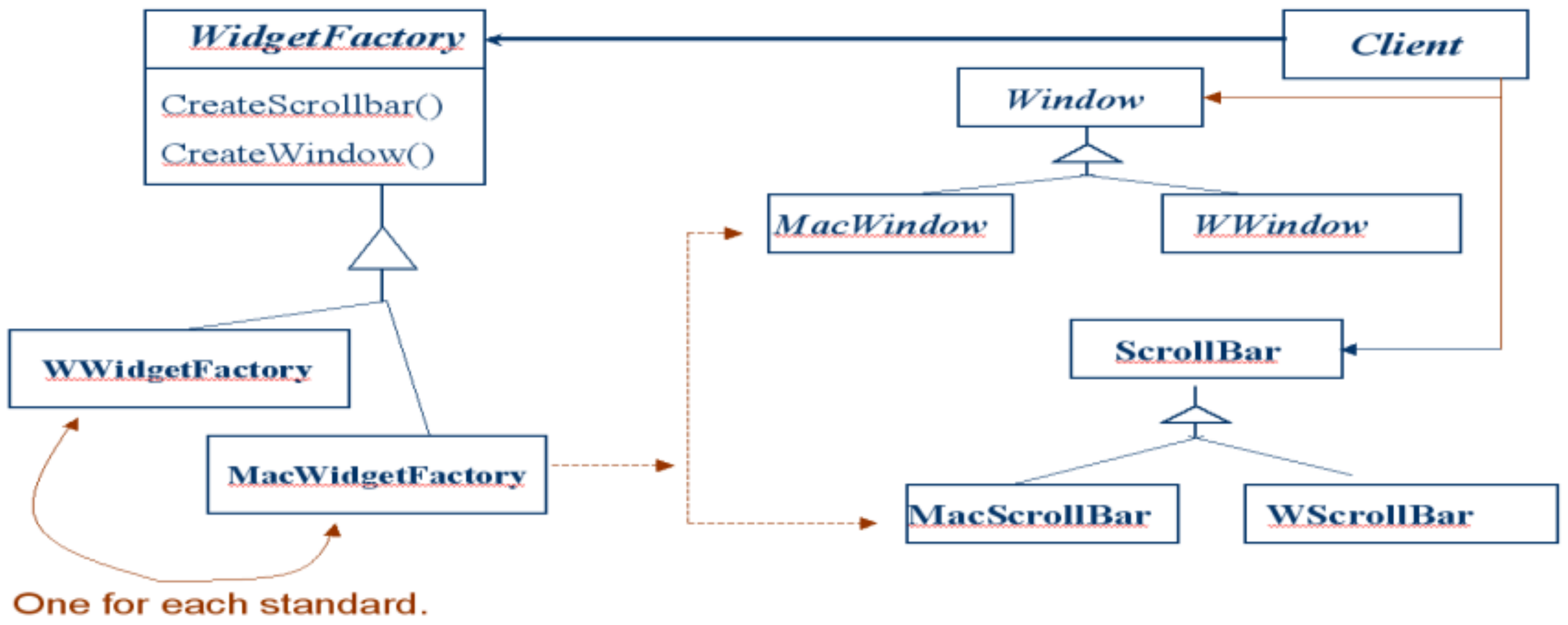
Can have descriptive names to instantiate
(e.g., Java constructor = class name)

```
class Complex {
  public static Complex fromCartesian(double real, double imaginary) {
    return new Complex(real, imaginary);
  }
  public static Complex fromPolar(double modulus, double angle) {
    return new Complex(modulus * cos(angle), modulus * sin(angle));
  }
  private Complex(double a, double b) {
     //...
  }
}
Complex c = Complex.fromPolar(1, pi);
```

Examples

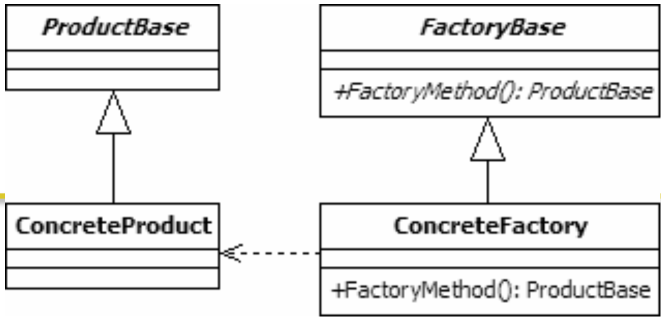
Widgets

Example: Widgets



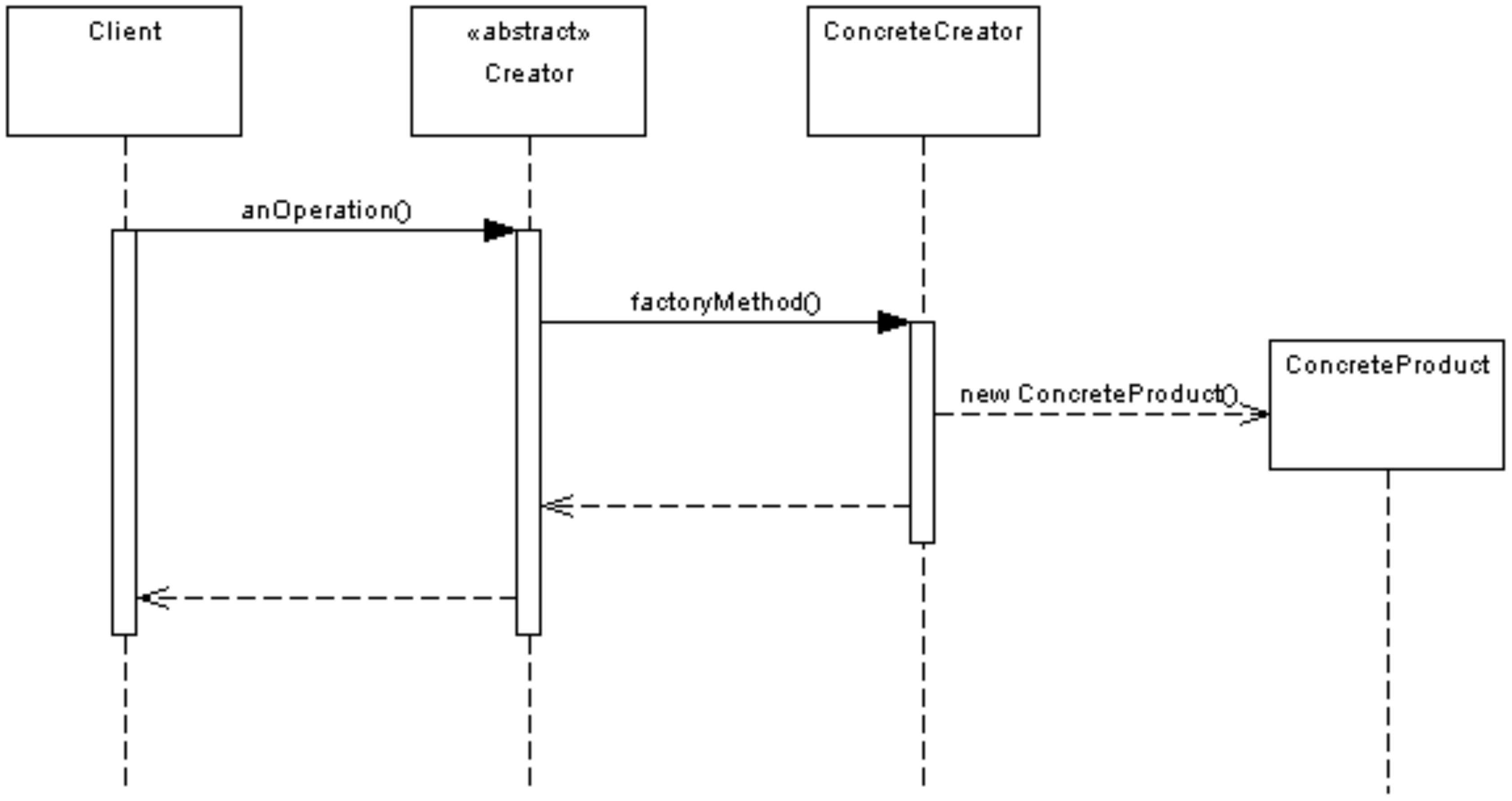
How

Factory Method Structure



- **FactoryBase**
  - An abstract base class for the concrete factory classes that will actually generate new objects. This class could be a simple interface containing the signature for the factory method. However, generally an abstract class will be used so that other standard functionality can be included and inherited by subclasses. In simple situations the factory method may be implemented in full here, rather than being declared as abstract.
- **ConcreteFactory**
  - Inheriting from the FactoryBase class, the concrete factory classes inherit the actual factory method. This is overridden with the object generation code unless already implemented in full in the base class.
- **ProductBase**
  - Abstract class is the base class for the types of object that the factory can create. It is also the return type for the factory method. Again, this can be a simple interface if no general functionality is to be inherited by its subclasses.
- **ConcreteProduct**
  - Multiple subclasses of the Product class are defined, each containing specific functionality. Objects of these classes are generated by the factory method.

Factory Method Pattern Sequence Diagram



How

1. If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
2. Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
3. Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
4. Consider making all constructors private or protected.

How - Note

Consider what you need to have in your system before you concern yourself with how to create it… That is, we define our factories after we decide what our objects are.

Comparisons

- **Factory** - Creates objects without exposing the instantiation logic to the client and Refers to the newly created object through a common interface. Is a simplified version of Factory Method

- **Factory Method** - Defines an interface for creating objects, but let subclasses to decide which class to instantiate and Refers to the newly created object through a common interface.
- **Abstract Factory** - Offers the interface for creating a family of related objects, without explicitly specifying their classes.

Comparisons

Abstract Factory vs. Factory Method

- Factory Method is similar to Abstract Factory but without the emphasis on families.
- The methods of an Abstract Factory are implemented as Factory Methods.
- Both the Abstract Factory Pattern and the Factory Method Pattern decouples the client system from the actual implementation classes through the abstract types and factories.
  The Factory Method creates objects through inheritance where the Abstract Factory creates objects through composition.

Summary

Summary

- **Factory Method** makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

- People often use Factory Method as the standard way to create objects; but it isn't necessary if:
  - the class that's instantiated never changes, or
  - instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).
- The use of factories is a natural result of hiding what varies.
- In a lot of cases, a simple factory pattern will work fine.
  The FactoryMethod just allows further decoupling, leaving it to the subclasses of the Creator to decide which type of concrete Product to create.

Summary

An increasingly popular definition of factory method:

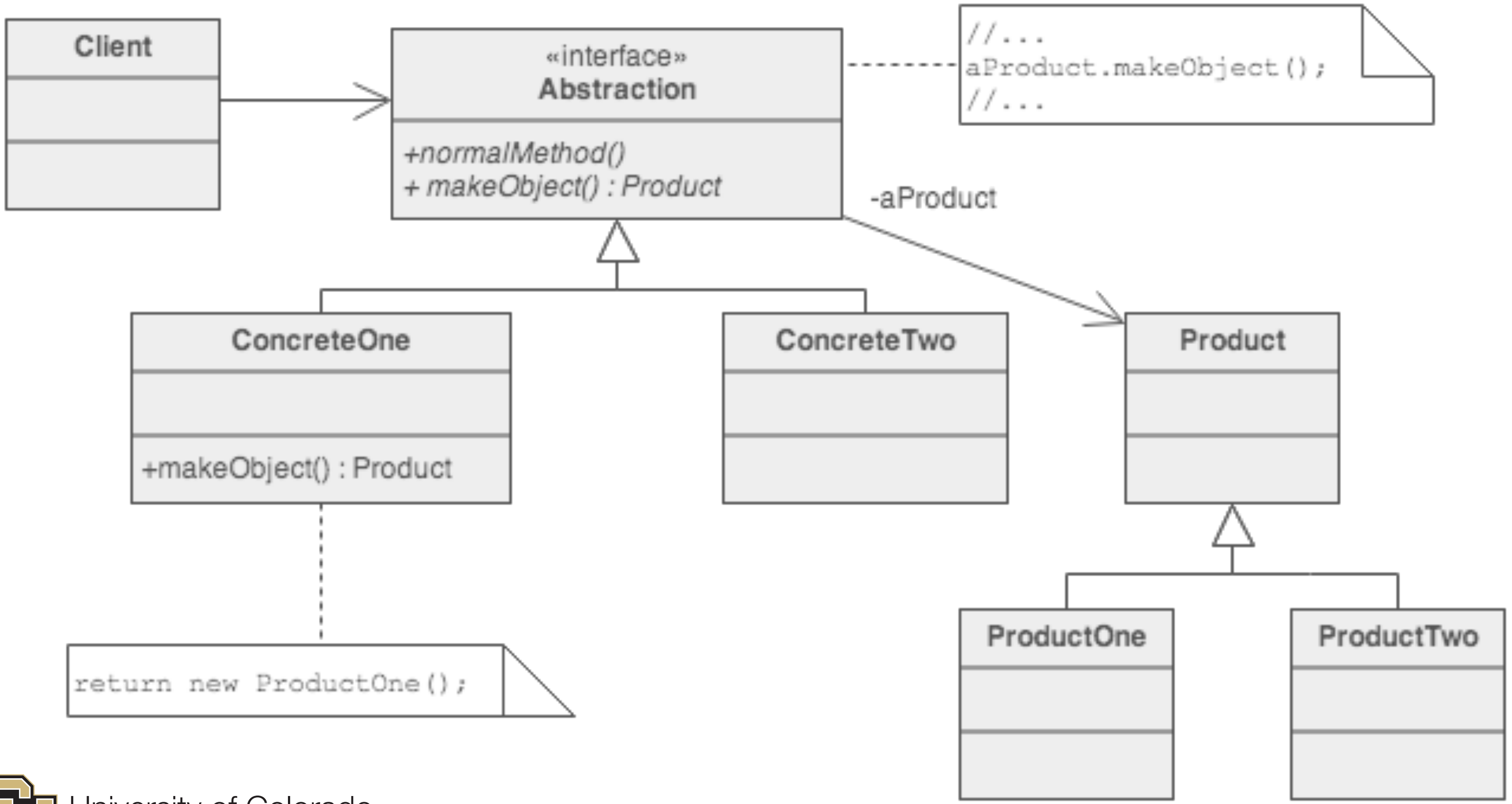
- A static method of a class that returns an object of that class' type.
- But unlike a constructor,
  - *The actual object it returns might be an instance of a subclass.*
  - *An existing object might be reused, instead of a new object created.*
  - *Factory methods can have different and more descriptive names*
    E.g.,
    - *Color.make_RGB_color(float red, float green, float blue)*
    - *Color.make_HSB_color(float hue, float saturation, float brightness)*

So what exactly does it mean when we say that "*the Factory Method Pattern lets subclasses decide which class to instantiate?*"

- It means that Creator class is written without knowing what actual ConcreteProduct class will be instantiated. The ConcreteProduct class which is instantiated is determined solely by which ConcreteCreator subclass is instantiated and used by the application.
- It does not mean that somehow the subclass decides at runtime which ConcreteProduct class to create
- E.g., If the document should be printed, instantiate basic shapes without shading… if for the screen, instantiate original shapes with shading…

Subclasses

Example demonstrating subclasses choosing what to instantiate



Client
«interface»
Abstraction
+normalMethod()
+ makeObject() : Product
//...
aProduct.makeObject();
//...
-aProduct
ConcreteOne
+makeObject() : Product
ConcreteTwo
Product
return new ProductOne();
ProductOne
ProductTwo