GE Java Sessions

Creational Patterns: Factory Patterns

Outline

- Student presentation schedule
- Factory Method Pattern
- Abstract Factory Pattern

Describing Design Patterns

- Pattern name and classification
  - Scope (Class, Object)
  - Purpose (Creational, Structural, Behavioral)
- Intent
  - What does the design pattern do?
  - What particular design issue or problem does it address?
- Also Known As
  - Any other known name for the pattern

Describing Design Patterns

- Motivation
  - Scenario of a design problem
  - How the class and object structures in the pattern solve the problem
- Applicability
  - Situations in which the design pattern can be applied?
  - How to recognize these situations?
- Structure
  - Graphical representation of the classes in the pattern (diagrams etc.)

Describing Design Patterns

- Participants
  - Classes/Objects within the design pattern and their responsibilities.
- Collaborations
  - Participants collaboration to fulfill responsibilities.
- Consequences
  - graphical representation of the classes in the pattern (diagrams etc.)

Describing Design Patterns

- Implementation
  - Techniques required?
  - Language-specific issues?
- Sample Code
  - Code fragments for demonstration
- Known uses
  - Examples of this pattern found in real world systems
- Related Patterns
  - Examples of patterns related to this one, what are the differences?

Design Patterns in GoF: The Big Picture

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

What Are Creational Patterns?

- Creational Patterns
  - Abstract the instantiation process
  - Hide how objects are created
- Two categories
  - Object creational patterns
    - Focus on the delegation of the instantiation to another object (e.g., Abstract Factory)
  - Class creational patterns
    - Focus on the use of inheritance to decide the object to be instantiated (e.g., Factory Method)

Factory Method Pattern

Pattern name and classification

- Factory Method
- Scope : Class
- Purpose : Creational

Intent

- Defines an interface for creating an object, but let other system subclasses decide which class to instantiate.
- Factory Method lets a class have customized instantiation based on subclasses.

Also Known As

- Virtual Constructor

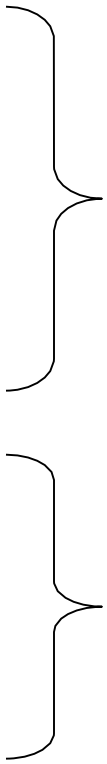
Motivation

```
Pizza orderPizza()
  {
     Pizza pizza = new Pizza ();

     Pizza.prepare()
     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 GrekPizza();
     } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
     }

     Pizza.prepare()
     Pizza.bake();
     Pizza.cut();
     Pizza.box();
     return pizza;
  }
```

Motivation

```
Pizza orderPizza (String type)
  {
     Pizza pizza;

     if (type.equals("cheese")){
        pizza = new CheesePizza();
     } else if (type.equals("greek") {
        pizza = new GrekPizza();
     } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
     } else if (type.equals(“clam”) {
        pizza = new ClamPizza ();
     }

     Pizza.prepare()
     Pizza.bake();
     Pizza.cut();
     Pizza.box();
     return pizza;
  }
```



This is what varies. As the pizza selection changes over time, you will have to modify this code over and over

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years. So we don’t expect this code to change

Motivation

```
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;
     }
}
```

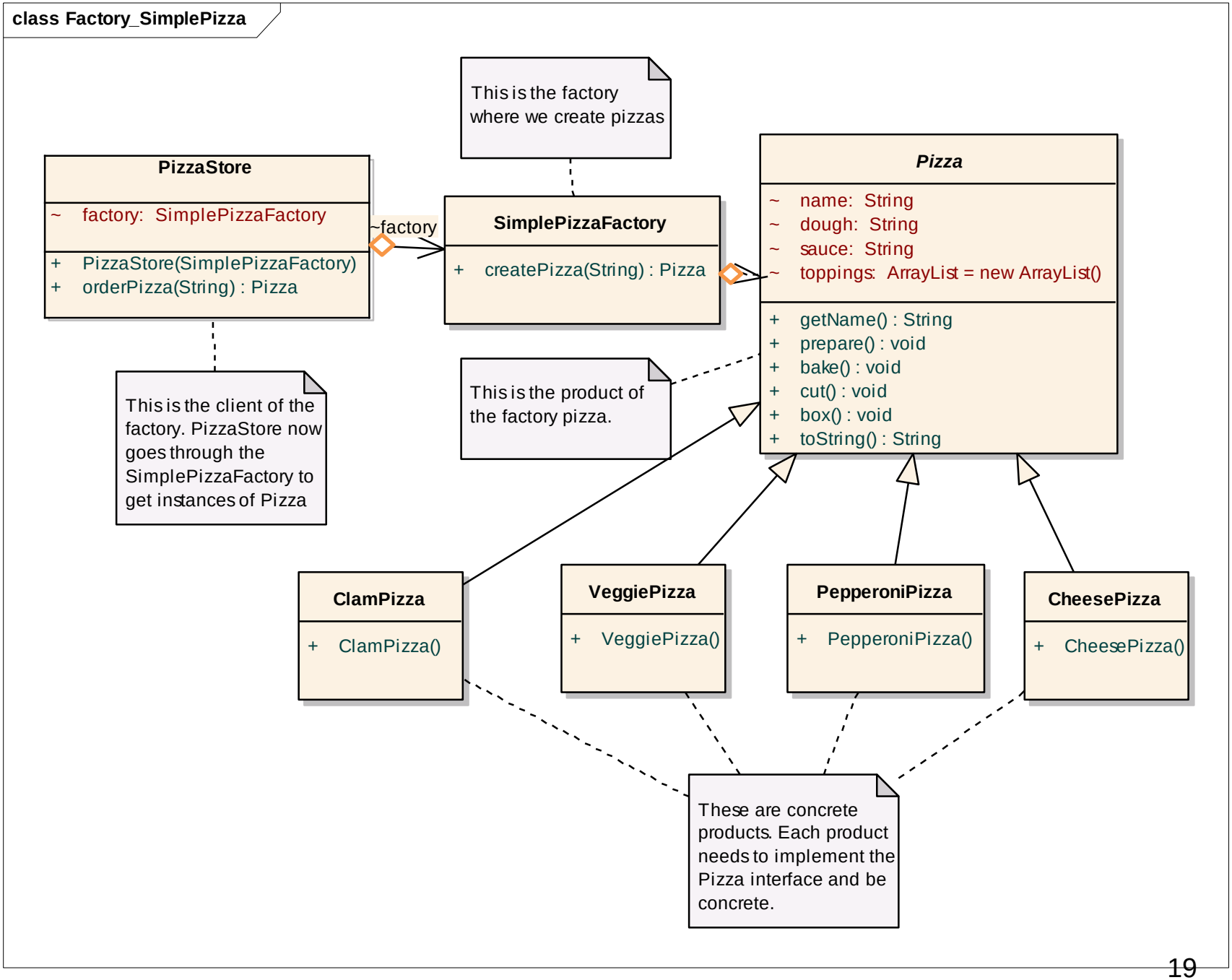
Motivation

```
public class PizzaStore
  {   // give PizzaStore a reference to a SimplePizzaFactory
      SimplePizzaFactory factory;

      // PizzaStore gets the factory passed to it in the constructor
      public PizzaStore(SimplePizzaFactory factory)
      {
          this.factory = factory;
      }

      public Pizza orderPizza(String type){
          Pizza pizza;

          //use factory to create its pizza by simply passing on the type of the order
          pizza = factory.createPizza(type);
          pizza.prepare();
          pizza.bake();
          pizza.cut();
          pizza.box();
          return pizza;
      }
  }
```

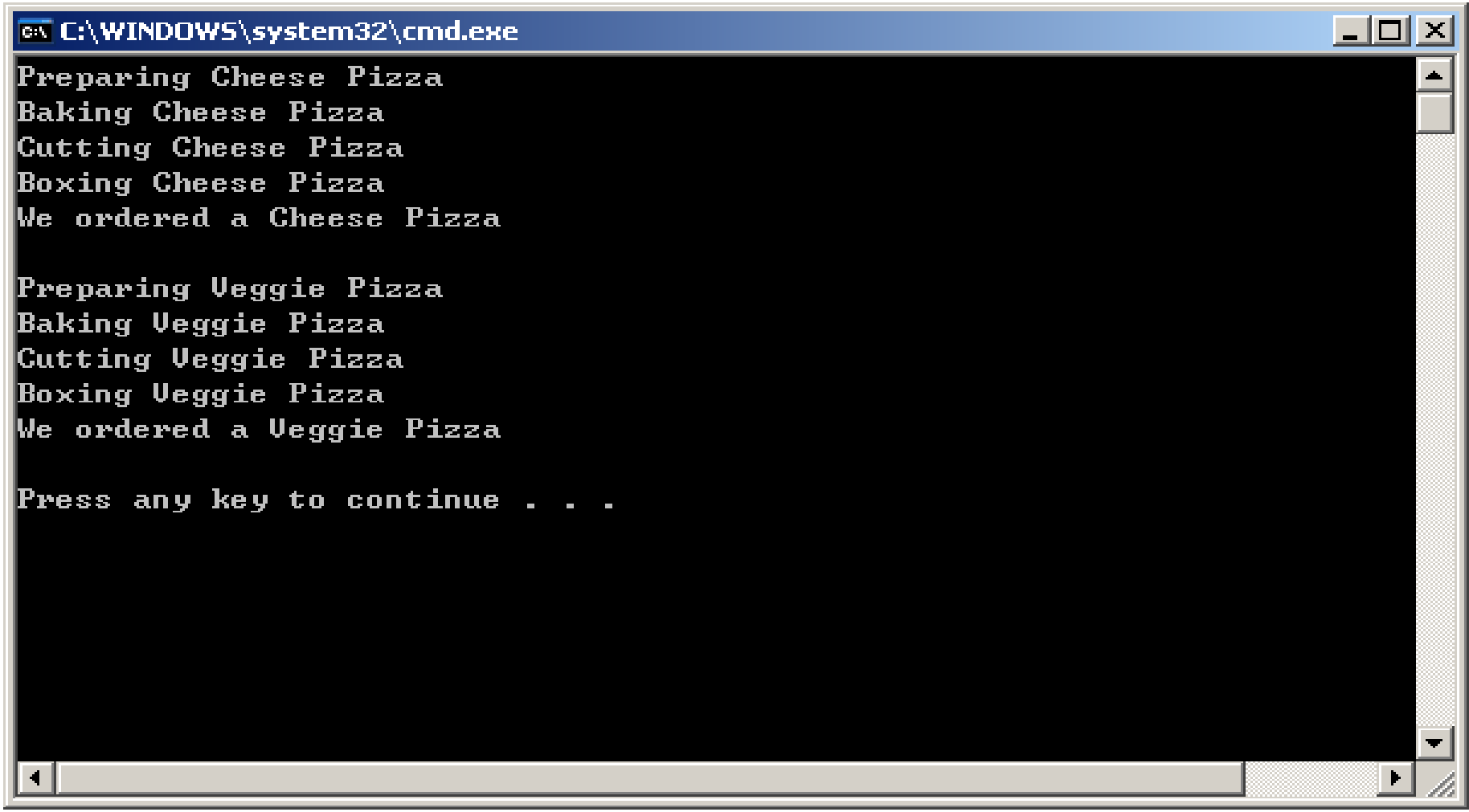
```
abstract public class Pizza {
	String name;
	String dough;
	String sauce;
	ArrayList toppings = new ArrayList();

	public String getName() {
		return name;
	}

	public void prepare() {
		System.out.println("Preparing " + name);
	}

	public void bake() {
		System.out.println("Baking " + name);
	}

	public void cut() {
		System.out.println("Cutting " + name);
	}

	public void box() {
		System.out.println("Boxing " + name);
	}
```

Main program in simple factory

```
public static void main(String[] args)
    {
        SimplePizzaFactory factory = new SimplePizzaFactory();
        PizzaStore store = new PizzaStore(factory);

        Pizza pizza = store.orderPizza("cheese");
        System.out.println("We ordered a " + pizza.getName() + "\n");

        pizza = store.orderPizza("veggie");
        System.out.println("We ordered a " + pizza.getName() + "\n");
        //
        // TODO: Add code to start application here
        //
    }
```


class Factory_SimplePizza
This is the factory where we create pizzas
PizzaStore
~ factory: SimplePizzaFactory
+ PizzaStore(SimplePizzaFactory)
+ orderPizza(String) : Pizza
~factory
SimplePizzaFactory
+ createPizza(String) : Pizza
Pizza
~ name: String
~ dough: String
~ sauce: String
~ toppings: ArrayList = new ArrayList()
+ getName() : String
+ prepare() : void
+ bake() : void
+ cut() : void
+ box() : void
+ toString() : String
This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of Pizza
This is the product of the factory pizza.
ClamPizza
+ ClamPizza()
VeggiePizza
+ VeggiePizza()
PepperoniPizza
+ PepperoniPizza()
CheesePizza
+ CheesePizza()
These are concrete products. Each product needs to implement the Pizza interface and be concrete.

Result window



Applicability

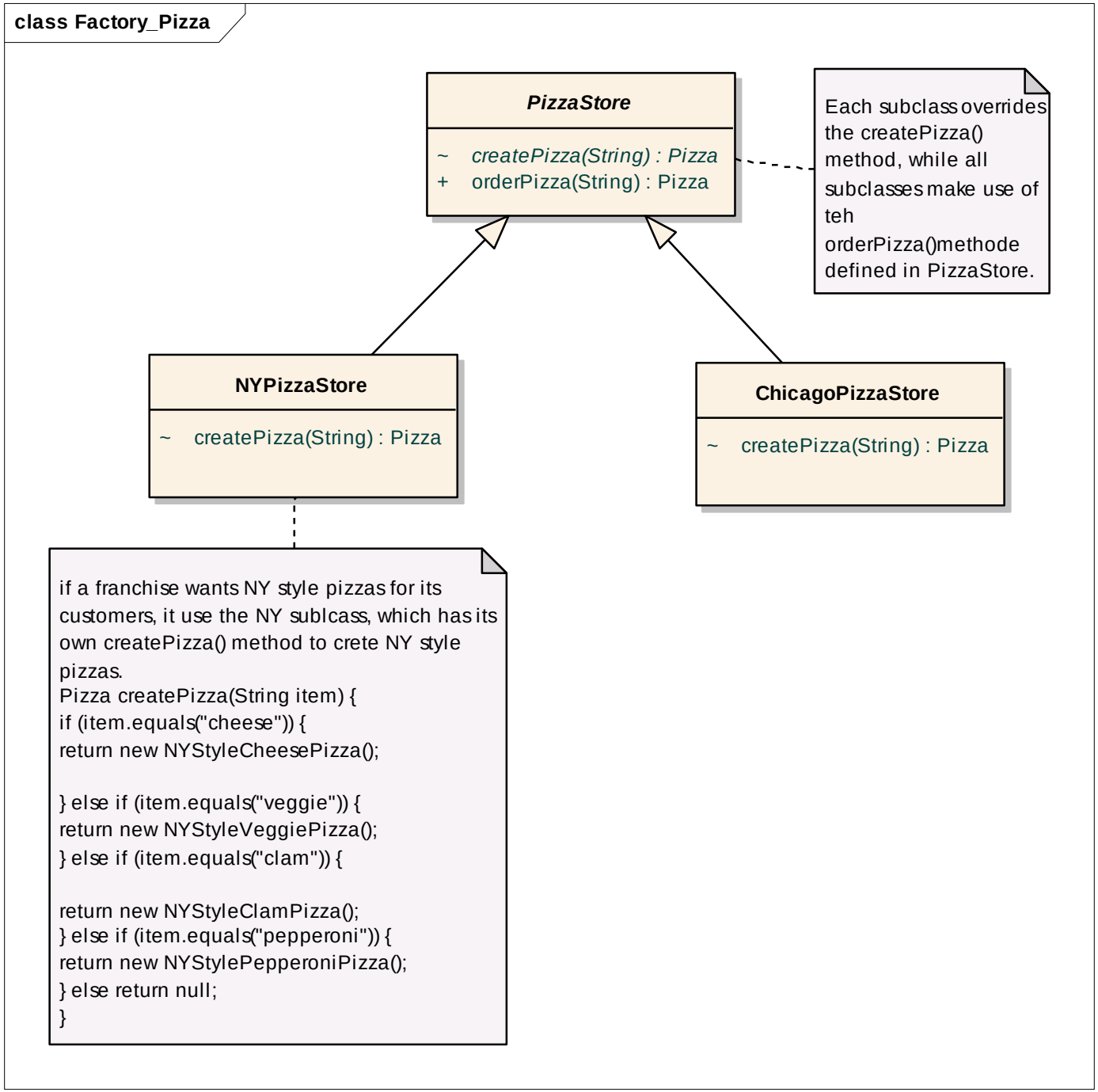
- A class cannot anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- A factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it.

Motivation cont.

- Suppose in previous example, we open new stores in different places
- Different places might want to offer different styles of pizzas

A framework for the pizza store

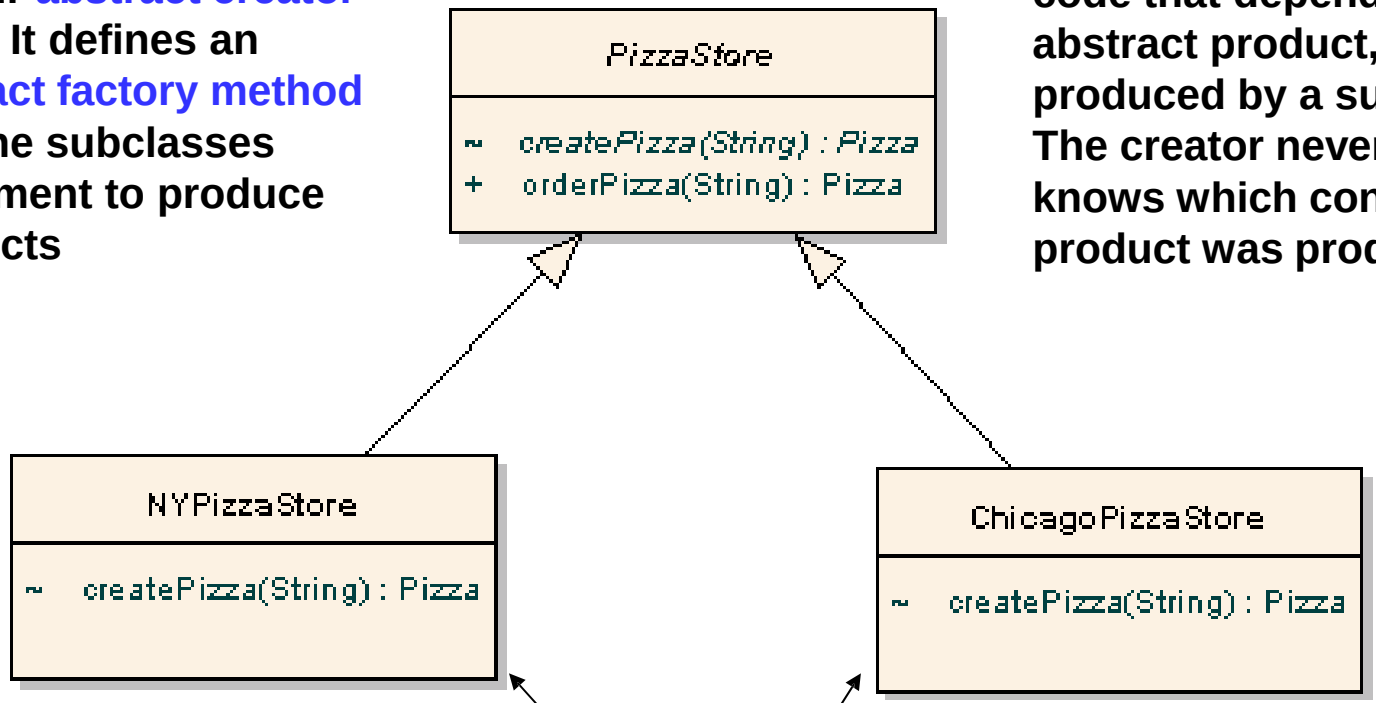
```
//PizzaStore is now abstract
public abstract class PizzaStore {


    public Pizza orderPizza(String type) {
//Now create Pizza is back to being a call to a method in the PizzaStore rather than on a
    factory object
            Pizza pizza = createPizza(type);
            System.out.println("--- Making a " + pizza.getName() + " ---");
            pizza.prepare();
            pizza.bake();
            pizza.cut();
            pizza.box();
            return pizza;
    }
//Now we've moved our factory object to this method and is now abstract in PizzaStore.
    abstract Pizza createPizza(String item);

}
```


class Factory_Pizza
PizzaStore
~ createPizza(String) : Pizza
+ orderPizza(String) : Pizza
Each subclass overrides the createPizza() method, while all subclasses make use of teh orderPizza()methode defined in PizzaStore.
NYPizzaStore
~ createPizza(String) : Pizza
ChicagoPizzaStore
~ createPizza(String) : Pizza
if a franchise wants NY style pizzas for its customers, it use the NY sublcass, which has its own createPizza() method to crete NY style pizzas.
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;
}

The Creator Classes

This is our abstract creator class. It defines an abstract factory method that the subclasses implement to produce products

Often the creator contains code that depends on an abstract product, which is produced by a subclass. The creator never really knows which concrete product was produced

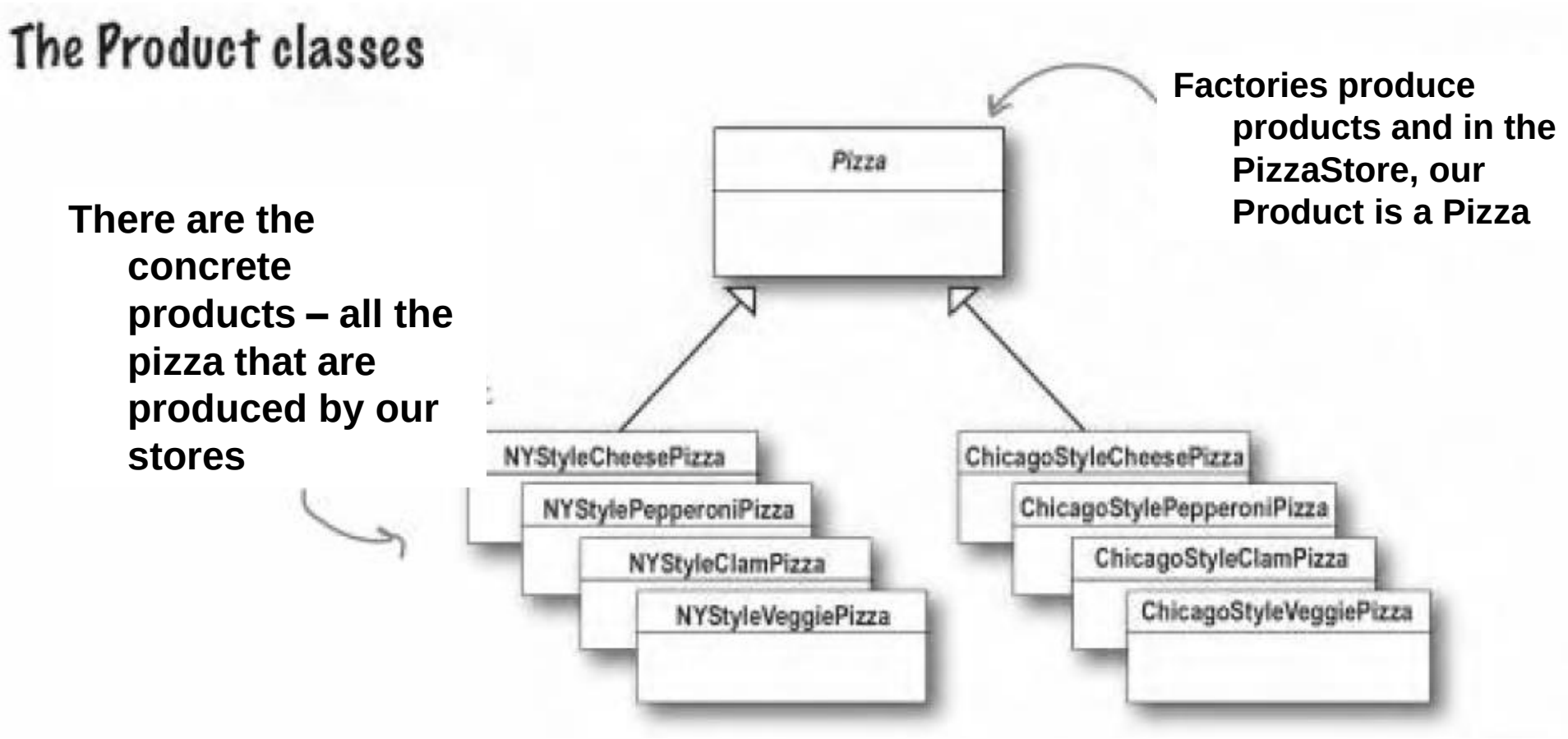


The createPizza() method is our factory method. It produces products

Classes that produce products are called concrete creators

Since each franchise gets its own subclass of PizzaStore, it is free to create its own style of pizza by implementing createPizza()

The Product Classes



Source codes- PizzaStore

```
public abstract class PizzaStore {

    abstract Pizza createPizza(String type);

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        System.out.println("--- Making a " + pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Source code - NYPizzaStore

```
public class NYPizzaStore extends PizzaStore {

    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;
    }
}
```

SourceCode - Pizza

```
public abstract class Pizza {
	String name;
	String dough;
	String sauce;
	ArrayList toppings = new ArrayList();

	void prepare() {
		System.out.println("Preparing " + name);
		System.out.println("Tossing dough...");
		System.out.println("Adding sauce...");
		System.out.println("Adding toppings: ");
		for (int i = 0; i < toppings.size(); i++) {
			System.out.println("   " +
	toppings.get(i));
		}
	}

	void bake() {
		System.out.println("Bake for 25 minutes at 350");
	}
```

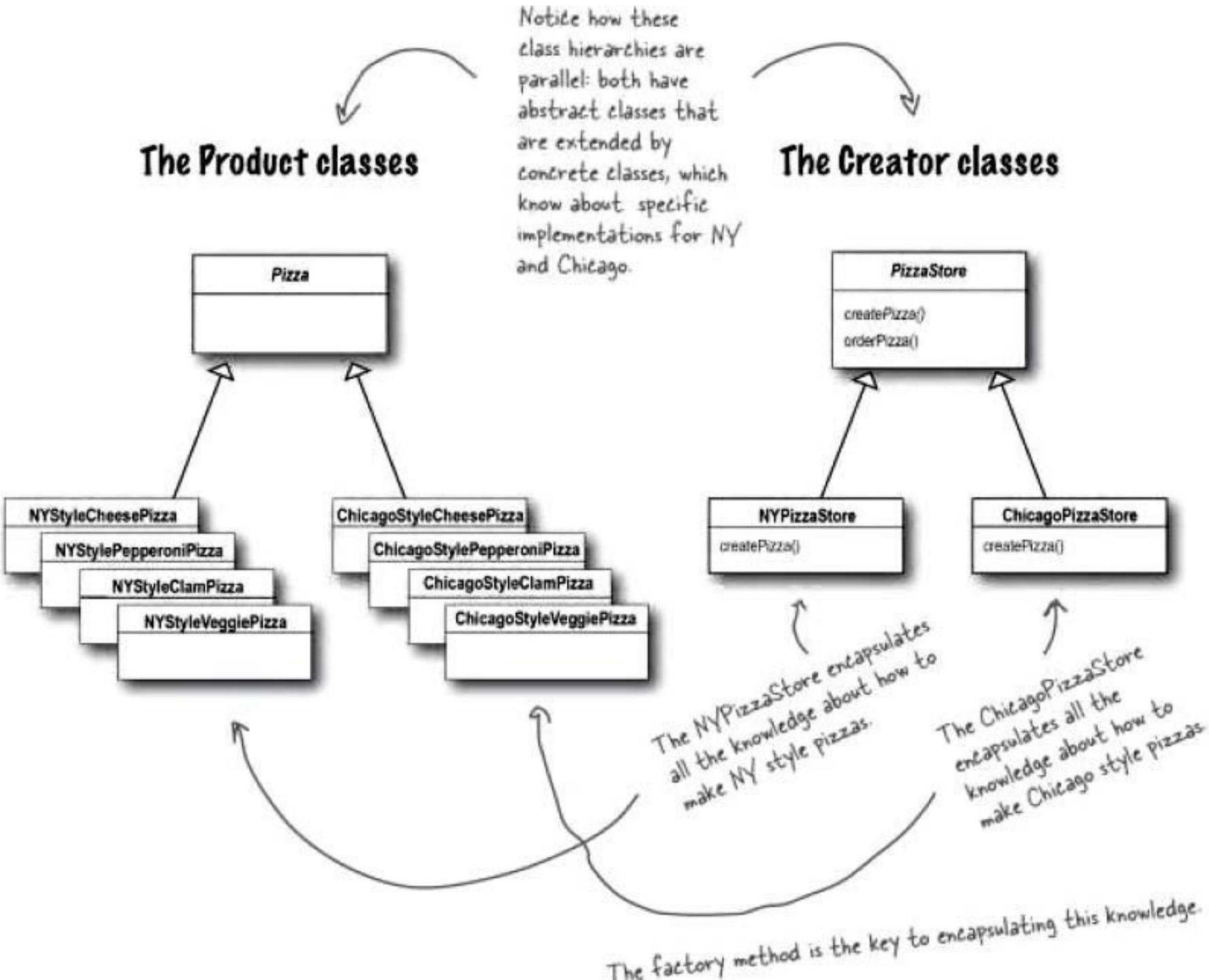
```
	void cut() {
		System.out.println("Cutting the pizza into diagonal slices");
	}

	void box() {
		System.out.println("Place pizza in official PizzaStore box");
	}

	public String getName() {
		return name;
	}

	public String toString() {
		StringBuffer display = new StringBuffer();
		display.append("---- " + name + " ----\n");
		display.append(dough + "\n");
		display.append(sauce + "\n");
		for (int i = 0; i < toppings.size(); i++) {
			display.append((String )toppings.get(i) + "\n");
		}
		return display.toString();
	}
```

SourceCode - NYStyleClamPizza

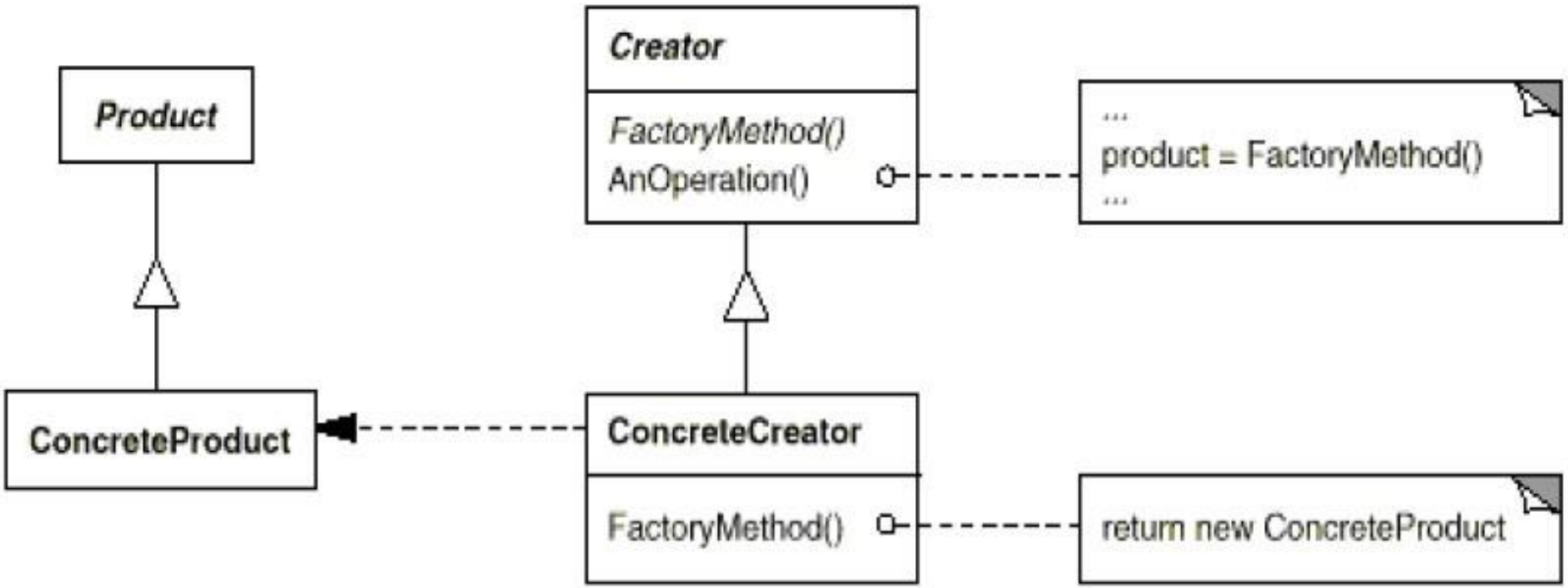
```
public class NYStyleClamPizza extends Pizza {

    public NYStyleClamPizza() {
        name = "NY Style Clam Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
        toppings.add("Fresh Clams from Long Island
    Sound");
    }
}
```

Source Code – Main program

```
public static void main(String[] args)
    {
        PizzaStore nyStore = new NYPizzaStore();
       PizzaStore chicagoStore = new ChicagoPizzaStore();



        pizza = nyStore.orderPizza("clam");
        System.out.println("Ethan ordered a " +
    pizza.getName() + "\n");

                    }
```


Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.
The Product classes
The Creator classes
Pizza
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
PizzaStore
createPizza()
orderPizza()
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
The NYPizzaStore encapsulates all the knowledge about how to make NY style pizzas.
The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago style pizzas.
The factory method is the key to encapsulating this knowledge.

The Factory Method Structure



Participants

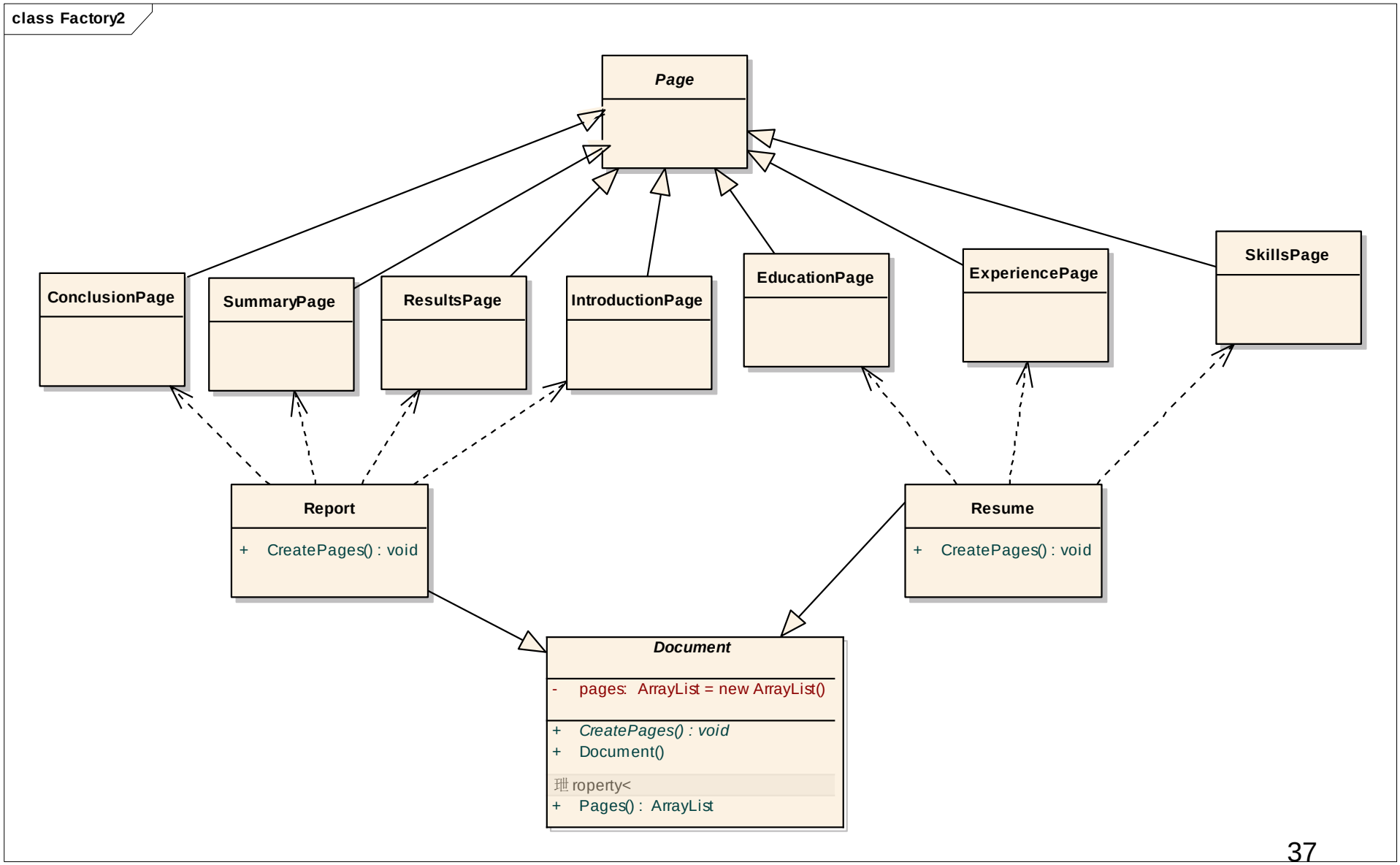
- Product Pizza
  - Defines the interface for the type of objects the factory method creates
- ConcreteProduct NYStyleCheesePizza
  - Implements the Product interface
- Creator PizzaStore
  - Declares the factory method, which returns an object of type Product
- ConcreteCreator NYPizzaStore
  - Overrides the factory method to return an instance of a ConcreteProduct

Collaborations

- Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct
- 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.

Execise1

- Create two types of Documents: Resume and Report (the concrete creators)
- Each Document consists of Pages (Page is the Product)
  - Factory Method: CreatePages()
- A Resume has the following pages (concrete products)
  - SkillsPage
  - EducationPage
  - ExperiencePage
- A Report has the following pages (concrete products)
  - SummaryPage
  - IntroductionPage
  - ResultsPage
  - ConclusionPage


class Factory2
Page
ConclusionPage
SummaryPage
ResultsPage
IntroductionPage
EducationPage
ExperiencePage
SkillsPage
Report
+ CreatePages() : void
Resume
+ CreatePages() : void
Document
- pages: ArrayList = new ArrayList()
+ CreatePages() : void
+ Document()
roperty<
+ Pages() : ArrayList

Execise1 – Implementation

```
// "Product"

abstract class Page
 {
 }
```

Execise1 – Implementation (cont'd)

```
// Concrete Products for resume

class SkillsPage : Page
      {
         public GetPageName()
          { Console.WriteLine("SkillsPage") }          }
class EducationPage : Page
      {
         public GetPageName()
          { Console.WriteLine("EduationPage") }      }
class ExperiencePage : Page
      {
         public GetPageName()
          { Console.WriteLine("ExperiencePage") }    }
```

Execise1 – Implementation (cont'd)

```
// Concrete Products for report

class SummaryPage : Page
      {
         public GetPageName()
          { Console.WriteLine("SummaryPage") }          }
class IntroductionPage : Page
      {
         public GetPageName()
          { Console.WriteLine("IntroductionPage") }      }
class ResultsPage : Page
      {
         public GetPageName()
          { Console.WriteLine("ResultsPage") }          }
class ConclusionPage : Page
      {
         public GetPageName()
          { Console.WriteLine("ConclusionPage") }       }
```

Execise1 – Implementation (cont'd)

```
//  Abstract Creator
abstract class Document
 {
   public ArrayList Pages = new ArrayList();

  // Constructor calls abstract Factory method
   public Document()
   {
     this.CreatePages();
   }

  public abstract GetDocName();

  // Factory Method
  public abstract void CreatePages();
 }
```

Execise1 – Implementation (cont'd)

```
// First Concrete Creator

class Resume : Document
{

 // Factory Method implementation
 public override void CreatePages()
  {
    Pages.Add(new SkillsPage());
    Pages.Add(new EducationPage());
    Pages.Add(new ExperiencePage());
  }

  public overide void GetDocName()
    {  Console.WriteLine("Resume")  }

}
```

Execise1 – Implementation (cont'd)

```
// Second Concrete Creator

class Report : Document
{

 // Factory Method implementation

 public override void CreatePages()
  {
    Pages.Add(new SummaryPage());
    Pages.Add(new IntroductionPage());
    Pages.Add(new ResultsPage());
    Pages.Add(new ConclusionPage());
  }
public overide void GetDocName()
   {  Console.WriteLine("Report") }

}
```

Execise1 - Implementation

```
// MainApp test application
 class MainApp
 {
   static void Main()
   {
     // Note: constructors call Factory Method
     Document[] documents = new Document[2];
     documents[0] = new Resume();
     documents[1] = new Report();

     // Display document pages
     foreach (Document document in documents)
     {
      Console.WriteLine("\n" + document.GetDocName()+ "--");
       foreach (Page page in document.Pages)
                 {
                 Console.WriteLine(" " + page.GetPageName());
                 }
     }
   }
 }
```

Example - Output

```
Resume -------
SkillsPage
EducationPage
ExperiencePage

Report -------
 IntroductionPage
 ResultsPage
 ConclusionPage
 SummaryPage
 BibliographyPage
```

Abstract Factory Pattern

Pattern name and classification

- Abstract Factory
- Scope : Object
- Purpose : Creational

Intent

- Provide an Interface for creating families of related or dependent objects without specifying their concrete classes
  - Objects instantiated in a coordinated fashion
- The pattern ensures that the system always gets the correct set of objects for the situation

Families of objects

- Families may be defined according to any number of reasons
  - Different operating systems (cross-platform applications)
  - Different performance guidelines
  - Different versions of applications
  - …

Motivating Example

- Select device drivers (display and print) according to the machine capacity

For Driver	In a Low-capacity machine, use	In a High-capacity machine, use
Display	LRDD – Low Resolution Display Driver	HRDD – High Resolution Display Driver
Print	LRPD – Low Resolution Print Driver	HRPD – High Resolution Print Driver

Define Families based on a Unifying Concept

- Two Families
  - Low-resolution family: put low demands on the system
    - LRDD and LRPD
  - High-resolution family: put high demands on the system
    - HRDD and HRPD
- Families are not always exclusive
  - Mid-resolution family: for mid-range machines
    - LRDD and HRDP

Applicability

1. A system should be independent of how its objects (or product objects) are created, composed, and represented
2. A system should be configured with one of multiple families of objects
3. A family of related objects is designed to be used together

Solution 1 – A Switch to Control Which Driver to Use

```
Class ApControl {
  …
  public doDraw() {
    …
    switch (Resolution) {
      case LOW:
        // use LRDD
      case HIGH:
        // use HRDD
    }
  }
  public doPrint() {
    …
    switch (Resolution) {
      case LOW:
        // use LRPD
      case HIGH:
        // use HRPD
    }
  }
}
```

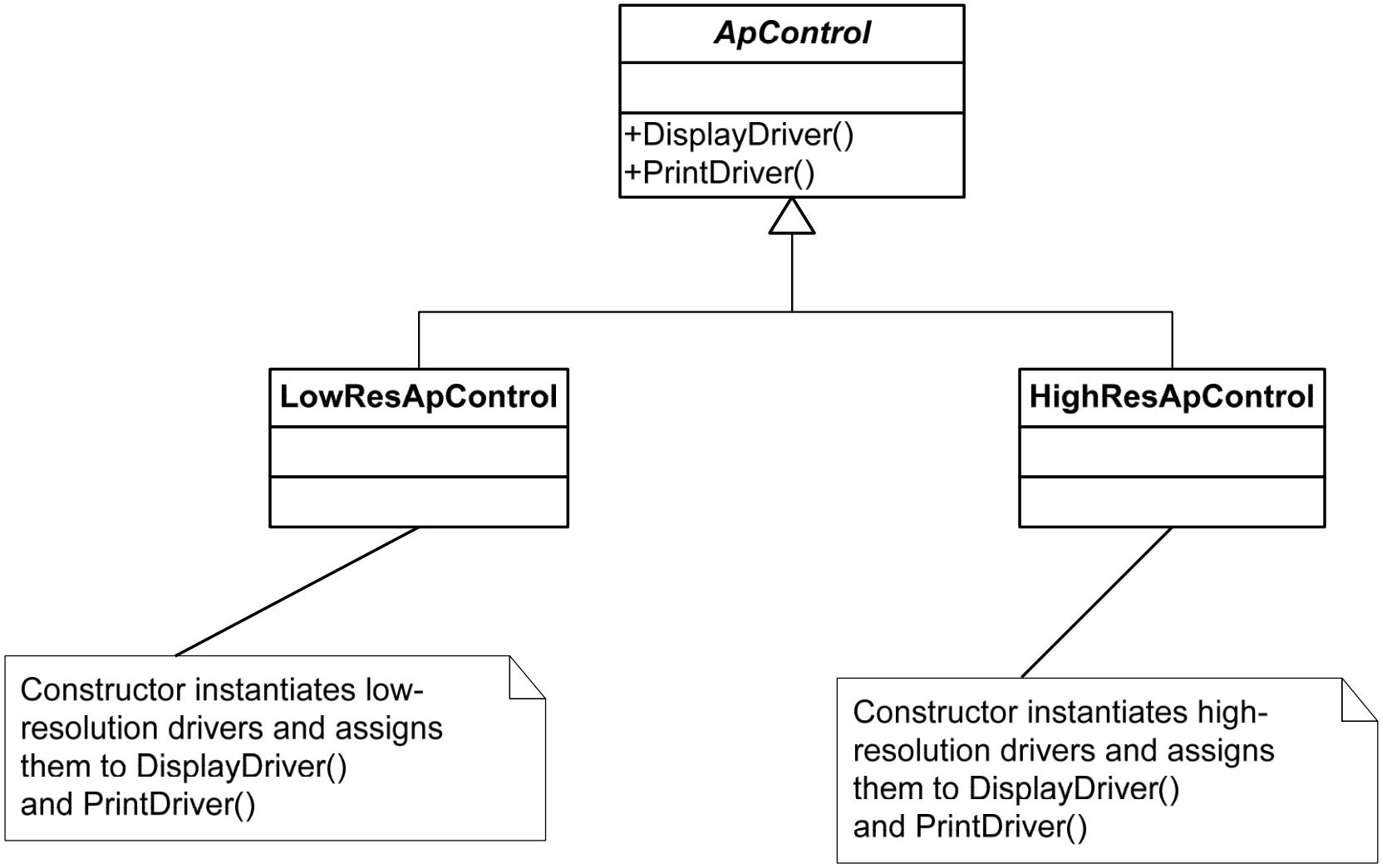
Solution 1 - Cons

- Unclear code
  - Code for determining the driver to use are intermixed with the code for using the driver
- Tight coupling
  - Adding a MIDDLE value requires changing the code in two places (otherwise not related)
- Weak cohesion
  - Each method (doDraw and doPrint) has two unrelated assignment: select driver and create shape

Solution 2 – Use Inheritance

- Use one abstract class *Apcontrol*
- Have two different concrete classes LowResApControl and HighResApControl
  - LowResApControl and HighResApControl are derived from *Apcontrol*
  - LowResApControl uses low-resolution drivers
  - HighResApcontrol uses high-resolution drivers

Solution 2 – The Class Diagram



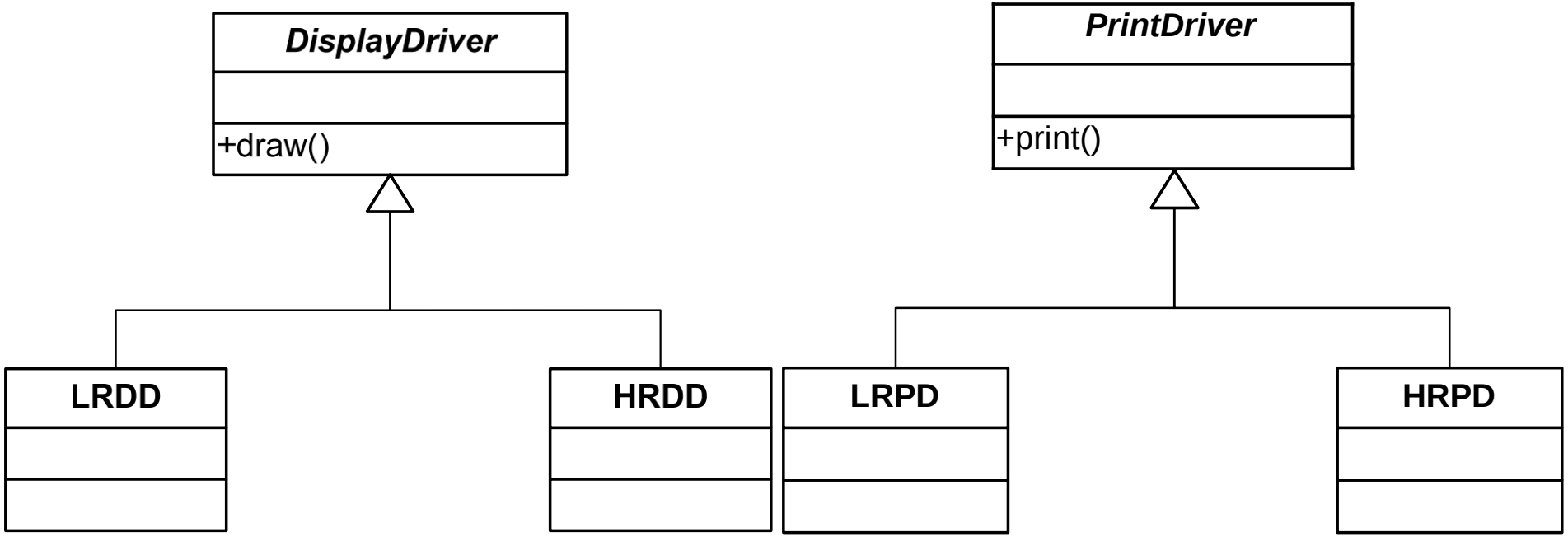
Solution 2 - Cons

- Combinatorial explosion: new concrete class for each new family
  - New concrete class for (LRDP,HRDD)
  - New concrete class for (HRDP, LRDD)
- Unclear meaning: we specialized each class to a particular special case

Solution 3 – Replace Switches With Abstraction

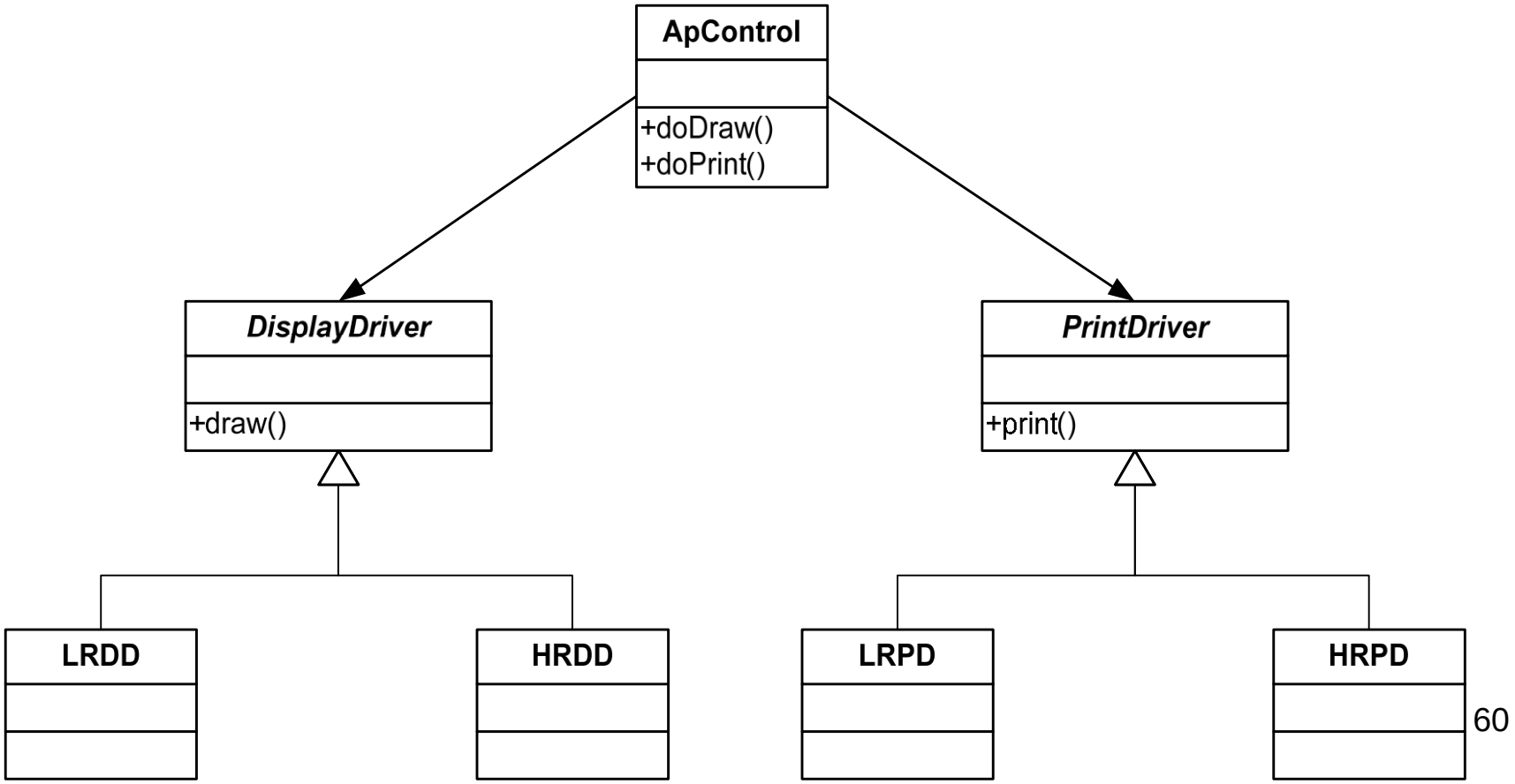
- Opportunity for abstraction
  - LRDD and HRDD are both display drivers
  - LRPD and HRPD are both print drivers
- Abstractions
  - Display Drivers
  - Print Drivers

Drivers and their Abstractions



ApControl Using Display and Print Drivers

- Code is simpler to understand: ApControl uses a DisplayDriver object or a PrintDriver object without concerning itself about the driver's resolution



Code Fragment

```
Class ApControl {
  …
  public doDraw() {
    …
    myDisplayDriver.draw()
  }
  public doPrint() {
    …
    myPrintDriver.print()
  }
}
```

Question

How do we create the appropriate objects?

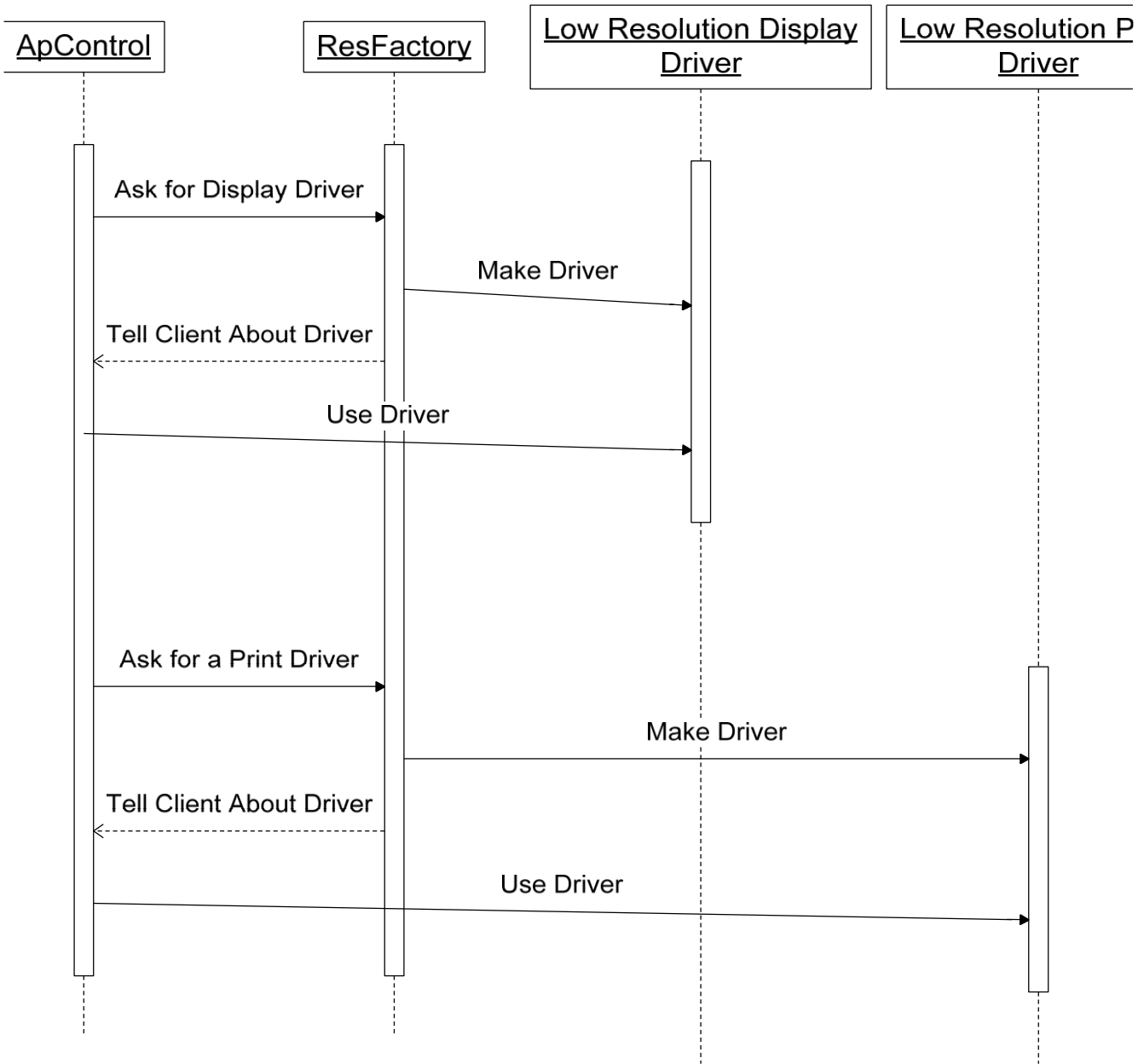
Creating Appropriate Objects

- Option 1: Have ApControl do it
  - Maintenance problem: new set of objects
  - Unclear code: intermixing creation code with other
- Option 2: delegate the creation of the appropriate family of objects to another object
  - Call it the factory object (ResFactory in our example)

The Factory Object

- Decomposition by responsibility
  - ResFactory has the responsibility for deciding which objects are appropriate
  - ApControl has the responsibility for knowing how to work with the appropriate objects
    - ApControl does not need to worry about whether a low or high resolution driver is returned because it uses both in the same way

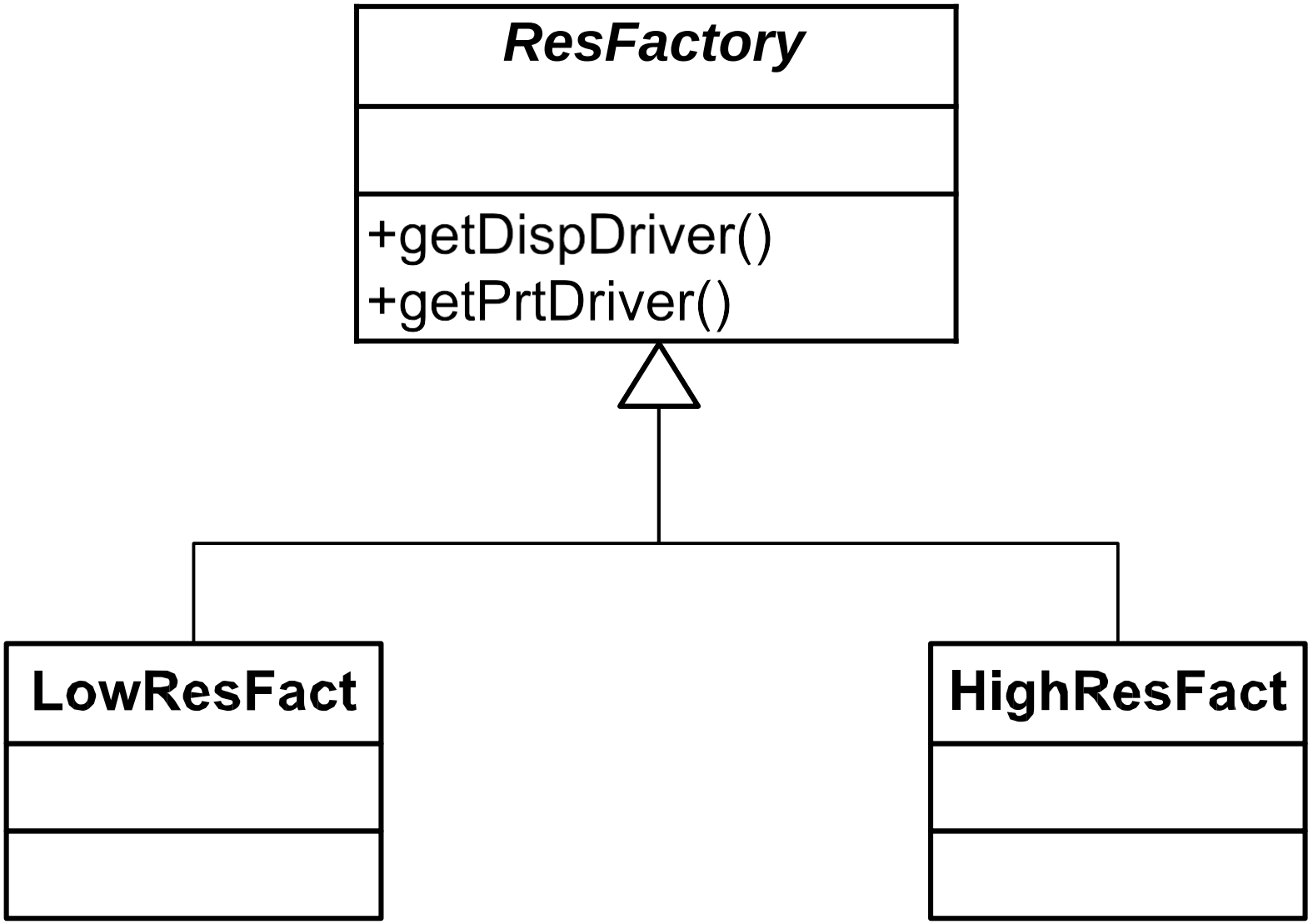
Sequence Diagram



The ResFactory Abstract Class

- Define each factory for the same family of objects as a concrete class
  - LowResFact and HighResFact are derived from the ResFactory abstract class
- The ResFactory abstract class has two methods
  - Give me the display driver I should use
  - Give me the print driver I should use

The ResFactory Abstract Class (cont'd)



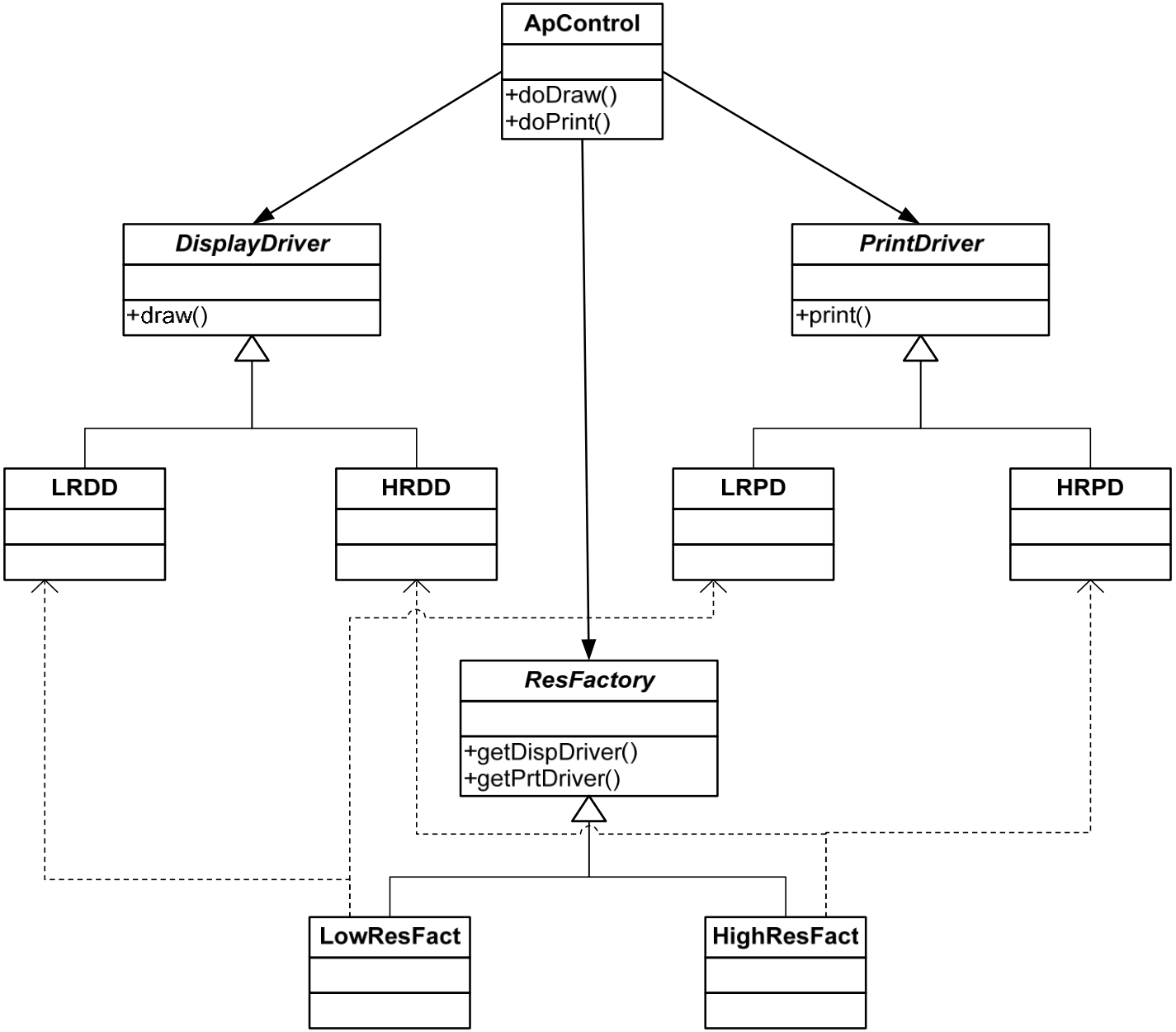
Implementation of ResFactory-Code Fragments

```
abstract class ResFactory {
    abstract public DisplayDriver getDispDriver();
    abstract public PrintDriver getPrtDriver();
}
class LowResFact extends ResFactory {
    public DisplayDriver getDispDriver() {
        return new LRDD();
    }
    public PrintDriver getPrtDriver() {
        return new LRPD();
    }
}
```

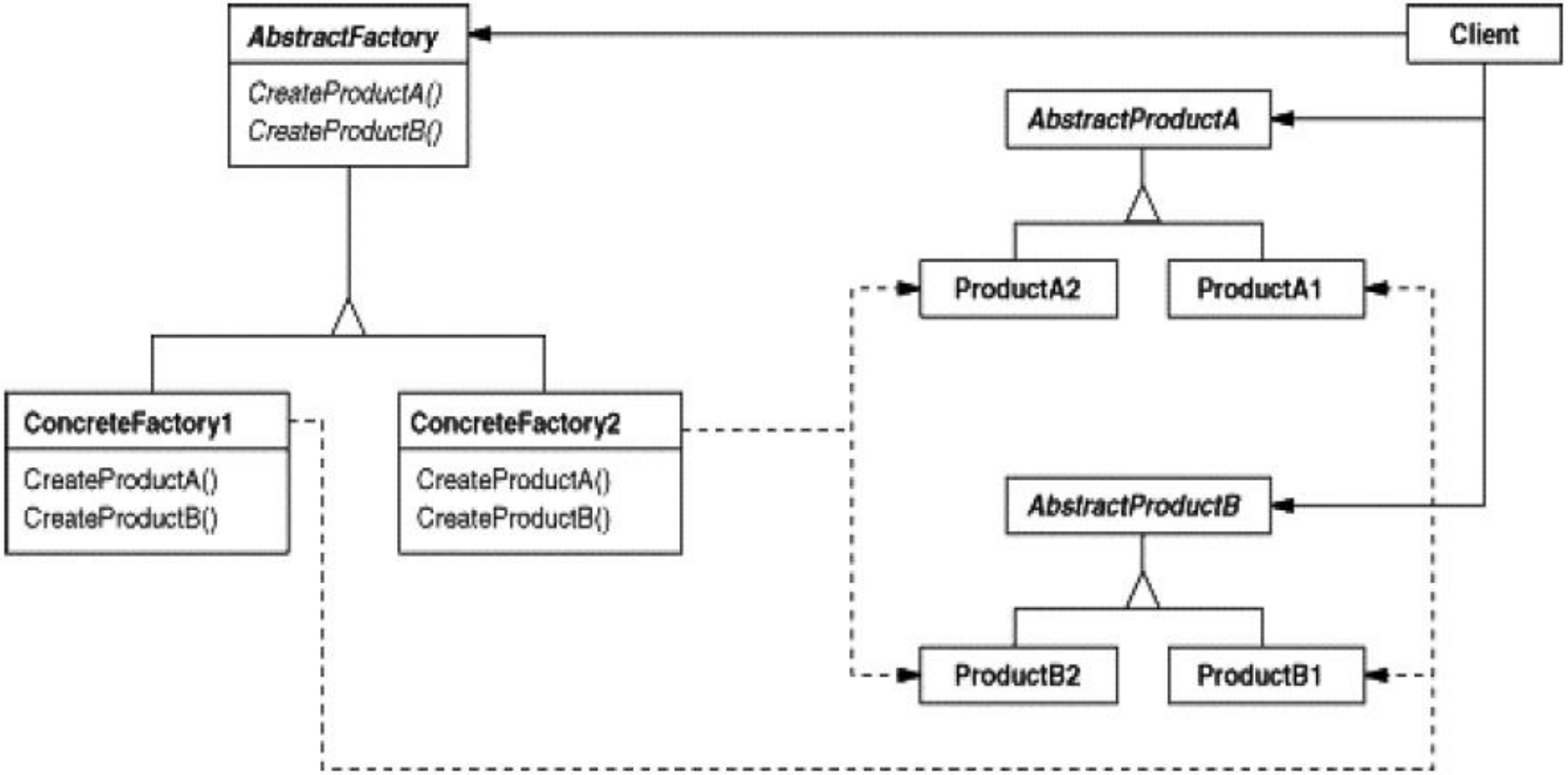
Implementation of ResFactory-Code Fragments (cont'd)

```
class HighResFact extends ResFactory {
    public DisplayDriver getDispDriver() {
        return new HRDD();
    }
    public PrintDriver getPrtDriver() {
        return new HRPD();
    }
}
```

Putting Things Together



The Abstract Factory Structure



Participants

- AbstractFactory
  - Declares an interface for operations that create abstract product objects
- ConcreteFactory
  - Implements the 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 the corresponding concrete factory
  - Implements the AbstractProduct interface
- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

Collaborations

- Normally a single instance of a ConcreteFactory class is created at runtime.
- This concrete factory creates product objects having a particular implementation.
- To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory

Consequences

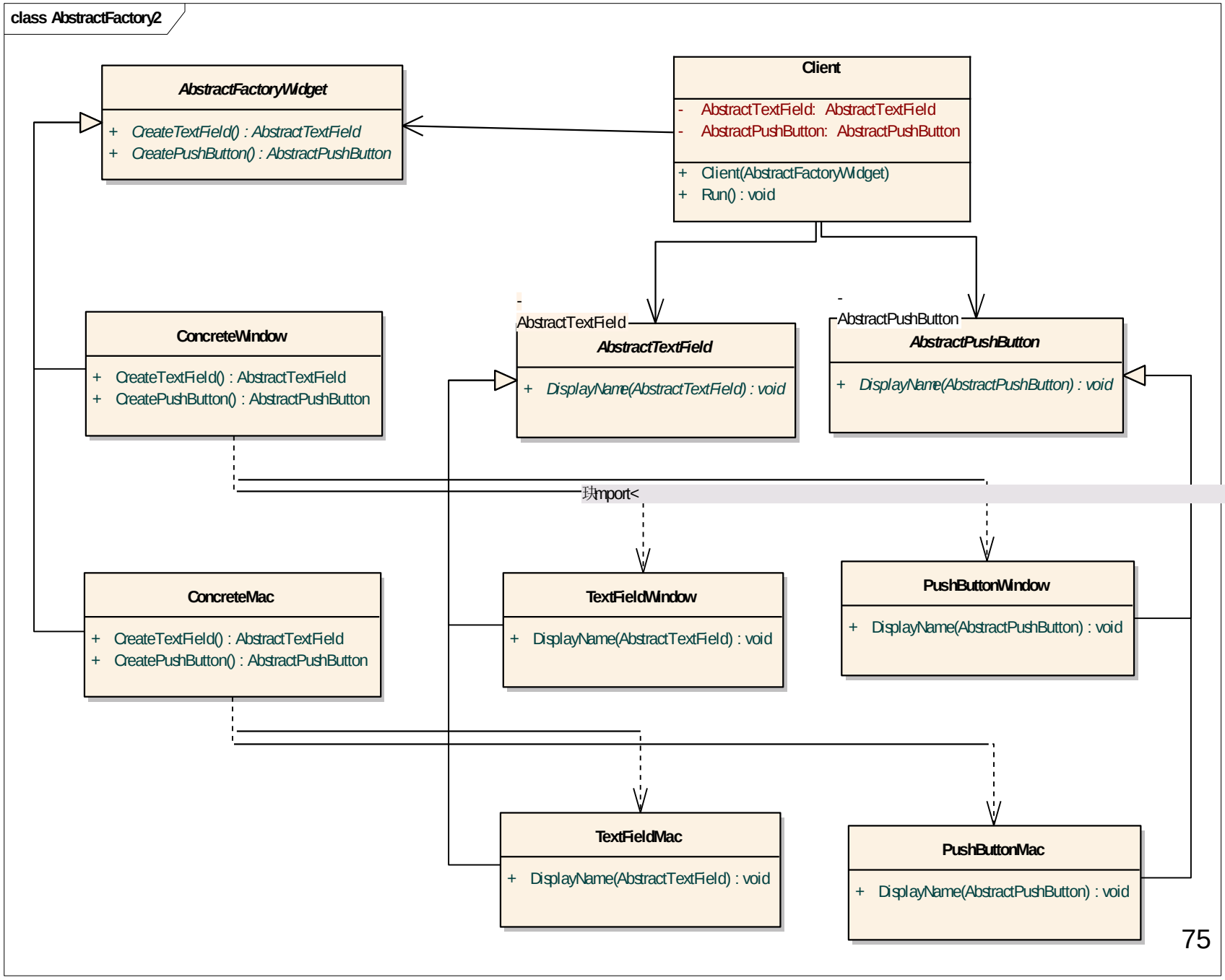
- Benefits
  - Isolates clients from concrete implementation classes
  - Makes exchanging product families easy, since a particular concrete factory can support a complete family of products
  - Enforces the use of products only from one family
- Liabilities
  - Supporting new kinds of products requires changing the AbstractFactory interface

Exercise

- Suppose you have the task of building a user interface framework that works on top of MS-Windows, Mac OS. And. It must work on each platform with the platform's native look and feel. You organize it by creating an abstract class for each type of widget. We consider the following two types: text field, push button. The main program use client to display

  "This is Window Button as PushButtonWindow"
  "This is Window TextField as TextFieldWindow"
  "This is Mac Button as PushButtonMac"
  "This is Mac Text Field as TextFieldMac"

- in the case of MS-Windows and Mac OS platforms respectively
- Question 1: Give the UML class diagram
- Question 2: Implement the class diagram


class AbstractFactory2
AbstractFactoryWidget
+ CreateTextField() : AbstractTextField
+ CreatePushButton() : AbstractPushButton
Client
- AbstractTextField: AbstractTextField
- AbstractPushButton: AbstractPushButton
+ Client(AbstractFactoryWidget)
+ Run() : void
ConcreteWindow
+ CreateTextField() : AbstractTextField
+ CreatePushButton() : AbstractPushButton
AbstractTextField
AbstractTextField
+ DisplayName(AbstractTextField) : void
AbstractPushButton
AbstractPushButton
+ DisplayName(AbstractPushButton) : void
import
ConcreteMac
+ CreateTextField() : AbstractTextField
+ CreatePushButton() : AbstractPushButton
TextFieldWindow
+ DisplayName(AbstractTextField) : void
PushButtonWindow
+ DisplayName(AbstractPushButton) : void
TextFieldMac
+ DisplayName(AbstractTextField) : void
PushButtonMac
+ DisplayName(AbstractPushButton) : void

Implementation

```
abstract class AbstractFactoryWidget
  {
      public abstract AbstractTextField
  CreateTextField();
      public abstract AbstractPushButton
  CreatePushButton();
  }
```

```
    class ConcreteWindow : AbstractFactoryWidget
    {
        public override AbstractTextField CreateTextField()
        {
            return new TextFieldWindow();
        }
        public override AbstractPushButton CreatePushButton()
        {
            return new PushButtonWindow();
        }
    }

class ConcreteMac : AbstractFactoryWidget
  {
      public override AbstractTextField CreateTextField()
      {
          return new TextFieldMac();
      }
      public override AbstractPushButton CreatePushButton()
      {
          return new PushButtonMac();
      }
  }
```

```
// "AbstractTextField"
abstract class AbstractTextField
    {
        public abstract void DisplayName(AbstractTextField a);
    }


// "AbstractPushButton"

 abstract class AbstractPushButton
    {
        public abstract void DisplayName(AbstractPushButton a);
    }
```

```
// "Concrete TextFieldWindow"

   class TextFieldWindow : AbstractTextField
   {
      public override void DisplayName(AbstractTextField a)
      {
         Console.WriteLine(" This is Window Text Field as " + a.GetType().Name);
      }
   }

   // "Concrete PushButtonWindow"

   class PushButtonWindow : AbstractPushButton
   {
      public override void DisplayName(AbstractPushButton a)
      {
         Console.WriteLine(" This is Window Button as " + a.GetType().Name);
      }
   }
```

```
// "Concrete TextFieldMac"

   class TextFieldMac : AbstractTextField
   {
      public override void DisplayName(AbstractTextField a)
      {
         Console.WriteLine(" This is Mac TextField as " + a.GetType().Name);
      }
   }

   // "Concrete PushButtonMac"

   class PushButtonMac : AbstractPushButton
   {
      public override void DisplayName(AbstractPushButton a)
      {
         Console.WriteLine(" This is Mac Button as " + a.GetType().Name);
      }
   }
```

```
class Client
  {
     private AbstractTextField AbstractTextField;
     private AbstractPushButton AbstractPushButton;

     // Constructor
     public Client(AbstractFactoryWidget factory)
     {
        AbstractPushButton = factory.CreatePushButton();
        AbstractTextField = factory.CreateTextField();
     }

     public void Run()
     {
        AbstractPushButton.DisplayName(AbstractPushButton);
        AbstractTextField.DisplayName(AbstractTextField);
     }
  }
```

```
static void Main(string[] args)
        {
            AbstractFactoryWidget    FactoryWindow = new ConcreteWindow();
           Client c1 = new Client(FactoryWindow);
           c1.Run();

            AbstractFactoryWidget    FactoryMac = new ConcreteMac();
           Client c2 = new Client(FactoryMac);
           c2.Run();


           Console.Read();
        }
```