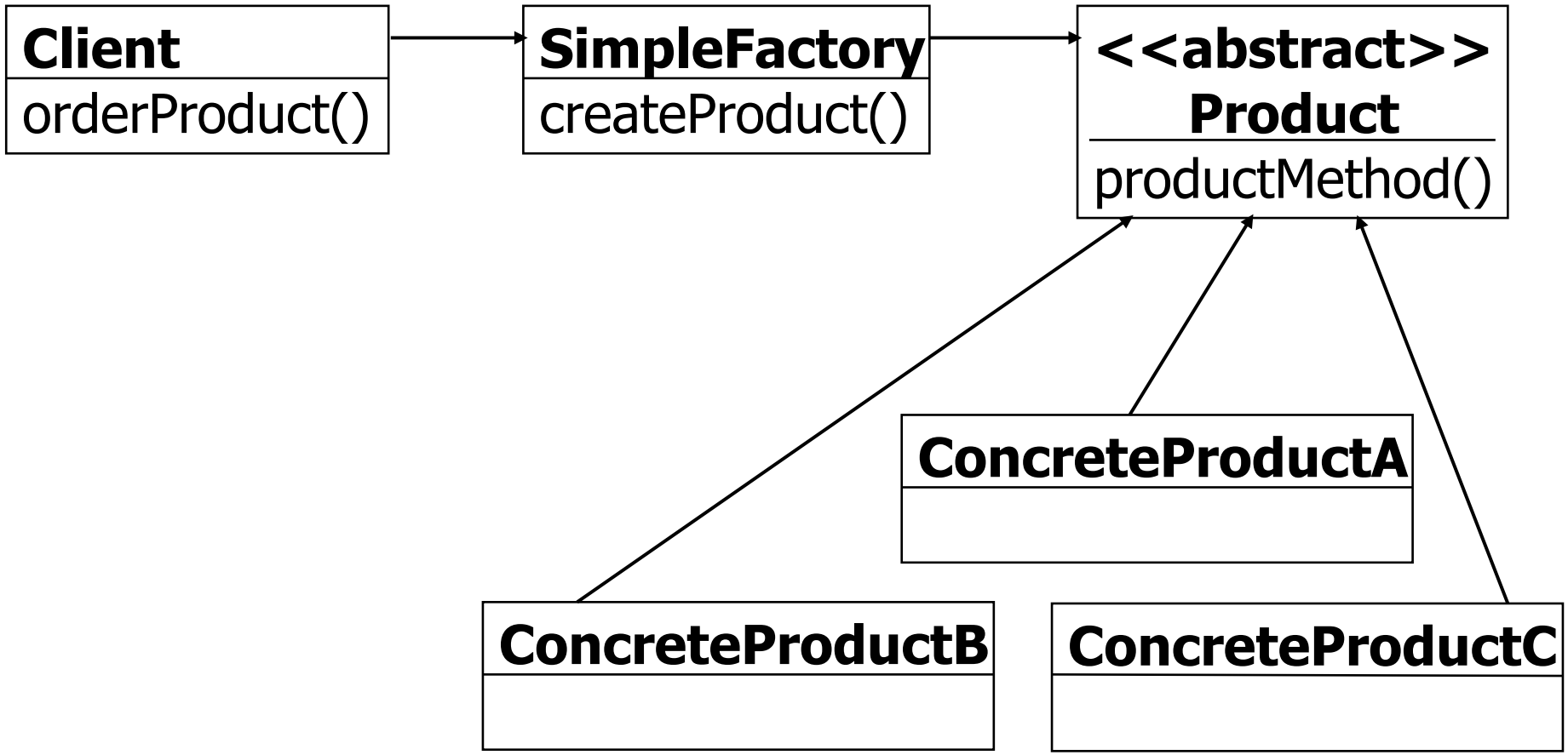
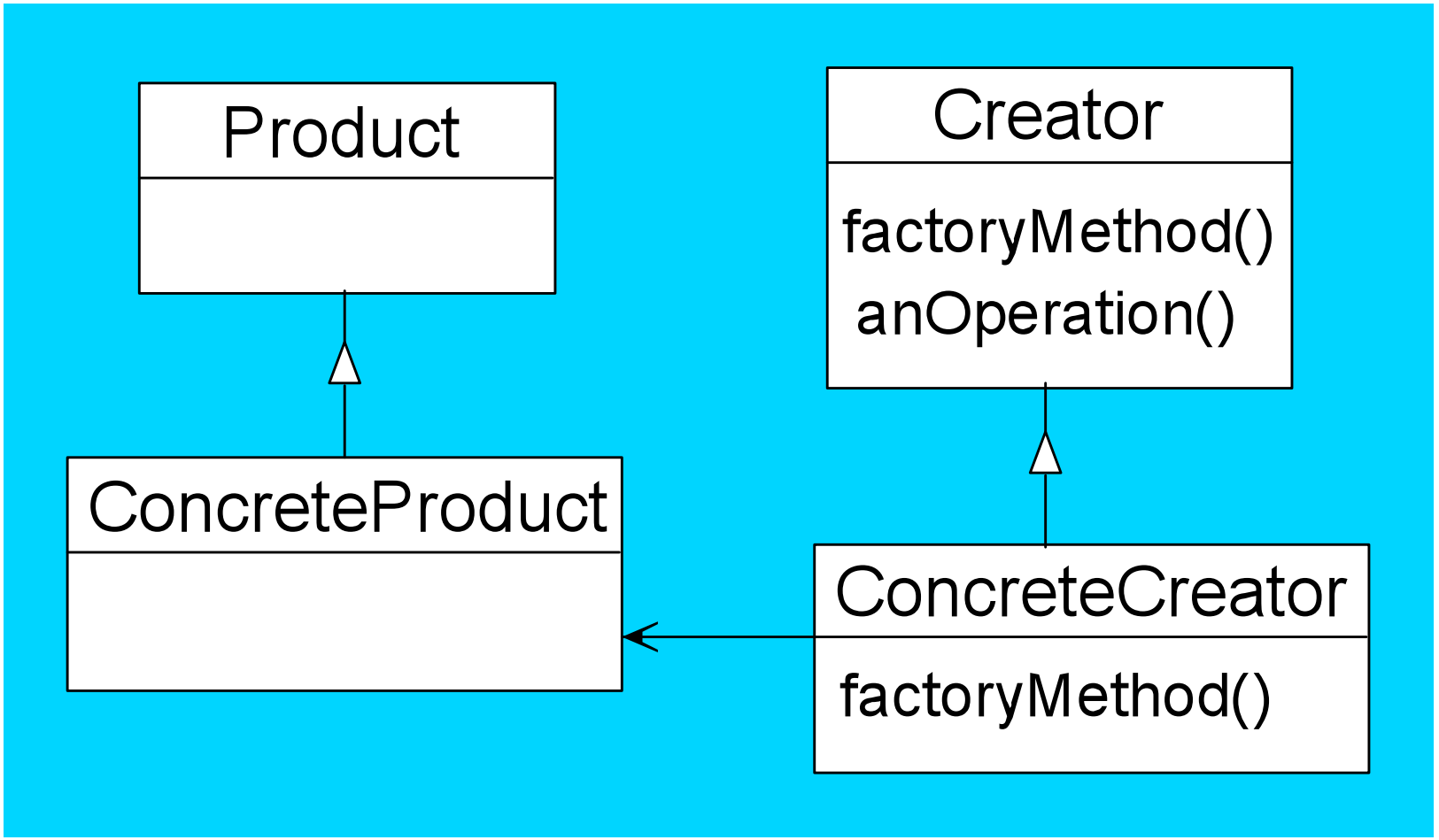
The Abstract Factory Pattern and Singleton Pattern

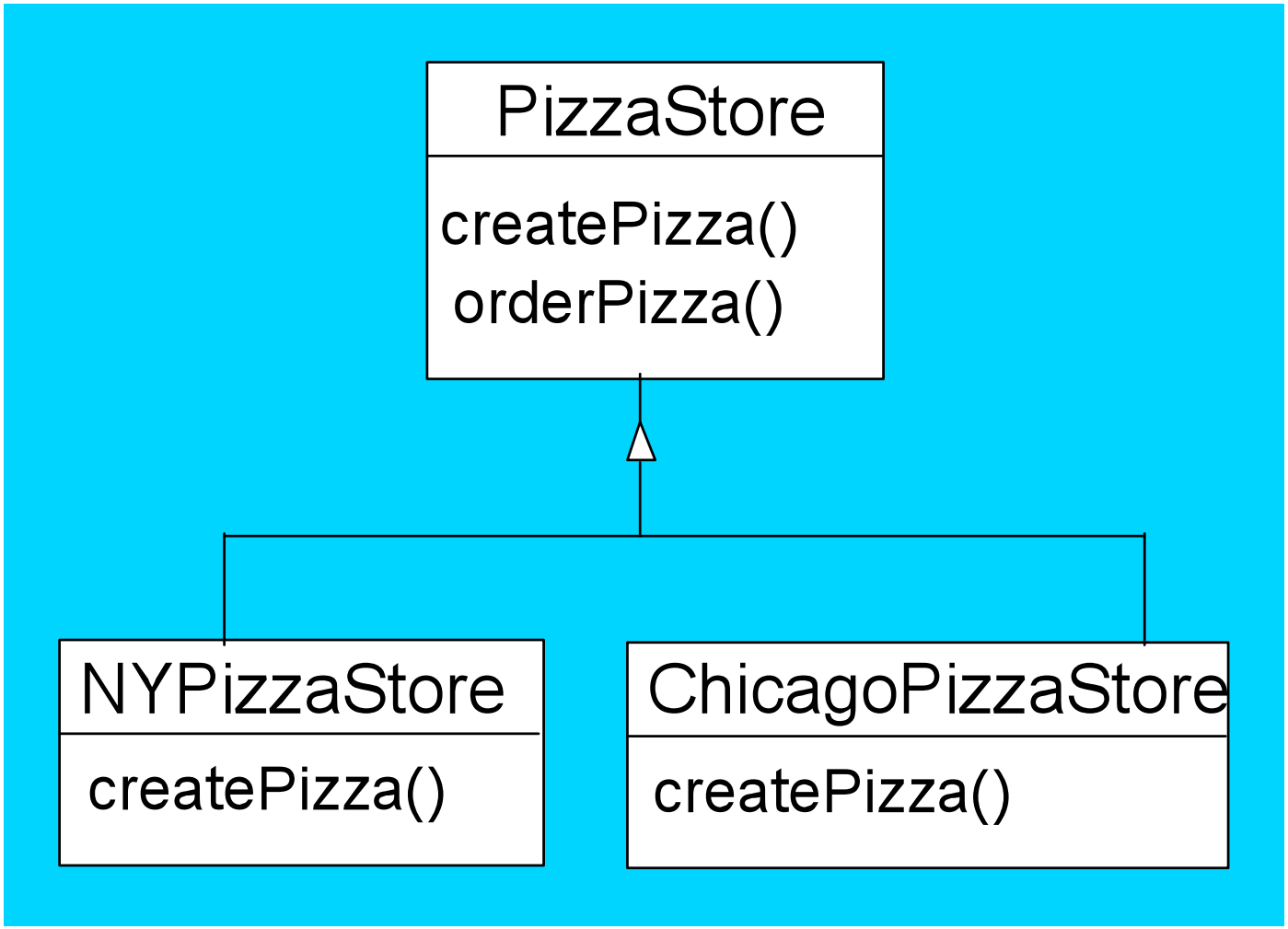
Simple Factory Pattern



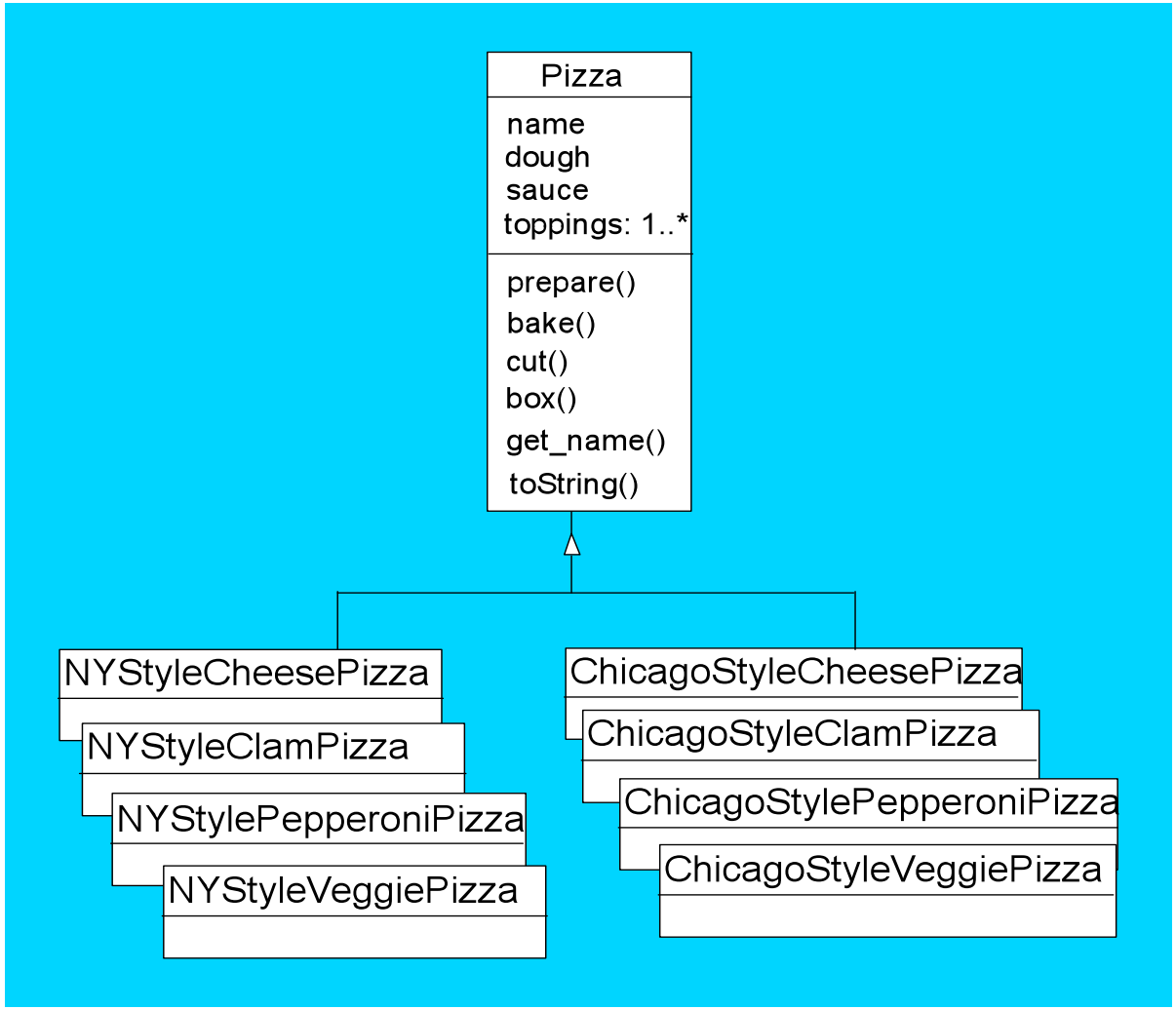
Factory Method Structure



Creator Class

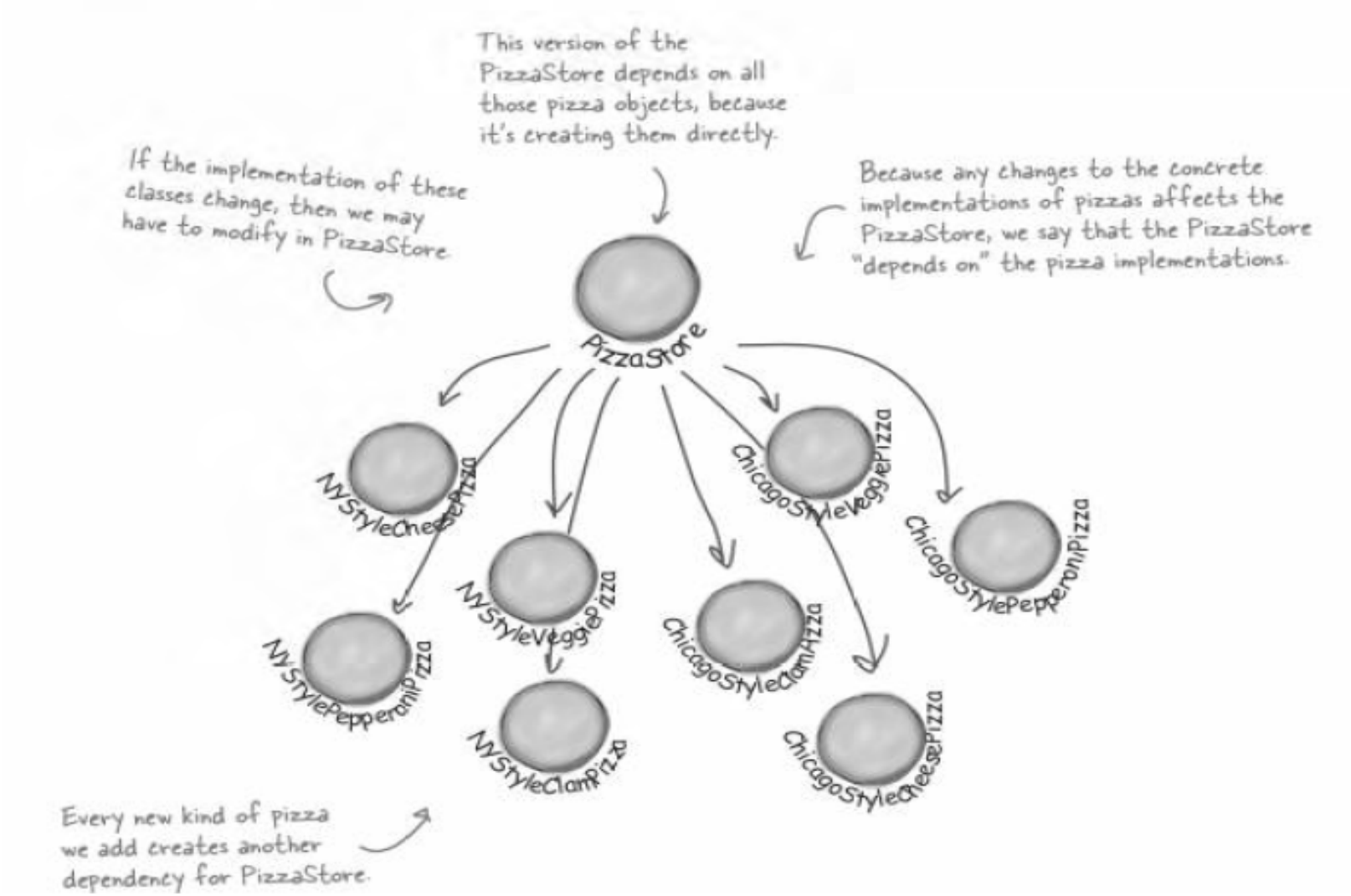


Product Class



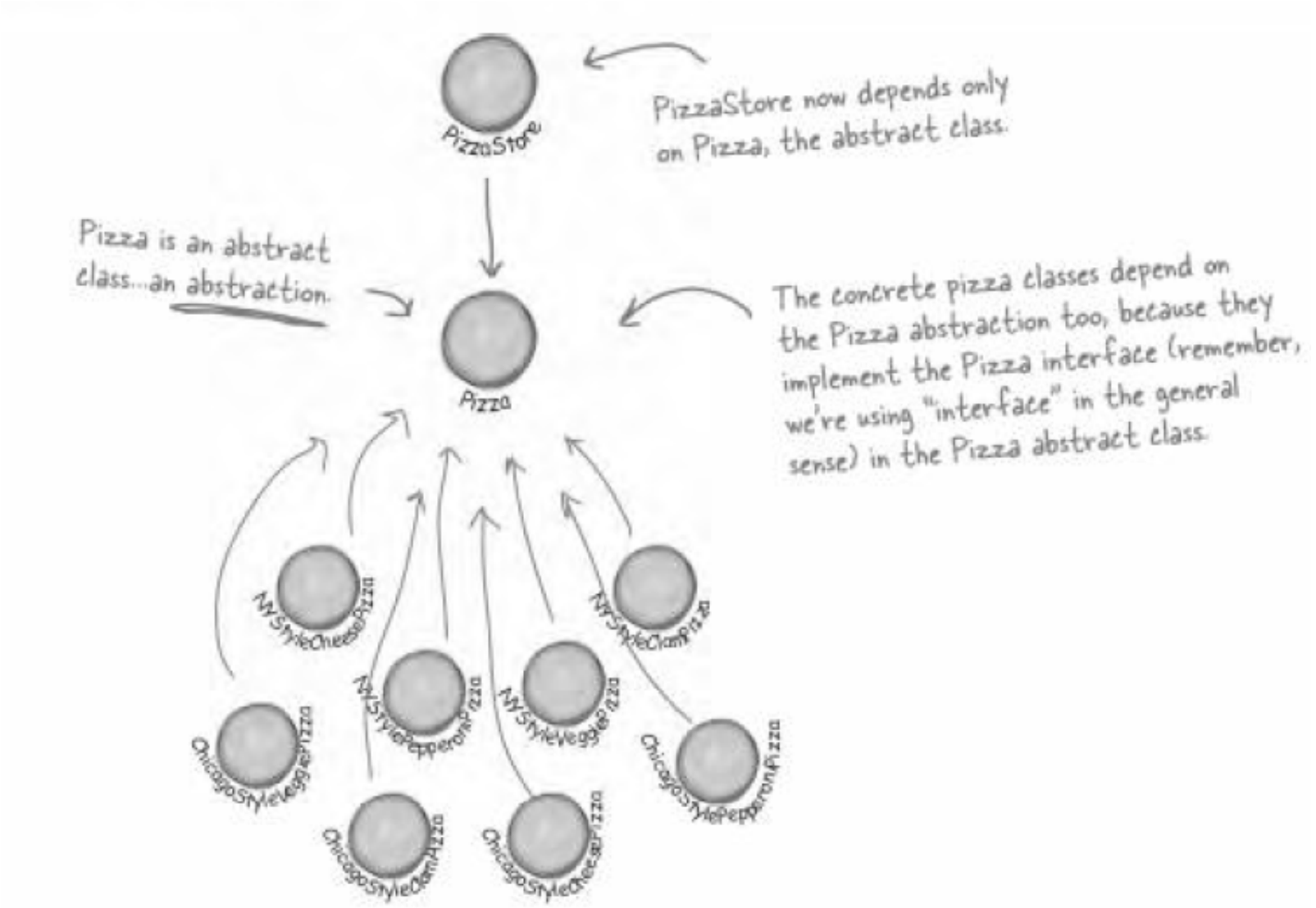
Advantage

- The Creator (PizzaStore) is not tightly coupled to any concrete product (Pizza).
- Instantiating concrete classes is an area of frequent change. By encapsulating it using factories we avoid code duplication (which is a code smell) and make it easier to embrace change during development or to perform maintenance.
- The Factory Pattern also illustrate the principle of coding to abstractions (the Pizza product and the PizzaStore client are abstracts)



- This is possibly a development and maintenance nighmare

- This is what the Factory Method pattern has achieved:

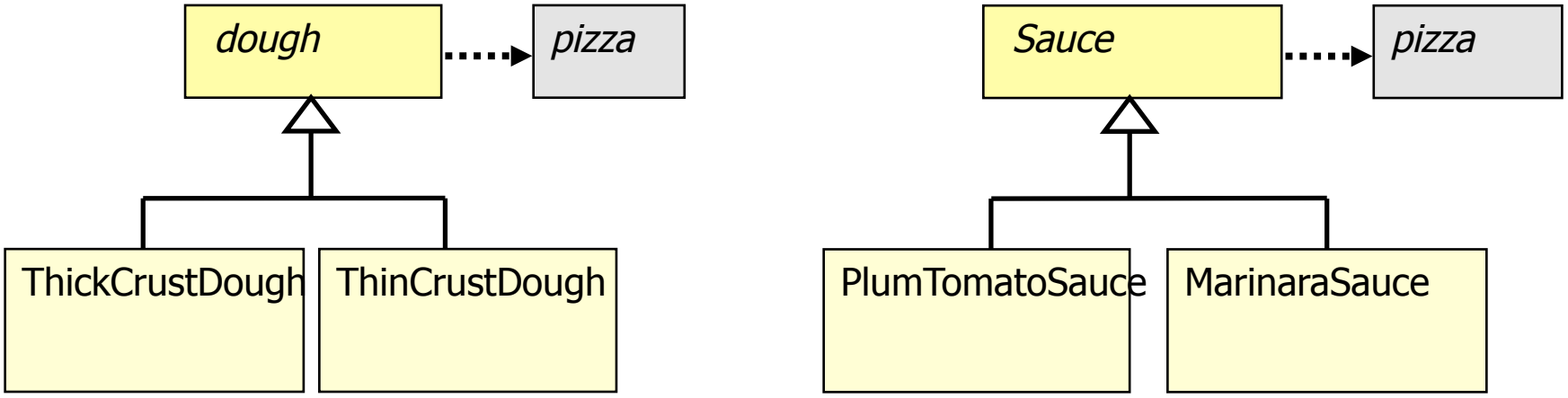


The Abstract Factory

- Our code can now work with different concrete factories, through a *Factory* interface (Pizzastore)
- What if we need to create several types of ”products”, not just a single type?
  - Pizza dough (Thin in NY, Thick in Chicago)
  - Sauce (…)

The Abstract Factory

- Answer seems simple: just use Factory Method pattern twice



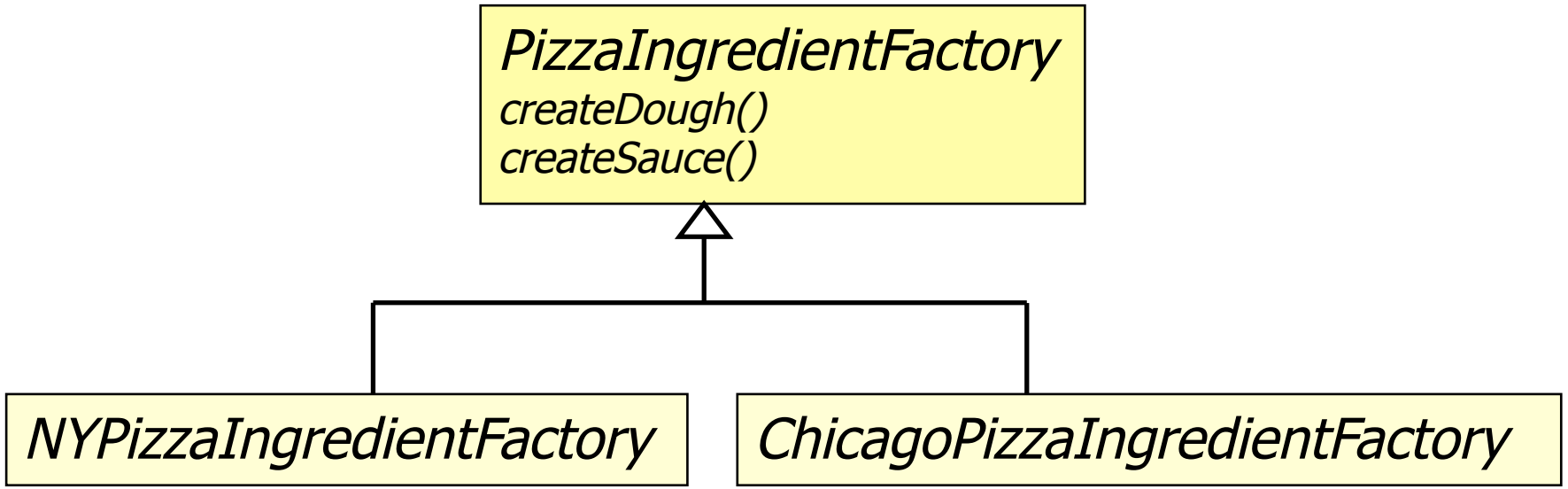
The Abstract Factory

- This looks fine…
- …but does it reflect our intention?
- Would it make sense to have a pizza, with
  - Chicago Thick Crust + NY Marinara Sauce?
- Model does not include any ”binding” between related products

The Abstract Factory

- A ***Dough*** and a ***Sauce*** are not – as seen from a type point-of-view – related
- Would be somewhat artificial – or perhaps even impossible – to introduce a common base class
- However, we can enforce the binding through a shared factory class!

The Abstract Factory



More Ingredience

- We want to list the exact list of ingredients for each concrete pizza. For example :
  - Chicago Cheese Pizza : Plum tomato Sauce, Mozzarella, Parmesan, Oregano;
  - New York Cheese Pizza : Marinara Sauce, Reggiano, Garlic.
- Each “style” uses a different set of ingredients,
- We could change implement Pizza as in:

```
public class Pizza {
   Dough dough;
   Sauce sauce;
   Veggies veggies[];
   Cheese cheese;
   Pepperoni pepperoni;
   Clams clam;
    . . .
}
```

- and the constructor naturally becomes something like :

```
public Pizza(Dough d, Sauce s, Cheese c, Veggies v,
                Pepperoni p, Clams c)
{
  dough = d;
  sauce = s;
  veggies = v;
  cheese = c;
  pepperoni  = p;
  clam = c;
}
```

- but then:

```
public class NYPizzaStore extends PizzaStore {
  Pizza createPizza(String item) {
    if (item.equals("cheese")) {
      return new NYStyleCheesePizza(new ThinCrustDough(),
             new Marinara(), new Reggiano(), null, null );
    } else if (item.equals("veggie")) {
      return new NYStyleVeggiaPizza(new ThinCrustDough(),
            new Marinara(), new Reggiano(), mew Garlic() ,
            null);
    }
  }
```

This will cause a lot of maintenance headaches! Imagine what happens when we create a new pizza!

- We know that we have a certain set of ingredients that are used for New York..yet we have to keep repeating that set with each constructor. Can we define this unique set just once?
- After all we are creating concrete instances of dough, ingredients etc. :
  - let's use the factory of ingredients!

```
public interface PizzaIngredientFactory {

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

- We then “program to the interface” by implementing different concrete ingredients factories. For example here are the ingredients used in New York pizza style:

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

- Our Pizza class will remain an abstract class:

```
public abstract class Pizza {
  protected string name;
  protected Dough dough;
  protected Sauce sauce;
  protected ArrayList toppings = new ArrayList();

  abstract void Prepare(); //now abstract
  public virtual string Bake() {
    Console.WriteLine("Bake for 25 minutes at 350 \n");
  }
  public virtual string Cut() {
    Console.WriteLine("Cutting the pizza into diagonal slices \n");
  }
// ...
}
```

- and our concrete Pizza simply bevome:

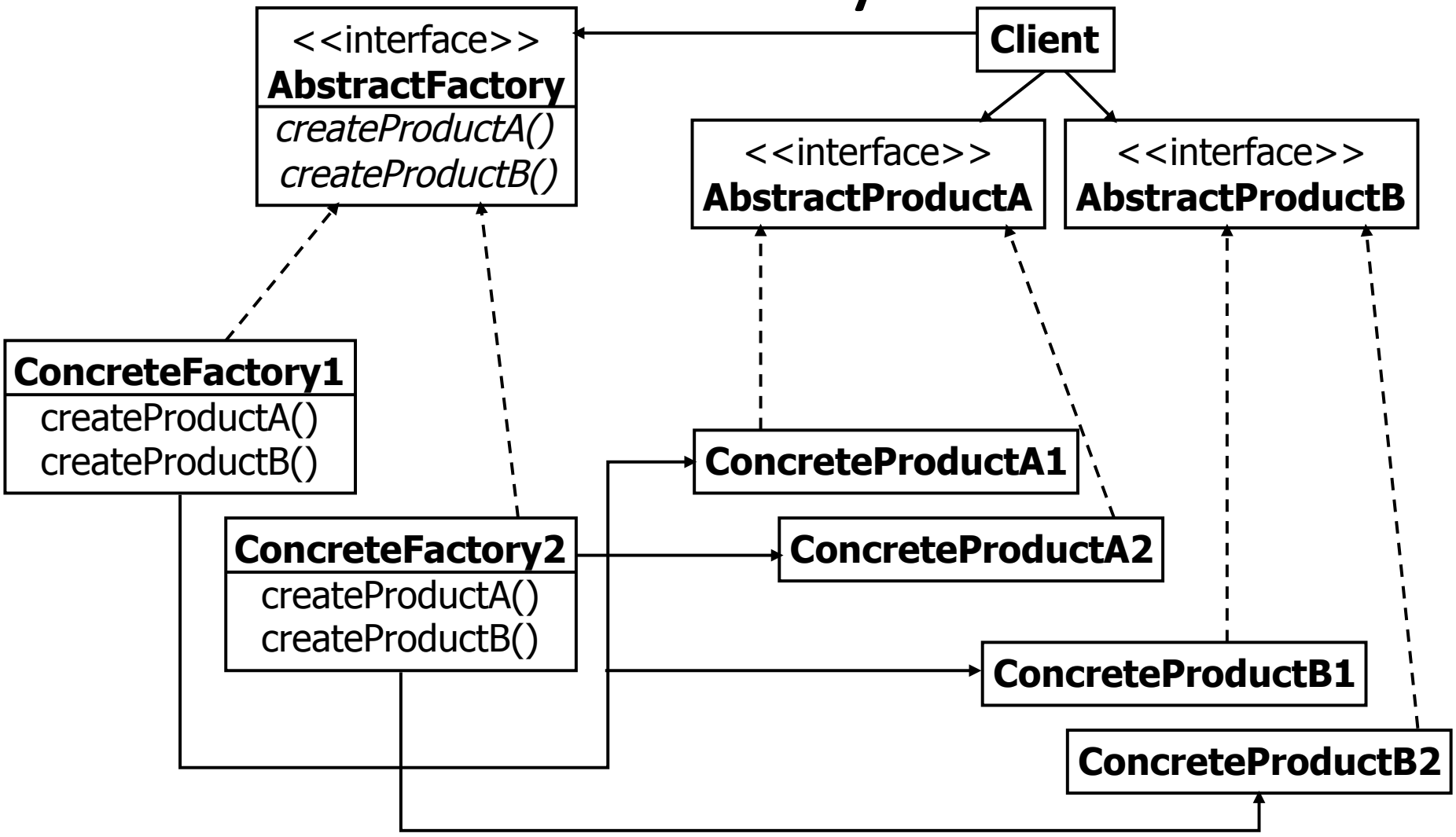
```
public class CheesePizza : Pizza {
   PizzaIngredientFactory ingredientFactory;
   public CheesePizza(PizzaIngredientFactory ingredientFactory) {
    this.ingredientFactory = ingredientFactory;
   }
   void prepare() {
    dough = ingredientFactory.createDough();
    sauce = ingredientFactory.createSauce();
    cheese = ingredientFactory.createCheese();
   }
}
```

the creation of the ingredients is delegated to a factory.

- finally we must have our concrete Pizza store e.g.

```
public class NYPizzaStore : PizzaStore {
   protected Pizza createPizza(String item) {
    Pizza pizza = null;
    PizzaIngredientFactory ingredientFactory =
        new NYPizzaIngredientFactory();
    if (item.equals("cheese")) {
        pizza = new CheesePizza(ingredientFactory);
    } else if (item.equals("veggie")) {
        pizza = new VeggiePizza(ingredientFactory);
    } else if (item.equals("clam")) {
        pizza = new ClamPizza(ingredientFactory);
    } else if (item.equals("pepperoni")) {
        pizza = new PepperoniPizza(ingredientFactory);
    }
    return pizza;
  }
}
```

Abstract Factory Pattern



Abstract Factory Summary

- The Abstract Factory Pattern "provides an interface for creating families of related or dependent objects without specifying their concrete classes".
- Factory Method:
  - Uses inheritance to create a Concrete Product
  - Sub classes decide which Concrete Product to use
- Abstract Factory:
  - Uses composition to create objects
  - The objects created were a part of a family of objects. For example, NY region had a specific set of ingredients.
  - An abstract factory actually contains one or more Factory Methods!

The Abstract Factory

- By making a creator class with several **`create`**… methods, we restrict the product combinations the client can create
- The methods in the **Abstract Factory** are product-type dependent, so if we add another product, we need to change the interface of the base class
- This is a price we must pay for binding (formally) non-related types together

Creating a Single Instance of a Class

- In some cases it maybe necessary to create just one instance of a class.
- This maybe necessary because:
  - More than one instance will result in incorrect program behavior
  - More than one instance will result in the overuse of resources
  - More than one instance will result in inconsistent results
  - There is a need for a global point of access
- How would you ensure that just one instance of a class is created?

Singleton Pattern Overview

- In some cases there should be at most one instance of a class.
- This one instance must be accessible by all “clients”, e.g. a printer spooler.
- This usually occurs when a global resource has to be shared.
- The singleton pattern ensures that a class has only one instance and provides only one point of entry.

Implementing the Singleton Pattern

- Implement a private constructor to prevent other classes from declaring more than one instance.
- Implement a method to create a single instance. Make this method static.
- Create a lazy instance of the class in the class.
- Make the data element static.

Thread Example

A A dual processor machine, with two threads calling the *getInstance()* method for the chocolate boiler

Thread 1

```
public stat ChocolateBoiler
        getInstance()

if (uniqueInstance == null)

uniqueInstance =
       new ChocolateBoiler()

return uniqueInstance;
```

Thread 2

```
public stat ChocolateBoiler
        getInstance()

if (uniqueInstance == null)

uniqueInstance =
       new ChocolateBoiler()

return uniqueInstance;
```

Problems with Multithreading

- In the case of multithreading with more than one processor the getInstance() method could be called at more or less the same time resulting in to more than one instance being created.
- Possible solutions:
  - Synchronize the getInstance() method
  - Do nothing if the getInstance() method is not critical to the application.
  - Move to an eagerly created instance rather than a lazily created one.

Synchronizing the *getInstance()* Method

- Code

```
public static synchronized Singleton getInstance()
{…
}
```

- Disadvantage – synchronizing can decrease system performance by a factor of 100.

Use an Eagerly Created Instance Rather than a Lazy One

- Code:

```
//Data elements
public static Singleton uniqueInstance = new
   Singleton()

private Singleton() {}

public static Singleton getInstance() {
   return uniqueInstance
}
```

- Disadvantage – Memory may be allocated and not used.

Singleton Pattern Summary

- The singleton pattern ensures that there is just one instance of a class.
- The singleton pattern provides a global access point.
- The pattern is implemented by using a private constructor and a static method combined with a static variable.
- Possible problems with multithreading.
- Versions of Java earlier that 1.2 automatically clear singletons that are not being accessed as part of garbage collection.