The Factory Pattern

The Factory Pattern and the problem with **new**

- We are wanting to program to interfaces, not implementations
- By programming to interfaces, new types that are of the type of the interface can seamlessly be added to the system
- Every time we use **new**, we program to a concrete implementation

An example of where **new** is bad

```
Duck* duck;

if (picnic) {
   duck = new MallardDuck( );
} else if (hunting) {
   duck = new DecoyDuck( );
} else if (inBathTub) {
   duck = new RubberDuck( );
}
```

If we add a new situation to this, e.g., playWithDog, we'll need to modify this code to add another *else if* branch

Pretty much makes a mockery of closed to change

Moreover, because this code often appears it multiple places, not fixing them all will lead to bugs

We return to our first principle: Identify the aspects that vary and separate them from what stays the same.

An example using a pizza shop (See Example1 code)

```
Pizza* PizzaStore::orderPizza( ) {
   Pizza* pizza = new Pizza( );
   pizza->prepare( );
   pizza->bake( );
   pizza->cut( );
   pizza.box( );
   return pizza;
}
```

Since we cannot create an abstract class object, we’re forced to make a concrete object

But the shop sells many kinds of pizza (See Example2 code)

```
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( ); // bake, etc.
}
```

Instantiates a concrete object that implements the Pizza abstract class based on the value of the string *type*

Each pizza knows how to prepare itself, i.e., how to prepare, bake, cut and box itself

We have not separated the parts that can change

```
Pizza 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;
}
```

orderPizza is not closed to modification.

Stuff that will change whenever the types of pizza that are sold changes. Every time a pizza is dropped or added we have to change this code.

Stuff that has been done this way for hundreds of years, and is unlikely to change.

Let’s encapsulate the object creation

```
Pizza 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->prepare( );
pizza->bake( );
pizza->cut( );
pizza->box( );
return *pizza;
}
```

Pull out the code that is likely to change and put it into a new class

We’ll call a *factory* method on an object of the new class, passing the type of the pizza.

`orderPizza` will no longer care about different kinds of pizza.

Let’s encapsulate the object creation (See Example3 code)

```
Pizza SimplePizzaFactory::createPizza(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( );
  }
  return *pizza;
}
```

All pizza objects are created by the createPizza method

Code taken from orderPizza(…)

What is the advantage of this?

- This appears to have just pushed the complexity off into another class
- Advantages:
  - Methods better follow the “each method does one thing” rule
  - Code that is likely to change is separated from code that is unlikely to change
  - If Pizza objects could be created in multiple places, there is now only one piece of pizza creation code to maintain
  - Some code uses a static method. Why not here?
    - Both are common
    - Static methods allow createPizza to be called without instantiating an object
    - Static methods cannot be overridden

Updating the pizza ordering class

```
class PizzaStore {
public:
  PizzaStore(SimplePizzaFactory);
  virtual Pizza* orderPizza(std::string);
private:
  SimplePizzaFactory* factory;
};
```

```
Pizza* PizzaStore::orderPizza(std::string type) {
  Pizza* pizza = factory->createPizza(type);
  pizza->prepare( );
  pizza->bake( );
  pizza->cut( );
  pizza->box( );
  return pizza;
}
```

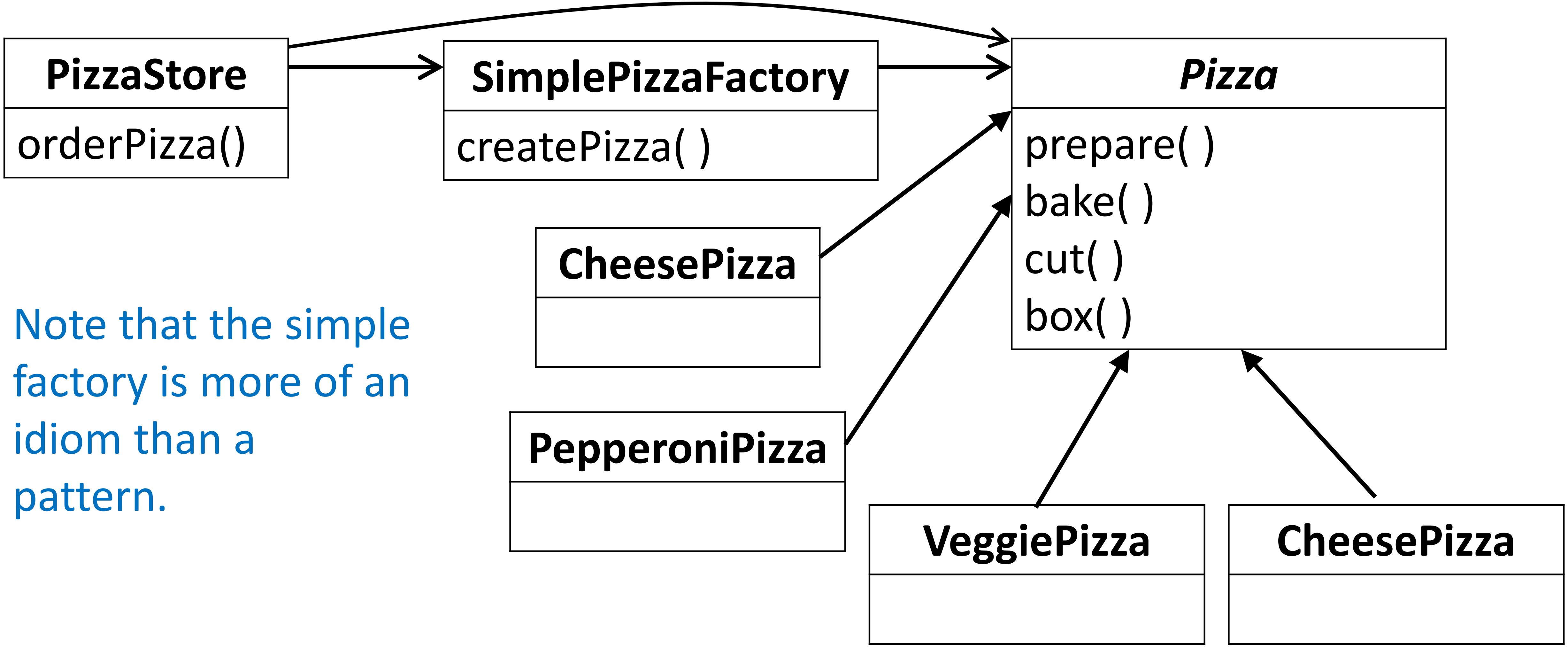
The factory method is used to create a pizza of a style supported by the factory

```
PizzaStore::PizzaStore(SimplePizzaFactory* _factory) {
  factory = _factory;
}
```

A factory is passed in. Different factories can make different styles of pizza

Note that there are no new calls on concrete implementations, i.e., no concreate instantiations. All programming is to the Pizza interface.

The simple factory



Note that the simple factory is more of an idiom than a pattern.

Design Principles

- Encapsulate what varies
- Favor composition over inheritance
- Program to an interface, not an implementation
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Depend on abstractions. Do not depend on concrete classes.