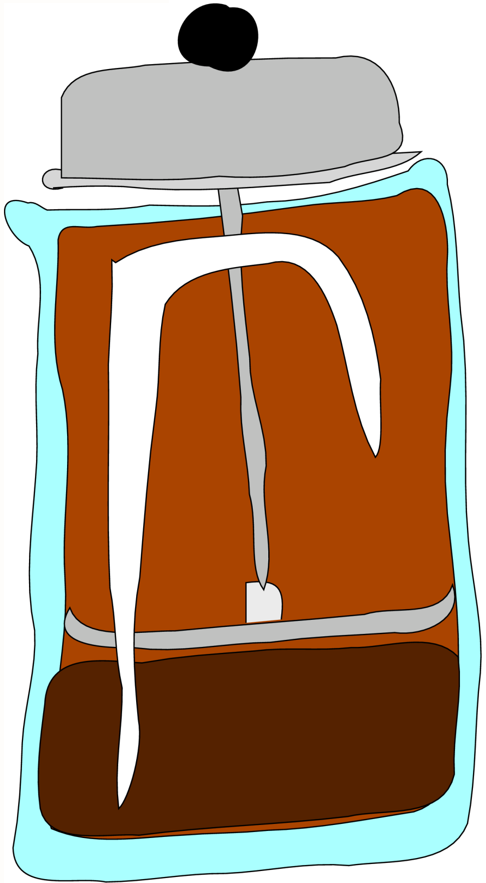
- **Template Method Pattern**

Example

Coffee recipe:

- boil some water
- brew coffee in the water
- pour coffee in cup
- add sugar and milk

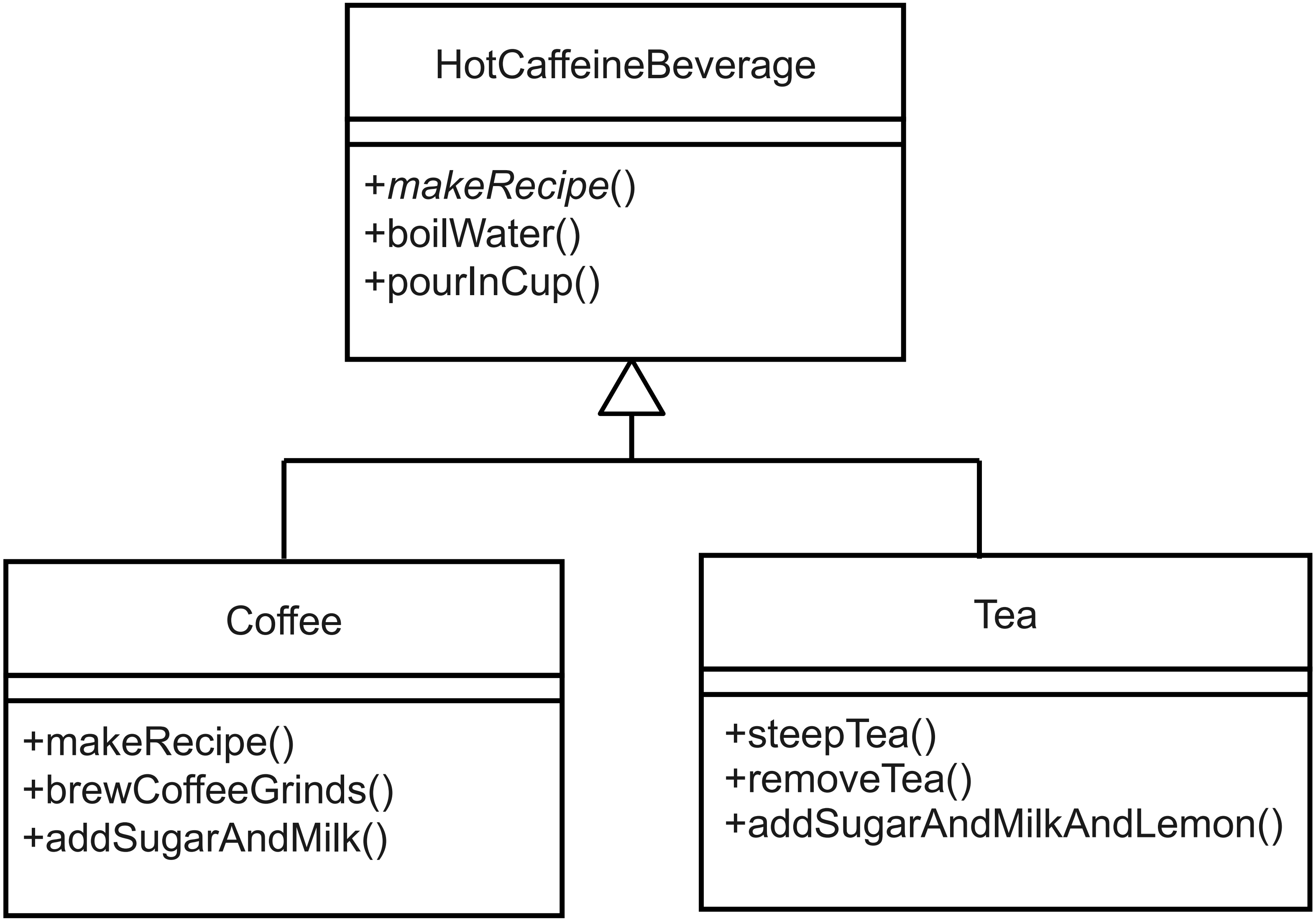


Tea recipe:

- boil some water
- steep tea in the water
- Remove tea from water
- Pour cup of tea
- Add sugar, milk or lemon

```
public class Coffee {

    public void makeRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println( “Boiling water” );
    }
    public void brewCoffeeGrinds() {
        System.out.println( “Brewing the coffee” );
    }
    public void pourInCup() {
        System.out.println( “Pouring into cup” );
    }
    public void addSugarAndMilk() {
        System.out.println( “Adding sugar, milk” );
    }
}
```

```
public class Tea {

    public void makeRecipe() {
        boilWater();
        steepTea();
        removeTea();
        pourInCup();
        addSugarMilkLemon();
    }

    public void boilWater() {
        System.out.println( “Boiling water” );
    }
    public void steepTeaBag() {
        System.out.println( “Steeping the tea” );
    }
    public void removeTea() {
        System.out.println( “Remove Tea” );
    }
    public void pourInCup() {
        System.out.println( “Pouring into cup” );
    }

}
```


HotCaffeineBeverage

+*makeRecipe*()
+boilWater()
+pourInCup()

Coffee

+makeRecipe()
+brewCoffeeGrinds()
+addSugarAndMilk()

Tea

+steepTea()
+removeTea()
+addSugarAndMilkAndLemon()

Similar Algorithms

General recipe:

- boil some water
- use the water to extract coffee or tea
- pour resulting beverage into a cup
- add appropriate condiments to the beverage

Similar Algorithms

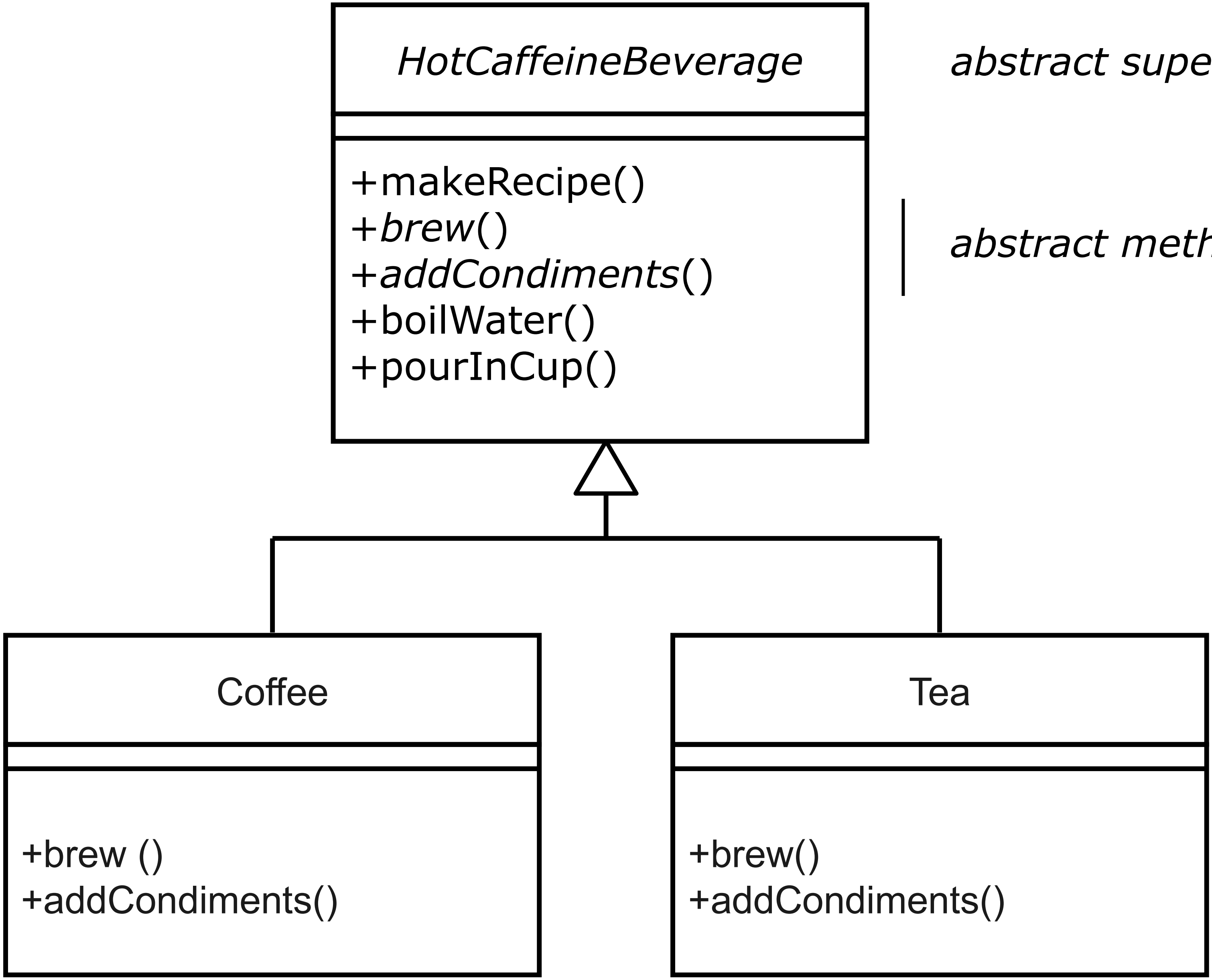
```
// in Coffee class

public void
makeRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

```
// in Tea class

public void
makeRecipe() {
    boilWater();
    steepTeaBag();
    RemoveTeaBag();
    pourInCup();
    addSugarMilkLemon();

}
```

template method

```
public abstract class HotCaffeineBeverage {

    // serves like a “template” for an algorithm,
    // where subclasses provide certain parts
    public final void makeRecipe() {
        boilWater();
        brew();            // from subclass
        pourInCup();
        addCondiments();   // from subclass
    }

    // let the subclasses determine how
    public abstract void brew();
    public abstract void addCondiments();

    public void boilWater() {
        System.out.println( “Boiling water” );
    }

    public void pourInCup() {
        System.out.println( “Pouring into cup” );
    }
}
```

```
// subclasses inherit
// makeRecipe, boilWater, pourInCup

public class Coffee extends HotCaffeineBeverage {

    public void brew() {
        System.out.println( “Brewing the coffee” );
    }
    public void addCondiments() {
        System.out.println( “Adding sugar, milk” );
    }
}

public class Tea extends HotCaffeineBeverage {

    public void brew() {
        System.out.println( “Steeping the tea” );
        System.out.println( “Removing the tea” );
    }
    public void addCondiments() {
        System.out.println( “Adding lemon” );
    }
}
```



HotCaffeineBeverage

+makeRecipe()
+*brew*()
+*addCondiments*()
+boilWater()
+pourInCup()

Coffee

+brew ()
+addCondiments()

Tea

+brew()
+addCondiments()

abstract superclass

abstract methods

Why Template Method?

Before:

- Coffee and Tea have the algorithm
- near duplicated code in Coffee and Tea
- changing the algorithm requires opening the subclasses and making multiple changes

After:

- HotCaffeineBeverage has the algorithm
- reduces duplication and enhances reuse
- algorithm is found in one place, so changes to it are localized

Why Template Method?

Before:

- original structure requires more work to add a new subclass (need to provide the whole algorithm again)

After:

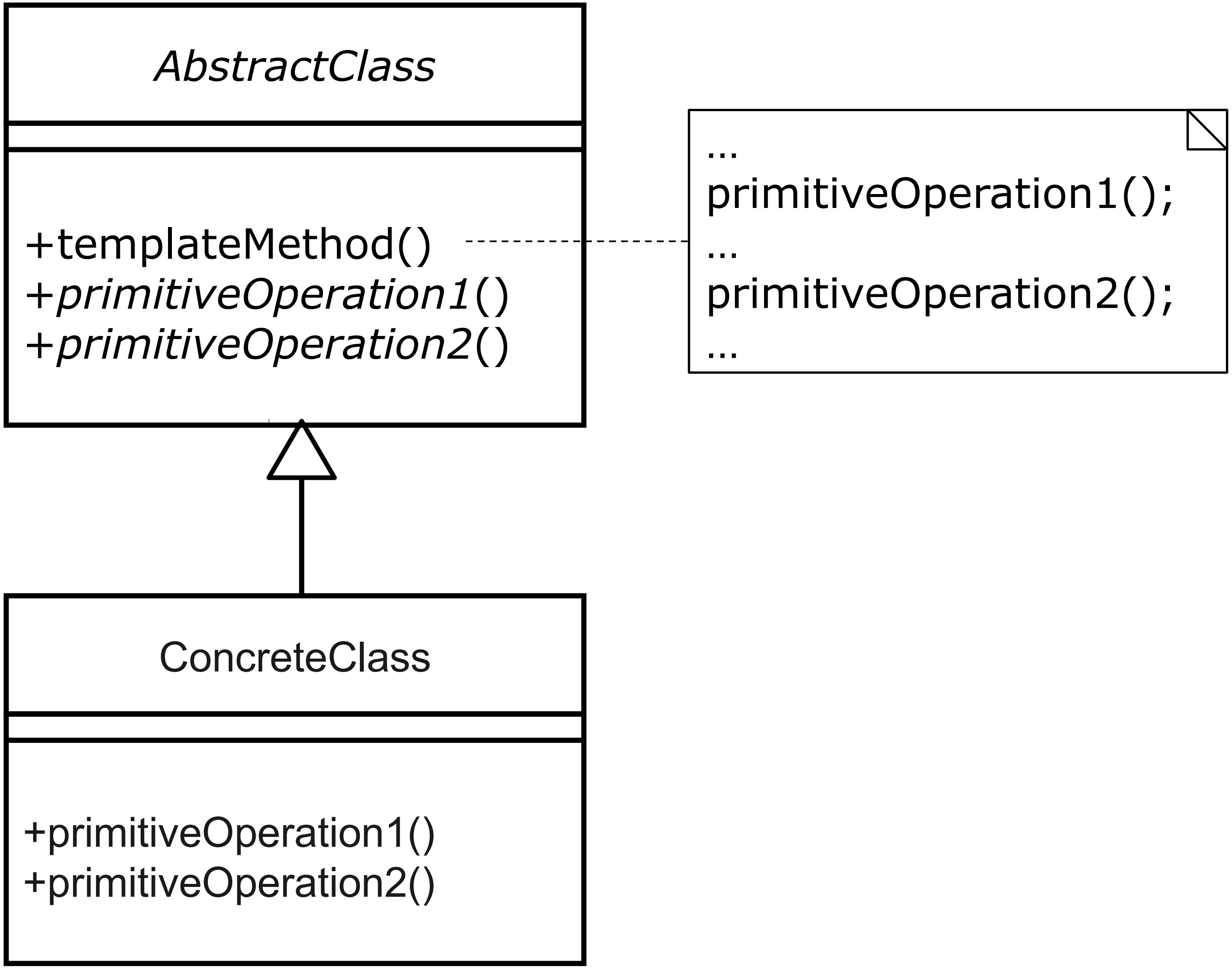
- new structure provides a framework to add a new subclass (need to provide just the distinctive parts of the algorithm)

Template Method Pattern

Design intent:

“define the skeleton of an algorithm in a method, deferring some steps to subclasses”

Template Method Structure



Consequences

Results:

inverted control

- superclass method calling subclass method

“Hollywood principle”

- “Don’t call us, we’ll call you.”

“Hooks”

Idea:

methods in the superclass which provide default behavior that the subclasses *may* override

often *hook* methods do nothing by default

“Hooks”

```
public abstract class AbstractClass {

    public final void templateMethod() {
        …
        primitiveOperation1();
        …
        primitiveOperation2();
        …
        hook();
    }

    // subclasses must override
    public abstract void primitiveOperation1();
    public abstract void primitiveOperation2();

    // do nothing by default;
    // subclass may override
    public void hook() { }
}
```

Exercise

Problem:

page object to be printed

customize for different header and footer

common body text

optional watermark

```
public abstract class Page {
    …

    // template method
    public final void print() {
        printHeader();
        printBody();
        printFooter();
        printWatermark();
    }

    // subclasses must provide header and footer
    public abstract void printHeader();
    public abstract void printFooter();

    // print the page body
    public void printBody() {
        …
    }

    // do nothing by default, i.e., no watermark
    public void printWatermark() { }
}
```

```
public class DraftPage extends Page {
    …
    // print the page header
    public void printHeader() {
        …
    }

    // print the page footer
    public void printFooter() {
        …
    }

    public void printWatermark() {
        // print a DRAFT watermark
        …
    }
}
```

Factory Method Pattern

Dealing with new

```
// limited, what if new pizza types?
PepperoniPizza pizza = new PepperoniPizza();

// code to bake, cut, box PepperoniPizza
…

// or have subclasses of a Pizza abstract superclass
if (pizzaType.equals( “pepperoni” ) {
    Pizza pizza = new PepperoniPizza();
} else if (pizzaType.equals( “veggie” ) {
    Pizza pizza = new VeggiePizza();
}

// code to bake, cut, box Pizza
…
```

Should depend upon abstractions,
not directly upon concrete classes.

Attempt 1

```
// general pizza ordering method
public Pizza orderPizza() {
    Pizza pizza = new Pizza();

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

    return pizza;
}
```

for flexibility, would like to use the superclass name here, but it is abstract

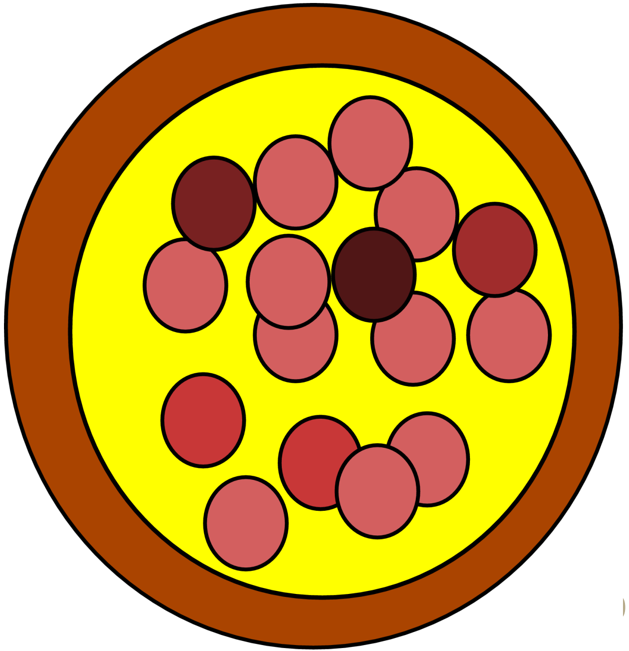
Attempt 2

```
// general pizza ordering method
public Pizza orderPizza( Pizza pizza ) {

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

    return pizza;
}
```

still need code somewhere to instantiate a specific type of pizza, and pass it in

Attempt 3

```
// general pizza ordering method
public Pizza orderPizza( String pizzaType ) {
    Pizza pizza;

    if (pizzaType.equals( “pepperoni” ) {
        Pizza pizza = new PepperoniPizza();
    } else if (pizzaType.equals( “veggie” ) {
        Pizza pizza = new VeggiePizza();
    }

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

    return pizza;
}
```



Attempt 3 with Changes

```
// general pizza ordering method
public Pizza orderPizza( String pizzaType ) {
    Pizza pizza;
```

tends to change

```
    if (pizzaType.equals( “pepperoni” ) {
        Pizza pizza = new PepperoniPizza();
    } else if (pizzaType.equals( “veggie” ) {
        Pizza pizza = new VeggiePizza();
    } else if (pizzaType.equals( “hawaiian” ) {
        Pizza pizza = new HawaiianPizza();
    }
```

tends to stay the same

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

    return pizza;
}
```

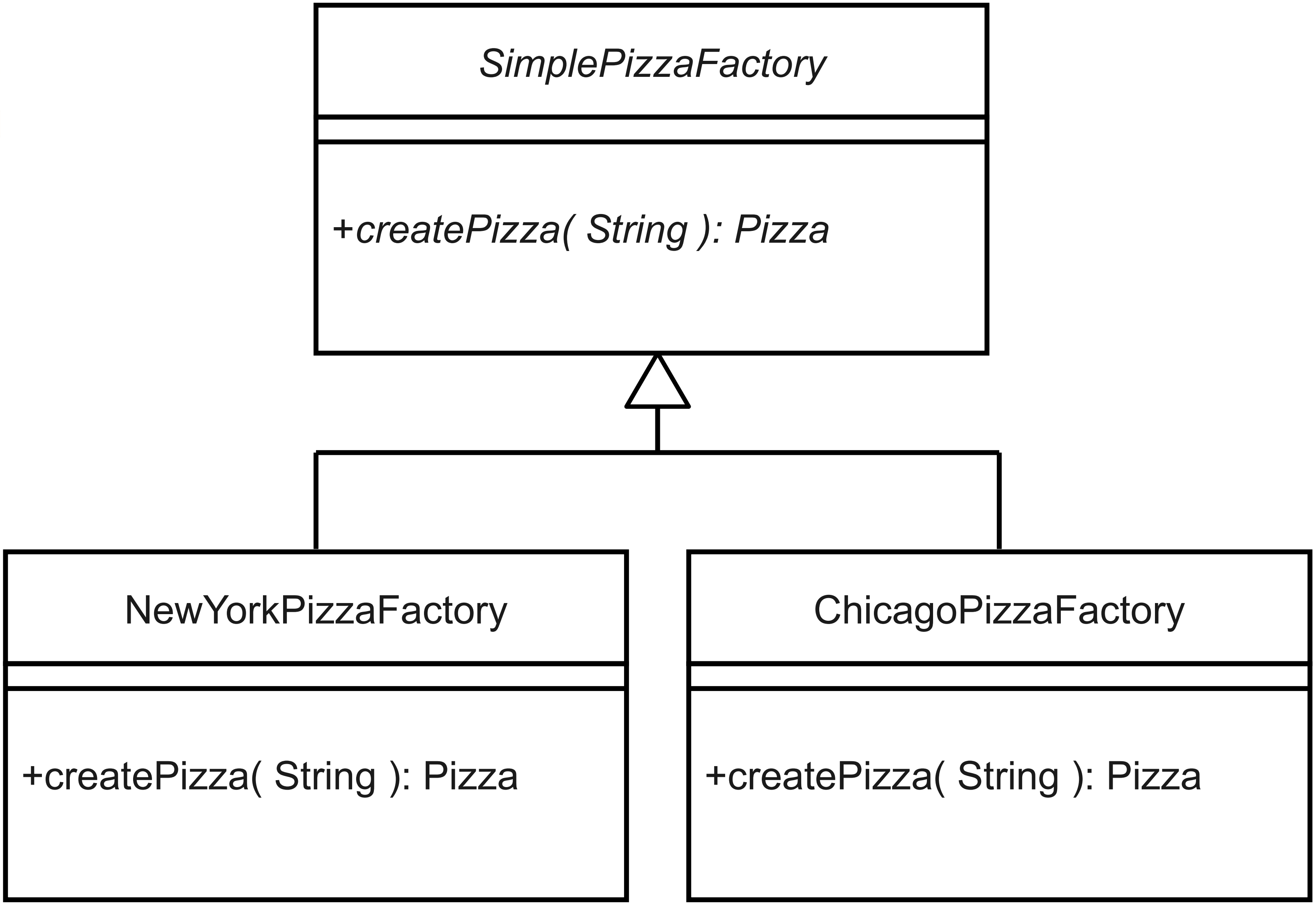
Simple Factory Approach

```
// separate factory class to create a Pizza

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

        if (pizzaType.equals( “pepperoni” ) {
            Pizza pizza = new PepperoniPizza();
        } else if (pizzaType.equals( “veggie” ) {
            Pizza pizza = new VeggiePizza();
        }

        return pizza;
    }
}
```

Using a Factory Object

```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore( SimplePizzaFactory factory ) {
        this.factory = factory;
    }

    public Pizza orderPizza( String pizzaType ) {
        Pizza pizza;

        pizza = factory.createPizza( pizzaType );

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

        return pizza;
    }
}
```

Factories


SimplePizzaFactory
+createPizza(String): Pizza
NewYorkPizzaFactory
+createPizza(String): Pizza
ChicagoPizzaFactory
+createPizza(String): Pizza

Using Factories

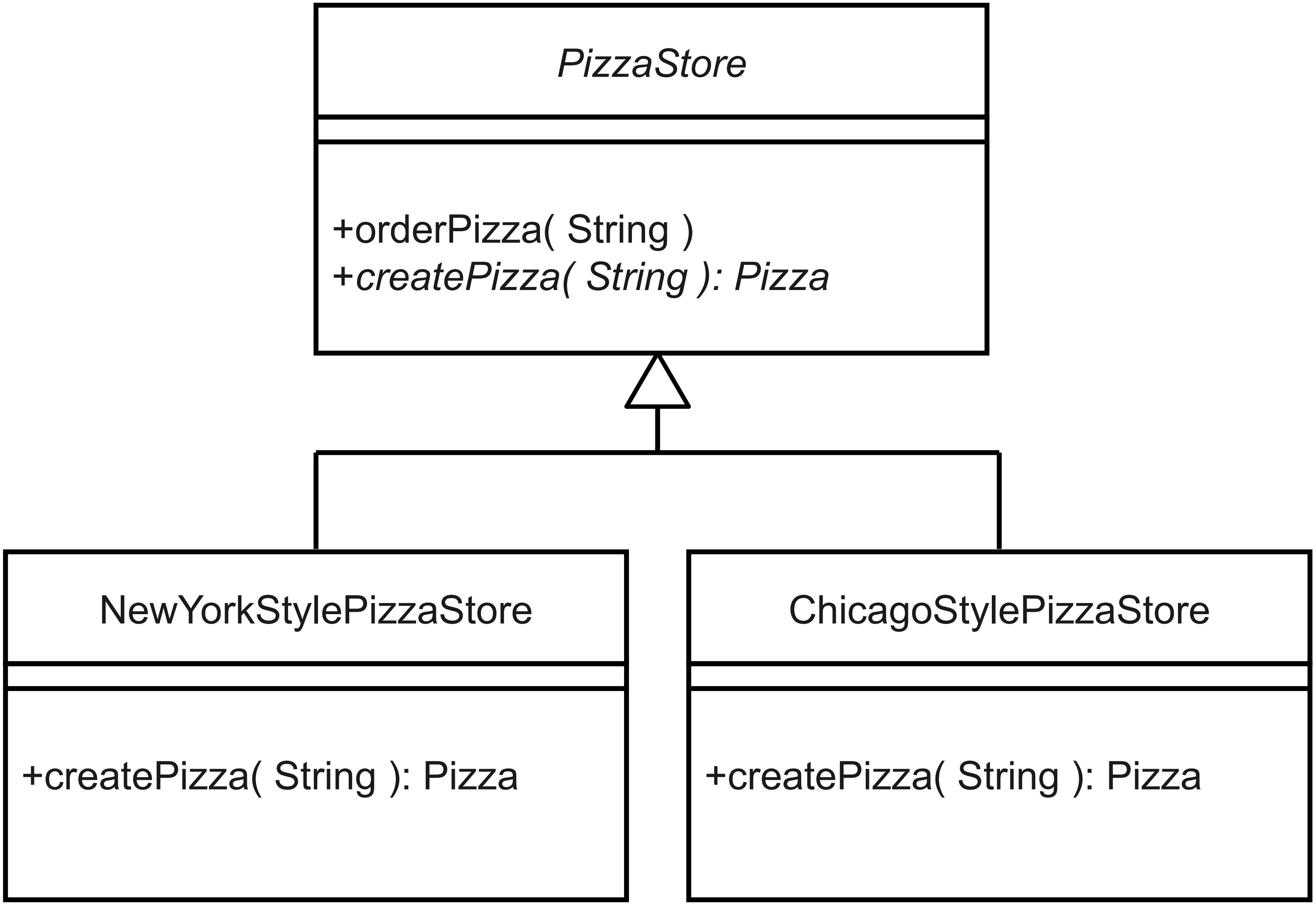
```
PizzaStore newYorkStore = new PizzaStore(
    new NewYorkPizzaFactory()
);
newYorkStore.order( “veggie” );



PizzaStore chicagoStore = new PizzaStore(
    new ChicagoPizzaFactory()
);
chicagoStore.order( “veggie” );
```

Factory Method Approach

keep orderPizza general and decoupled from specific pizza types

```
public abstract class PizzaStore {

    public Pizza orderPizza( String pizzaType ) {
        Pizza pizza;

        pizza = createPizza( pizzaType );

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

        return pizza;
    }

    // defer to subclass to instantiate
    // Pizza of the appropriate type
    public abstract Pizza createPizza(
        String pizzaType );
}
```

factory method


PizzaStore
+orderPizza(String)
+*createPizza(String): Pizza*
NewYorkStylePizzaStore
+createPizza(String): Pizza
ChicagoStylePizzaStore
+createPizza(String): Pizza

Factory Method Approach

```
public class NewYorkStylePizzaStore
    extends PizzaStore {

    public Pizza createPizza( String pizzaType ) {
        if (pizzaType.equals( “pepperoni” ) {
            Pizza pizza =
                new NewYorkStylePepperoniPizza();
        } else if (pizzaType.equals( “veggie” ) {
            Pizza pizza =
                new NewYorkStyleVeggiePizza();
        }
        return pizza;
    }
}
```

Factory Method Pattern

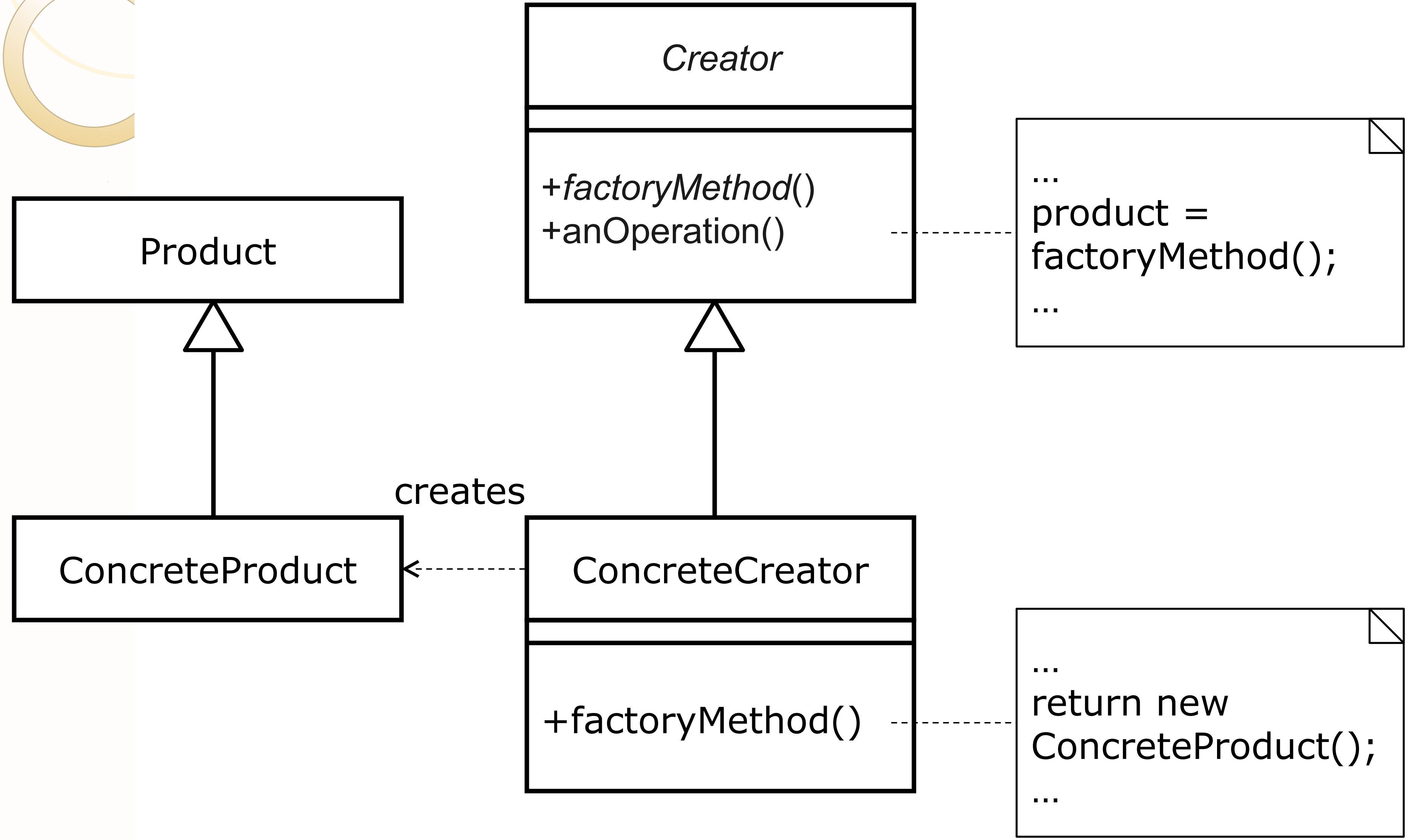
Design intent:

> “define an interface for creating an object, but lets subclasses decide which actual class to instantiate”

```
abstract Product factoryMethod( String type );
```

decouple client code in the superclass from the object creation code in the subclass

Factory Method Structure



…
product =
factoryMethod();
…

…
return new
ConcreteProduct();
…

Exercise

Problem:

designing a framework

- Application and Document superclasses

an actual application would subclass these

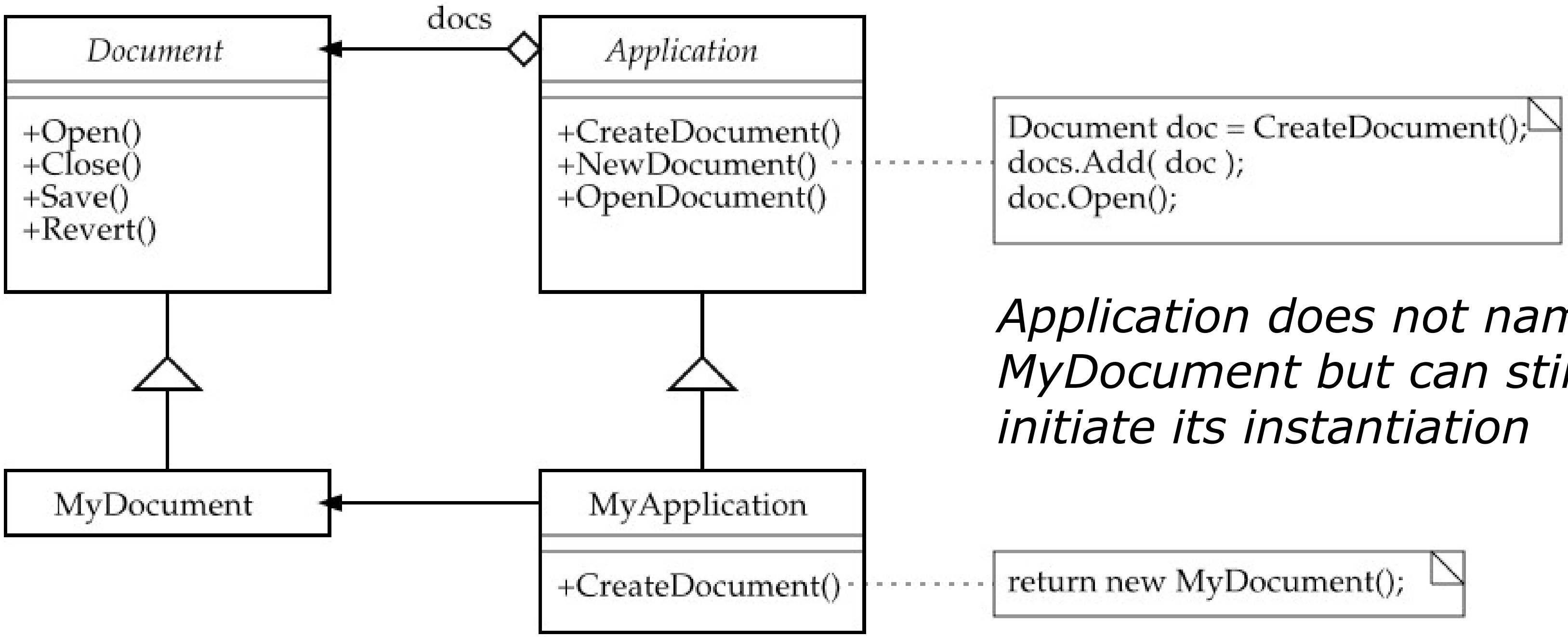
- add MyApplication and MyDocument subclasses
- but do not change the code of the superclasses

write a general NewDocument method in Application that ultimately instantiates a MyDocument

Example Structure

Product

Creator



```
Document doc = CreateDocument();
docs.Add( doc );
doc.Open();
```

Application does not name MyDocument but can still initiate its instantiation

```
return new MyDocument();
```

factory method

also known as Virtual Constructor