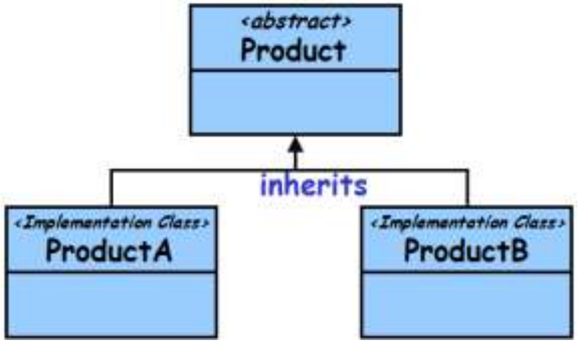
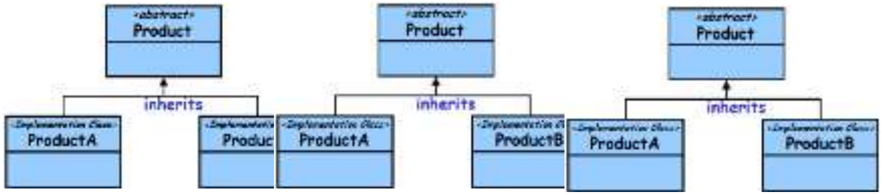
Factory Pattern

Factory Variations

- Three main Variants:
- **Simple Factory:**
  - Returns an object of a class from a class hierarchy



- **Factory Method pattern:**
  - Produces objects of one type
  - Uses an overridable method to create its objects
  - Subclassed to make new kinds of factories



- **Abstract Factory pattern:**
  - Produces objects of many different types (families)

```
Pizza orderPizza() {
  Pizza pizza = new Pizza(); //Base
  pizza.garnish();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

Motivation for Simple Factory

```
Pizza orderPizza(String type) {
  Pizza pizza;
  if (type.equals(“cheese”)) {
  pizza = new CheesePizza();
  } else if (type.equals(“greek”)) {
  pizza = new GreekPizza();
  } else if
  (type.equals(“pepperoni”)){
  pizza =new PepperoniPizza(); }
```

```
  pizza.garnish();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

```
Pizza orderPizza(String type) {
  Pizza pizza;
  if (type.equals("cheese")) {
  pizza = new CheesePizza();
  } else if (type.equals("greek")) {
  pizza = new GreekPizza();
  } else if (type.equals("pepperoni")) {
  pizza = new PepperoniPizza();
  } else if (type.equals("sausage")) {
  pizza = new SausagePizza();
  } else if (type.equals("veggie")) {
  pizza = new VeggiePizza();
  }
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

Closed for changes!

Encapsulate

Want to introduce new base pizzas…

```
public class SimplePizzaFactory {
  public static Pizza createPizza(String type) {
    Pizza pizza;
    if (type.equals(“cheese”)) {
      pizza = new CheesePizza();
    } else if (type.equals(“pepperoni”)) {
      pizza = new PepperoniPizza();
    } else if (type.equals(“sausage”)) {
      pizza = new SausagePizza();
    } else if (type.equals(“veggie”)) {
      pizza = new VeggiePizza();   }
    return pizza;
  }
}
```

Motivation for Simple Factory

Now orderPizza() would be tidy

```
public class PizzaStore {
  SimplePizzaFactory factory;
```

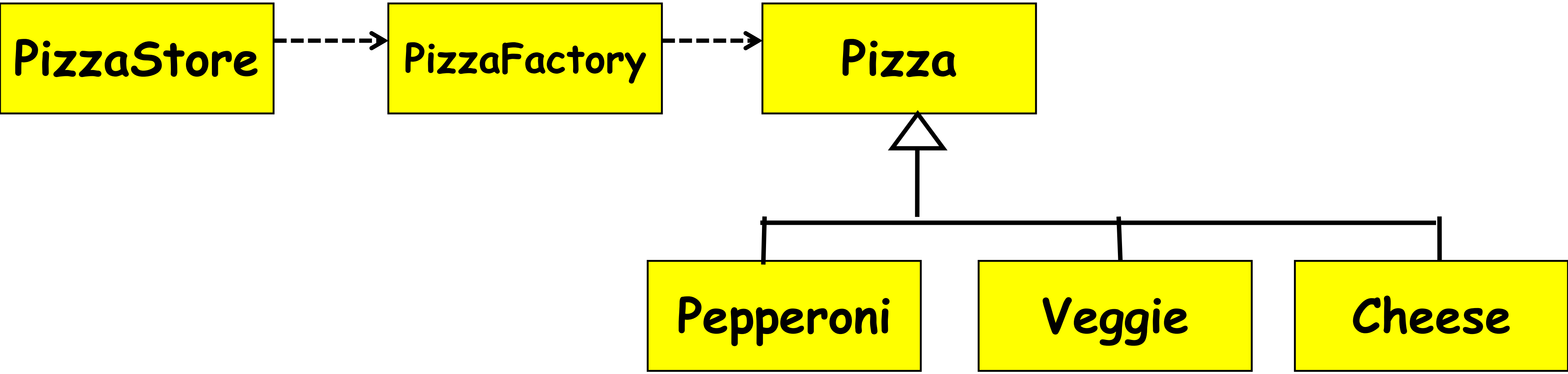
No new operator

```
  public static Pizza
  orderPizza(String type) {
    Pizza pizza;
    pizza =
  factory.createPizza(type);

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

    return pizza;
  }
}
```

Pizza Factory Class Diagram



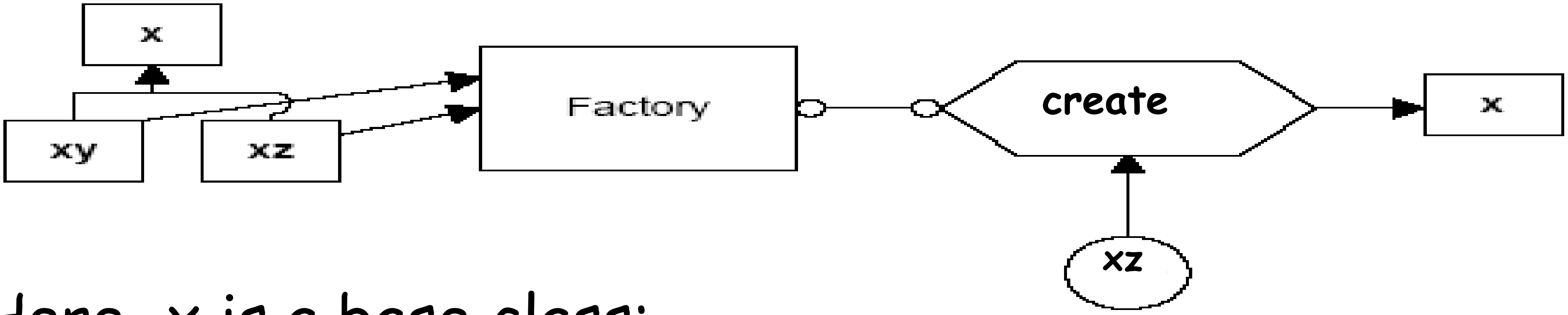
- ***A Simple Factory:***
  - Not quite the Factory pattern, to do so we would need an abstract **PizzaFactory** class.

Simple Factory: An Explanation

- Pull out the code that builds the instances, and put it into a separate factory class:
  - **Principle:** Identify the aspects of your application that vary and separate them from what stays the same…

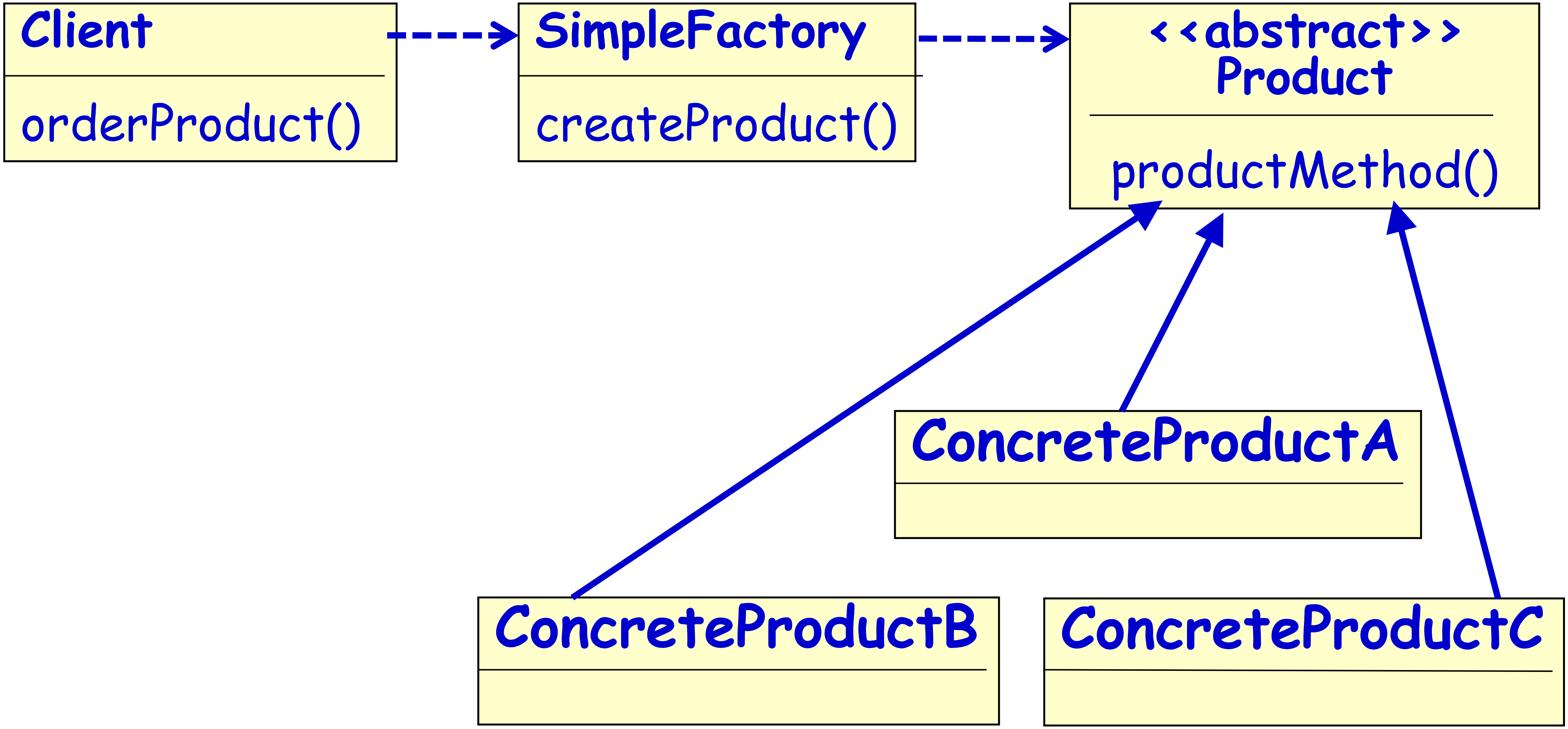
Simple Factory Pattern: Explanation

The simple Factory returns an instance of one of several possible classes depending on the data provided to it.



- Here, x is a base class:
  - Classes xy and xz are derived from it.
  - The Factory class decides which of these subclasses to return depending on the arguments you give it.
- The create() method gets value xz, and returns an instance of the class xz.
  - Which one it returns doesn't matter to the programmer since they are all of type X, but different implementations.

Simple Factory Pattern



Client

orderProduct()

SimpleFactory

createProduct()

<<abstract>>
Product

productMethod()

ConcreteProductA

ConcreteProductB

ConcreteProductC

Why Would We do This?

- Two main reasons:
  - Ensure consistent object initialization when multiple clients need the same types of objects.
  - Open for modification

Case for Simple Factory: 2 Examples

- Code to construct many GUI components:

```
homestarItem = new JMenuItem("Homestar Runner");
 homestarItem.addActionListener(this);
 viewMenu.add(homestarItem);
 crapItem = new JMenuItem("Crappy");
 crapItem.addActionListener(this);
 viewMenu.add(crapItem);
```

- Another example (with buttons):

```
button1 = new JButton();
 button1.addActionListener(this);
 button1.setBorderPainted(false);

 button2 = new JButton();
 button2.addActionListener(this);
 button2.setBorderPainted(false);
```

Factory Example 1

```
public class ButtonFactory {
   private ButtonFactory() {}
  public static JButton createButton(
    String text, ActionListener listener, Container
  panel){
    JButton button = new JButton(text);
    button.setMnemonic(text.charAt(0));
    button.addActionListener(listener);
    panel.add(button);
    return button;
  }
}
```

Simple Factory Advantages…

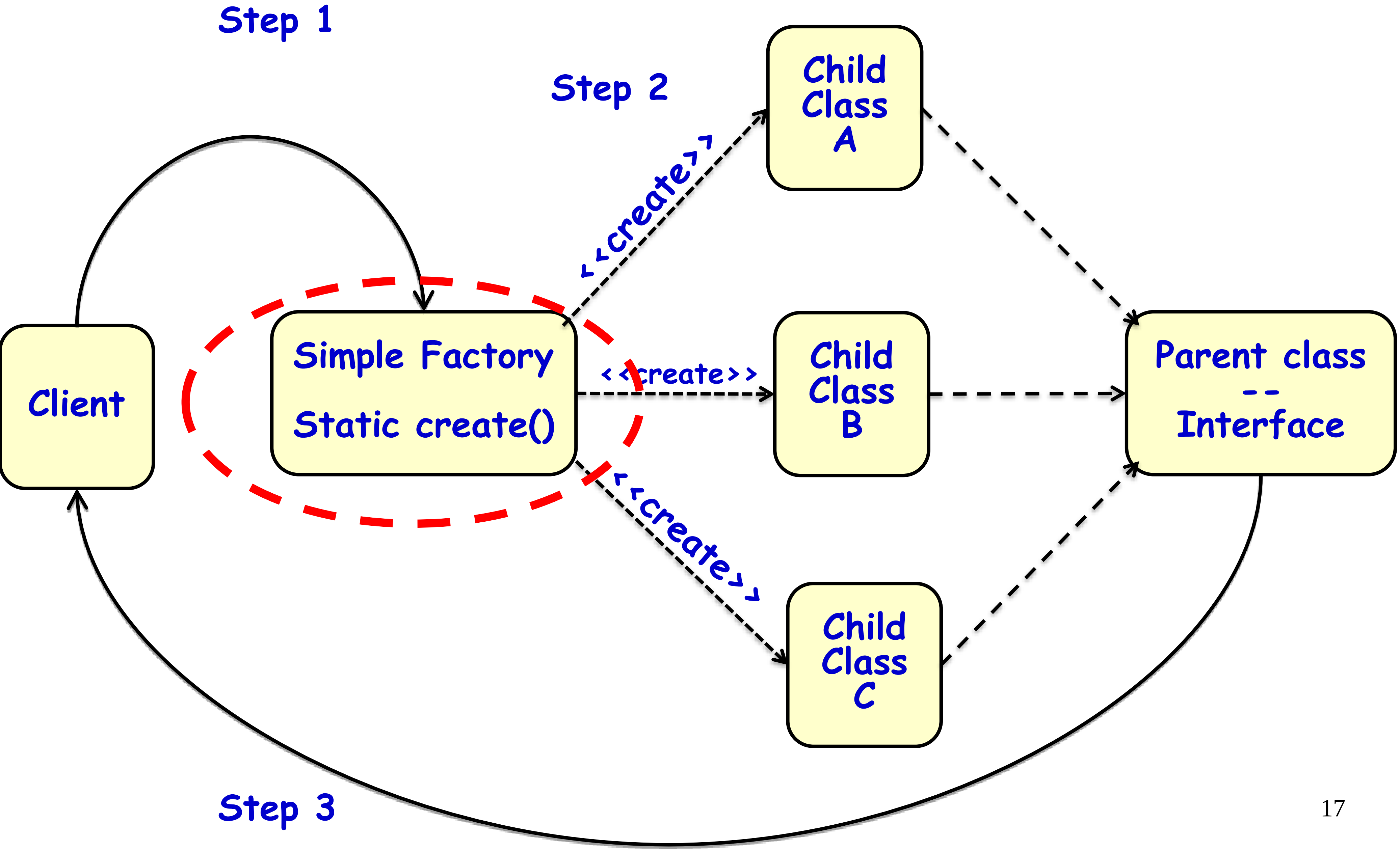
- Creation of buttons etc. by an application object:
  - Avoids significant duplication of code.
  - Makes the client class work at a suitable level of abstraction as these may not be part of the composing object's concerns.

```
public class SimplePizzaFactory {
   public static Pizza createPizza(String type) {
      Pizza pizza;
      if (type.equals(“cheese”)) {
         pizza = new CheesePizza();
      } else if (type.equals(“pepperoni”)) {
         pizza = new PepperoniPizza();
      } else if (type.equals(“sausage”)) {
         pizza = new SausagePizza();
      } else if (type.equals(“veggie”)) {
         pizza = new VeggiePizza();   }
      return pizza;
   }
}
```

Draw Class Diagram for Simple Factory

Simple Factory

Shortcomings?



Step 1

Step 2

Step 3

Simple Factory: Working

- **Step One:**
  - Call the static create method of factory.
  - The parameters tell the factory which class to create.
- **Step Two:**
  - The factory creates required object.
  - Note that the objects have the same parent class, or implement the same interface.
- **Step Three:**
  - Factory returns the object.

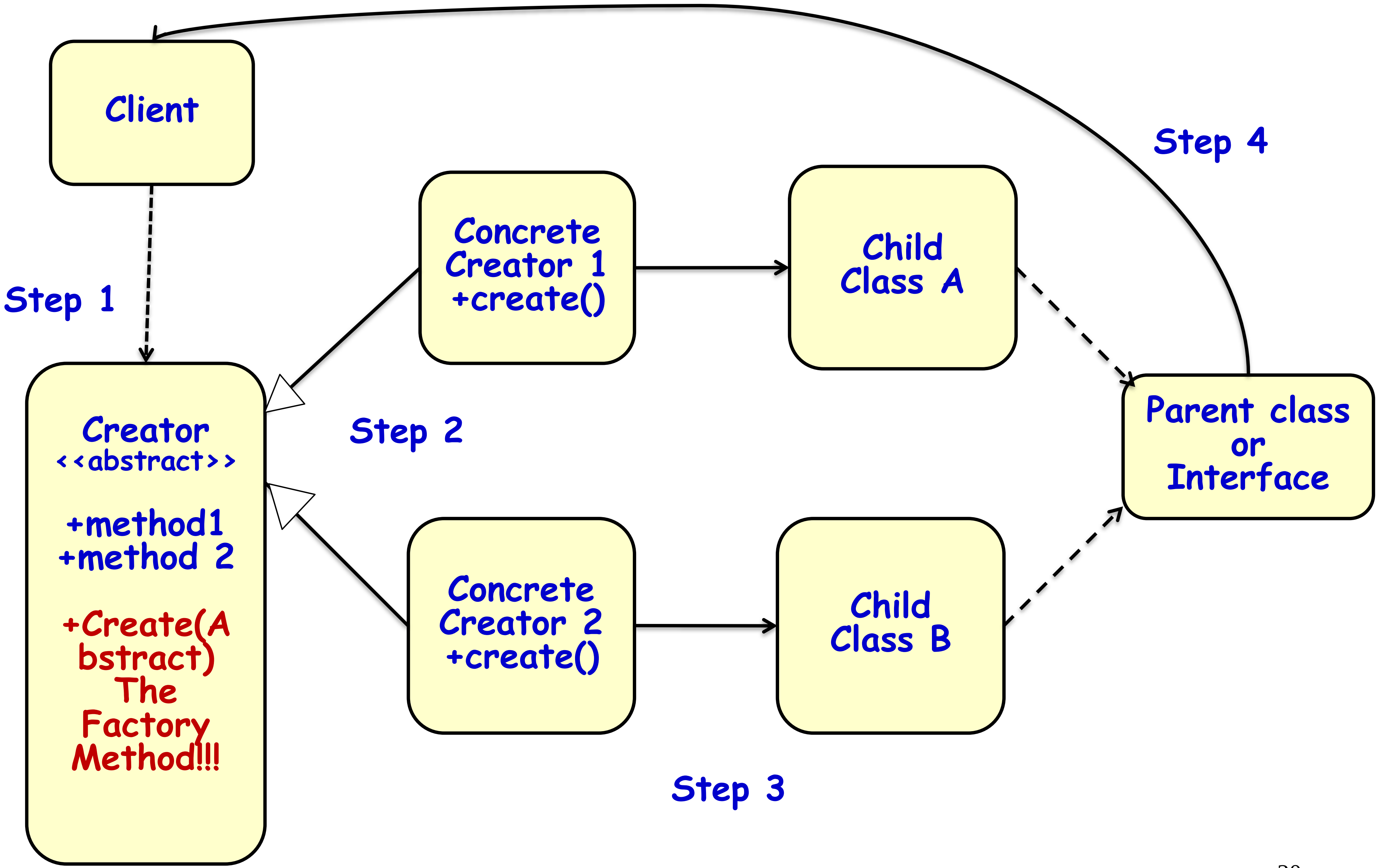
Problems with Simple Factory

- Simple factory makes the application unaffected by changes

- **But the factory itself needs to be changed each time a new class needs to be instantiated…**

- Solution: **Factory method:**
  - **Subclass the factory!**

Need a Factory Method!



Client

Step 1

Creator
<<abstract>>

+method1
+method 2

+Create(Abstract)
The Factory Method!!!

Concrete Creator 1
+create()

Concrete Creator 2
+create()

Step 2

Child Class A

Child Class B

Step 3

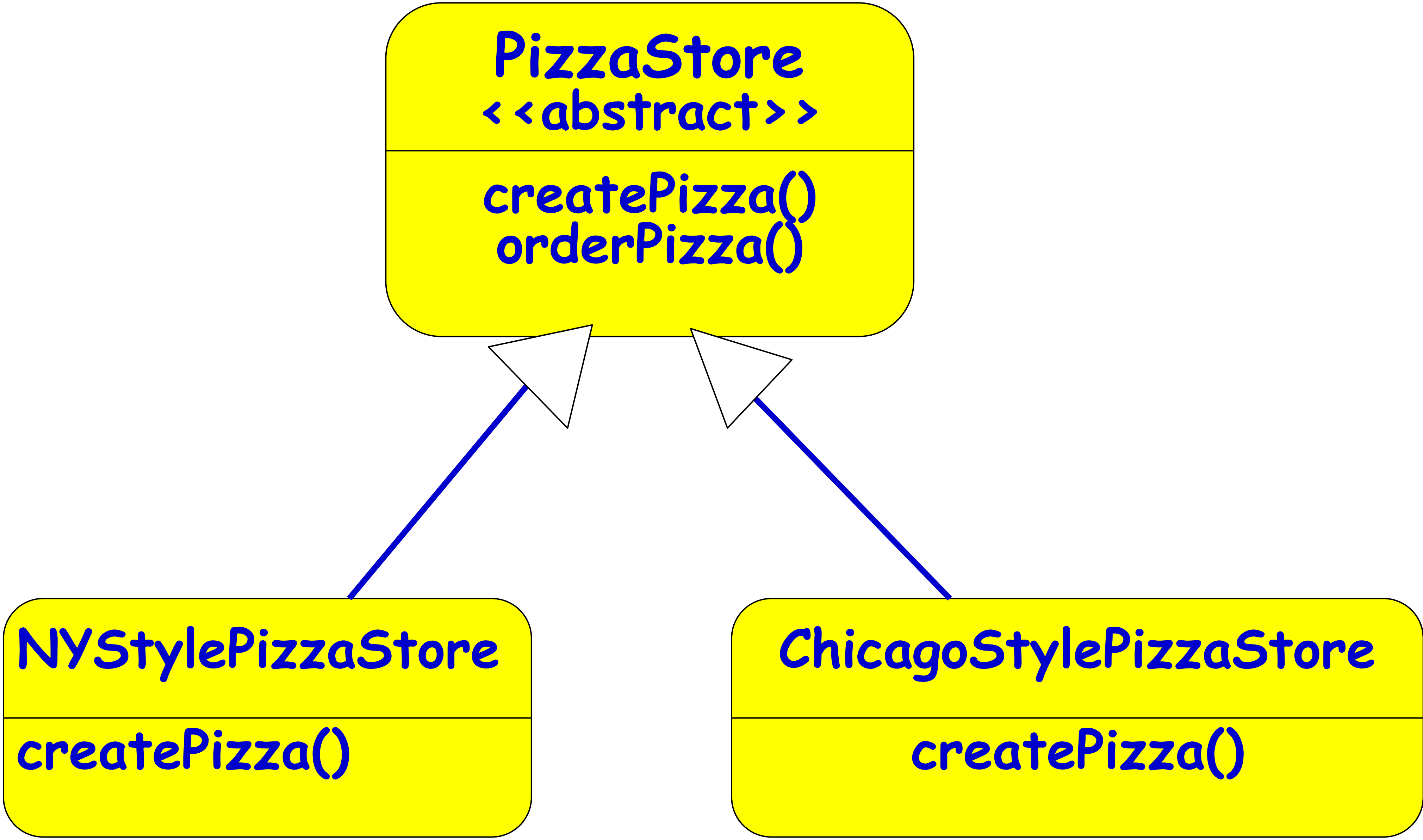
Parent class or Interface

Step 4

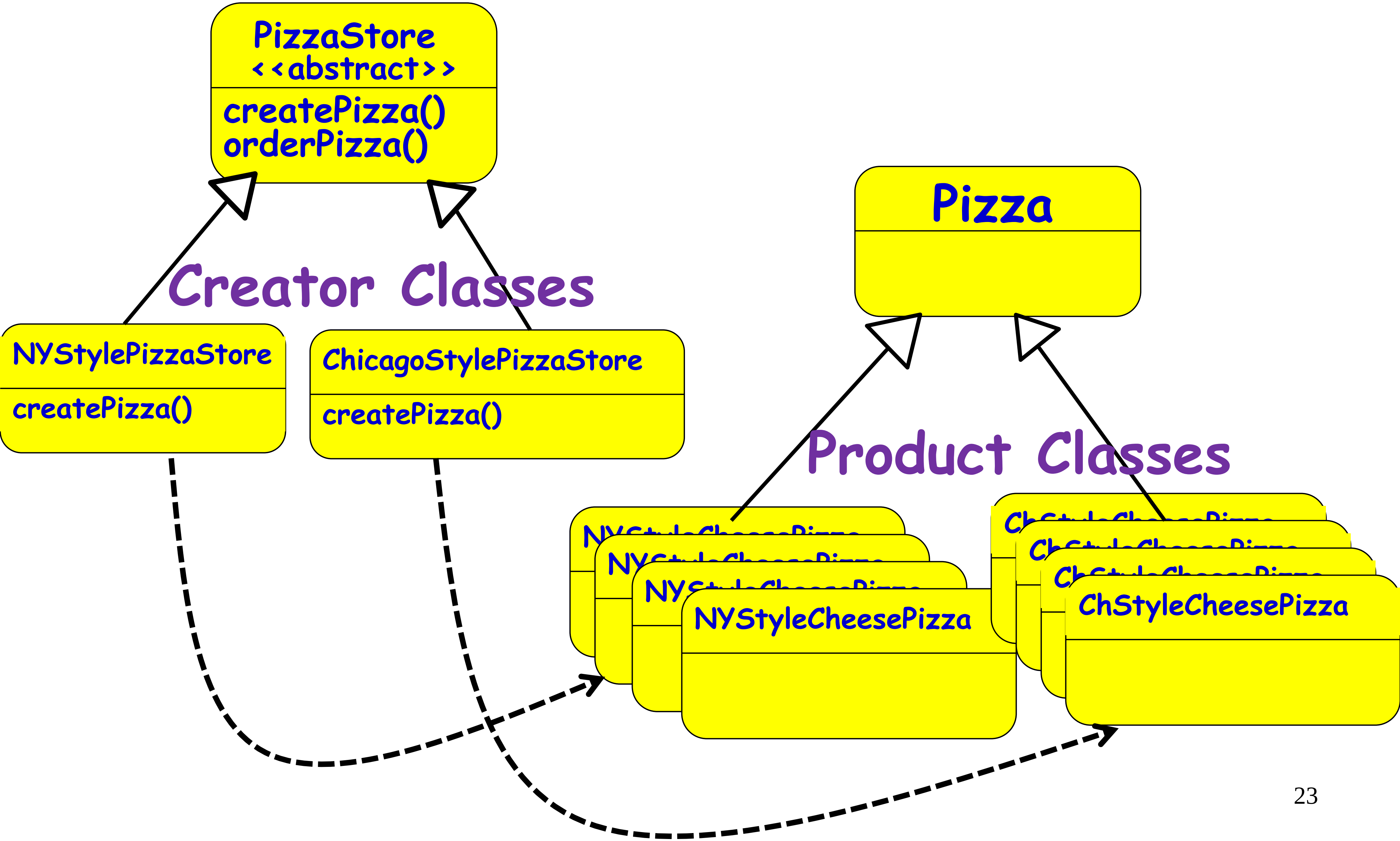
Factory Method

- **Step One:**
  - The client maintains a reference to the abstract Creator, but instantiates it with one of the subclasses. **(i.e. Creator c = new ConcreteCreator1();)**
- **Step Two:**
  - The Creator has an abstract method for creation of an object, which we'll call “create".
  - All child classes must implement “create”.
- **Step Three:**
  - The concrete creator creates the concrete object.
- **Step Four:**
  - The concrete object is returned to the client.

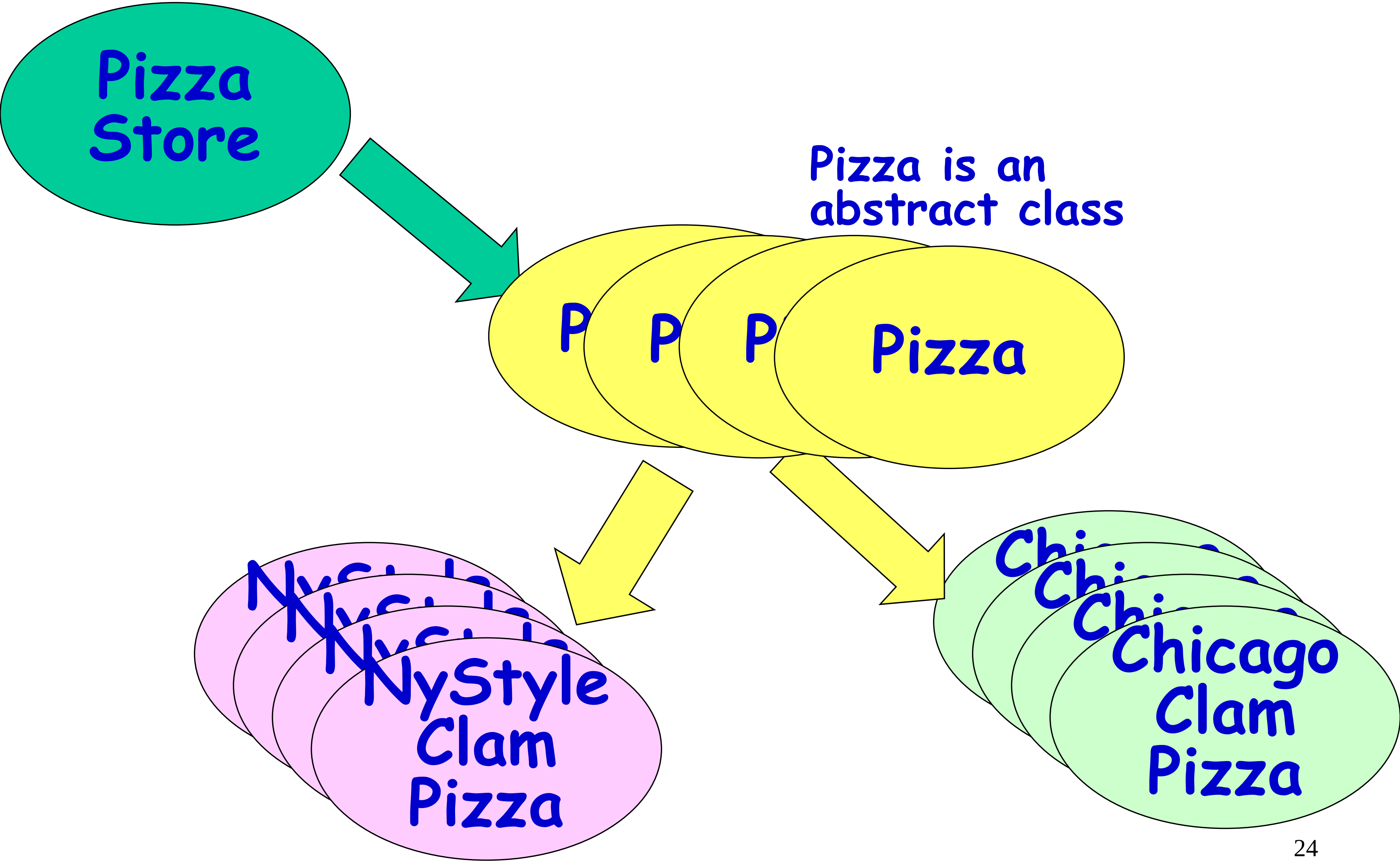
Factory Method: Example



Factory Method: Example



Example: Pizza Store

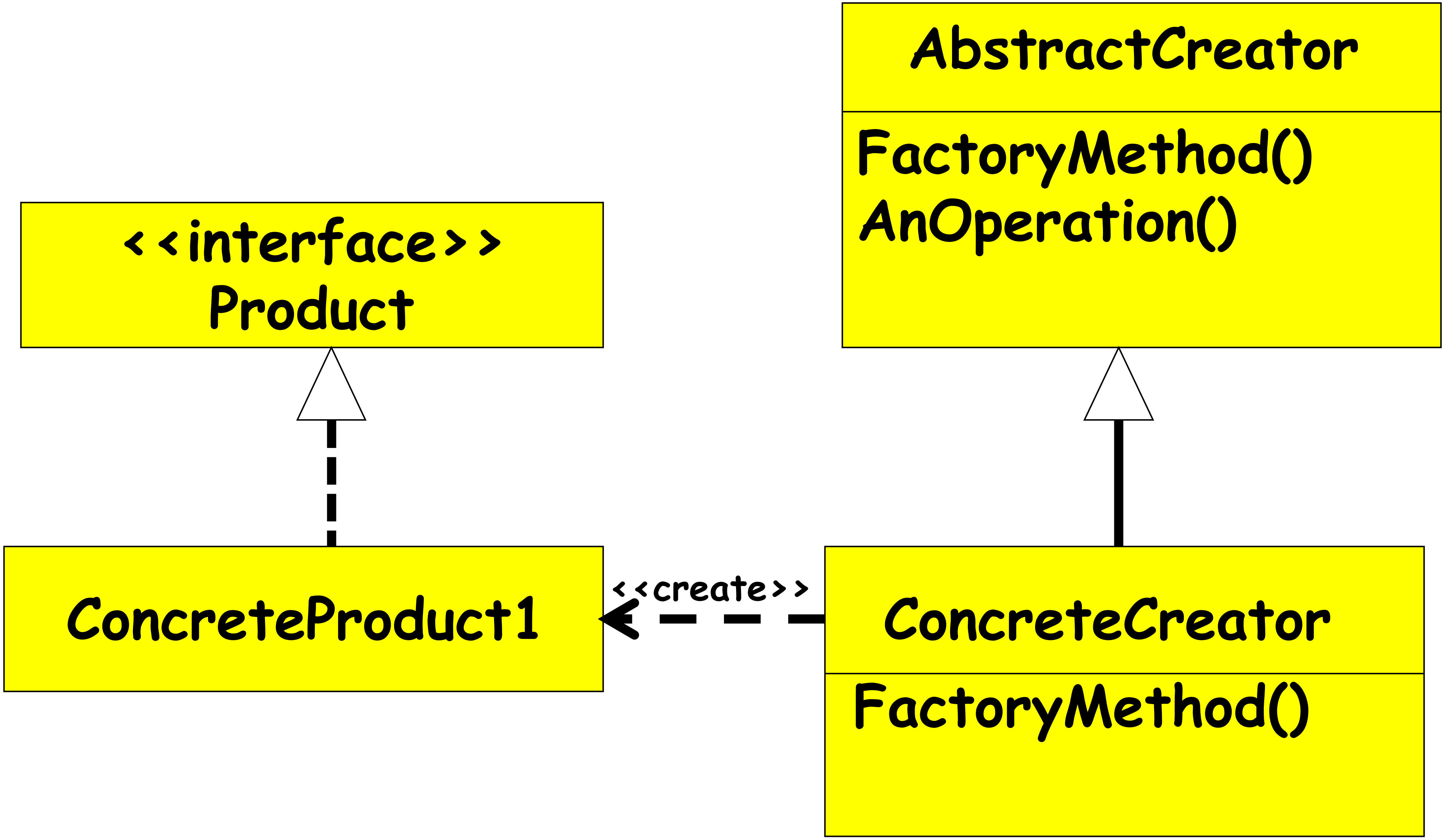


Pizza is an abstract class

Factory Method Pattern Defined

- The factory method pattern defines an abstract class or interface for creating an object:
  - But leaves it to subclasses regarding which class to instantiate.
  - Factory method lets a class defer instantiation to subclass.

Factory Method: Class Structure



AbstractCreator

FactoryMethod()
AnOperation()

<<interface>>
Product

ConcreteProduct1

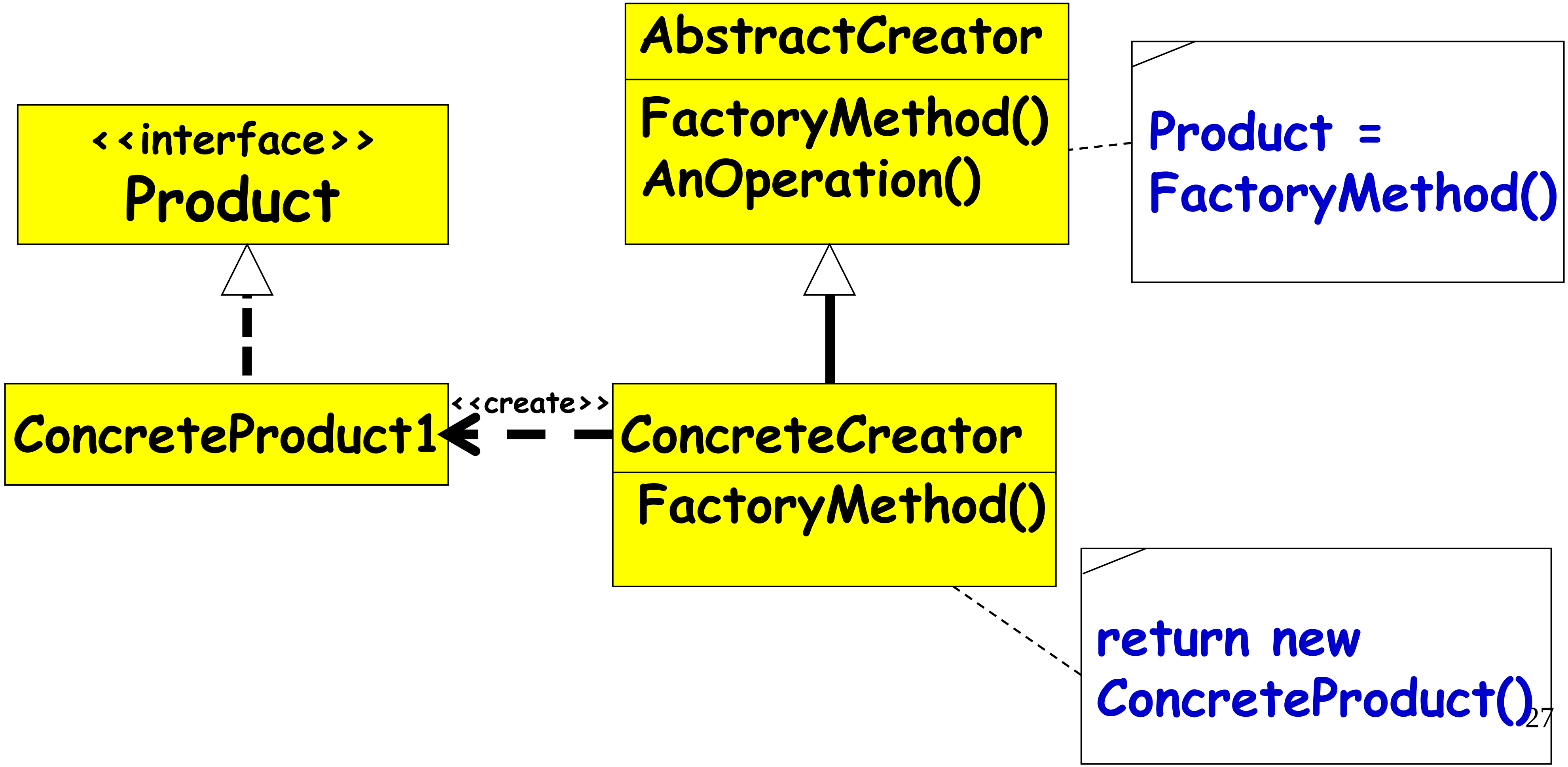
<<create>>

ConcreteCreator

FactoryMethod()

Factory Method Pattern

- Define an interface for creating an object,
  - but lets subclasses to instantiate.

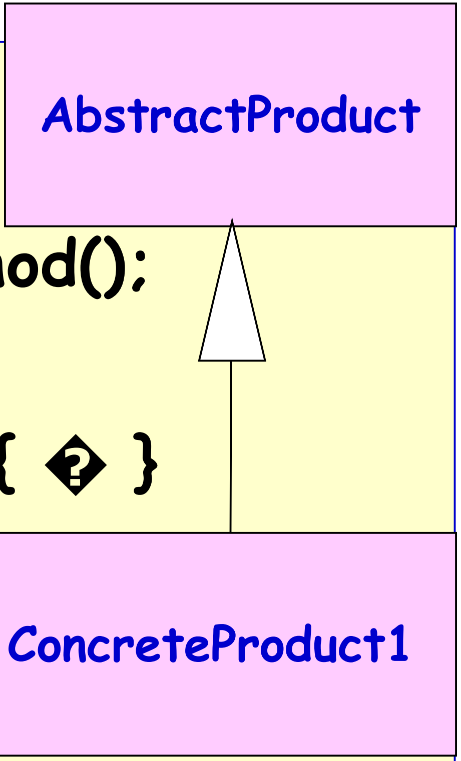


Participants

- **Product:** defines the interface for the factory method to create objects.
- **ConcreteProduct:** implements the Product interface.
- **Creator**(aka **Factory**): is an abstract class
  - Declares the method **FactoryMethod**, which returns a Product object.
  - Calls the generating method for creating Product objects .
- **ConcreteCreator:** overrides the generating method for creating **ConcreteProduct** objects

```
public interface Product { � }
public abstract class Creator {
        protected abstract  Product factoryMethod();
}
public class ConcreteProduct implements Product { � }

public class ConcreteCreator1 extends Creator {
         protected Product factoryMethod() {
                     return new ConcreteProduct1(); } }
public class Client {
          public static void main( String arg[] ) {
          Creator c = new ConcreteCreator1();
          Product p= c.factoryMethod();
     }
}
```



Factory Method Generic Code

Factory Method: Example 1

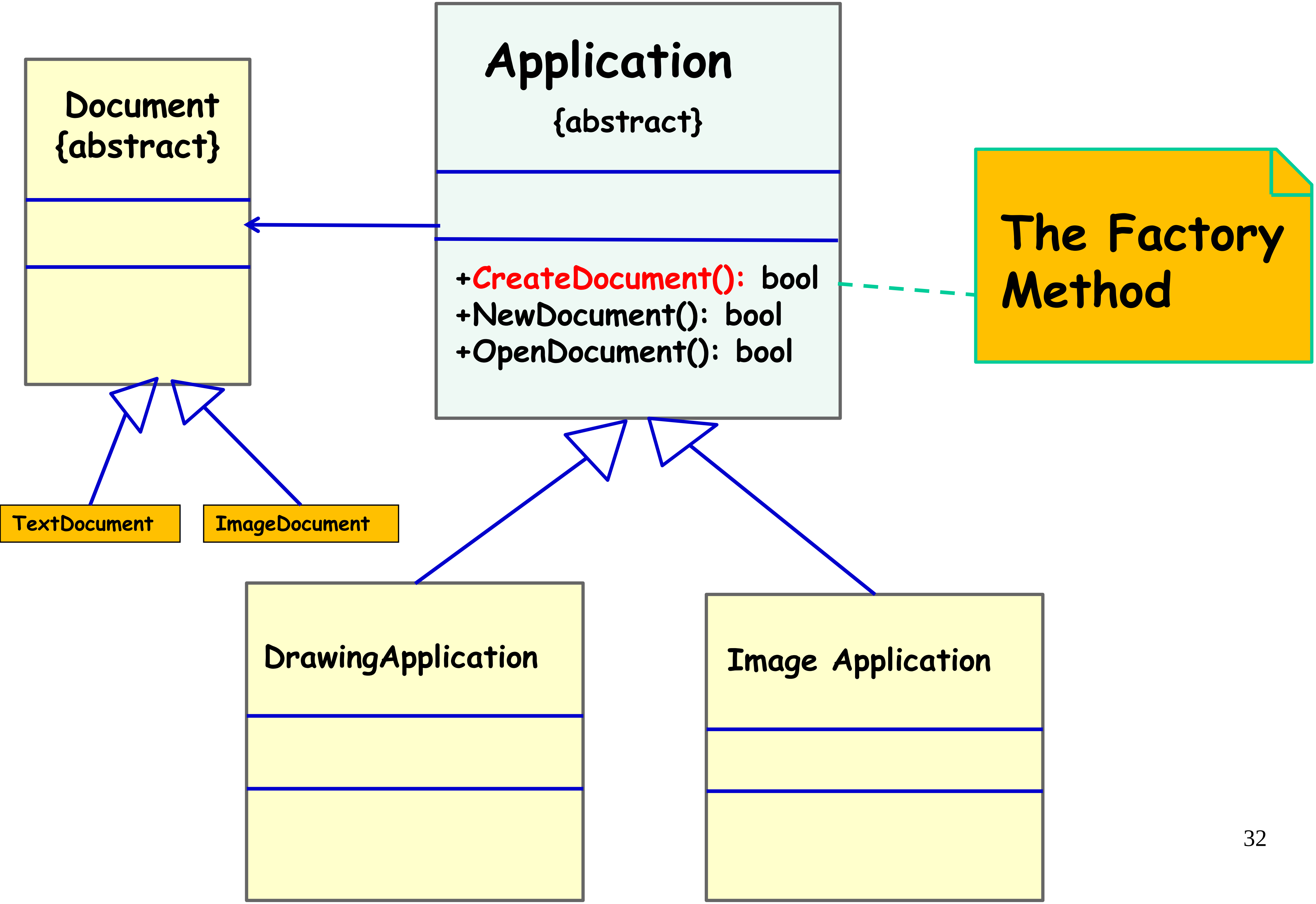
- We need to create an application that can read and display multiple types of documents.

- Two key abstractions:
  - **Application**
    - Create and Manage Document
  - **Document**
    - Specific type of documents

Factory Method: Example 1

- We want to support a wide variety of applications:
  - Text editors
  - Video processors
  - Vector drawing applications
  - Image Viewers
- Our application be able to manage the documents.

Document Presenter Application Example



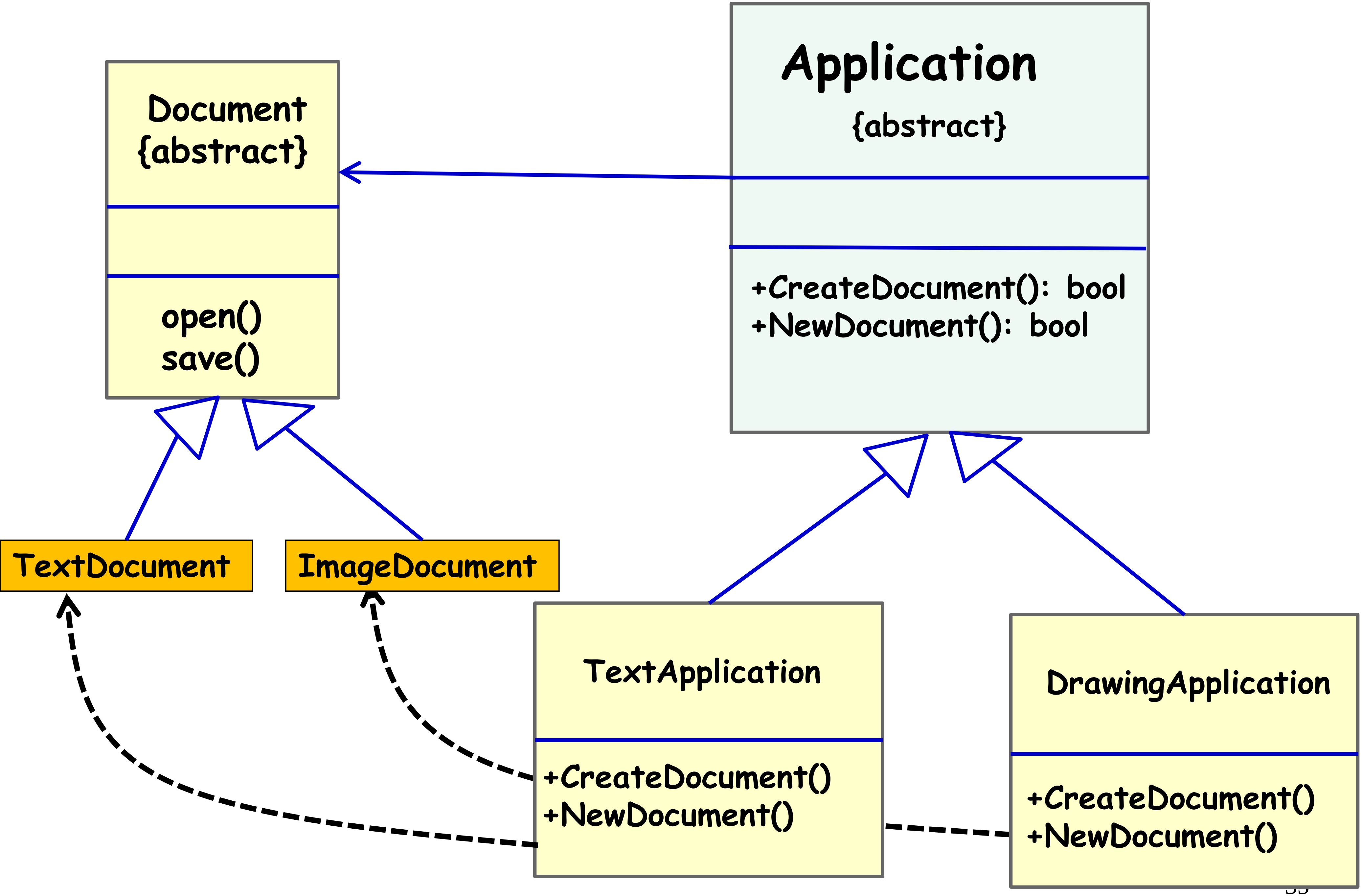
```
public abstract class Document {
   public abstract void open();
   public abstract void close(); }

public abstract class Application {
   private List docs = new ArrayList();
   public void newDocument() {
       Document doc = createDocument();
       docs.add(doc);
       doc.open(); }
... public abstract Document createDocument();
                                          // factory method
}
```

```
public class TextDocument extends Document {
  … // implementation of the abstract methods
}


public class TextApp extends Application{
   public Document createDocument() {
      return new TextDocument();
     }
}
```

Document Presenter Application Example



Document
{abstract}

open()
save()

TextDocument

ImageDocument

Application
{abstract}

+CreateDocument(): bool
+NewDocument(): bool

TextApplication

+CreateDocument()
+NewDocument()

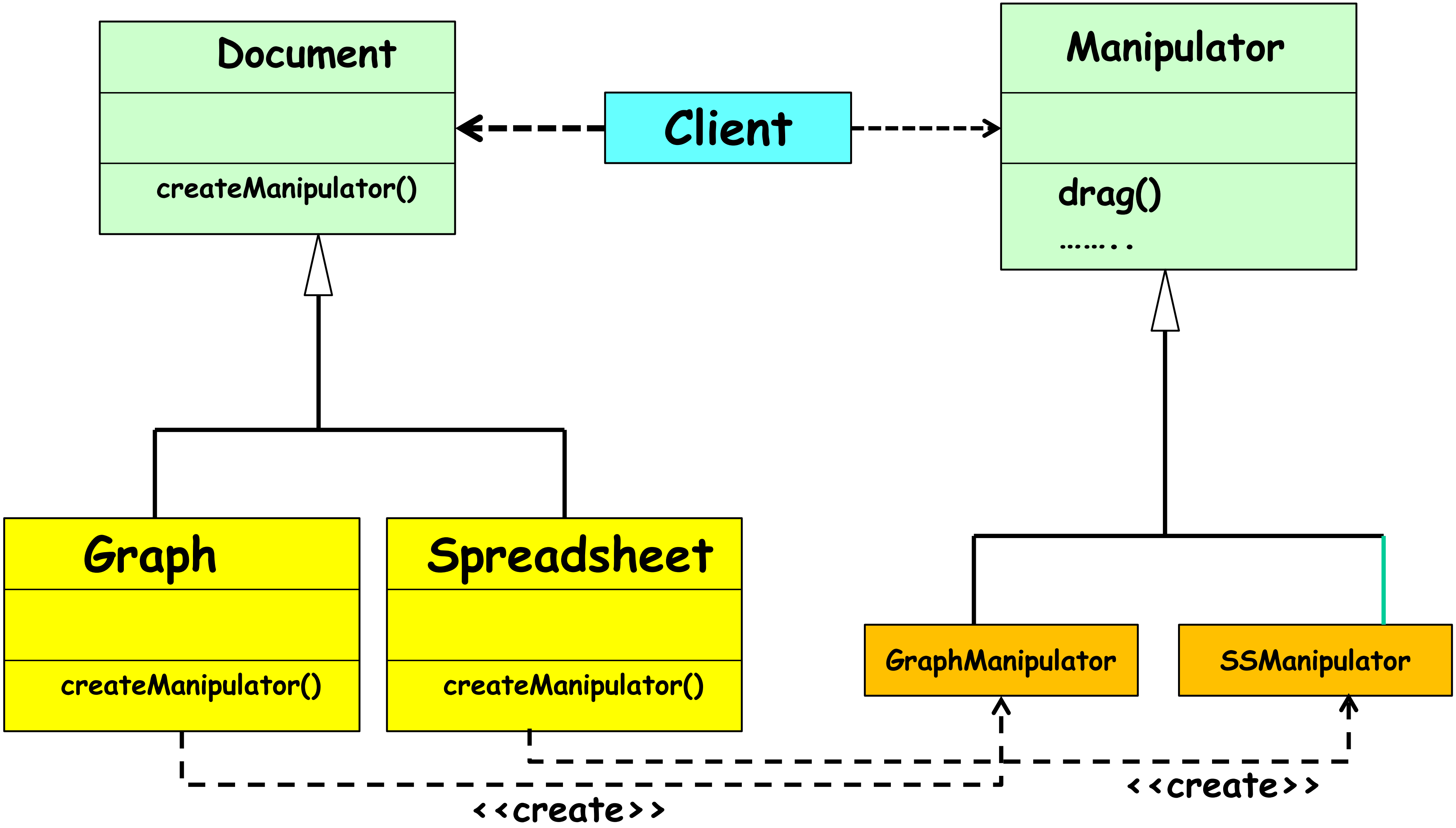
DrawingApplication

+CreateDocument()
+NewDocument()

Exercise

- A Composite document contains several types of documents such as graphics, Spreadsheet, CAD application, CASE tool application, etc.
  - Double clicking on a specific type of document should bring up the corresponding Editor (manipulator)

- Give the class diagram of your solution.


Document
createManipulator()
Client
Manipulator
drag()
……..
Graph
createManipulator()
Spreadsheet
createManipulator()
GraphManipulator
SSManipulator
<<create>>
<<create>>

Factory Method: Applicability

You should consider using a Factory method pattern when:

- Not possible anticipate which kind of objects must be created.
- Choice may depend on:
  - The state of the running application.
  - User input.
  - Changes or enhancements.
- The objects to be created are instances of classes that form a hierarchy.

Advantages of Factory Method Pattern

- Separates responsibility of complex creation into cohesive helper classes
  - Hides complex creation logic, such as initialization from a file
  - Create classes of hierarchy of objects as required
- The client of Creator can ask for the production of different Products in a uniform way:
  - And use them uniformly
  - Without knowing the nitty-gritty details

Factory Pattern: Pros and Cons

- Factory pattern introduces separation between the application and a family of classes:
  - It removes tight coupling by hiding concrete classes from the application.

- It provides a simple way to extend the family of products with minor changes in application code.

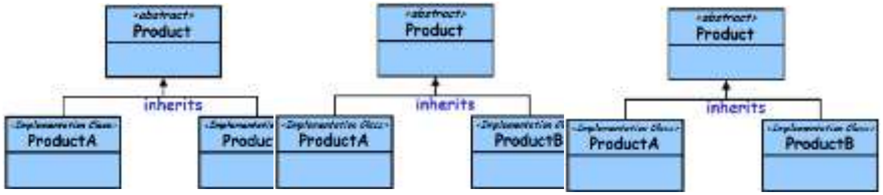
- When the objects are created directly inside the class:
  - It's hard to replace them by objects which extend their functionality.
  - **When a factory is used one can easily replace the original objects, configuring the factory to create them.**

Known Uses

- It is a pervasive pattern.
- It is used in several places in the Java API.
  - For example, **URLConnection** class has a method **getContent** that returns the content as an appropriate object (html, gif etc.)
- In .Net Framework Class Library, the Factory method is used in:
  - Systems.Collections.IEnumerable,
  - System.Net.WebRequest
  - System.Security.Cryptography

Abstract Factory

- Provide an interface for creating families of related or dependent objects:
  - **Without specifying their concrete classes**



Abstract Factory Pattern

- The Abstract Factory pattern:
  - Works at a higher level of abstraction than the factory pattern.
  - Abstract Factory returns one of several factoryobjects.
  - Each of which can create and return several different types of objects on request.

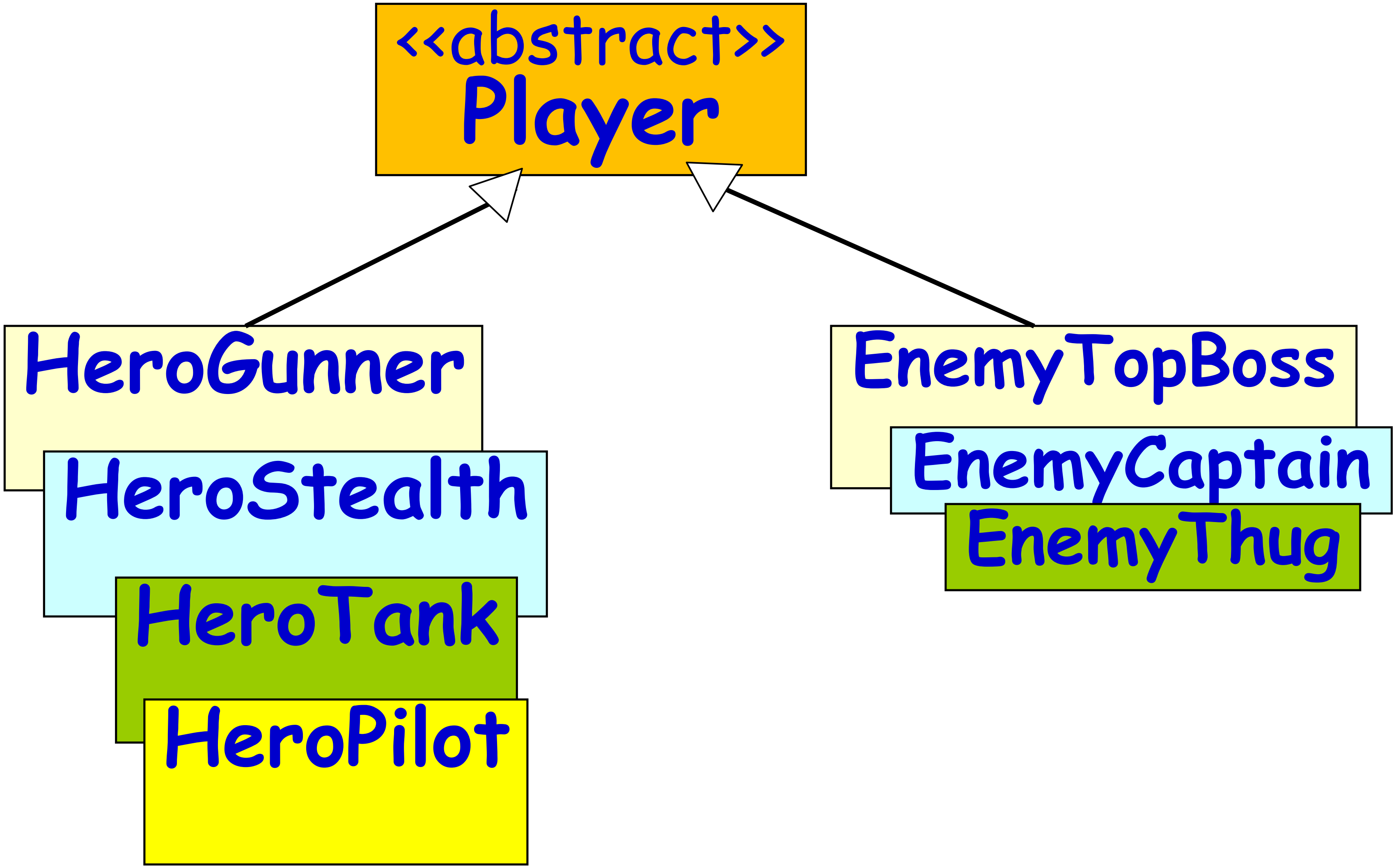
Abstract Factory Analogy

- You want the capability of making different products in the same production plant:
  - Simply by pushing a switch

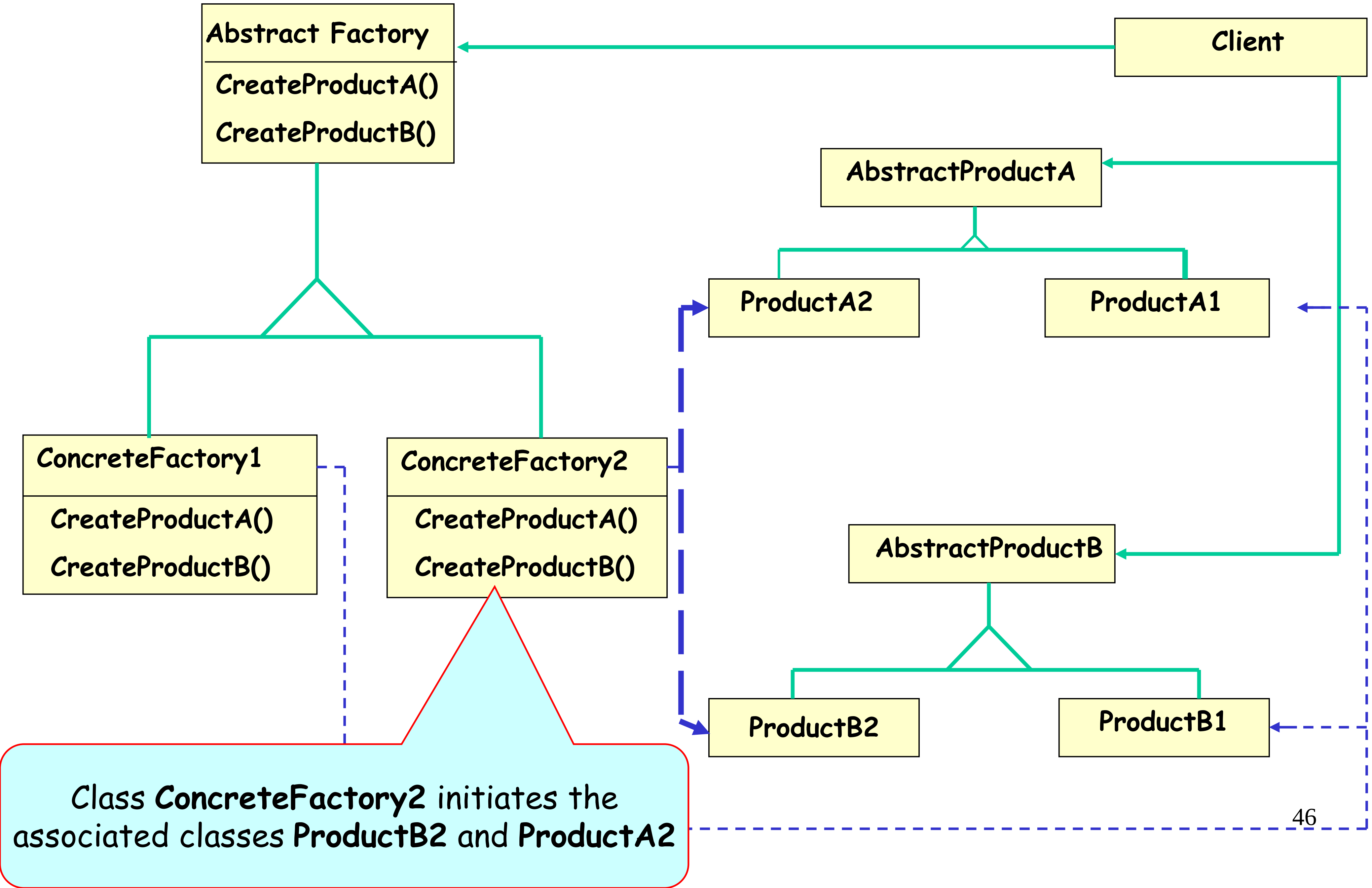
- The production procedure followed by the factory is the same :
  - Independent from the product being produced
  - **The switch controls what machinery is activated during the production process**

- Result: Different final products

Motivating Example



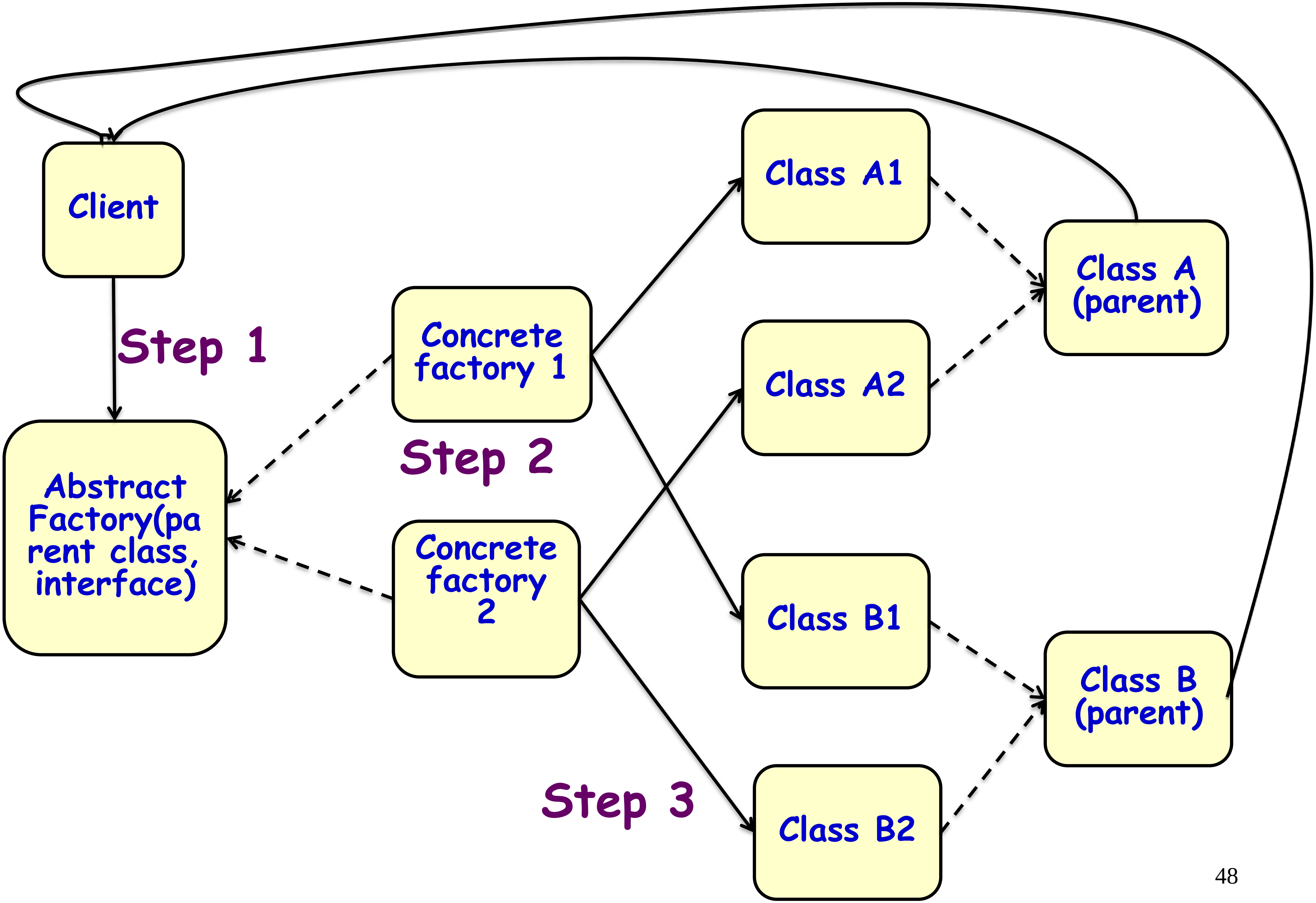
Abstract Factory Structure



Class **ConcreteFactory2** initiates the associated classes **ProductB2** and **ProductA2**

Abstract Factory Participants

- **AbtractFactory**
  - Declares interface for operations to create abstract product objects
- **ConcreteFactory**
  - Implements operations to create concrete product objects
- **AbstractProduct**
  - Declares an interface for a type of product object
- **ConcreteProduct**
  - Defines a product object to be created by concrete factory
  - Implements the abstract product interface
- **Client**
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes


Client
Step 1
Abstract Factory(parent class, interface)
Concrete factory 1
Step 2
Concrete factory 2
Step 3
Class A1
Class A2
Class B1
Class B2
Class A (parent)
Class B (parent)

- **Step One:**
  - The client maintains a reference to an abstract Factory class, which all Factories must implement.
  - The abstract Factory is instantiated with a concrete factory.
- **Step Two:**
  - the factory is capable of producing multiple types. This is where the “family of related products” comes into play.
  - The objects which can be created still have a parent class or interface that the client knows about, but the key point is there is more than one type of parent.
- **Step Three:**
  - The concrete factory creates the concrete objects.
- **Step Four:**
  - The concrete objects are returned to the client.

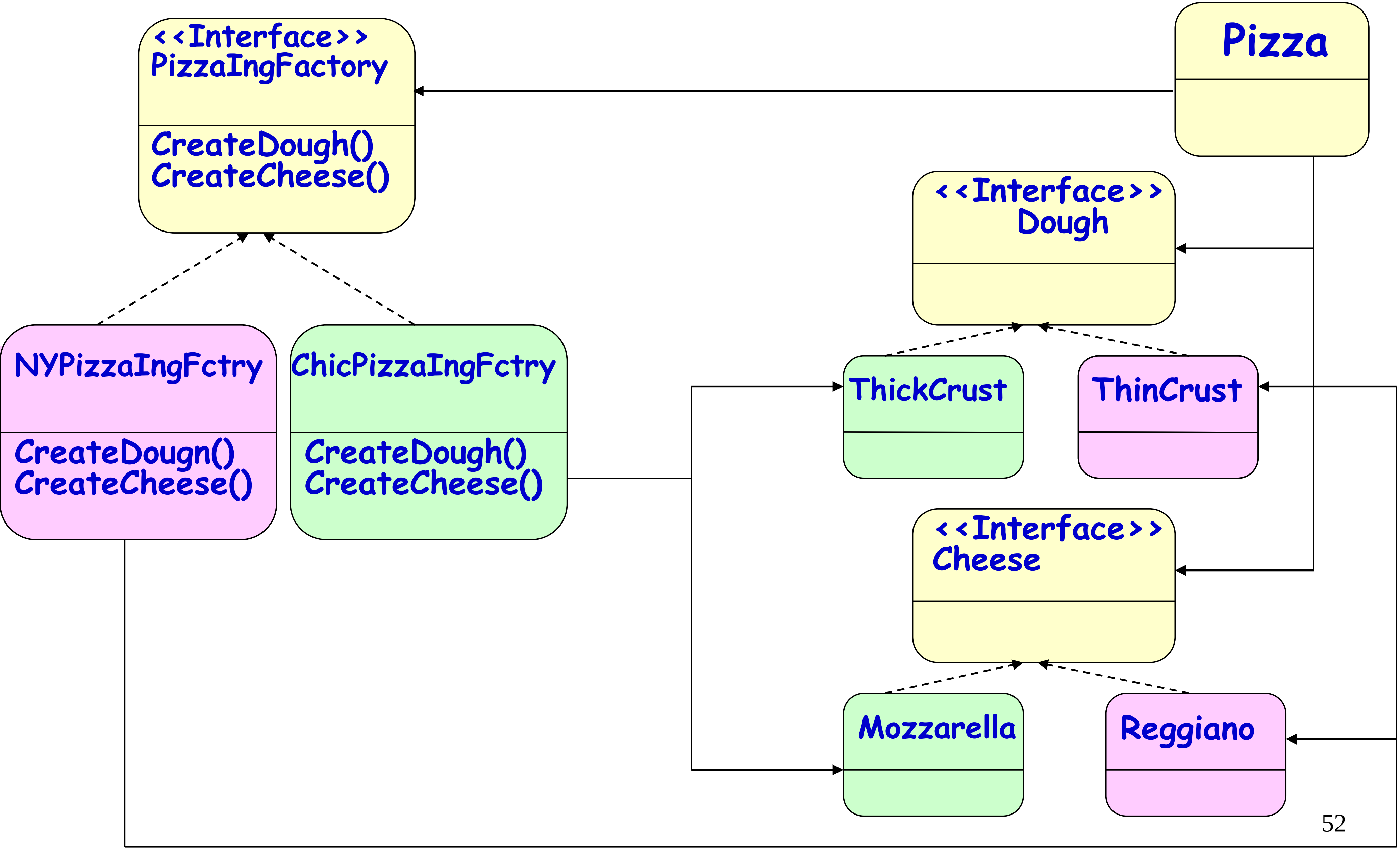
Exercise 1

- Extending the Pizza store example…
- **How do we deal with families of ingredients?**
  - Chicago: FrozenClams, PlumTomatoSauce, ThickCrustDough, MozzarellaCheese
  - New York: FreshClams, MarinaroSauce, ThinCrustDough, ReggianoCheese
  - California: Calamari, BruuuschettaSauce, VeryThinCrust, GoatCheese

Abstract Factory

```
public interface PizzaIngredientFactory {
  public Dough createDough();
   public Sauce createSauce();
   public Cheese createCheese();
   public Veggies[] createVeggies();
   public Pepperoni createPepperoni();
   public Clams createClam();
}
```

Abstract Factory Pattern example



Building NY ingredient factory

```
public class
 NYPizzaIngredientFactory
 implements
 PizzaIngredientFactory {
 public Dough createDough() {
 return new ThinCrustDough();
 }
 public Sauce createSauce() {
 return new MarinaraSauce();
 }
 public Cheese createCheese() {
 return new ReggianoCheese();
 }
```

```
public Veggies[] createVeggies() {
Veggies veggies[] = { new
Garlic(), new Onion(), new
Mushroom(), new RedPepper() };
return veggies;
}
public Pepperoni createPepperoni()
{
return new SlicedPepperoni();
}
public Clams createClam() {
return new FreshClams();
}
}
```

Applicability

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented
- A system should be configured with one of multiple families of produces
- A family of related product objects is designed to be used together, and you need to enforce this constraint
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Exercise 2

- Suppose you are writing a program to plan the layout of gardens.
- These could be annual gardens, vegetable gardens or perennial gardens.
- No matter which kind of garden you are planning, you want to ask the same questions:
  - What are good border plants?
  - What are good center plants?
  - What plants do well in partial shade?

We want a base *Garden* class that can answer these questions:

```
public abstract class Garden {
    public abstract Plant getCenter();
    public abstract Plant getBorder();
    public abstract Plant getShade();
}
```

Abstract Factory Pattern

Plant class simply contains and returns the plant name:

```
public class Plant {
      String name;
      public Plant(String pname) {
       name = pname; //save name
      }
      public String getName() {
       return name;
  }
}
```

Abstract Factory Pattern

A Garden class simply returns one kind of each plant. For the vegetable garden :

```
public class VegieGarden extends Garden {
    public Plant getShade() {
    return new Plant("Broccoli");
    }
    public Plant getCenter() {
    return new Plant("Corn");
    }
    public Plant getBorder() {
    return new Plant("Peas");
   }
}
```

Abstract Factory Pattern

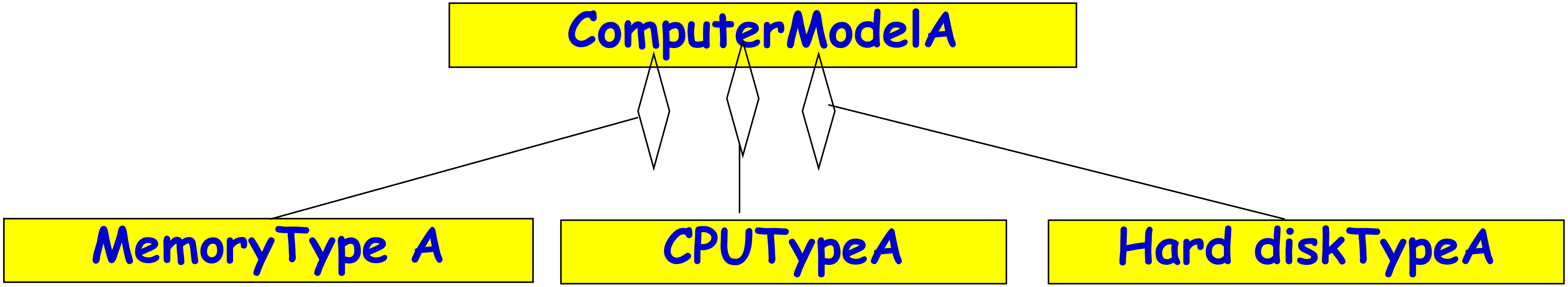
Next, we construct our **abstract factory** to return an object instantiated from one of these Garden classes and based on the string it is given as an argument:

```
class GardenMaker { //Abstract Factory
    private Garden gd;
    public Garden getGarden(String gtype) {
        gd = new VegieGarden(); //default
        if(gtype.equals("Perennial"))
        gd = new PerennialGarden();
        if(gtype.equals("Annual"))
        gd = new AnnualGarden();
       return gd;
     }
}
```

Exercise 3

- Every Computer is madeup of RAM, CPU and hard disk.
- The actual memory, CPU, and hard disk that is used depends on the actual computer model being used.
  - Server, workstation, desktop
- We want to provide a configure function that will configure any computer with appropriate parts.

Elaboration



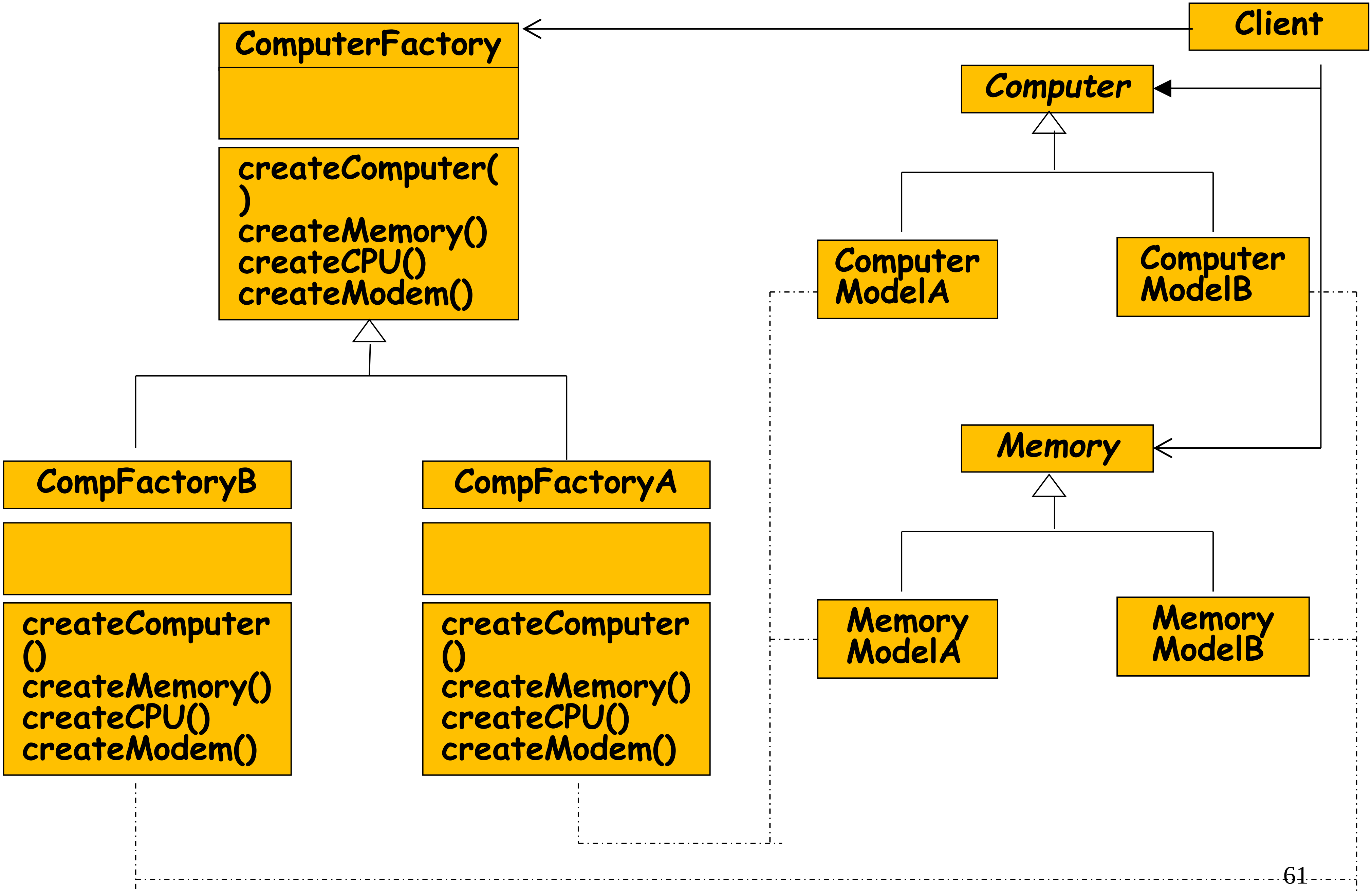
```
CreateComputer(ComputerModelA  comp){
      comp.Add(new MemoryTypeA);

      comp.Add(new CPUTypeA);

      comp.Add(new HDiskTypeA);

}
```

Exercise 2: Solution



```
public interface Computer {
  public Parts getHarddisk();
  public Parts getRAM();
  public Parts getProcessor();
}
```

```
public class PC extendss
  Computer {
public Parts getRAM() {
return new Parts("256 MB");
}
public Parts getProcessor() {
return new Parts("Pentium3");
}
public Parts getHarddisk() {
return new Parts("40GB");
}
}
```

```
public class Workstation
   extends Computer {
public Parts getRAM() {
return new Parts("1 GB");
}
public Parts getProcessor() {
return new
   Parts("Pentium4");
}
public Parts getHarddisk() {
return new Parts("80GB");
}
}
```

```
public class Server extends
   Computer{
public Parts getRAM() {
return new Parts("2 GB");
}
public Parts getProcessor() {
return new Parts("DualCore");
}
public Parts getHarddisk() {
return new Parts("160GB");
}
}
```

```
public Computer getComputer(String catagoryType)
  {
if (catagoryType.equals("PC"))
comp = new PC();
else if(catogoryType.equals("Workstation"))
comp = new Workstation();
else if(catagoryType.equals("Server"))
comp = new Server();
return comp;
}
}
```

Applicability of Abstract Factory

- **Independence from Initialization or Representation:**
  - System should be independent of how its products are created, composed and represented
- **Manufacturer Independence:**
  - System should be configured with one of multiple families of products
- **Constraint that need to be enforced**
  - A family of related product objects must be used together
- **Cope with upcoming change:**
  - You are using one particular product family, but you expect that the underlying technology would change very soon, and new product should quickly appear in the market.

Consequences of Using Abstract Factory

- Isolates concrete classes
- Makes modifying products families easy
- Promotes consistency among products
  - Enforces, that products from one family are used together
- Supporting entirely new kinds of products is difficult:
  - AbstractFactory interface fixes the set of products that can be created
  - involves changing AbstractFactory and all its subclasses interfaces

Summary

- **Simple factory:**
  - Normally called by client via a static method
  - Returns one of several objects that all inherit/implement the same parent.
- **Factory Method**
  - A "create" method implemented by sub classes.
- **Abstract Factory:**
  - Returns a family of related objects to client.
  - It normally uses the Factory Method to create the objects.