Intro to Design Patterns:
The Strategy Pattern
The Factory Pattern

CSCI-3081: Program Design and Development

(Slides adapted from Head First Design Patterns)

Now, remember our Ducks…

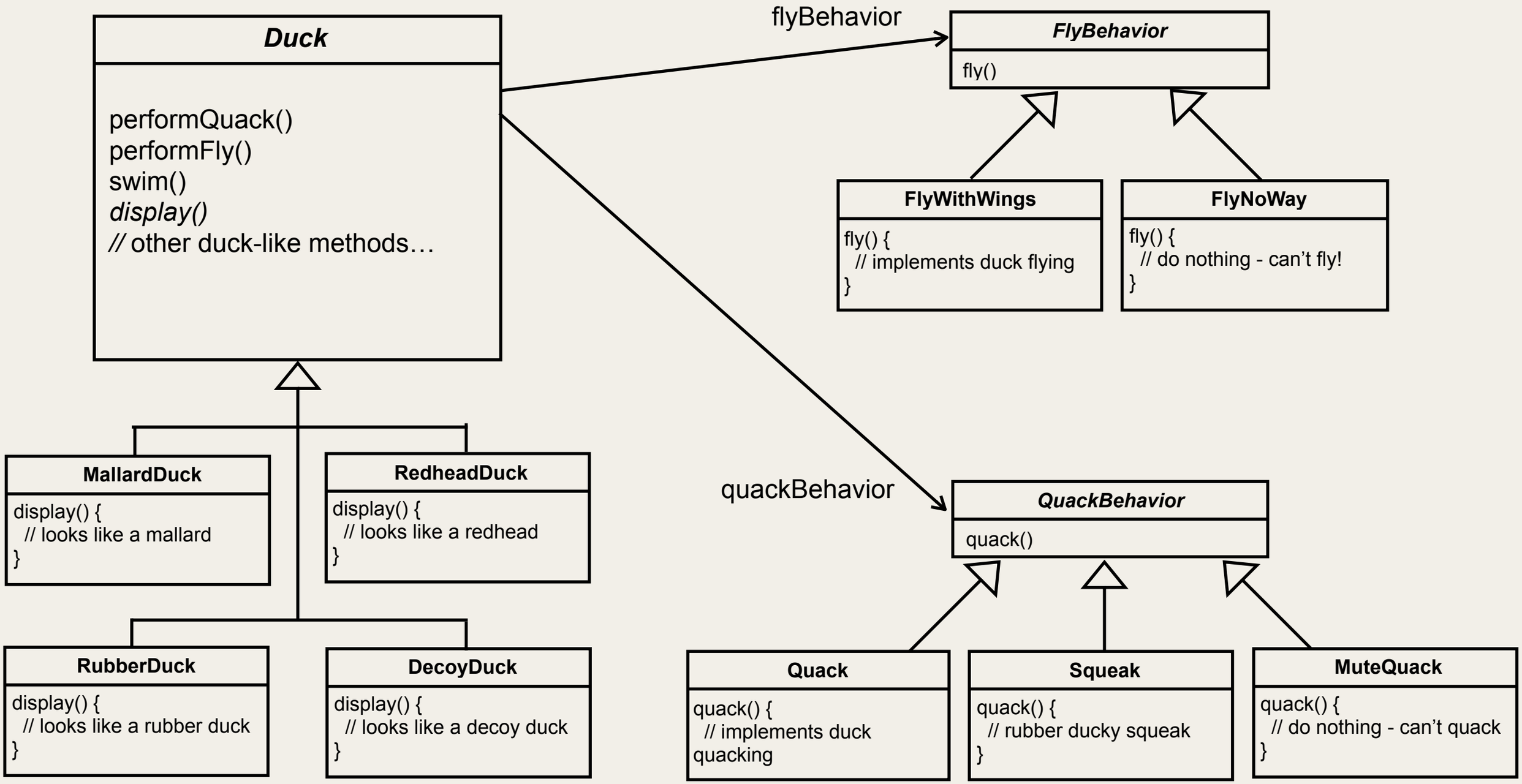
UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Your Design Challenge

How else could you design this program?

- Add a new type of duck - DecoyDuck

Our Recommended Solution:



Client

Duck

performQuack()
performFly()
swim()
display()
// other duck-like methods…

MallardDuck

display() {
  // looks like a mallard
}

RedheadDuck

display() {
  // looks like a redhead
}

RubberDuck

display() {
  // looks like a rubber duck
}

DecoyDuck

display() {
  // looks like a decoy duck
}

Encapsulated fly behavior

flyBehavior

FlyBehavior

fly()

FlyWithWings

fly() {
  // implements duck flying
}

FlyNoWay

fly() {
  // do nothing - can't fly!
}

Encapsulated quack behavior

quackBehavior

QuackBehavior

quack()

Quack

quack() {
  // implements duck quacking
}

Squeak

quack() {
  // rubber ducky squeak
}

MuteQuack

quack() {
  // do nothing - can't quack
}

Think of these as a family of interchangeable algorithms!

If we think of these duck "behaviors" as "algorithms":

- What we have here is a very powerful design approach.
- We can substitute any fly or quack algorithm, and we can even change the algorithm to use on the fly at runtime.

- This is an example of…

The Strategy Pattern

- **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Why do I care? What is a Design Pattern?

- Design patterns help you design software by recognizing common design scenarios.
- If you know the 5-10 most important/frequent design patterns, then you can often recognize that a software design problem can be solved using these (or a combination of them), and then this can often lead you to a very nice software design.
- Great, can I just download an open source library of design patterns?
- No, design patterns are at a higher level than source code. We're talking about patterns that you can recognize using your brain when you come across some new design challenge.

Example Design Activity

1. Arrange the classes.
2. Identify one abstract class, one interface, and eight classes.
3. Draw arrows between them, using the correct style arrows for “has a” and “is a”.

Character
WeaponBehavior weapon;
fight();

KnifeBehavior

useWeapon() {
// implements cutting with a knife
}

BowAndArrowBehavior

useWeapon() {
// implements shooting an arrow with a bow
}

WeaponBehavior
useWeapon();

Queen

SwordBehavior

useWeapon() {
// implements cutting with a sword
}

King

AxeBehavior

useWeapon() {
// implements cutting with an axe
}

Troll

Knight

The Factory Pattern

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Hungry?

```
Pizza* PizzaStore::orderPizza(std::string type) {
  Pizza *pizza;

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

  pizza->prepare();
  pizza->bake();
  pizza->cut();
  pizza->box();

  return pizza;
}
```

What's wrong with this (if anything)?

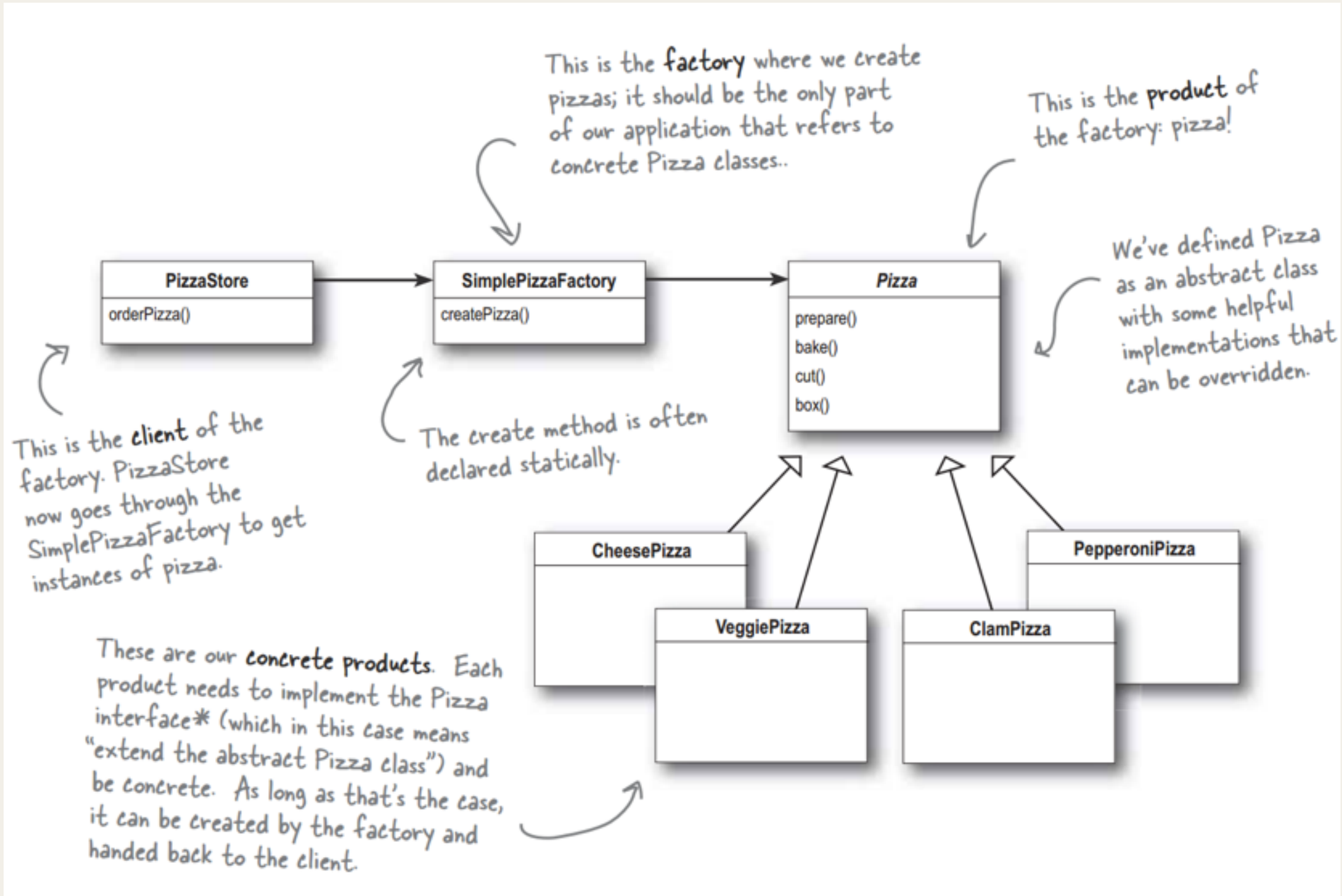
```
Pizza* PizzaStore::orderPizza(std::string type) {
  Pizza *pizza;

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

  pizza->prepare();
  pizza->bake();
  pizza->cut();
  pizza->box();

  return pizza;
}
```

```
Pizza* PizzaStore::orderPizza(std::string type) {
  Pizza *pizza;

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

  pizza->prepare();
  pizza->bake();
  pizza->cut();
  pizza->box();

  return pizza;
}
```

An Alternative Design

- Isolate change: Pull all the object creation code out of the Pizza class.
- Encapsulate object creation in its own class.
- The only purpose of this new class is to create pizzas. It is the only place in the program where pizzas can be created.
- We call this special kind of class a Factory, e.g., SimplePizzaFactory.

SimplePizzaFactory



Inside the Factory

```
class SimplePizzaFactory {

public:

  Pizza* createPizza(std::string type) {
    Pizza *pizza = NULL;

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

    return pizza;
  }

};
```

The Reworked PizzaStore class

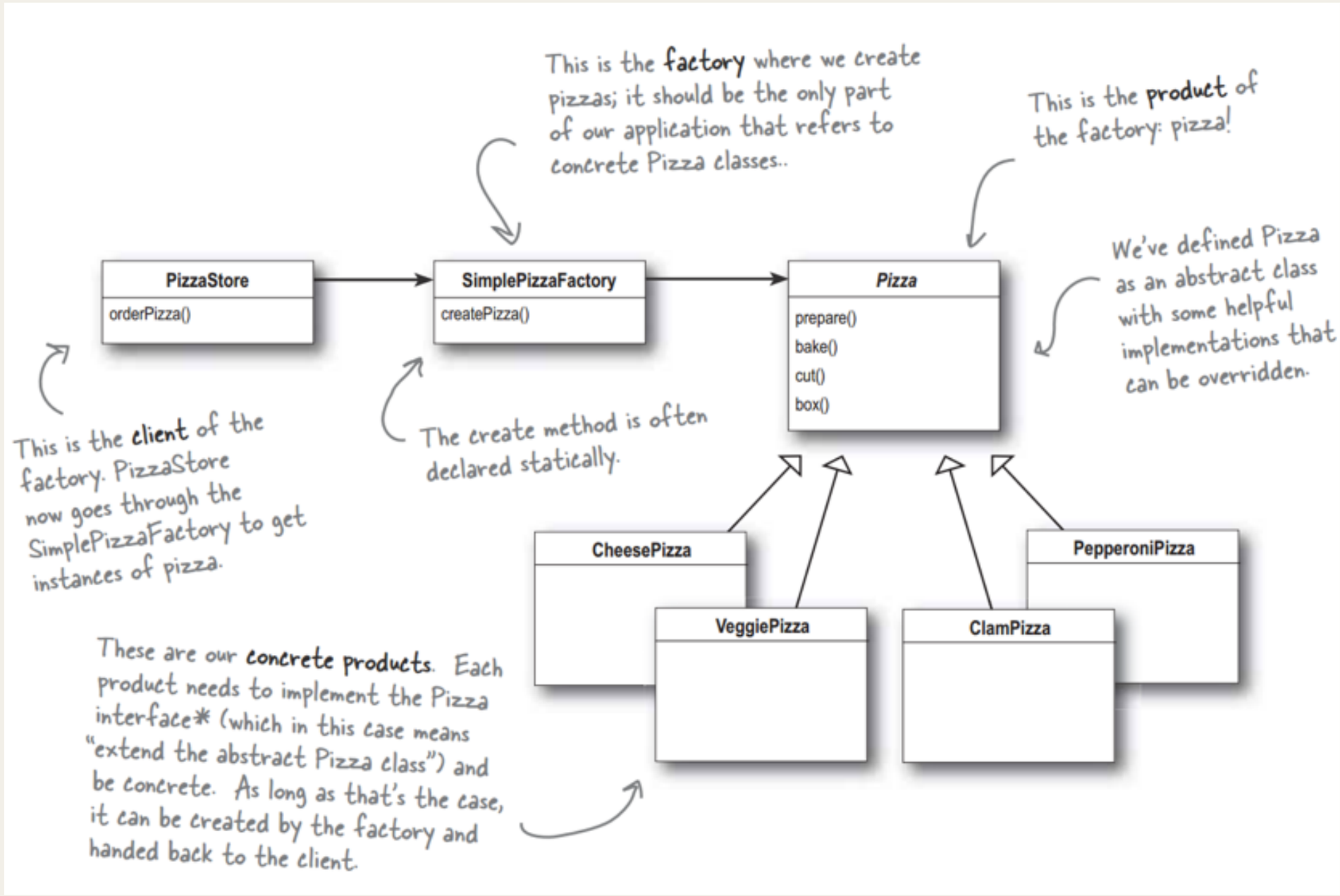
```
class PizzaStore {

public:

  PizzaStore(SimplePizzaFactory *factory) {
    m_factory = factory;
  }

  ~PizzaStore() {}

  Pizza* orderPizza(std::string type) {

    Pizza *pizza = m_factory->createPizza(type);

    pizza->prepare();
    pizza->bake();
    pizza->cut();
    pizza->box();

    return pizza;
  }

private:

  SimplePizzaFactory *m_factory;
};
```

Extensions to keep in mind — we'll discuss these in detail later in the course.

- Making the factory a **static** class.
- Using **enum**s rather than a string as the parameter.

What if you have stores in NYC, Chicago, and California, each with their own variations on Cheese, Veggie, Clam, and Pepperoni?

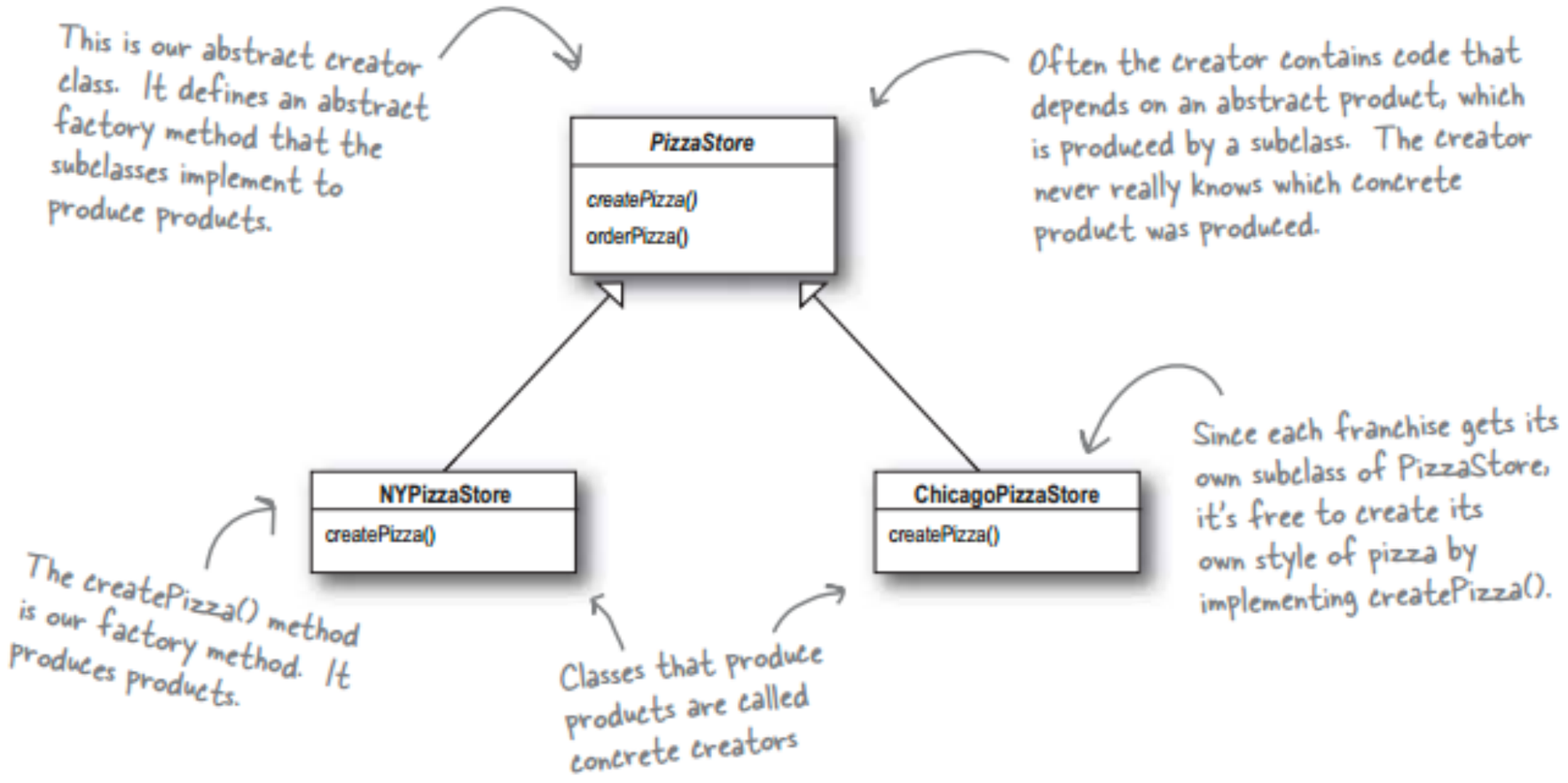
- Can you change this design to work for a Pizza franchise?



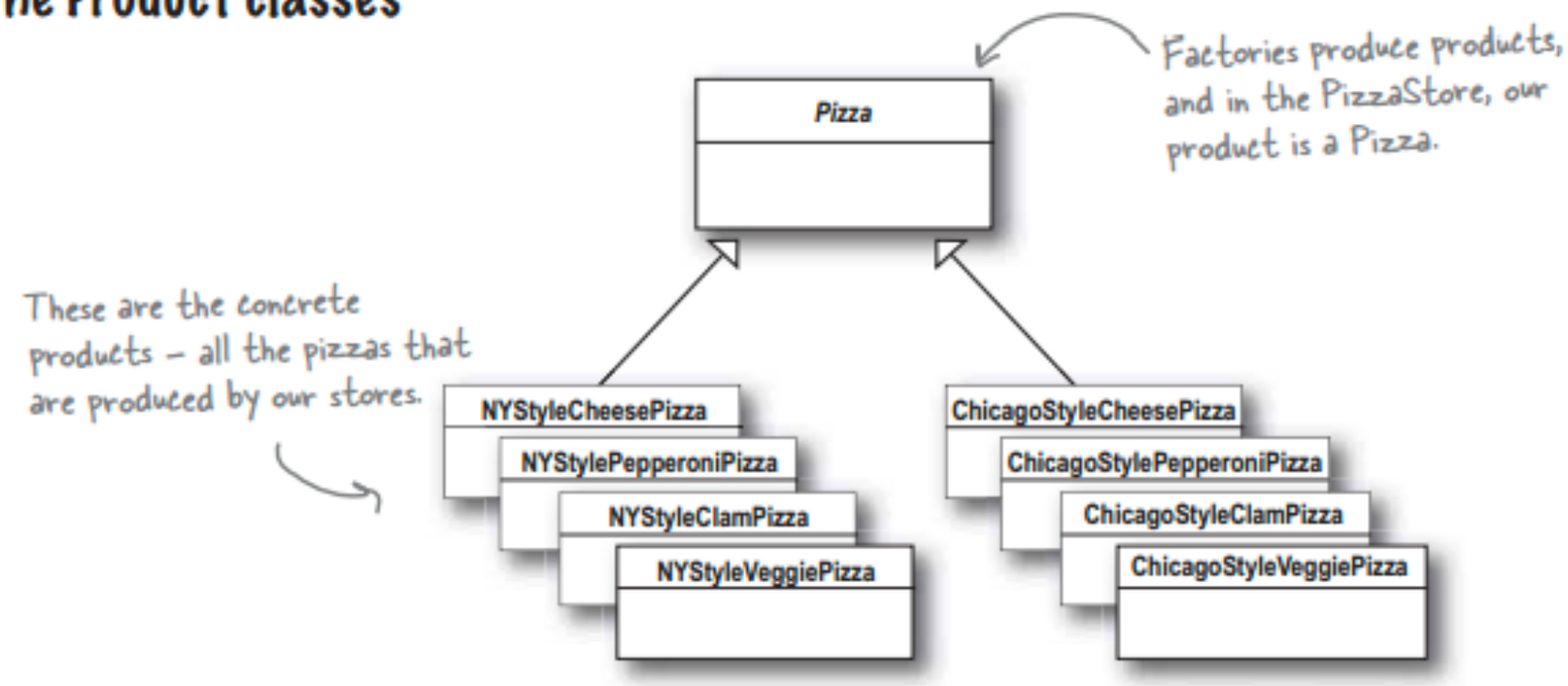
Suggested Design: Use a Factory Method

- This is roughly the same idea as the factory class that we just built, but here the factory is an abstract method that subclasses implement.
- So, a PizzaStore knows that it needs to createPizzas, but since each store creates pizzas in a slightly different way, we leave it to the subclasses to do the actual creation.

The Creator classes



The Product classes



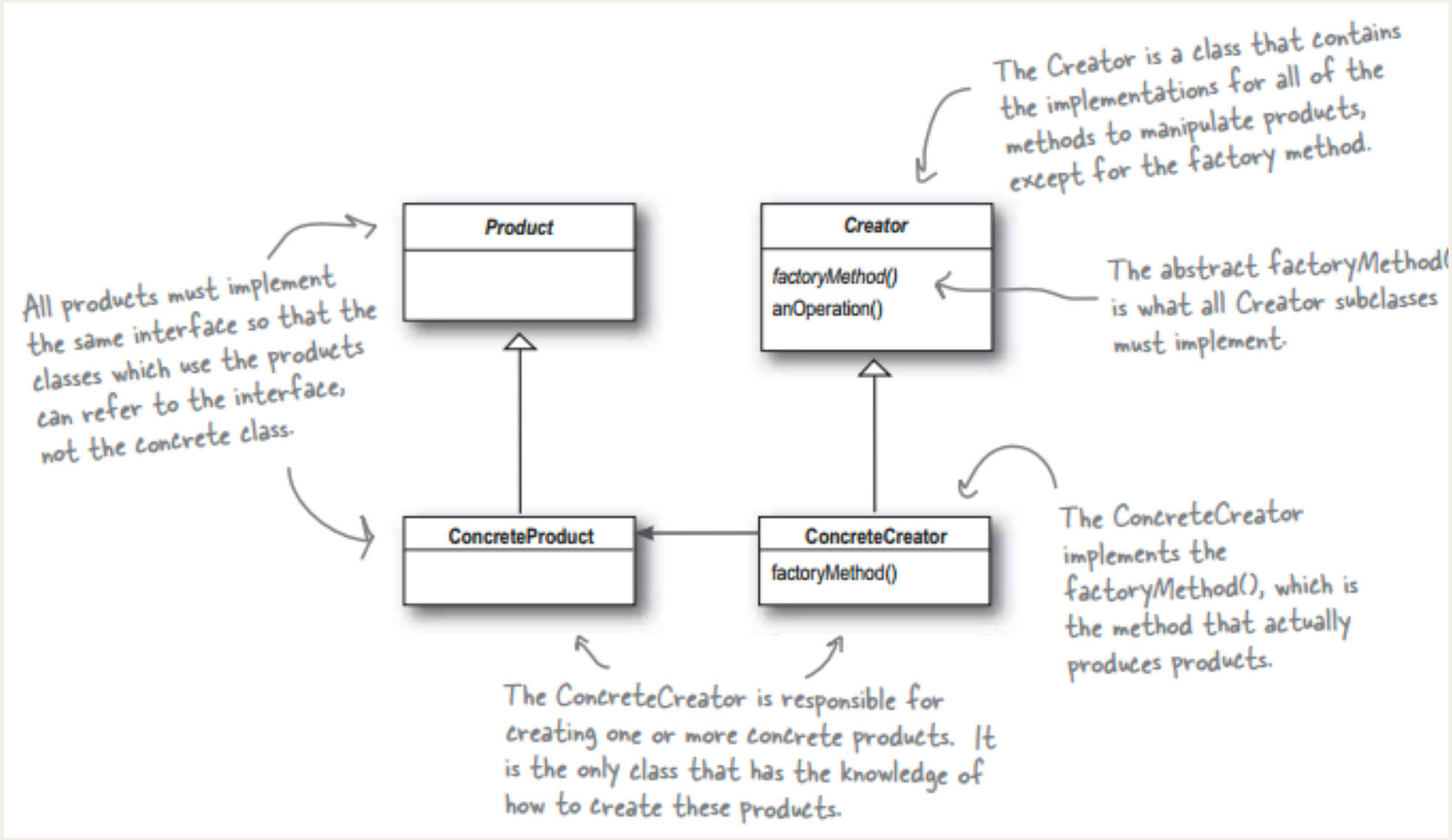
What are the advantages of this?

- The key is that PizzaStore can continue handling all Pizzas the same way. The code to order a pizza is the same regardless of the type of store or the type of pizza.

```
class PizzaStore {

public:

  PizzaStore() {}
  virtual ~PizzaStore() {}


  // This method is pure abstract
  virtual Pizza* createPizza(std::string type) = 0;


  // This method calls createPizza, but as long as it gets
  // a pizza back it doesn't matter which specific
  // PizzaStore creates that pizza.
  Pizza* orderPizza(std::string type) {

    Pizza *pizza = createPizza(type);

    pizza->prepare();
    pizza->bake();
    pizza->cut();
    pizza->box();

    return pizza;
  }

};
```

The Factory Method Pattern

- **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate.



Summary of Two Types of Factories

- **Simple Factory:**
  - You identify that the code that is changing a lot has to do with figuring out which type of object (e.g., Pizza) to create because you keep needing to add new Pizzas and take away old Pizzas.
  - Move all code that creates objects Pizzas to a single Factory class. This is the only place in your application where Pizzas are created.
  - This isolates change and encapsulates object creation, so this is a good programming design decision.
- **Factory Method:**
  - If we have more variation in our program, for example, multiple PizzaStores that each create their own versions of pizzas. Then, we need a bit more flexibility.
  - Using a factory method is a good choice here because it has the same advantages of the simple factory while also allowing us to vary the products we create based on the subclass we are in.