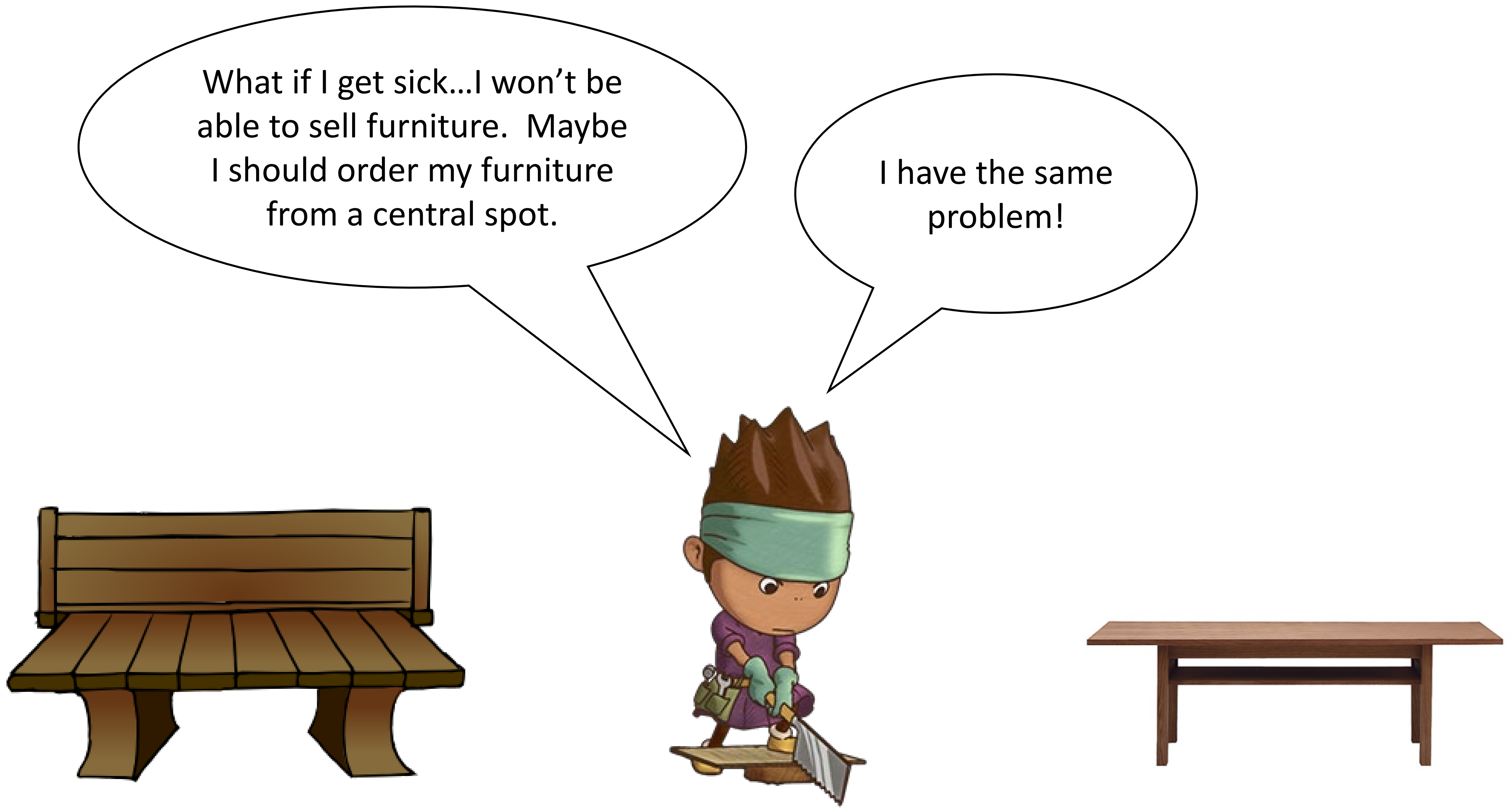
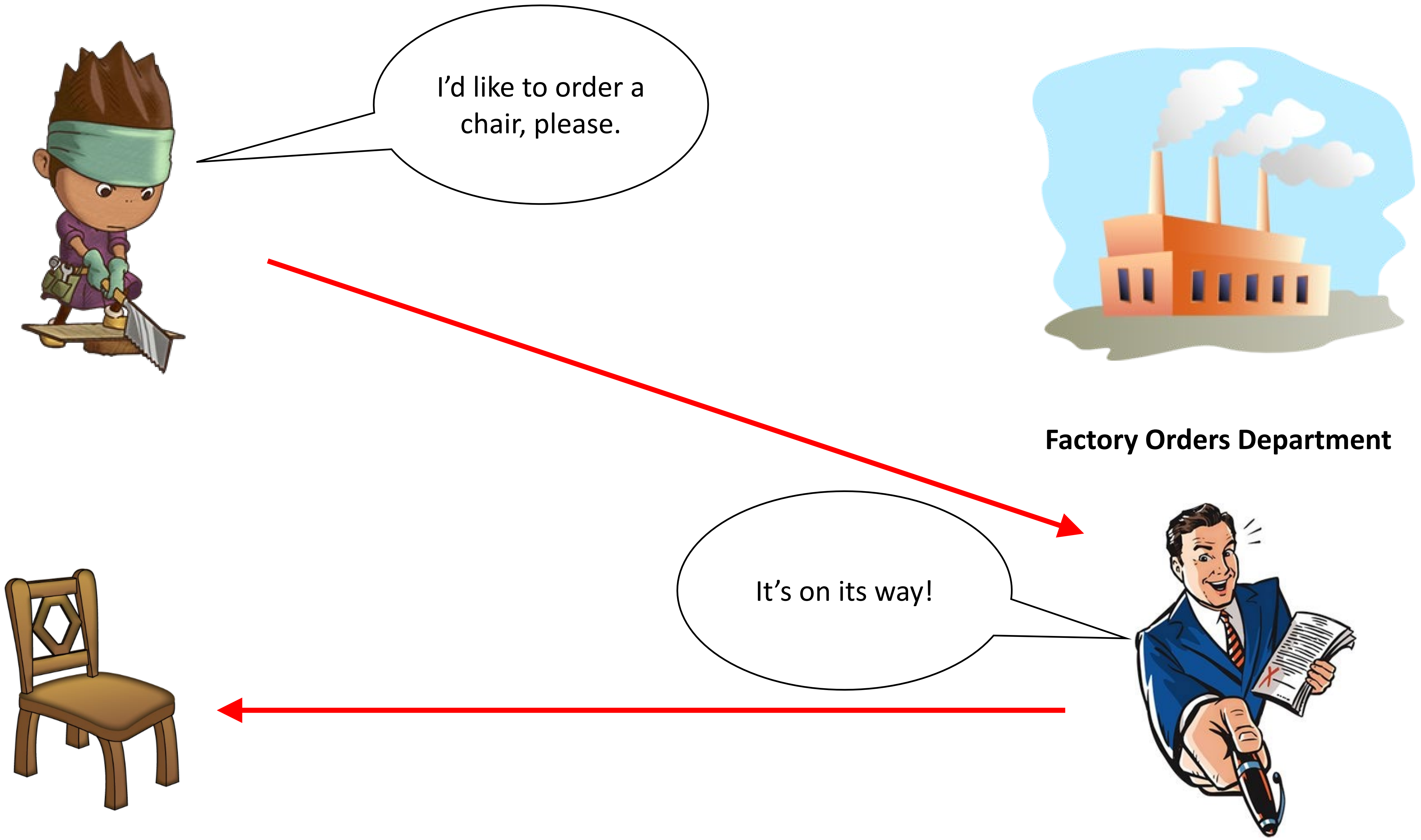
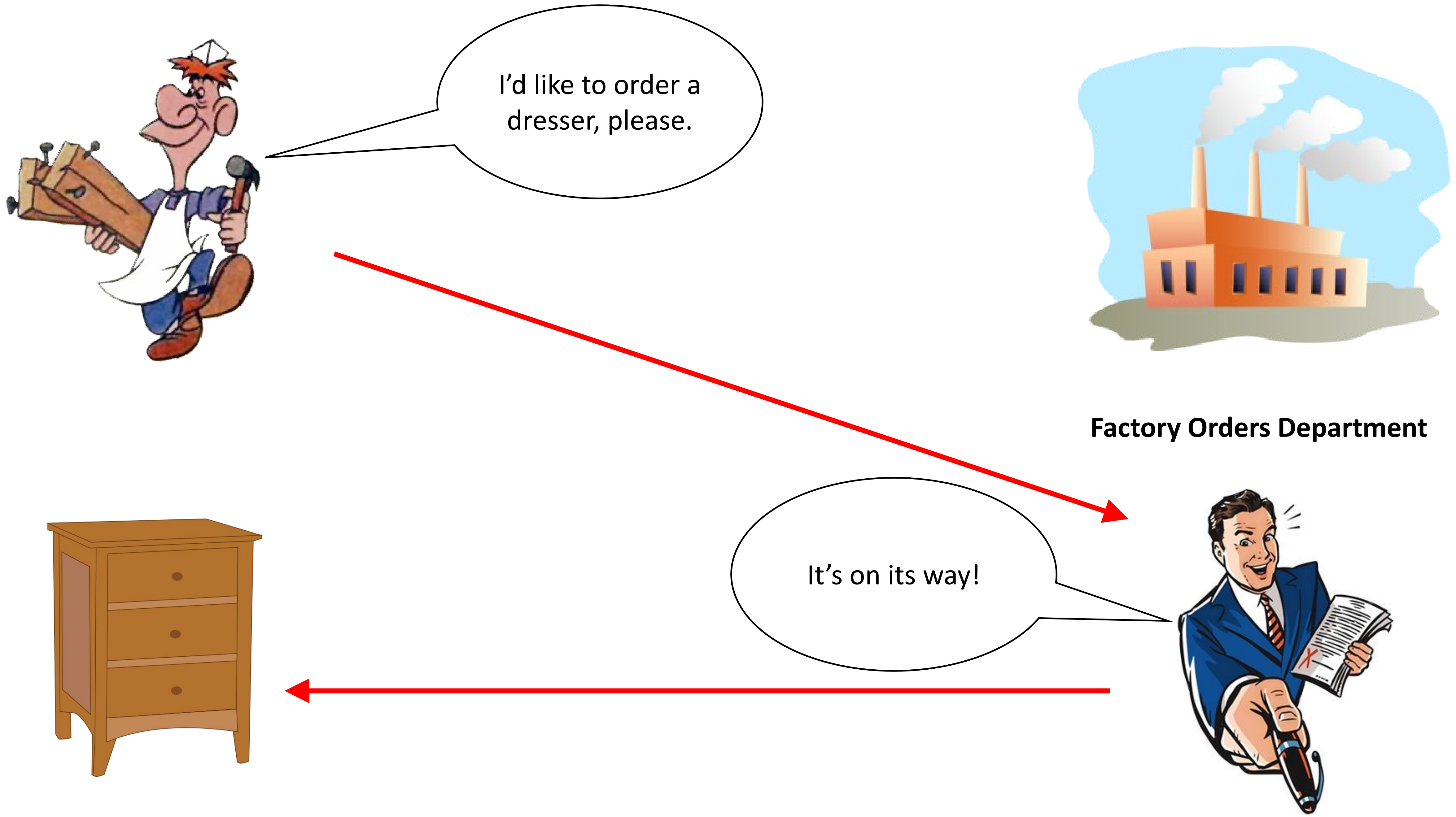
The kinds of furniture I create is tightly-coupled to me.
I can make all kinds of furniture for my shop!


What if I get sick…I won't be able to sell furniture. Maybe I should order my furniture from a central spot.
I have the same problem!


I'd like to order a chair, please.
Factory Orders Department
It's on its way!


I'd like to order a dresser, please.
Factory Orders Department
It's on its way!

# The Factory Method Pattern

## Theory

## Definition

*The factory method pattern is a design pattern that allows for the creation of objects without specifying the type of object that is to be created in code. A factory class contains a method that allows determination of the created type at run-time.*

- This is a *creational* pattern as it is used to control class instantiation.
- The factory pattern is used to replace class constructors, **abstracting the process of object generation** so that the type of the object instantiated can be determined at run-time.

## Intent

- Defines an interface for creating objects, but let subclasses to decide which class to instantiate
- Refers to the newly created object through a common interface

## Benefits

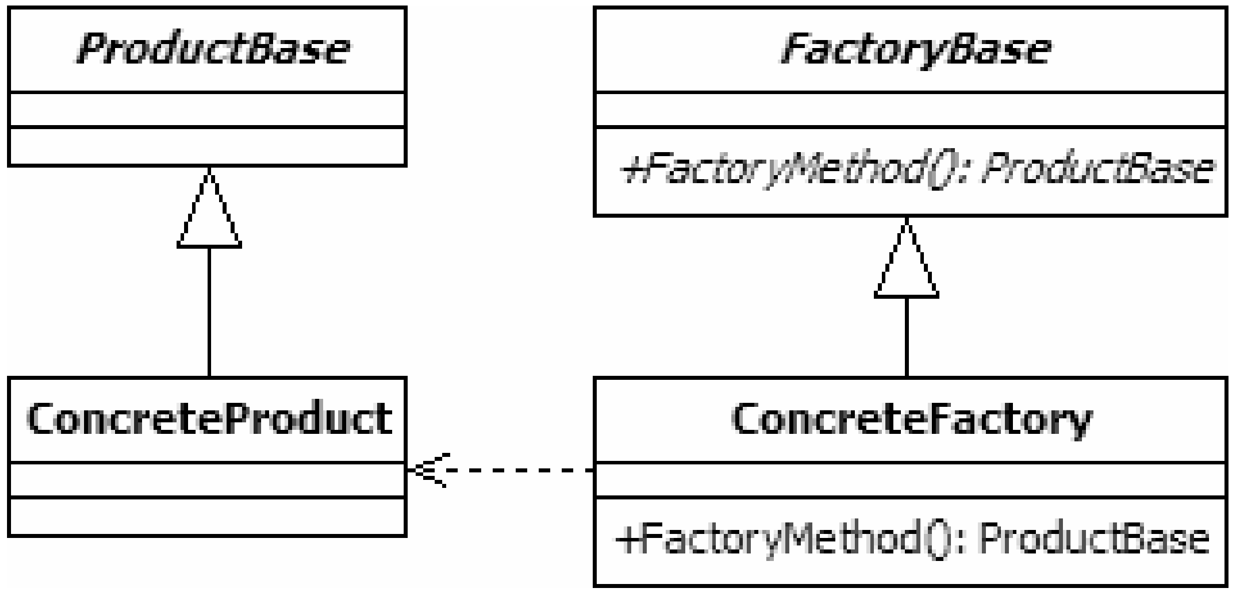
- **The main reason for which the factory pattern is used is that it introduces a separation between the application and a family of classes** (it introduces weak coupling instead of tight coupling hiding concrete classes from the application). It provides a simple way of extending the family of products with minor changes in application code.
- **It provides customization hooks.** When the objects are created directly inside the class it's hard to replace them by objects which extend their functionality. If a factory is used instead to create a family of objects the customized objects can easily replace the original objects, configuring the factory to create them.

## Drawbacks

The factory has to be used for a family of objects. If the classes doesn't extend common base class or interface they can not be used in a factory design template.

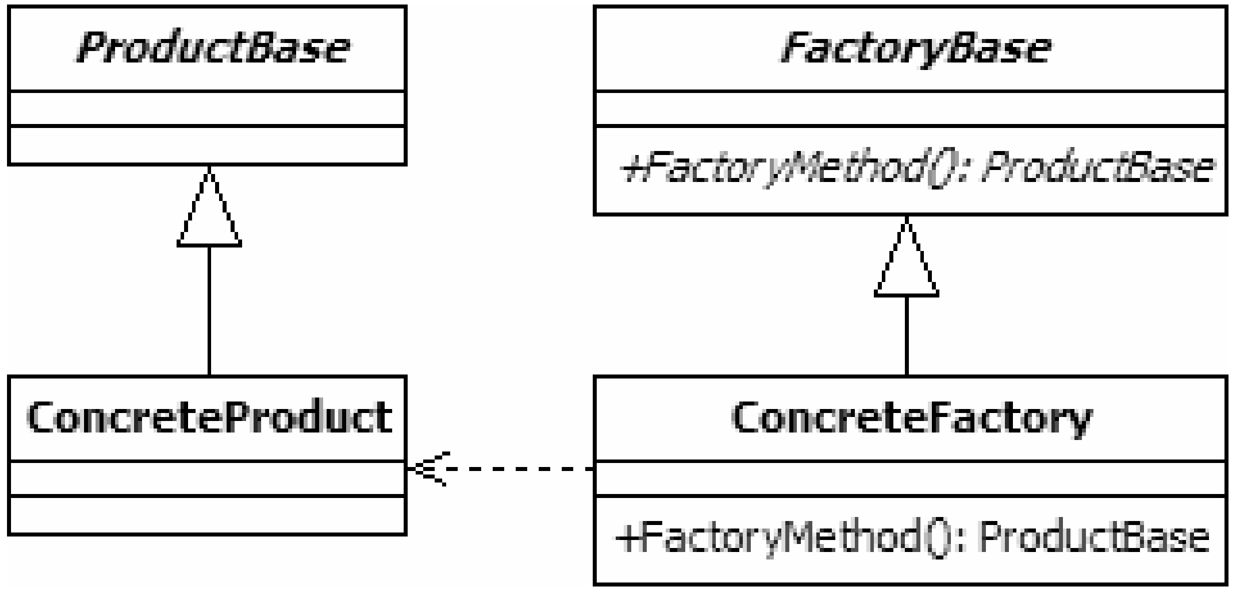
## Implementation

# Factory Method Pattern UML Diagram



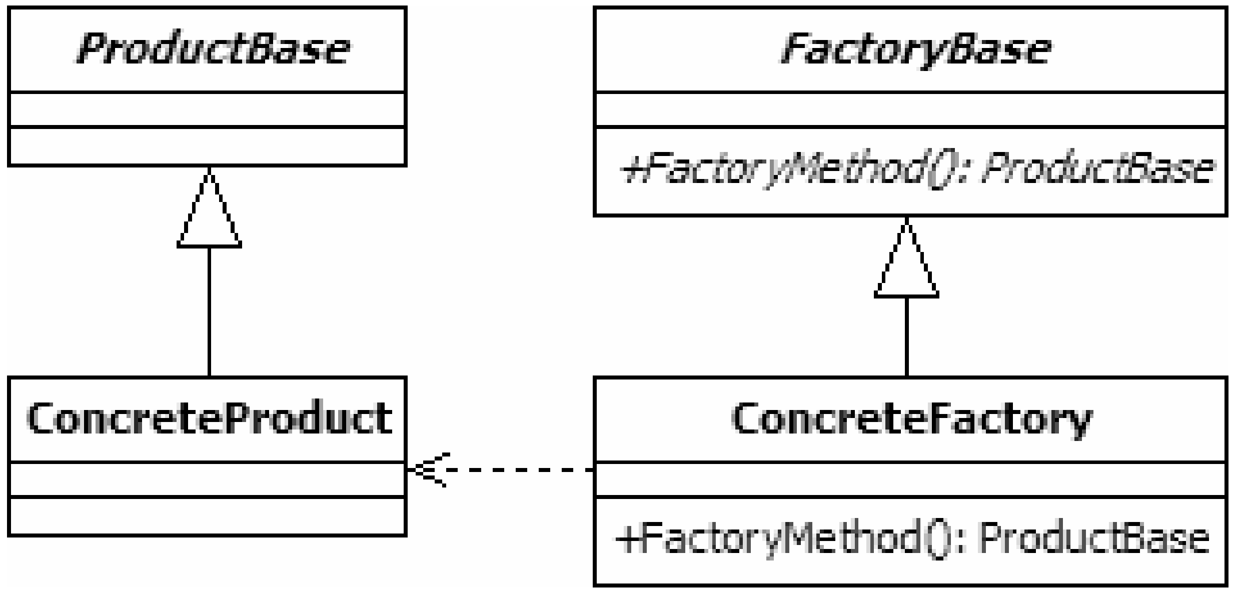
**FactoryBase**. This is an abstract base class for the concrete factory classes that will actually generate new objects. This class could be a simple interface containing the signature for the factory method. However, an abstract class will generally be used so that other standard functionality can be included and inherited by subclasses. In simple situations the factory method may be implemented in full here, rather than being declared as abstract.

# Factory Method Pattern UML Diagram



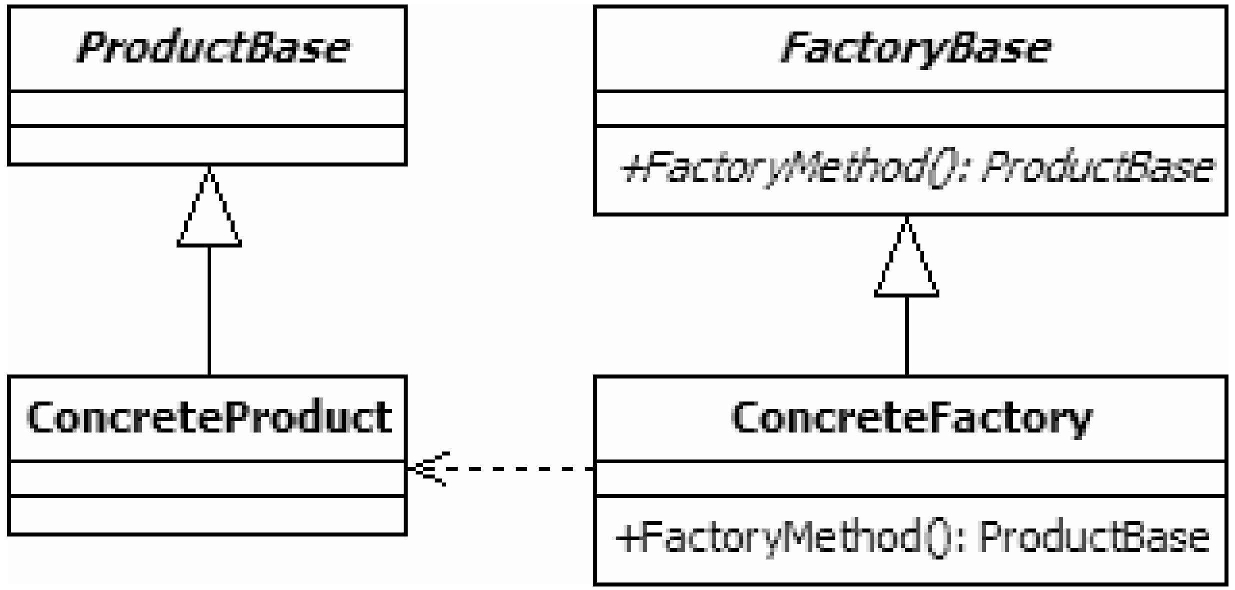
**ConcreteFactory**. Inheriting from the FactoryBase class, the concrete factory classes inherit the actual factory method. This is overridden with the object generation code unless already implemented in full in the base class.

## Factory Method Pattern UML Diagram



**ProductBase**. This abstract class is the base class for the types of object that the factory can create. It is also the return type for the factory method. Again, this can be a simple interface if no general functionality is to be inherited by its subclasses.

# Factory Method Pattern UML Diagram



**ConcreteProduct**. Multiple subclasses of the Product class are defined, each containing specific functionality. Objects of these classes are generated by the factory method.

# Factory Method Pattern Basic Code

```
abstract class FactoryBase
{
    public abstract ProductBase FactoryMethod();
}
```

# Factory Method Pattern Basic Code

```
class ConcreteFactoryA:FactoryBase
{
    public override ProductBase FactoryMethod()
    {
        return new ConcreteProductA();
    }
}

class ConcreteFactoryB : FactoryBase
{
    public override ProductBase FactoryMethod()
    {
        return new ConcreteProductB();
    }
}
```

## Factory Method Pattern Basic Code

```
abstract class ProductBase
{
}

class ConcreteProductA : ProductBase
{

}

class ConcreteProductB : ProductBase
{

}
```

# Factory Method Pattern Basic Code

```
static void Main(string[] args)
{
    FactoryBase theFactory = new ConcreteFactoryA();
    ProductBase theProduct = theFactory.FactoryMethod();
    Console.WriteLine("Created " + theProduct.GetType().Name);
    theFactory = new ConcreteFactoryB();
    theProduct = theFactory.FactoryMethod();
    Console.WriteLine("Created " + theProduct.GetType().Name);
    Console.ReadLine();
}
```

Any questions?